SQUARE – A Migration Language for Robust Data Importing

Version of March 14, 2016

Maikel Krause

SQUARE – A Migration Language for Robust Data Importing

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Maikel Krause
born in Eelderwolde, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

M-industries
Rotterdamseweg 183c
2629 HD Delft
The Netherlands
`m-industries.com`

SQUARE – A Migration Language for Robust Data Importing

Author:	Maikel Krause
Student id:	1511475
Email:	`maikelkrause@gmail.com`

Abstract

When information systems managed by different organizations are integrated, the different parties often must collaborate to define a migration system that transforms the data from one data model to the other. Such a migration step is a common source of system failures. We present a migration language to define transformation definitions for real-time unidirectional data migration (i.e. "importing"), from a relational schema to a data model defined in the Alan modeling language. This migration language has been designed to be transparent (simple to inspect and validate), easy to debug in case of failures, but expressive enough to support common patterns encountered in real-world projects. We examine three different case studies based on real-world software projects to illustrate the characteristics, capabilities, and limitations of the tool.

Preface

Looking up from my workspace every day during the last year or so at M-industries, I would see the following quote hanging on the wall:

> *Perfection is achieved not when there is nothing more to add, but when there is nothing left to take away.*
>
> — Antoine de Saint Exupéry

I think this sums up the philosophy of the company I had the pleasure of working for, and it is that spirit that I have tried to apply to this thesis project.

This thesis report is the end result of my graduation project for the master's degree in Computer Science at Delft University of Technology. The project was performed as part of an intership at M-industries, Delft.

I would like to thank my supervisor, Andy, for his guidance and insight. From day one, he pushed me to strive for the best of my ability and not settle for less. In addition, I want to thank Jan and Guido, for lending me their expertise and teaching me about a great deal of topics.

Special thanks go out to Corno, my daily supervisor at M-industries, without whom this project would not have been possible. His particular brand of thinking has been an inspiration and will probably influence the way I approach problems for the rest of my career. And thank you to rest of the team at M-industries for providing a great, fun, working environment, and a ton of assistance.

Lastly, I am incredibly grateful for my family and friends, who have supported and encouraged me all the way.

Maikel Krause
Delft, the Netherlands
March 14, 2016

Contents

List of Figures

Chapter 1

Introduction

In today's increasingly decentralized business world, information systems are rarely constrained to a single organization. New information systems are commonly created by integrating multiple independent systems together, over a network that crosses organizational boundaries [1].

A consequence is that there is often no longer just one centrally owned database. Different systems within different organizations may need to work with the same data. Exchanging this data requires a *data migration* from the source system's data model to the target system's data model. In practice, this migration step turns out to be a common source of system failures [2].

One of the major challenges in developing a data migration system is that the correctness of the migration cannot be judged by any of the parties in isolation, because we need to have intricate knowledge of both the source and target systems (and how they relate to each other). Being at the intersection of different organizations means that any communication problems will manifest as faults in the migration. Either by crashing the system through a runtime error, or worse, not causing any error at all and simply manifesting as semantically incorrect output at a later stage.

In this thesis report, we focus specifically on unidirectional, real-time data migration, commonly referred to to as *data importing*. Our goal is to find an approach to data importing that leads to more *robust* import systems. By being robust, we mean not just preventing as many errors from occurring as possible, but also helping the developer locate and solve any errors that do occur.

We propose Square, a language to write data import definitions that are:

- *transparent*, in such a way that both parties can easily understand and validate the import logic,
- *easy to debug*, in the sense that any failing assumptions should be reported with clear, human-readable error reports, and
- *expressive* enough to be able to express most import rules found in real-world applications.

There is a trade-off between the first two points (transparency and debuggability), and the third (expressivity). A language which is more expressive is generally more complex and therefore more difficult to read, analyze, and debug. We prioritize the first two, as long as the language is still expressive enough to be practical.

Because we want Square to integrate with pre-existing systems, we need two schema languages that define the kind of concepts we are able to work with, one for the source data and one for the target data. Due to the ubiquity of SQL in business applications, we have chosen to use a source schema language compatible with SQL to represent source data. For the target, we use Alan, a high-level, structured data modeling language that is compatible with common (semi-)structured formats such as XML and JSON.

In this report we provide a conceptual overview of the language, followed by a more formal definition, and finally an evaluation. We evaluate Square using a number of case studies based on real-world projects. This thesis was performed at M-industries, a developer of industrial software located in Delft, The Netherlands.

1.1 Problem Description

The problem we investigate may be characterized as follows. Suppose a software developer needs to build an application that provides some service to a client organization. In order to function, the application requires up-to-date information from the client's operational databases, collectively referred to as the *source data*. In the worst case, this source data may be:

- Heterogeneous. There may be multiple data sources, in different technical spaces.
- Low-quality. The data generally requires data cleaning in order to meet the requirements of the consuming application.
- Dynamic. The operational data is constantly being updated at the client, which means that the target database must be refreshed regularly.

We make no assumptions about the application that consumes this data. However, we assume the existence of a single *target schema* that dictates the format of the data as required by the application. The problem of data importing can then be defined more precisely as follows:

Def. A **data import** is the process of taking a set of source data, and transforming it to a valid instance of the target schema.

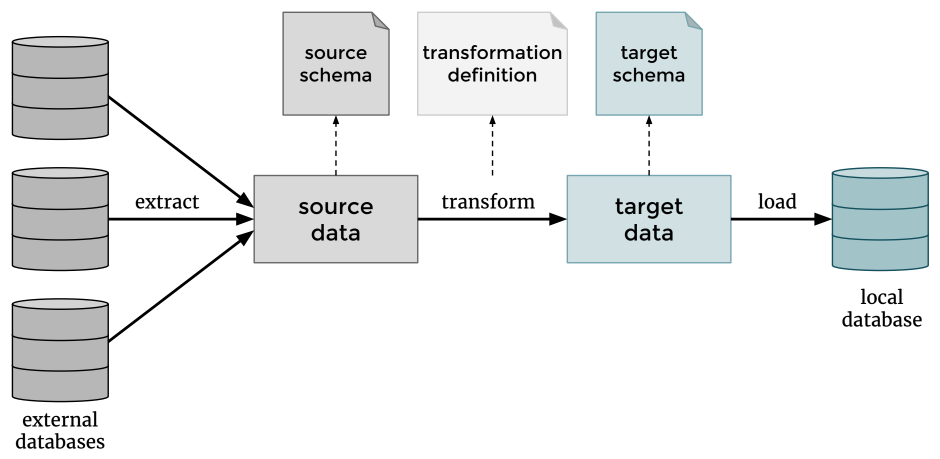

Figure 1.1: Data import process.

This process is closely related to Extract-Transform-Load (ETL) tooling as found in data warehousing [3, 4]. The main difference being that ETL in data warehousing is used to collect data for analytical purposes. Data warehousing is interested in storing large amounts of historical data in an efficient way in order to provide access to this data to knowledge workers. In contrast, we are interested in the more limited use case of integrating different information systems together.

The transformation part of the process is specified by a set of transformation rules called the *import definition*. Such a definition expresses the target schema in terms of the source schema. Note that although there may be multiple sources, we abstract over the integration of the different technical spaces and provide a single "source schema" in a language we can reason about.

This effectively reduces the transformation to the concept of a model transformation, a concept which is well studied in the field of model-driven engineering [5, 6, 7].

1.1.1 Import failures

Converting data from one schema to another is not a novel problem. The challenge we concern ourselves with in this thesis is what happens when an import *fails*. We observe from practice that despite best efforts, import errors still occur frequently.

Def. An **import failure** occurs when the importer is unable to produce a valid instance of the target schema for a given set of source data.

One obvious solution to this problem is to ensure that we *always* produce a valid instance of the target, by enforcing in the import language that the programmer must handle all possible cases. This, however, is impractical. A poorly designed source schema may allow many possible instances that we judge to be invalid, and yet are not properly excluded through schema constraints in the client's database.

Consider, for example, the following table:

Transport log			
id	**order id**	**delivery date**	**transport time (seconds)**
42	106	NULL	NULL
43	101	2016-01-08	3840
44	102	2016-01-10	4120
45	103	2016-01-14	3500

This table contains a log for the transportation of products from a producer to a consumer. By default, the delivery date and transport time are set to NULL. Whenever a delivery truck finishes the delivery of a particular order, the delivery date is updated along with the time it took to transport the goods.

We want to import this data to a logistics scheduler, which aims to optimize inventory management by estimating future transport times based on this historical data. As an application developer, we base our program design on the assumptions we made by looking at the above example data. In a sense, we are *reverse engineering* the underlying business rules, purely by looking at the data.

At some point in time, we receive the following new records:

Transport log (cont.)			
id	**order id**	**delivery date**	**transport time (seconds)**
281	305	2016-02-30	4590
282	308	2016-01-15	0
283	310	NULL	8760

Notice the following “quirks” (highlighted in the table):

- Row `281` contains an illegal delivery date value (the 30th of February does not exist). It appears that the date field can contain an arbitrary string, instead of having been constrained to a proper date type.
- Row `282` contains a transport time of 0. Such a transport time is not possible if we assume different physical start and end locations, and may break our algorithm if we had assumed a positive duration (think division by zero errors).
- Row `283` has a transport time, and yet no delivery date. There is an implicit dependency between these fields that appears to have been violated.

Each of these unexpected values identify an assumption on the developer’s part that has been violated. Either because the input contains a mistake (e.g. typo), *or* because there is a particular business rule that the developer was simply not aware of. For example, a *transport time* of 0 might simply be an encoding that the client failed to communicate. In this example, it may turn out that a transport time of 0 is not a mistake, but rather systematically encodes an order where no physical goods need to be transported (e.g. transfer of digital information).

In order to be able to resolve these types of mistakes as soon as possible, we need a methodology that makes the kind of implicit assumptions illustrated in the example above explicit. By forcing the programmer to explicitly annotate assumptions on the data, the import system itself may be able to provide useful and timely feedback in the case of import errors.

1.2 Research Method

In this section we present our approach in designing and evaluating our research proposal. We formulate a number of research questions (RQ) to guide the rest of the text. We revisit these research questions in the conclusion (chapter 9).

First, in order to study this problem, we need to know about existing research on the topic. To this end, we need to look at current methods.

RQ1 What are the characteristics of current methods? In particular, what kind of static guarantees do these methods give us about runtime failures?

Once we have a clear picture of the current landscape of data importing, we can design a programming model to tackle the problem. We take an empirical approach and use practical case studies to discover, motivate, and evaluate design choices. The case studies should answer the following research question:

RQ2 What kind of data import problems typically occur in industrial projects?

Next, we can design the solution itself. We want to design a language that fulfills the functional needs identified by the case studies, but which succeed at the goals stipulated in the previous section.

RQ3 What kind of language constructs do we need to be able to implement common use cases in way that is transparent and easy to debug?

This report is structured as follows. First, we present a number of relevant concepts as background material for the reader in chapter 2. Next, we give a conceptual overview of Square in chapter 3. In chapter 4, we dive deeper into the type system and other static constraints using a more formal notation. Chapter 5 details the implementation we have built for Square. This is followed by a set of case studies, and an evaluation in chapters 6 and 7. Finally, we present related work and a final conclusion in chapters 8 and 9.

Chapter 2

Background

The topic of data importing touches upon diverse areas of study. In this chapter, we review some of the basic concepts and technologies that we will assume the reader to be familiar with in the remainder of the text.

2.1 Technical Spaces

Data importing inherently requires us to jump technological boundaries. We go from one set of technologies to another. In order to abstract over the concept of a particular technological context, we will make use of the term *technical space* (TS), as discussed by Bézivin et al. in [8]. The notion of a technical space gives us a common framework to talk about and compare different technologies.

A *technical space* is defined as "a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities" [9, 8]. Examples include modeling frameworks like OMG's Model-Driven Architecture (OMG/MDA) [10], programming languages like Java, semi-structured data languages like XML, and relational database systems. All of these spaces are different, but they also have certain interesting aspects in common.

2.1.1 Three-level organization

When we look at different artifacts within technical spaces, we see a common pattern. Different artifacts are often *described* by a meta-level artifact. The relation between the two is characterized by a "conforms-by" relation. For example, an UML-model in MDA conforms to the UML modeling language, which itself conforms to the MOF (Meta-Object Facility). An XML document conforms to an XML Schema, which itself conforms to the XML Metaschema. An SQL (Structured Query Language [11]) database conforms to an SQL schema, which itself may be described using an SQL metaschema (many databases support the ANSI-standard Information Schema [12]).

This hierarchy is common across a surprising amount of contexts. This is captured by the three-level organization shown in figure 2.1. In this organization, we identify three different levels. At level M1, we find the elementary models that serve as our abstract representation of the system under consideration (the latter is sometimes included in this context as "M0"). The language that these models are written in, or *metamodel*, is at M2. Finally, at M3 we find a *metametamodel* which serves as the

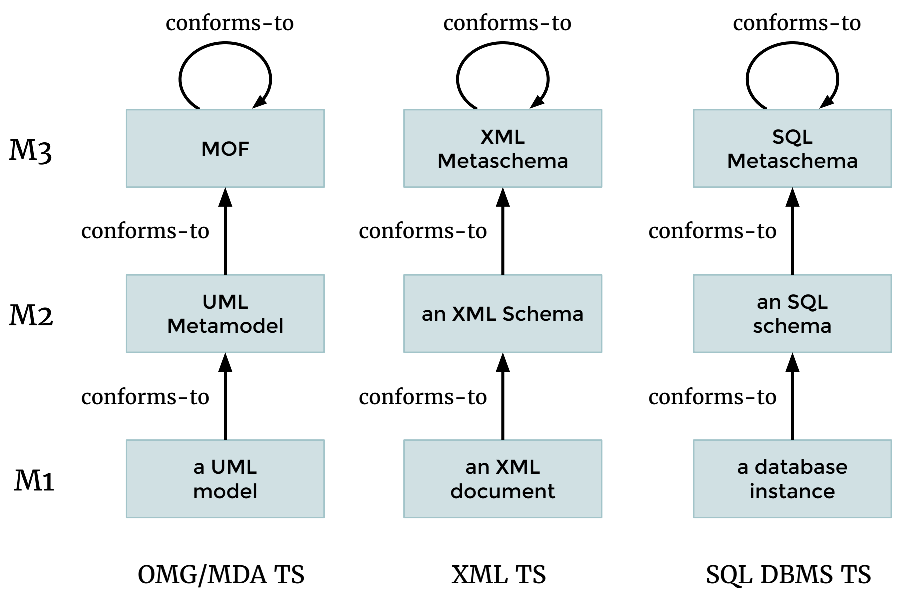

Figure 2.1: Three-level model organization.

general framework to talk about all metamodels. Usually, the metametamodel is self-describing (i.e. conforms to itself), providing a natural end to the chain.

2.1.2 Tool support

A technical space is usually accompanied by tools that work on artifacts in that space. These tools are made to work with the general concepts captured by the metametamodel.

A basic function needed to work with any sort of model is the access and retrieval of information. A *navigation language* (or *query language*) is a language to express this kind of access specifically for a particular TS. Examples include the Object Constraint Language (OCL) for MOF [13], the XQuery and XPath languages for XML [14], and SQL for relational databases.

Also common are *transformation languages* to operate on artifacts, returning another artifact of the same type. XSLT [15] and XQuery fulfill this role for XML, as does SQL for relational databases.

2.1.3 Technical space projectors

Having a formal notion of a technical space is especially useful when we start to integrate different technologies. In this case, we want to extract a model in the one space (the target) from a model in another space (the source). This kind of transformation is different from the kinds of transformations we mentioned in the previous paragraph, which only work *within* a particular space. Bézivin et al. introduce the concept of a *technical space projector* for transformations that cross technical spaces [8].

Note that, in general, a technical space projector is not necessarily the same thing as a data import system. An important distinction here is that a projector may create any new model from a source model. This allows it to be defined very generally (potentially, for any model in a TS). In a data importer, both the source and target model are predefined. Generating an import definition for any pair of source and target models is impossible, because the transformation depends on the semantics of each model element. We can view each data importer as a technical space projector which only works on a specific subset of source and target models.

2.2 Relational Databases

The first technical space we examine is the space of relational databases. This is the space that serves as the source of our importer.

The relational model, first formulated by Edgar F. Codd in his seminal 1970 paper [16], is a database model based on first-order logic. In this model, data is organized into *relations*, containing a set of tuples that map attribute names to values from some data domain. More formally, we say that a relation R is defined as:

$$R = (\text{heading}, \text{body})$$

Here, the *heading* specifies a set of k attribute names and corresponding domains $d_1, \ldots, d_k$. The *body* is then a set of tuples, each of which a member of the Cartesian product $d_1 \times \cdots \times d_k$. We call the number of attributes the *degree* of R, and the number of tuples the *cardinality* of R (denoted $|R|$).

A relation may be thought of as a representation of some first-order logic predicate. For example, the predicate:

$$W = \text{“Person P works in department D.”}$$

This is a relation with two attributes: P of domain *persons*, and D of domain *departments*. Every tuple contained in the relation encodes a true proposition, where the attribute values correspond to a particular combination of person and department (e.g. $\langle \text{John}, \text{Marketing} \rangle$).

Following common terminology in database systems, we may also refer to relations, attributes, and tuples as *tables*, *columns*, and *rows*, respectively. Equivalently, we may borrow from data modeling and use the terms *entities*, *fields*, and *records*.

2.2.1 Schemas

A *relation schema* is a definition for some class of relations. Such a schema consists of two parts: the heading defining the attributes and data domains, and a set of integrity constraints that must hold for any given instance. Typical constraints that may be specified in a relation schema are:

1. **Key constraints**. A *key* is a subset of attributes for which holds that no two tuples in the relation contain the same combination of values. We commonly identify a single *primary key* (denoted $\text{key}(R)$) to serve as the identity of the tuple, and in addition there may be zero or more *alternative keys* that serve as additional uniqueness constraints.

2. **Referential constraints**. A relation schema may define zero or more *foreign keys*, which map a subset of attributes in R to attributes in another relation S. Each tuple in R must then refer to some tuple present in S through the foreign key correspondence (with the exception of NULL values, discussed in section 2.2.2).

We add the restriction (common in database systems, although not universally enforced) that each foreign key is defined in terms of a key in S, which ensures that each foreign key value can only ever refer to a single tuple in S.

A *relational database schema* (often abbreviated as *relational schema*, not to be confused with the term "relation schema") is a set of named relation schemas.

2.2.2 Three-valued logic

Although not part of the original relational model, most database systems support a form of three-valued logic. A special NULL value is allowed wherever a regular value is, indicating a "missing" or "unknown" value. NULL values have the special property that they are ignored by foreign key constraints, i.e. they can be used to specify the *lack of* a link to another table (for example, the "parent" of the root in a tree). They may also arise as the result of particular operations. For example, outer joins in SQL use NULL values to fill up values where no record could be matched.

For models that support three-valued logic, we allow the following schema constraint type in addition to the ones defined in the previous section:

3. **Non-nullability constraints**. Declared for a particular attribute to prevent that attribute from taking on a NULL value.

2.2.3 Query languages

Part of the relational theory are two mathematical systems that serve as the foundation for query languages: relational algebra and relational calculus (proven to be equivalent by Codd [17]). These systems have inspired a number of database languages, the most popular among which is SQL.

SQL consists of two main parts: (1) a data definition language, that allows a database designer to define a relational schema, and (2) a query language based on relational algebra. A *query* in a relational database is the primary mechanism to retrieve or transform data. The main operations, using the terminology from relational algebra, are:

- Basic set operations, such as **set union** and **set difference**.
- **Selection** of records based on a predicate function.
- **Projection** of a relation to a new set of attributes.
- **Join** operations, combining records from two different relations.

The expressive power we get when combining these operations is well-known, as discussed in [18]. SQL goes further and adds **arithmetic**, **grouping**, **aggregation**, and even **recursion** operations that extend the expressive power of the language (at the cost of increased computational complexity) [19].

A *view* is a named query, which may itself be queried like any other relation. Views are the basic tool for mappings in the relational model, with sources (input relations) and a target (output relation).

2.2.4 Constraint languages

The integrity constraints mentioned so far (key constraints, referential constraints, and non-nullability) do not provide much expressive power. In particular, we cannot express the following types of constraints:

- Complex value constraints (e.g. a regular expression match on a text field).
- Constraints that describe dependencies between different fields or relations. For example, we cannot specify that fields A and B must either both be NULL, or both not NULL.
- Constraints that prevent recursive links in a data structure. There is, for example, no general way to represent a tree in a relation with the guarantee that there are no cycles.

Such constraints are useful to encode the *business rules* of a particular organization. Usually, business rules are considered separate from the data store itself, enforced through additional layers (systems or policy) on top of a database system.

SQL does define mechanisms to express more powerful constraints. *Check constraints* may be specified as part of the schema to check the validity of a row after an insert or update. However, these checks are limited to the scope of a single row. For more power, *triggers* allow arbitrary procedures to run on specific events.

There have also been attempts to extend the relational model with constraint languages more powerful than first-order predicate logic. In particular, fixpoint logic has been applied to relational schemas in order to restrict recursive structures [20].

2.3 Data Modeling

In the previous section we looked at the data format of databases from which we want to import. We use the relational model because that is what is commonly used in industry. For the target schema, we have more freedom to chose a modeling language. The relational model is ubiquitous, but not necessarily the best way to create a high-level, conceptual model. In the next chapter (and appendix A), we describe our chosen data modeling language. In this section, we provide some background on data modeling in general.

A *data model* is a conceptual representation of some real world problem domain. The term is slightly ambiguous. Some authors use the term “data model” to mean a specific kind of “model”, as defined in the three-level model organization (i.e. level M1 in figure 2.1). In other contexts, it is used to mean a meta-level artifact that describes data (i.e. a metamodel, which is level M2 in figure 2.1). Following the terminology common in industry, we use the term in the latter sense (a metamodel). To avoid ambiguity, in this report we will mostly avoid the term altogether and use “schema” instead, and use “(schema) instance” to denote the data itself.

2.3.1 Model transformations

A lot of data processing may be viewed as a transformation from one data model to another. For example, when we need to convert from a relational database to an object-oriented model (the well-known Object-Relational Mapping). But also other types of processing where traditionally we might not think in terms of “data models”, such as compilers (converting from one language to another).

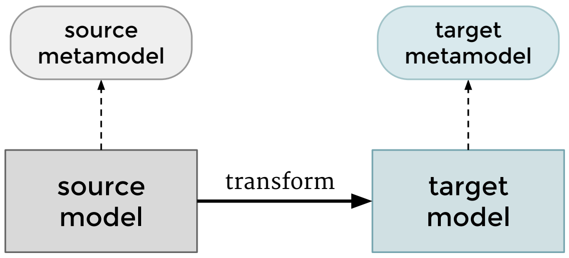

Figure 2.2: Model transformation.

A *model transformation* is a mapping from some input model to an output model [5, 6, 7]. Model transformations may be specified in a general-purpose programming language, but there are also specialized transformations languages that are designed for particular input and output metamodels. QVT and ATL are transformation languages for metamodels defined within the MOF framework [21]. There are also frameworks specialized in transformation of programs, including Stratego/XT [22].

2.3.2 Model-driven engineering

The idea of a model transformation is a fairly powerful concept. We can use it to develop a data processing tool, but also a language compiler for example. Using the same principle, we can take a data model, and transform that model to an executable format. *Model-driven engineering* (MDE) is a methodology that takes uses this concept to build entire software applications [23, 24, 25].

Using this approach, we can solve problems by creating high-level, domain-specific models or languages specially tailored for the problem at hand, and use generic tooling to generate the required software artifacts (like database schemas, user interfaces, business logic).

A number of toolsets have been developed to support this type of development, including “language workbenches” like Spoofax, and Ensō [26, 27]. These tools provide an integrated environment to define new domain-specific languages and transform them using specialized transformation languages.

2.4 Data Quality

Data quality is a measure of how fit data is for use by data consumers [28]. Databases tend to contain a lot of errors – in the form of inconsistencies, redundancies, incompleteness, etc. – that make it harder (or impossible) for a data consumer to perform the task at hand. When we import data into a system, we want to detect and possibly rectify ("clean") as many of these errors as we can. To this end, we will give an overview of the different classes of data quality issues that may occur in databases. The classification we present is based on the work done by Strong et al. [28], as well as the analysis of Rahm et al. in [29].

Note that in our definition of data quality, we place special emphasis on the data consumer. Rather than seeing data quality as an intrinsic property of data, research has shown that it is important to see these issues within the broader context of the users [30].

2.4.1 Accuracy

The *accuracy* of data is a measure of correctness. In other words, does the data accurately reflect the real situation it is supposed to represent? The producer of the data, being the one with the knowledge of the true facts underlying the data, is often the only one who can judge this particular quality.

For this reason, we split up inaccuracies into **detectable** and **indetectable** issues.[1] Detectable issues are issues that are easily recognized by an automated process, simply by looking at the data. Indetectable issues on the other hand, are indistinguishable from accurate data, but are semantically wrong. This type of issue should not cause the consuming application to crash (because the "form" is correct), but may lead to wrong results.

Detectable inaccuracies		
Scope	**Problem**	**Example**
Field	Illegal values	`birthdate=1990-01-99`
Record	Violated dependencies	`<gender="male", num_pregnancies=2>`
Record type	Uniqueness violation	`<name="Smith", SSN="123456">` `<name="Miller", SSN="123456">` (SSN should be unique)
Source	Referential integrityviolation	`<name="John Smith", deptno=78>` (Reference does not exist)

Table 2.1: Examples of detectable inaccuracies in data.

[1]These definitions are similar to the "schema-level" and "instance-level" classification as defined by Rahm et al. [29]. We have slightly changed the terminology to decouple it from the expressive power of any particular schema language.

Indetectable inaccuracies		
Scope	**Problem**	**Example**
Field	Dummy values	`phone="999-99999"`
	Misspellings	`city="Liipzig"`
	Misfielded values	`city="Germany"`
Record type	Duplicated records	`<name="John Smith">` `<name="J. Smith">`
	Contradicting records	`<name="J. Smith", bdate=1990-01-01>` `<name="J. Smith", bdate=1984-05-31>`
Source	Wrong references	`<name="John Smith", deptno=17>` (Reference exists, but is wrong)

Table 2.2: Examples of indetectable inaccuracies in data.

Tables 2.1 and 2.2 show a few examples of detectable and indetectable accuracy issues. Some of the issues in table 2.1 could be prevented by typical relational database constraints (e.g. referential integrity constraints), but in general a more powerful constraint language is needed to catch all of these errors.

2.4.2 Representational quality

Representational issues are those issues where the data is not “wrong” per se, however the manner in which the data has been represented makes it unnecessarily difficult for the consumer to interpret.

Representational Issues		
Scope	**Problem**	**Example**
Field	Cryptic values	`experience="B"`
	Embedded values	`name="J. Smith, New York"`
Record type	Word transpositions	`<name="J. Smith">` `<name="Miller P.">`

Table 2.3: Examples of representational issues in data.

Table 2.3 shows a few examples. Cleaning these issues requires some normalization procedure.

2.4.3 Contextual quality

The third class of data quality issues are the *contextual quality* issues. These are issues which make it harder to consume data when we consider what the data will be used for. For example, a database where we are only interested in a small part of the data contains mostly *irrelevant data*. Another example is localization, where the data consumer needs to perform extra conversions (e.g. currency conversions) to get the data in a suitable format.

2.5 Data Warehousing

A *data warehouse* is a system used by organizations to collect data from different sources, and present that data to end-users [3]. Among the functionality associated with data warehousing are the integration of data from various (possibly heterogeneous) sources, cleaning of the data to improve the data quality, and the provision of various analytics tools for end-users (data exploration, visualization, reporting, etc.)

2.5.1 Extract-Transform-Load

There are several inherent problems involved in migrating data from external sources:

- The incoming data must be mapped to a common schema.
- Data needs to be cleaned to ensure the end-user gets high-quality data.
- The data must constantly be refreshed in order to have up-to-date information.

The process that tackles these problems is traditionally subdivided into a three-phase process called *Extract-Transform-Load*, or ETL [4]. In the extraction phase, the different data sources are consolidated and possibly translated to an intermediate format suitable for data processing. In the transformation phase this data is then processed and transformed into an instance of the target schema. Finally, the processed data is loaded into the target database.

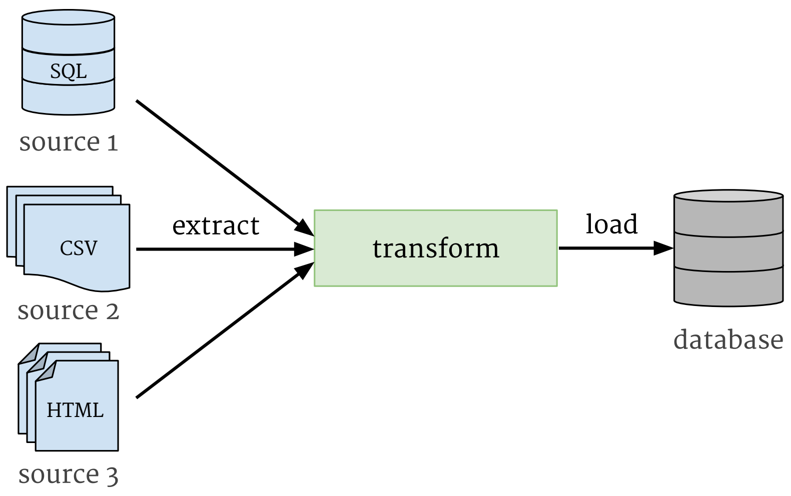

Figure 2.3: Overview of the ETL process.

What makes ETL different from just a view over a set of relations, is that ETL needs to map data from external sources to a local target. That means that we are dealing with separate systems, each of which in its own technical space and thus having a different *metamodel*.

Vassiliadis et al. [31] present one of the earliest conceptual models for ETL. Their work presents a formal conceptualization of ETL activities, a set of commonly used ETL operators as identified by the authors, and mechanisms for custom transformations and constraints. The authors use a custom graph-based metamodel consisting of a few types of nodes, including *concepts* and *attributes* for data, and *providers* and *transformations* to relate target nodes to source nodes.

More recent research includes ETL models using more standard metamodels such as UML [32], alternative transformation methods such as those based on the logical programming paradigm [33], and semi-automatic methods using semantics-aware domain models [34].

2.5.2 Data cleaning

Data cleaning is the process of detecting, and possibly correcting, quality issues in data. This process is often performed as part of ETL.

A number of tools exist to interactively explore and analyze data sets in order to identify problems, and possibly rectify these directly. Examples of such tools are DataWrangler and OpenRefine [35, 36]. Once applied to a specific data set, the operations performed may be saved into an automated process.

Certain classes of problems can be discovered automatically through data mining and knowledge discovery techniques [37]. For example, an automated tool may look for patterns in data that are likely to indicate duplicate records [38, 39].

Chapter 3

Square

In this chapter, we introduce *Square*. Square is an import definition language, which aims to make data import systems more robust in the face of runtime failures. We achieve this through a methodology that takes into account the *uncertainty* inherent in predicting what kind of data the system may encounter in the future. This methodology is supported by a custom type system and two specialized definition languages. We focus heavily on static (compile-time) guarantees to verify certain properties of the transformation.

The purpose of this chapter is to give an informal introduction to Square. A programmer reading this chapter should be able to get a feel for the programming model, and be able to write their own programs. To illustrate the different concepts, we provide code samples written in our own custom syntax. In section 3.5 we describe the methodology. A more formal description of the type system and other static constraints is given in chapter 4.

3.1 Overview

A Square execution extracts data from one or more external sources, transforms it according to an import definition, and finally loads it into a local database. Figure 3.1 shows the data flow for a given execution.

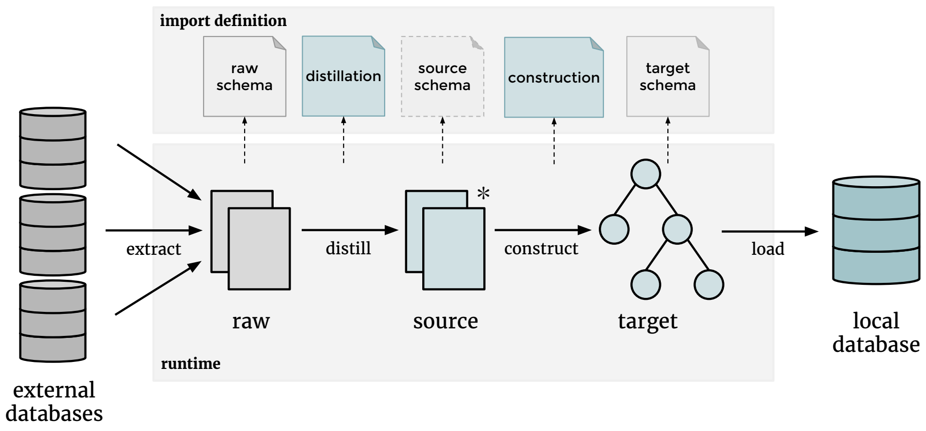

Figure 3.1: Data flow of a Square import.

At the two boundaries we have the extraction of data from external databases, and the loading of the data into the local database. These two steps are beyond the scope of Square itself, and are instead described when we look at the implementation in chapter 5.

The tranformation is split into two phases: *distillation* and *construction*, each of which is configured through a corresponding definition language. The choice for this split comes from the observation that we often want to first break apart the high-level structure of the data in order to obtain a more convenient structural description of the data. When this is done, we can focus on "building up" the individual bits of data to match the target schema.

3.1.1 Import definition

There are two schemas that define the input and the output, respectively. These schemas are given beforehand by the two different parties involved:

- The **raw schema**, which describes the external databases collectively.
- The **target schema**, which describes valid output instances.

The transformation itself is defined by two definitions, one for each of the two-step process:

- The **distillation**, which defines the distillation step that transforms an instance of a raw schema to a source schema.
- The **construction**, which defines the construction step that transforms an instance of a source schema to the target schema.

The **source schema**, which is the output of the distillation step, is defined implicitly by the distillation (indicated by the dotted lines in figure 3.1). Both the raw schema and the source schema are described by an SQL-compatible language, which we call the *source schema language*.

3.1.2 Source schema language

Because we want to be able to import data from pre-existing enterprise databases, we need a source schema language that matches most existing database schemas as closely as possible. Due to the prevalence of SQL-based database systems in the enterprise world, we have chosen a source schema language that is compatible with the SQL data definition language (DDL).

The data types that we support in this schema language are listed in table 3.1. Since we are not concerned with data storage, the data types are defined to be as broad as possible in order to be able to handle values regardless of precision (limited only by system memory). Note that we also include an `enum` type, which is non-standard, but supported by a number of popular database systems (most notably, MySQL and PostgreSQL).

Name	SQL equivalents	Description
`text`	`CHARACTER`, `VARCHAR`	Arbitrary-length textual value
`integer`	`INTEGER`	Arbitrary-precision integer
`decimal`	`FLOAT`, `DECIMAL`	Arbitrary-precision floating point number
`boolean`	`BOOLEAN`	Boolean (true or false)
`date`	`DATE`	Absolute date value
`datetime`	`TIMESTAMP`	Absolute time value
`enum`	`ENUM`	One of an enumerated set of options

Table 3.1: Data types.

In addition, we support the following features from the SQL DDL:

- Basic SQL constraints (key constraints, referential constraints)
- NULL values, and non-nullability constraints

We currently do not support arbitrary check constraints (i.e. SQL `CHECK` clauses). Recall from section 2.2.4 that check constraints allow database designers to specify predicates on single rows, to be checked by the database engine on insert or update. Supporting check constraints would require us to map the entire SQL constraint sublanguage over to our own schema language, which is possible but requires some effort. Support for check constraints varies strongly among database systems, with some systems (e.g. MySQL) not supporting this feature at all, and some supporting a superset (e.g. support for subqueries that can check beyond the scope of a single row).

We have not encountered any check constraints in our case studies thus far, and therefore have chosen not to support this feature. In the future however, check constraints could provide a useful additional source of integrity constraints.

3.1.3 Target schema language

The target schema serves as the importer's interface to the application that consumes the data. Such a schema should detail the exact requirements of the data as imposed by the application.

The schema language we have selected for this purpose is *Alan*. Alan is a high-level data modeling language created at M-industries, to serve as the modeling language for their model-driven engineering (MDE) platform. The language is comparable to schema languages for (semi-)structured data languages like XML Schema and JSON Schema, although unlike the former languages it was designed specifically with MDE in mind. We provide a more detailed description of Alan in appendix A. A formal grammar of the language can be found in appendix B.

The fact that Alan was designed for MDE makes it a convenient fit for our purposes, because it means we know that *any* instance that conforms to an Alan schema must correspond to a supported use case in applications generated from this model (i.e. the application should not crash on any valid instance). Hence, we can treat the target schema as being completely representative of the consuming application. Furthermore, the fact that Alan is compatible with languages like XML Schema means that we can use Alan as an intermediate format for many pre-existing systems.

```
(
    "description" :: text
    "version" :: integer
    "product types" :: collection (       // Key-value pairs
        "description" :: text
        "operations" :: collection (
            "instruction" :: text
        )
    )
    "orders" :: collection (
        "product" :: reference <../"product types">
        "quantity" :: integer
        "status" :: option (
            "pending" :: ()
            "finished" :: (
                "finish time" :: integer
            )
        )
        "required operations" :: collection of <"product"/"operations"> (
            "additional remarks" :: text
        )
    )
)
```

Code Snippet 1: An Alan model.

The code sample above shows an Alan model for a simple manufacturing company, with two main entity types: `product types` that describe the kinds of products the company is able to manufacture, and `orders` that represent the actual incoming orders from clients for a particular amount of a product.

An Alan *instance* that conforms to this model is a tree structure, which fills in a valid value for each Alan property. A formal algorithm describing conformance rules is given in appendix A.

3.2 Distillation

The distillation phase gives the programmer the opportunity to clean up structural aspects of the data. The goal is to identify the logical entity types and fields embedded in the raw data, as well as create traversable links between the types.

If we think in terms of the ANSI/SPARC three-schema architecture [40, 41], the goal of this phase may be thought of as “undoing” the optimizations that have been made in going to a *physical schema*, and converting it back to a *conceptual schema*. Some examples of cleanup operations we may want to perform during this phase are:

- Combine tables that have been partitioned (either horizontal or vertical), despite being part of a single logical entity type.
- Normalize data that has been denormalized (e.g. for performance reasons).

- Identify links between tables that exist despite lacking an explicit foreign key constraint.
- Rename identifiers (table and column names) to more human-readable equivalents. For example, IBM DB2's 10-character identifier limit means that names are often "fudged" (forcefully shortened to fit within the character limit).

All of these operations either undo bad design, or undo optimizations made for secondary requirements, like performance or availability. In an ideal scenario, none of this would be necessary, and the schema of the input data would already match our desired source schema. In other words, we want to stick to the *intent* of the system designer as much as possible. This is important, because we want the client to be able to study and validate the correctness of the resulting source schema, in order to prevent faults caused by communication errors.

3.2.1 Table definitions

Code snippet 2 shows a basic distillation of a table containing specifications of products.

```
table "products" <= map external "CMPNY.PRODUCT"
    column * "product id" :: text <= .["PROD_ID"]
    column "description" :: text <= .["DESC"]
    column "stock quantity" :: integer <= .["STOCK"]
```

Code Snippet 2: Basic distillation example.

This example describes a table `products` with three columns. The primary key is denoted by the columns marked with an asterisk (∗), which in this case is just the single product ID. Each table and column takes an *import expression* (marked by `<=`) that describes how to import a new instance from an input.

A table import expression must start with one of several possible *operations*:

- `map` (or `map on <key name>`) performs an exact one-to-one mapping from the input table to the output. To guarantee that we do not end up with conflicting indices, the table definition must preserve the primary key (or an alternative key) of the input table. This is checked by the compiler (cf. chapter 4).
- `aggregate` allows a new primary key definition, but forces the programmer to handle conflicting values at the column level, by using the available aggregation operators (e.g. `min`, `max`, `concat`).

The input must be in the form of a table. The keyword `external` may be used to refer to a table in the external schema, but we can also use sibling table definitions in the distillation, using the syntax `table <table name>`.

A table import definition is similar to a "view" in relational terms, but instead of defining one big query we express the result in terms of a table definition instead. This gives us the benefit of adding integrity constraints on the result, something which is not possible in SQL views.

3.2.2 Filters

We can filter input tables to reduce them to a relevant subset, using the `filter` operator. This is similar to a *selection* operation in relational algebra, or a *WHERE* clause in SQL terms.

```
table "products"
    <= map external "CMPNY.PRODUCT"
        filter .["ACTIVE"] == "1"
        filter .["STATE"] == "P"
    //...
```

Code Snippet 3: Filters on an import expression.

3.2.3 Keys

Keys are sets of columns that represent a uniqueness constraint on the table. The primary key is defined by annotating columns with an asterisk (∗), but we also support alternative, named keys.

```
table "users" <= map external "CMPNY.USER"
    column * "user id" :: text <= .["USER_ID"]
    column "email" :: text <= .["EMAIL"]

    key "unique email address": { "email" }
```

Code Snippet 4: Alternate key definition.

3.2.4 Links

We can add links from one table to another, as shown in snippet 5. Links are the primary mechanism to traverse from one entity to another, which will become an important mechanism later in the construction phase.

```
table "orders" <= map external "CMPNY.ORDER"
    column * "order id" :: text <= .["ORDER_ID"]
    column "product id" :: text <= .["PROD_ID"]
    link "product": single "products"
        <= { "product id" <= .["product id"] }
```

Code Snippet 5: Links on a table.

Links can be marked as `single` or `many` (the "multiplicity" of the link), depending on whether the link resolves to exactly one or a set of rows, respectively. The import expression that follows specifies the foreign key that links the two tables.

Besides these "forward links", where the foreign key columns are part of the table itself, we also allow "reverse links", that allow you to name the other side of a relationship (thus creating a bidirectional link).

```
table "products" <= map external "CMPNY.PRODUCT"
    //...
    link "orders": many "orders"
        <= reverse "product"
```

Code Snippet 6: Reverse links.

In general, forward links tend to be singular, and reverse links plural. This is due the way foreign keys work in the relational model. Forward links are singular, because we constrain foreign keys to only ever map to a key of the referenced table (as is customary in most relational database systems). There are some cases where we allow singular reverse links, in order to model one-to-one relations. The exact semantics are provided in section 4.3.

3.3 Construction

A construction is a projection over the target schema (the Alan model), where each property in the target schema is annotated with an import expression, which tells the importer how to create an instance of that property. From a high-level perspective, a construction is a function that takes as input a source instance (i.e. the result of a distillation), and returns as output either (1) a valid target instance, or (2) an error report in case of failure.

```
(
    // Basic expressions
    "description" :: text <= "Example."
    "version" :: integer <= 42

    // Collections
    "orders" :: collection <= map table "orders": encode key -> (
        // Field access
        "quantity" :: integer <= .["quantity"]
        // Option selection
        "status" :: option <= case
            .["status"] == "pending"
                => "pending" :: ()
            .["status"] == "finished"
                => "finished" :: (
                    "finish time" :: integer <= .["finish time"]
                )
            otherwise
                => error <!>
    )
)
```

Code Snippet 7: A basic construction example.

When a construction definition is compiled, the compiler checks whether the construction represents a correct projection over the target schema (i.e. the construction tree must match the target schema tree). The language is statically typed; the compiler checks that all construction expressions are of a type that corresponds with the type of the property.

A complete overview of the construction syntax is found in appendix B.

3.3.1 Property types

Alan offers the following property types:

Property type	Instance result
`text`	Textual value (a string).
`integer`	Arbitrary integer value.
`decimal`	Arbitrary decimal value.
`option`	A choice out of several pre-defined options.
`collection`	Mapping from keys (strings) to entries.
`collection-of`	Collection where the key set is a subset of another collection.
`reference`	Reference to a single entry in a collection.
`component`	An instance of a named component type.

Table 3.2: Alan property types.

For a given property in a construction, the programmer is required to specify an operation that results in a value of the corresponding type. There are a fixed set of operations per property type to choose from. Examples of each are given below.

3.3.2 Property expressions

In the following code samples, the placeholder `<exp ::` *type*`>` represents a primitive expression of the type *type*. Expressions are written using a generic expression language, described in section 3.4.

Primitive operations

```
"name" :: text <= <exp :: text>
"quantity" :: integer <= <exp :: integer>
"weight" :: decimal <= <exp :: decimal>
```

Option operations

```
"type" :: option <= "a" :: ()  // Unconditional choice selection
"type" :: option <= error <!>  // Unconditional failure

// Select the first choice where the predicate expression is true
"status" :: option <= case
    <exp :: boolean> => "pending" :: ()
    <exp :: boolean> => "finished" :: ()
    otherwise => "unknown" :: () // Fallback (required)

// Cover all the options of an enum
"gender" :: option <= match enum <exp :: enum>
    "M" => "male" :: ()
    "F" => "female" :: ()

// Select either possibility of an optional-typed value
"has task" :: option <= settle
    "task" = <exp :: optional<T>>
        => "yes" :: (
            // Constants with the expression results are available in this scope
            "task" :: reference <../"tasks"> <= $"task"
        )
    none => "no" :: ()

// Try to evaluate an unsafe expression, use fallback in case of failure
"has parent" :: option <= try
    "parent" = <exp :: unsafe<T>>
        => "yes" :: (
            // Constants with the expression results are available in this scope
            "parent" :: reference <..> <= $"parent"
        )
    catch => "no"
```

Collection operations

```
// Map a set of rows to a collection, using a text expression as collection key
"machines" :: collection <= map <exp :: rowset>:
    key <!> <exp :: text> -> ( // Note: unsafe (risk of key clashes)
        "description" :: text <= .["description"]
    )

// Map a set of rows to a collection, using the primary key as collection key
"products" :: collection <= map <exp :: rowset>:
    encode key -> ( // Safe (uniquely encodes primary key values as key)
        "quantity" :: integer <= .["quantity"]
    )

// Map a set of rows to a collection, using an alternate key as collection key
"users" :: collection <= map <exp :: rowset>:
    encode key "email" -> ( // Encodes the alternate key named "email"
        "name" :: text <= .["name"]
    )
```

Reference operations

Reference operations (and the collection-of operations below) are slightly tricky, because to be correct, we must select an *existing* entry in the referenced collection. We provide several operation types, some of which may result in a failure (i.e. are “unsafe”), and one which the compiler guarantees to be safe (but is not always possible).

```
// Select an element from the collection by specifying the key directly
// (Note: unsafe, because the key might not resolve to an entry)
"machine" :: reference <../"machines"> <= key <!> <exp :: text>

// Select an element from the referenced collection by arbitrary row
// (Note: unsafe, because the row might not be part of the collection)
"product" :: reference <../"products"> <= index <!> <exp :: row>

// Select an element from the referenced collection by row entry
// (Note: safe, but only allowed if the compiler can verify correctness)
"order" :: reference <../"orders"> <= entry <exp :: row>
```

Collection-of operations

Similar to references, but here we need to verify that we have a *subset* of the entries in the referenced collection.

```
// Arbitrary rowset (unsafe)
"my products" :: collection of <../"products">
    <= intersect <!> <exp :: rowset>

// Rowset that forms a subset, as verified by compiler (safe)
"custom operations" :: collection of <../"operations">
    <= subset <exp :: rowset>
```

3.4 Expression Language

The distillation and construction languages are purposefully kept separate, but share the need for a common bit of functionality: the ability to perform operations on simple data types as well as more complex relational data. In particular, the construction phase requires the output of a distillation as its input, which means we would like the two languages to share a common type system.

We have chosen to implement this with a common sublanguage, which we call the *expression language*. This language is a simple, functional language for data manipulation, which supports our basic data types as well as table data types, as defined in our source schema language.

This language is (at least in its current version) very basic. In particular, we do not support mutable data, user-defined functions, higher-level functions, recursion, and no user-defined types beyond the table types defined in the distillation language.

3.4.1 Primitive types

The basic data types are the column types defined as part of the source schema language (introduced in section 3.1.2): **text**, **integer**, **boolean**, and so forth. We will refer to these as the *primitive types*.

Literals

```
"foo"                       // Text literal
42                          // Integer literal
42.0                        // Decimal literal
true, false                 // Boolean literals
```

Basic operators

```
"foo" ++ " " ++ "bar"       // Text concatenation
(42 * 2) + 1                // Arithmetic
true and (not false)        // Boolean logic
```

Type conversions

There are a few constructs available for type conversions. In particular, every primitive type may be converted from/to text values using the `parse` and `serialize` operators, respectively.

```
parse "42" :: integer       // Text parsing
serialize 42                // Text representation
```

Scoping

Each expression is evaluated in a particular *scope*. A scope consists of named constants, which may be accessed using the `$"<name>"` syntax. We do not include any kind of mutable state, thus constants cannot be assigned to. We can however create a new scope for subexpressions using the `where` syntax.

```
// Access constants from parent scope
$"first name" ++ " " ++ $"last name"

// Define a new local scope
$"first name" ++ " " ++ $"last name" where {
    "first name" = "John"
    "last name" = "Doe"
}
```

Context

A special scope constant called the *context* is usually (although not necessarily) available. This constant is expressed using the dot syntax (`.`).

```
// Append the text literal "foo" to the current context
// Note: only valid if the type checker confirms that . is of type text
. ++ "foo"
```

3.4.2 Row sets

Because we are dealing with relational data, we need to include tabular data types into our type system. We introduce a single concept, the **row set** type, to represent this information. Note that although we use the term “set”, these are not technically *sets* in the mathematical sense, because we allow duplicates (i.e. they are multisets, or bags).

Table expressions

We can access the contents of a table simply by referring to `table <name>`, which returns the table body as a row set.

```
table "foo"                                // Table contents
(table "foo") union (table "bar")          // Merge two (compatible) tables
(table "foo") filter (.["type"] == "x")    // Filter a subset of rows
```

Row expressions

Individual rows are typically passed down to subexpressions through the context (.) mechanism. For example, when we `map` over a table in a construction definition, the current row is used as context value. Given a row, we can access the column values, as well as traverse links.

```
.["desc"]          // Access the column "desc"
. -> "products"    // Follow the link named "products"
```

Note that, although conceptually we are manipulating single rows, there is no “row” type in our type system. Instead, the operators `[]` and `->` work on row sets. Ensuring that an expression like `.["desc"]` resolves to a single text value is done by the type checker, through the *link chain* mechanism, which we describe in chapter 4.

3.4.3 Type constructors (unsafe + optional)

Some expressions may result in an error. For example, the following expression fails to return an integer:

```
parse "foo" :: integer // Fails
```

We say that the `parse` operator is *unsafe*. This fact is represented in its return type: `unsafe<T>` (where `T` = `integer` in this case). We take inspiration from functional languages (e.g. `Maybe` in Haskell), and represent errors using a type constructor that "wraps around" the type in question.

Unsafe expressions

We provide two mechanisms to deal with unsafe-typed values. The first is to explicitly handle both cases using the `try ... catch` operator.

```
try (parse "foo" :: integer) catch 0 // Evaluates to 0
```

There is a second mechanism, called an *assumption marker* (`<!>`), which forcefully "unwraps" the result of an unsafe operator, without regard for whether it succeeds. If this does not succeed, a runtime failure is introduced and handled specially by the runtime engine. The motivation for this construct is given in the next section (3.5).

```
(parse "42" :: integer) <!>   // Evaluates to 42
(parse "foo" :: integer) <!>  // Causes a runtime failure
```

Optional expressions

The type constructor `optional<T>` is similar to `unsafe<T>`, in that it encapsulates either a successful value, or a fallback. The major difference is that we use optional values to represent values where the fallback is not an error, but rather an explicitly defined "default" (called the *none* case).

It is used, among other things, to represent nullable columns (as found in SQL). Expressions that want to manipulate an optional value *must* handle both cases using the `settle` operator.

```
// Optional value constructor (is "none" if boolean expressions fails)
optional .["foo"] on (. != "")

settle (.["optional value"]) otherwise 0 // Handle both cases
```

3.5 Methodology

In this section we present the methodology that eventually led to the creation of Square. These are a set of general concepts and guidelines that were developed at M-industries through practical experience in systems integration. We believe this methodology makes for a good workflow to write importers in a way that is natural and practical, and which converges to a robust importer through successive iterations.

3.5.1 Process

Data importers have to deal with varying data over time. Thus, we can view writing an import definition as a matter of making hypotheses about future inputs. For example, at some point during the transformation, we might require a particular value to be an integer, or the result of a join operation to contain exactly one row, or for a tree encoding to contain no cycles. Even if the source schema does not guarantee such a property, we might still have good reason to *suppose* that it will always hold and perform our transformation rule regardless.

If such a hypothesis were to ever break (through a counterexample), the import would fail, because the target schema – and thus any consuming application – was not designed to handle that particular use case. Moreover, we *want* the import to fail in this case, because it means somewhere we must have made a design flaw, which makes manual intervention necessary.

Notice that this is functionally equivalent to the more conventional approach of applying a set of validation rules prior to performing the transformation itself. However, in this new perspective the validation rules become an integrated part of the tranformation itself, instead of a separate phase upfront.

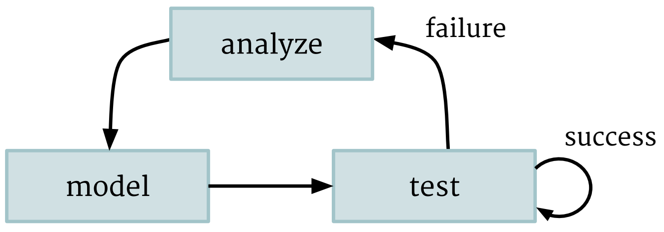

Figure 3.2: Feedback loop.

We apply this process iteratively in a kind of trial and error approach. The motivation for such an approach is the following observation: business operations are often complex systems for which there is no perfect formal description. Using this type of iterative process allows us to work towards a correct import definition purely based on the output data coming from this complex system.

3.5.2 Certainty levels

When making a hypothesis, we distinguish between three different cases based on how certain we are that the hypothesis will hold. We denote these three *certainty levels* as follows:

- Know
- Think
- Hope

Consider a particular hypothesis we want to make (e.g. a value is integer). We may be able to statically prove that the hypothesis is true. For instance, it may be guaranteed by the source schema. We say that we **know** that the hypothesis is true. In this case we are certain we can safely apply the operation without risk of failures.

Alternatively, we may not be able to prove it in an automated fashion, but we might have good reason to believe it nonetheless. For example, the client may have informed us that the property in question is checked through alternative means, such as an internal software system, or company policy. In this case, we **think** the hypothesis holds (because we have some rationale), but we cannot prove it automatically.

In the last scenario, we have (at best) circumstantial evidence. For example, we may have studied data samples and observed that it *looks* as though a particular field is always integer. We would like the hypothesis to hold, but because we have no proof that it does, we feel compelled to handle both cases. We say that we **hope** that the hypothesis is correct, but because there is a risk we explicitly handle the fallback case as well.

3.5.3 Assumptions

The three certainty levels are formalized in the Square programming model. Each scenario corresponds to a particular pattern in the language.

An operation in Square is inherently linked to a language construct called an *assumption*, which represents a hypothesis on the behalf of the programmer. For each assumption in an import definition, the compiler will attempt to check whether the assumption is true. For example, a simple field access operation cannot fail. The compiler will thus mark this operation as *safe*. This corresponds to the “know” case.

```
// Here, .["quantity"] is a field of type 'integer'
"quantity" :: integer <= .["quantity"]
```

On the other hand, if the compiler cannot verify an assumption, it will be marked as *unsafe*. This corresponds to the “think” scenario. The compiler forces the programmer to mark an unsafe assumption in the syntax itself using an *assumption marker* (<!>). For example, to parse a text value to an integer:

```
// Here, .["quantity"] is a field of type 'text'
"quantity" :: integer <= parse .["quantity"] :: integer <!>
```

The assumption marker serves to make it explicit that the operation is a potential failure. A reader of the code should consider it a red flag. A string called the *rationale* may be added to the marker to motivate the presence of the unsafe assumption:

```
"quantity" :: integer <= parse .["quantity"] :: integer
    <!"Validated at user input, cf. e-mail correspondence #456">
```

By making this part of the syntax, we can provide tool support. For example, a reporting tool can parse an import definition and produce an overview of all "risks" in the transformation, which may be shown to the client for verification. Or, when we encounter an import failure at runtime, the import system can use these markers to provide better diagnostics.

In the final "hope" scenario, we want to explicitly support the possibility of a failure. The importer, being a non-interactive background process, cannot deal with the error itself (it cannot inform the end user). Instead, we require each consuming application to add explicit error handling logic (e.g. it might ask the end user to "fix" the offending data).

The presence of such an error handling mechanism must be modeled in the target schema. In Alan terms, this means we need an `option` property with a fallback state.

```
"quantity status" :: option (
    "valid" :: (
        "quantity" :: integer
    )
    "error" :: ()
)
```

In the construction language, we can map this using a `try` operation.

```
"quantity status" :: option <=
    try "quantity" = parse .["quantity"] :: integer
        => "valid" :: (
            "quantity" :: integer <= $"quantity"
        )
    catch
        => "error" :: ()
```

A `try` operation takes a number of named expressions, where assumption markers are suppressed. If the expressions succeed, the result is made available within the child scope (accessed through `$"quantity"`). This idea is similar to the concept of a try/catch block in many programming languages.

Chapter 4

Formal Semantics

In the previous chapter, we introduced Square using informal descriptions and code examples. Here, we dive a little deeper into the semantics of Square using a more standard, formal notation. We focus in particular on the static semantics of the language, i.e. the type system and other semantic constraints imposed at compile time.

Throughout the chapter, we will use basic terminology and notation from type theory. We assume that the reader is at least somewhat familiar with: typed variants of the lambda calculus, typing rules (and their application in type checking), and denotational semantics (as covered for example in [42]).

4.1 Type System

As discussed in the previous chapter, the distillation and construction languages share a generic expression language, with a common type system, allowing outputs of the former to be passed as inputs to the latter. We will refer to this expression language as $\mathcal{E}$. A complete syntax for the language is given in appendix B.3. $\mathcal{E}$ is a statically typed, purely functional language with strict semantics. The language has been purposefully kept fairly simple, lacking user-defined functions, custom data types, and recursion.

Recall from chapter 3 that we want to be able to guarantee that an import definition produces no runtime errors *except* where explicitly annotated (using assumption markers). For this reason, we make heavy use of static type checking to make compile-time guarantees about the output.

The basic syntax of types in $\mathcal{E}$ (excluding row types, which we cover in the next section) is:

$$\begin{gathered}\mathrm{PrimitiveType} ::= \mathbf{text} \mid \mathbf{integer} \mid \mathbf{boolean} \mid ... \\ \mathrm{Type} ::= \mathrm{PrimitiveType} \mid \mathbf{optional}\langle\mathrm{Type}\rangle \mid \mathbf{unsafe}\langle\mathrm{Type}\rangle\end{gathered}$$

Here, $\mathrm{PrimitiveType}$ captures the primitive data types. A Type is defined as either a primitive type, or one of the type constructors $\mathbf{optional}$ or $\mathbf{unsafe}$. Note that a *type constructor* is a type which takes other types as arguments (i.e. a type-level function).

Per convention, we will denote concrete types in bold font, and type variables using Greek lowercase letters (τ, σ, ...).

4.1.1 Typing rules

$\mathcal{E}$ is parameterized by an alphabet Σ, and a set Ω containing typed constants $c : \tau$. There are constants defined for each primitive type (corresponding to the literals available for that type):

- $t : \textbf{text} \in \Omega$ for $t \in \Sigma^* = \{\, \texttt{""}, \texttt{"a"}, \texttt{"ab"}, \ldots \}$
- $n : \textbf{integer} \in \Omega$ for $n \in \mathbb{Z} = \{\, 0, +1, -1, +2, -2, \ldots \}$
- $b : \textbf{boolean} \in \Omega$ for $b \in \mathbb{B} = \{\, \texttt{true}, \texttt{false} \}$
- etc.

To determine whether an expression in $\mathcal{E}$ is well-typed, we need a set of *typing rules*. Typing rules are logical inference rules that tell us when a particular expression may be judged to be of a particular type. Each expression is given under a *typing context* Γ, which contains the types of free variables in the expression. A complete overview of all typing rules is provided in appendix C.

For constants, we introduce a trivial typing rule that says that constants of type τ may be judged to be of type τ under any context Γ.

$$\frac{c : \tau \in \Omega}{\Gamma \vdash c : \tau}$$

$\mathcal{E}$ does not allow "undefined" values, nor partial functions. In terms of type theory, we say that there is no *bottom value* $\bot$, and all types are thus *unlifted*. We do not need $\bot$ to express nontermination either, because an $\mathcal{E}$ expression always terminates (the language does not allow recursive definitions, and built-in operators are designed to terminate). This limits the expressiveness of the language, but at the benefit of stronger compile-time guarantees, and improved ability to reason about programs.

In place of a special value, expressions that want to indicate the "lack of" of a value need to use a different mechanism. The **optional** type constructor may be used for values that may or may not exist (similar to the `Maybe` type in Haskell, or `option` in ML). Manipulating such a value must be done using the `settle` operator, which forces the programmer to handle both cases.

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma,\, \textbf{.} : \tau \vdash e_2 : \textbf{boolean}}{\Gamma \vdash (\texttt{optional}\ e_1\ \texttt{on}\ e_2) : \textbf{optional}\langle\tau\rangle} \qquad \frac{\Gamma \vdash e_1 : \textbf{optional}\langle\tau\rangle \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\texttt{settle}\ e_1\ \texttt{otherwise}\ e_2) : \tau}$$

(Note that in the above, the syntax $\Gamma,\, x : \tau$ indicates a typing context, where the free variable "x" is assigned the type τ. In this case, we set the type of the context variable "**.**" because we pass this variable along to the subexpression e_2.)

Errors are handled using a similar construct; operators that may fail need to use the type constructor **unsafe**. Accessing its value may be done using a `try` operation. Values of type $\textbf{unsafe}\langle\tau\rangle$ are returned either by specific operators ("unsafe operators"), or directly constructed through the `error` $::\tau$ syntax.[1]

$$\frac{}{\Gamma \vdash (\texttt{error} :: \tau) : \textbf{unsafe}\langle\tau\rangle} \qquad \frac{\Gamma \vdash e_1 : \textbf{unsafe}\langle\tau\rangle \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\texttt{try}\ e_1\ \texttt{catch}\ e_2) : \tau}$$

Alternatively, we may simply grab the value directly, by applying the assumption marker `<!>`. This will either return the value or raise a runtime error. In fact, this is the *only* way a runtime error may be raised in $\mathcal{E}$.

$$\frac{e : \textbf{unsafe}\langle\tau\rangle}{\Gamma \vdash (e\ \texttt{<!>}) : \tau}$$

The language provides a number of built-in operators for primitive types, including basic arithmetic, text manipulation, boolean logic, etc. A few of these operators are listed with their typing rules below (refer to appendix C for the complete list).

$$\frac{\Gamma \vdash e_1, e_2 : \textbf{integer}}{\Gamma \vdash (e_1\ \texttt{+}\ e_2) : \textbf{integer}} \qquad \frac{\Gamma \vdash e_1, e_2 : \textbf{text}}{\Gamma \vdash (e_1\ \texttt{++}\ e_2) : \textbf{text}}$$

$$\frac{\Gamma \vdash e_1, e_2 : \textbf{boolean}}{\Gamma \vdash (e_1\ \texttt{and}\ e_2) : \textbf{boolean}} \qquad \frac{\Gamma \vdash e_1 : \textbf{boolean} \quad \Gamma \vdash e_2, e_3 : \tau}{\Gamma \vdash (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) : \tau}$$

$\mathcal{E}$ is strongly typed. There are no implicit type conversions, instead all conversions must be done through the type conversion mechanisms provided in the language. Of note, we allow any of the primitive types to be parsed from, or serialized as, **text** values.

$$\frac{\Gamma \vdash e : \textbf{text} \quad \tau \text{ is primitive}}{\Gamma \vdash (\texttt{parse}\ e :: \tau) : \textbf{unsafe}\langle\tau\rangle} \qquad \frac{\Gamma \vdash e : \tau \quad \tau \text{ is primitive}}{\Gamma \vdash (\texttt{serialize}\ e) : \textbf{text}}$$

Recall that the typing context Γ contains types of free variables in the current expression. In our expression language, these free variables correspond to scope constants. Such a scope constant x may be accessed through the `$x` syntax. We distinguish a special constant for the current *context*,[2] denoted with a dot (`.`).

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash \texttt{\$}x : \tau} \quad \text{or} \quad \frac{\texttt{.} : \tau \in \Gamma}{\Gamma \vdash \texttt{.} : \tau}$$

[1] The manual declaration of the type of `error` (using `::`) is necessary because we lack a polymorphic type system (i.e. we cannot declare `error` to be of a variable type).

[2] Be careful not to confuse the term "context", which is a special constant (`.`) in our language, with the "typing context" Γ, which is a term we appropriate from type theory.

Constants may be *bound* using a `where` expression.

$$\frac{\Gamma \vdash e_1 : \tau_1,\ \ldots,\ e_n : \tau_n \quad \Gamma,\ x_1 : \tau_1,\ \ldots,\ x_n : \tau_n \vdash e_0 : \tau_0}{\Gamma \vdash (e_0\ \texttt{where}\ \{\ x_1 = e_1,\ \ldots,\ x_n = e_n\ \}) : \tau_0}$$

4.1.2 Type checking

As we will see in the following sections, $\mathcal{E}$ expressions are always embedded in larger structures. The structure that embeds the expression determines its top-level scope, and thus its typing context Γ. This typing context will usually – although not necessarily – include a type τ_c for the context variable: $\Gamma = \{\ \textbf{.} : \tau_c\ \}$). Type checking an expression is done by recursively applying the typing rules defined above.

4.2 Source Schema

In the distillation language, we define a number of table schemas, which together form our *source schema*. We want to introduce new types for each of these table schemas, so that we can assign types to tabular data. Going beyond just rows and columns, we also want to be able to reason statically about integrity constraints, such as keys and foreign keys. Thus, we will formalize all of the information defined in the source schema directly in the type system, so that we can utilize this information in our typing rules.

4.2.1 Structure

A database schema is a fairly complex structure. Instead of basing ourselves on an elementary type theory (e.g. typed lambda theory), we find it easier to give a set-theoretical definition, and then introduce new types based on these mathematical objects. In other words, our goal here is to capture the data structure from section 3.2, but using set theory instead of a grammar.

First, a few basic definitions. Let Σ_t, Σ_c, Σ_l, Σ_k denote infinite sets of table names, column names, link names, and key names, respectively. In addition, let T denote the set of all possible data types, $LM = \{\,\text{single},\ \text{many}\,\}$ the set of the possible link multiplicities, and $LC = \{\,\text{none},\ \textbf{unsafe}\langle\rangle,\ \textbf{optional}\langle\rangle,\ \textbf{optional}\langle\textbf{unsafe}\langle\rangle\rangle\,\}$ the set of type constructors we can apply to link definitions.

We define a few domains to capture the structure of elementary table schema elements. Each member of the following sets is a single column, link, or key, respectively.

$$D_{\text{cols}} = \{\ \textit{datatype}\ \mid\ \textit{datatype} \in T\ \}$$
$$D_{\text{links}} = \{\ (\ \textit{target},\ \textit{mult},\ \textit{constr}\ )\ \mid\ \textit{target} \in \Sigma_t,\ \textit{mult} \in LM,\ \textit{constr} \in LC\ \}$$
$$D_{\text{keys}} = \{\ \textit{columnset}\ \mid\ \textit{columnset} \in \mathcal{P}(\Sigma_c)\ \}$$

The following examples illustrate the correspondence between the syntax we defined in the previous chapter, and the above set encoding:

`column "name" :: text`	$\textbf{text} \in D_{cols}$
`column "name" :: text !`	$\textbf{unsafe}\langle\textbf{text}\rangle \in D_{cols}$
`column "name" :: text ?`	$\textbf{optional}\langle\textbf{text}\rangle \in D_{cols}$
`column "name" :: text ! ?`	$\textbf{optional}\langle\textbf{unsafe}\langle\textbf{text}\rangle\rangle \in D_{cols}$
`link "machine": single "machines"`	$(\texttt{"machines"},\ \text{single},\ \text{none}) \in D_{links}$
`link "machines": many "machines"`	$(\texttt{"machines"},\ \text{many},\ \text{none}) \in D_{links}$
`link "machine": single "machines" !`	$(\texttt{"machines"},\ \text{single},\ \textbf{unsafe}\langle\rangle) \in D_{links}$
`link "machine": single "machines" ?`	$(\texttt{"machines"},\ \text{single},\ \textbf{optional}\langle\rangle) \in D_{links}$
`link "machine": single "machines" ! ?`	$(\texttt{"machines"},\ \text{single},\ \textbf{optional}\langle\textbf{unsafe}\langle\rangle\rangle) \in D_{links}$
`key "email": { "email address" }`	$\{\texttt{"email address"}\} \in D_{keys}$

Note that the grammar is slightly more restrictive than the set encoding. For example, the grammar does not allow a column of type $\textbf{unsafe}\langle\textbf{optional}\langle\textbf{text}\rangle\rangle$, despite this being a valid type. The reason for this discrepancy is that not all combinations of types correspond to a logical use case, and thus we expressively disallow them in the syntax. The difference is irrelevant for our typing rules.

Based on the above domains for columns, links, and keys, we can now define how we create mappings from names to values. Each mapping is a set of names, and a function that maps each of these names to an element of the corresponding domain.

$$
\begin{aligned}
M_{\text{cols}} &= \{\, (\, c,\ dom_c\,) \mid c \in \mathcal{P}(\Sigma_c),\ dom_c : c \to D_{\text{cols}} \,\} \\
M_{\text{links}} &= \{\, (\, l,\ dom_l\,) \mid l \in \mathcal{P}(\Sigma_l),\ dom_l : l \to D_{\text{links}} \,\} \\
M_{\text{keys}} &= \{\, (\, k,\ dom_k\,) \mid k \in \mathcal{P}(\{pk\} \cup \Sigma_k),\ dom_k : k \to D_{\text{keys}} \,\}
\end{aligned}
$$

(Note that we have added a special symbol $pk \notin \Sigma_k$ to the set of key names, which distinguishes the primary key, if present.)

A single table schema definition consists of mappings for columns, links, and keys. We thus define the domain of table schemas as follows:

$$
\begin{aligned}
D_{\text{tables}} = M_{\text{cols}} \times M_{\text{links}} \times M_{\text{keys}} \\
\textit{s.t. all key columns in } M_{\text{keys}} \textit{ match a column in } M_{\text{cols}}
\end{aligned}
$$

Finally, we can define the structure of an entire database schema. A database schema is a mapping from table names to table schemas. In other words, a database schema DB is an element of the following domain:

$$
\begin{aligned}
M_{\text{tables}} = \{\, (\, t,\ dom_t\,) \mid t \in \mathcal{P}(\Sigma_t),\ dom_t : t \to D_{\text{tables}} \,\} \\
\textit{s.t. all link targets match a valid table name } t
\end{aligned}
$$

For notational convenience, we introduce the following functions to access the individual components of any table schema $t \in D_{\text{tables}}$:

$$\begin{aligned}
\text{columns}(t) &= \{\, (\, \mathit{name},\ \mathit{domain}\,) \mid \mathit{name} \in c,\ \mathit{domain} = \mathit{dom}_c(c)\,\} \\
\text{links}(t) &= \{\, (\, \mathit{name},\ \mathit{domain}\,) \mid \mathit{name} \in l,\ \mathit{domain} = \mathit{dom}_l(l)\,\} \\
\text{keys}(t) &= \{\, (\, \mathit{name},\ \mathit{domain}\,) \mid \mathit{name} \in k,\ \mathit{domain} = \mathit{dom}_k(k)\,\} \\
&\text{for } \ t = (\,(\,c,\ \mathit{dom}_c\,),\ (\,l,\ \mathit{dom}_l\,),\ (\,k,\ \mathit{dom}_k\,)\,) \in D_{\text{tables}}
\end{aligned}$$

4.2.2 Row set types

Now that we have a structure for database schemas, we can introduce the corresponding types into our type system. The data types we introduce are *sets of rows*[3] that conform to a particular table schema. Each row in a row set should instantiate the table schema's columns and links. Additionally, all key constraints must hold for the set as a whole.

A naive approach would be to introduce a type $\textbf{rowset}[\,t\,]$ for each table t in the source schema. This, however, is not granular enough for our purposes. We want to be able to reason about the *origin* of rows as well. This because the chain of operators that led to a particular row can determine whether that row is (for example) a valid reference or not. We want to check these kind of properties at compile time.

For this reason, we introduce the concept of a *link chain*. An element in a link chain is a link (of type either single or many), where each link in the chain is applied in succession. We use the following short-hand syntax for a link element in a chain:

$$\begin{aligned}
\rightarrow t &\iff (\,t,\ \text{single},\ \text{none}\,) \in D_{\text{links}} \\
\twoheadrightarrow t &\iff (\,t,\ \text{many},\ \text{none}\,) \in D_{\text{links}}
\end{aligned}$$

In other words, $\rightarrow t$ and $\twoheadrightarrow t$ denote links with target t, of multiplicity single and many respectively, without any type constructors applied. A link chain, denoted $\vec{p}$, is then a sequence of such links. Some examples are given below.

$$\begin{aligned}
\vec{p} &= [\,\twoheadrightarrow t\,] \\
\vec{q} &= [\,\twoheadrightarrow t, \rightarrow u\,] \\
\vec{r} &= [\,\twoheadrightarrow t, \rightarrow u, \twoheadrightarrow v\,] \\
\vec{s} &= [\,\vec{r}, \twoheadrightarrow w\,] = [\,\twoheadrightarrow t, \rightarrow u, \twoheadrightarrow v, \rightarrow w\,]
\end{aligned}$$

By convention, the first element of the link chain always gives the base table. Thus, the result of the chain $\vec{p}$ is the entire table t. $\vec{q}$ produces a single row of the table u, accessed by following a link from some row in t. The chain $\vec{r}$ expresses one additional step, where a many link from a row in u is followed to the table v, resulting in 0 or more rows in table v. Lastly, the final link chain $\vec{s}$ builds upon an existing chain $\vec{r}$ and appends one more link element $\rightarrow w$.

For a given link chain $\vec{p} = [\,\vec{q}, \twoheadrightarrow t\,]$ (or $\vec{p} = [\,\vec{q}, \rightarrow t\,]$), we introduce a new type in our type system: $\textbf{rowset}[\,\vec{p}\,]$. An instance of this type is a set of rows conforming to the target of the final chain element t.

[3] Technically, we do allow duplicates in some cases, thus making these objects *multisets* (or *bags*) rather than sets. We stick to the term "row set" because it is a more familiar term.

4.2.3 Row set operators

Given a database schema $S \in M_{\text{tables}}$, we define a number of operators that work on $\mathbf{rowset}$ types. The simplest operator fetches the contents of an entire table.

$$\frac{(t, dom_t) \in S}{\Gamma \vdash \texttt{table}\ t : \mathbf{rowset}[\twoheadrightarrow t\,]}$$

We can take the union of two row sets, if their column sets are equal. The result of a union conforms to a new table schema u, where we can no longer guarantee the key constraints of the inputs (because t and s may conflict). We can, however, keep the links that are shared by both row sets.

$$\frac{\Gamma \vdash e_1 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad \Gamma \vdash e_2 : \mathbf{rowset}[\,\vec{q}, \twoheadrightarrow s\,] \quad \text{columns}(t) = \text{columns}(s)}{\Gamma \vdash e_1\ \texttt{union}\ e_2 : \mathbf{rowset}[\twoheadrightarrow u\,]}$$

$$\text{where } u = (\,\text{columns}(t),\ \text{links}(t) \cap \text{links}(s),\ \varnothing\,) \in D_{\text{tables}}$$

A filter takes a row set, and a boolean predicate, and returns a subset of the row set. The fact that the result is a subset is represented through the addition of a link in the chain back to the same table.

$$\frac{\Gamma \vdash e_1 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad \Gamma,\ \textbf{.} : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \vdash e_2 : \mathbf{boolean}}{\Gamma \vdash e_1\ \texttt{filter}\ e_2 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t, \twoheadrightarrow t\,]}$$

The operators above work on row sets of multiplicity many (i.e. $\mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,]$). If, on the other hand, we have a row set containing a single row ($\mathbf{rowset}[\,\vec{p}, \rightarrow t\,]$), we can perform row-specific operators.

The first operator we look at is the column indexing operator. The resulting type is exactly the type of the column as defined in the schema.

$$\frac{\Gamma \vdash e : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \quad (\,c,\ \tau\,) \in \text{columns}(t)}{\Gamma \vdash e\texttt{[}c\texttt{]} : \tau}$$

Similarly, links may be followed through the link traversal operator. This operator comes in different flavors, for each of the combinations of single/many and the different type constructors specified in the link definition:

$$\frac{\Gamma \vdash e : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \quad (\,l,\ (\,s,\ \text{single},\ \tau\,)\,) \in \text{links}(t)}{\Gamma \vdash e\ \texttt{->}\ l : \tau\langle\mathbf{rowset}[\,\vec{p}, \rightarrow t, \rightarrow s\,]\rangle}$$

$$\frac{\Gamma \vdash e : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \quad (\,l,\ (\,s,\ \text{many},\ \tau\,)\,) \in \text{links}(t)}{\Gamma \vdash e\ \texttt{->}\ l : \tau\langle\mathbf{rowset}[\,\vec{p}, \rightarrow t, \twoheadrightarrow s\,]\rangle}$$

(Note that the use of τ in the above typing rule represents one of the different possible type constructors, like $\mathbf{optional}\langle\rangle$, or possibly no constructor at all (none).)

The final operator we mention here is the inverse of the above link traversal. Given a row produced through a link chain, we can access its predecessor in the chain through a *parent link step* (`^`).

$$\frac{\Gamma \vdash e : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,]}{\Gamma \vdash e\ \texttt{^} : \mathbf{rowset}[\,\vec{p}\,]}$$

If we view the link chain as a stack, what this operator does is basically a "pop" operator, that returns us to the previous row set result. The typing rule ensures that there must be at least one item (a single link), or it will not type check.

4.2.4 Aggregation operators

A common use case is to take a set of rows, and flatten them to a single value. There are several aggregation operators available, for numerical (`min`, `max`, `avg`, `sum`) and text types (`concat`). For example, to take the minimum value over a row set, we can use the `min` operator, which has the following typing rule:

$$\frac{\Gamma \vdash e_0 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad \Gamma,\ \mathbf{.} : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \vdash e_1 : \mathbf{integer}}{\Gamma \vdash (\texttt{min}\ e_1\ \texttt{over}\ e_0) : \mathbf{unsafe}\langle\mathbf{integer}\rangle}$$

The expression e_0 gives the base row set to be mapped over. e_1 takes some row in this set as its context and converts it to an integer value. The operator returns the minimum value if successful. Note that the operator is *unsafe*, because the input may be empty. For some operators there is a logical "trivial" default value, but for others (like min/max/avg), we force the programmer to handle this case themselves.

Another aggregation operator we introduce is the `shared` operator:

$$\frac{\Gamma \vdash e_0 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad \Gamma,\ \mathbf{.} : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \vdash e_1 : \tau}{\Gamma \vdash (\texttt{shared}\ e_1\ \texttt{over}\ e_0) : \mathbf{unsafe}\langle\tau\rangle}$$

This operator checks that all of the given values are equal. If so, it returns that value, otherwise it will cause a failure. The motivation for this operator is due to a common pattern we observe in working with relational data, specifically in systems that lack "nested" tables. In these systems, the result of a join operation results in the same value being replicated across multiple rows. To "undo" such an operation during our distillation process, we want to be able to express the property that a particular set of rows *share* some value.

As with the previous sections, the complete list of typing rules are given in appendix C.

4.3 Distillation

Recall from chapter 3 that a *distillation* is a transformation that takes as input a set of raw tables, and produces an instance of a source schema as a result. We can now define this more formally. We are given a *raw schema* $R \in M_{\text{tables}}$, and a *source schema* $S \in M_{\text{tables}}$. The raw schema is obtained from some external database schema definition, and the source schema is defined by the database structure in the distillation program.

Def. Given $R, S \in M_{\text{tables}}$, a **distillation** is a transformation that takes an instance of R and produces a valid instance of S.

Note that – unlike typical transformations (e.g. SQL queries or views) – the structure of the output is *predetermined*. The output structure is not derived from the transformation rules themselves, rather, we limit the transformation rules such that it must produce a valid instance of the desired schema. This is a subtle but important difference, because it switches our perspective when it comes to defining static constraints on the language.

4.3.1 Table structure

A table schema definition consists of two parts: the table structure itself, and the import expressions that are annotated upon it (<=). The table structure determines its type. There is a simple correspondence between the formal notation from the previous section, and the Square syntax described in chapter 3, as illustrated by the following example:

```
table "products"

    column * "product id" :: integer
    column "order id" :: integer ?
    column "desc" :: text
    column "uniq" :: text


    link "order": single "orders" ?
    link "machines": many "machines"



    key "uniq": { "uniq" }
```

$$
\begin{aligned}
&t \in D_{\text{tables}} \;\; \text{s.t.} \\
&\quad \text{columns}(t) = \{ \\
&\qquad (\,\texttt{"product id"},\, \textbf{integer}\,), \\
&\qquad (\,\texttt{"order id"},\, \textbf{optional}\langle\textbf{integer}\rangle\,), \\
&\qquad (\,\texttt{"desc"},\, \textbf{text}\,), \\
&\qquad (\,\texttt{"uniq"},\, \textbf{text}\,) \\
&\quad \}, \\
&\quad \text{links}(t) = \{ \\
&\qquad (\,\texttt{"order"},\, (\,\texttt{"orders"},\, \text{single},\, \textbf{optional}\langle\rangle\,)\,), \\
&\qquad (\,\texttt{"machines"},\, (\,\texttt{"machines"},\, \text{many},\, \text{none}\,)\,) \\
&\quad \}, \\
&\quad \text{keys}(t) = \{ \\
&\qquad (\,pk,\, \{\,\texttt{"product id"}\,\}\,), \\
&\qquad (\,\texttt{"uniq"},\, \{\,\texttt{"uniq"}\,\}\,) \\
&\quad \}
\end{aligned}
$$

4.3.2 Table import expressions

The import for a table is given by an import expression e, prefixed by an operation type op:

```
table "products" <= <op> <e>
```

The expression e is constrained to be of type $\textbf{rowset}[\vec{p}, \twoheadrightarrow s]$. In other words, it should be a table, or the result of a table operator (e.g. `filter` or `union`). The expression may refer to table schemas in R, as well as sibling table schemas in the same source schema S. For namespacing purposes, table schemas in R are referred to using the syntax `external "<table name>"`.

The operation type op determines how to map the input rows to the output. There are two different operations we can choose from:

- `map` (or `map on "<key name>"`)
- `aggregate`

Map

The `map` operation performs a one-to-one mapping from each row in the input to a row in the output. To guarantee that we can map each input row to a *unique* output row (i.e. unique primary key value), the compiler forces the programmer to maintain the same primary key. In other words, if the primary key of the input **rowset** is { `"pk 1"`, `"pk 2"` }, then the table schema must contain precisely the following key columns:

```
column * "renamed pk 1" :: text <= .["pk 1"]
column * "renamed pk 2" :: text <= .["pk 2"]
```

> **Constraint:** Given: a table import expression of type $\textbf{rowset}[\vec{p}, \twoheadrightarrow s]$, on an output table of type t. A `map` operation is only valid **iff** s has a primary key $(k_1, ..., k_n)$, and the primary key of t is $(k'_1, ..., k'_n)$, where each column k'_i imports the column k_i.

Implementing this check requires us to look at the AST of each primary key column expression. In our current implementation, we simply check that each primary key column k of the input matches exactly one primary key column of the output, i.e. its expression is precisely `.["<k>"]`. There are more sophisticated ways of performing such an equivalence check, but for simplicity we have settled for this solution.

A different key of the table s may be selected using `map on "<key>"`. The same rules apply, but we use the specified key of the input rather than the primary key.

Aggregate

If the programmer wants to specify a *new* primary key, then they can use the `aggregate` operation. With this operation, rows are automatically grouped based on the new primary key. Thus, in general, n input rows may map to one output row. All non-primary key columns must deal with this by flattening the row set to a single value, using an aggregate operator.

```
table "employees" <= map external "EMPLOYEES"
    column * "employee id" :: text <= .["EID"]
    column "employee name" :: text <= .["NAME"]
    column "department" :: text <= .["DEP"]
    column "department manager" :: text <= .["DEPMANAGER"]

// Derive a new table of unique departments, based on the employees table
table "departments" <= aggregate table "employees"
    // Context (.) is of type rowset (single)
    column * "department id" :: text <= .["department"]
    // Context (.) is of type rowset (many), and must thus be flattened
    column "manager" :: text ! <= shared .["department manager"] over .
```

An example of an `aggregate` operation is shown above. Note that primary key columns are given a single row context as usual, because we use these columns as the basis for grouping rows. However, for non-primary key columns, **.** is of type **rowset**$[\vec{p}, \twoheadrightarrow s]$.

4.3.3 Column import expressions

For columns, the column type determines the expected type of the import expression. The compiler checks that the types match exactly.

```
// Type of <e> must be text
column "name" :: text <= <e>
// Type of <e> must be optional<date>
column "start date" :: date ? <= <e>
// Type of <e> must be optional<unsafe<integer>>
column "quantity" :: integer ! ? <= <e>
```

4.3.4 Key definitions

Non-primary keys provide alternate ways to index a table. To guarantee that a key does indeed form a unique index, the compiler must verify that an equivalent key is present on the input. Alternatively, we allow the programmer to specify "unsafe" keys (annotated with `!`). We can map over such a key, but because we may end up with multiple conflicting rows, any row accessed through an unsafe key is returned as an **unsafe**-typed value.

```
table "users" <= map external "USERS"
    column * "user id" :: text <= .["UID"]
    column "email address" :: text <= .["EMAIL"]
    column "alias" :: text <= .["ALIAS"]

    // Only valid if the input table has a key { "EMAIL" }
    key "by email address": { "email adress" }
    // Always possible, but accessing a row through this key is unsafe
    key "by alias": ! { "alias" }
```

Constraint: Given: a table import expression of type **rowset**$[\vec{p}, \twoheadrightarrow s]$, on an output table t. If t has a (safe) key $(k'_1, ..., k'_n)$, where each k'_i is imported from some column $k_i \in \text{columns}(s)$, then s **must** have a corresponding key $(k_1, ..., k_n)$.

4.3.5 Forward link definitions

A "forward link" is a link where the foreign key definition is on the table itself. There are several constraints we place on forward link definitions. The first is that all foreign key definitions must be done on the primary key of the link target table.[4]

Constraint: Given a table t, and a link $l = (u, \text{single}, \tau) \in \text{links}(t)$. A forward link definition for l is only valid **iff** the link target u has a primary key $(k_1, ..., k_n)$, and the foreign key mentions each primary key column k_i exactly once.

This constraint ensures that any forward link matches at most one unique row in the link target. A consequence of this is that forward many links are not possible, and the compiler will raise an error if you try to define one as such.

```
table "products" <= map external "PRODUCTS"
    column * "product id" :: text <= .["PID"]
    column "description" :: text <= .["DESC"]

table "orders" <= map external "ORDERS"
    column * "order id" :: text <= .["OID"]
    column "product id" :: text <= .["PID"]
    column "description" :: text <= .["DESC"]

    // Invalid (foreign key does not match the primary key of "products")
    link "desc": single "products" <= { "description" <= .["description"] }

    // Invalid (forward links can only be single)
    link "products": many "products" <= { "product id" <= .["product id"] }

    // Only valid if there is a corresponding link on the input table
    link "product": single "products" <= { "product id" <= .["product id"] }
```

As with keys, the presence of a link on a table represents a certain integrity constraint. Namely, that all rows refer to a valid row in the link's target table. Defining a link is only possible if we can statically guarantee that the link resolves for all rows, i.e. there must be an equivalent link on the input table.

[4] The reason for this constraint is that it matches the behavior of foreign keys in most relational database engines. Most of these systems also support foreign keys on alternate keys, a feature which we currently do not support.

Constraint: Given: a table import expression of type $\mathbf{rowset}[\vec{p}, \twoheadrightarrow s]$, on an output table of type t, and a link $l = (u, \text{single}, \tau) \in \text{links}(t)$. A forward link definition for l is only valid **iff** s has a link $l' = (u, \text{single}, \tau)$, and l preserves the foreign key definition of l'.

If any of the columns used in the foreign key definition is of type $\mathbf{optional}$, then the entire link must be marked optional as well (`?`). This matches the behavior of foreign key definitions in SQL using one or more nullable columns.

```
table "machines" <= map external "MACHINES"
    column ∗ "machine id" :: text <= .["MID"]

table "materials" <= map external "MATERIALS"
    column ∗ "material id" :: text <= .["MID"]

table "products" <= map external "PRODUCTS"
    column ∗ "product id" :: text <= .["PID"]
    column "machine id" :: text ? <= .["MACHID"]
    column "material id" :: text <= substring .["MATID"] from 0 to 2

    // Optional link (resolves to an optional<row>)
    link "machine": single "machines" ? <= { "machine id" <= .["machine id"] }

    // Unsafe link (resolves to an unsafe<row>)
    link "material": single "materials" ! <= { "material id" <= .["material id"] }
```

Lastly, if we cannot guarantee that a link exists (i.e. we are declaring a *new* link rather than preserving one), then we must annotate the link as unsafe (`!`). A common use case for this is when the external database lacks a certain foreign key constraint. Or, if the value we use in the foreign key is the result of some complex expression (like the substring operation in the example above). Unsafe and optional links may be combined.

4.3.6 Reverse link definitions

A "reverse link" resolves not through a foreign key on the table itself, but rather by referring back to a forward link. For each row, the result of this link is all rows that refer to it through the forward link. In general, there may be several such rows.

```
table "products" <= map external "PRODUCTS"
    column ∗ "product id" :: text <= .["PID"]

    // Safe
    link "orders": many "orders" <= reverse "product"

    // Unsafe
    link "order": single "orders" ! <= reverse "product"

table "orders" <= map external "ORDERS"
```

```
    column * "order id" :: text <= .["OID"]
    column "product id" :: text <= .["PID"]

    link "product": single "products" <= { "product id" <= .["product id"] }
```

If the reverse link is defined to be single, it **must** be marked unsafe (`!`), because the link fails when there is not exactly one row pointing to it. A reverse many on the other hand, is never unsafe nor optional.

4.4 Construction

A *construction* defines how to build an instance of a target schema (an Alan model), given an instance of a source schema as input. For this to be correct, we want to ensure that any target instance that results from a construction is a *valid* instance of the target schema.

In appendix A, we describe exactly what it means for a document (in JSON format) to be a valid instance of an Alan model. There are two main requirements:

1. The document must be **well-formed**, i.e. it must follow the tree structure defined by the model, with suitable values for each property type.
2. The document must pass **referential integrity** checks, i.e. all references must resolve properly.

In this thesis report, we do not give the full operational semantics for the language. Hence, we cannot formally *prove* that the target instance always meets the above requirements. We will, however, describe the function of each language construct, along with a set of static constraints that prevent schema violations – by reasoning about the semantics. Using this approach, we intend to motivate the soundness of the language despite the lack of a formal proof.

4.4.1 Primitive operations

Primitive property types take an expression of a corresponding type. We constrain the import expression to be of the right type. The expression type system was designed such that we can convert the source schema types to the corresponding Alan type without loss of information.

```
(
    "foo" :: text <= <e>       // <e> must be of type text
    "bar" :: integer <= <e>    // <e> must be of type integer
    "baz" :: decimal <= <e>    // <e> must be of type decimal
)
```

4.4.2 Option operations

An import expression on an `option` property is a kind of decision tree, which (eventually) leads to either:

- One of the choices defined for the option in the target schema, or
- A failure (`error<!>`)

```
(
    "foo" :: option <= "x" :: ()  // Unconditional choice selection
    "bar" :: option <= error <!>  // Unconditional failure
)
```

(Note that the `<!>` here is technically not the same as the `<!>` operator we have seen before, although syntactically and functionally they work the same.)

Several branching operations are available, as covered in the previous chapter. The only constraint we place on these operations is that any choice we make matches one of the valid choices defined in the target schema.

4.4.3 Collection operations

We provide one operation to build collections: the `map` operation. A `map` takes an expression of type $\mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,]$ (i.e. a set of rows conforming to the table schema t), and maps this one-to-one to a set of (key, entry) pairs. Each entry is passed the current row in the row set as its context (`.`).

```
(
    "foo" :: collection <= map <e>:
        <key exp> -> (
            // Entry fragment (context: current row)
        )

    // Examples:
    "products" :: collection <= map (table "products"):
        key <!> (.["product id"]) -> ()
    "machines" :: collection <= map (. -> "machines"):
        encode key -> ()
)
```

The collection key used for a particular entry is controlled through the *key expression*, which comes in two variants:

- `key<!>`
- `encode key` (or `encode key "<key name>"`)

The `key<!>` operation takes the current row and passes it to an arbitrary **text** expression. The operation is unsafe, because keys for different entries may conflict. When a conflict is detected, the entire subtree of the target instance at this point is invalidated.

If the **rowset** passed to the map has a primary key, we also allow the `encode key` operation (or any alternative key). This takes the key value of the current row, and encodes it as a text value to create a collection key that is guaranteed to be unique.

> **Constraint:** Given: a `map` operation on a collection with an input of type $\mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,]$. An `encode key` operation using the table key k (primary or otherwise) is valid **iff** $k \in \mathrm{keys}(t)$.

If the selected table key is *unsafe* (cf. section 4.3.4), the row that is passed to the entry as context is also unsafe. Any conflicting rows will cause a failure. To access the row safely, the use of an assumption marker is required. For convenience, we can do this for the entire entry fragment using the `with` syntax.

```
"orders" :: collection <= map table "orders":
    encode key "unsafe key" -> with . <!"Should be unique on 'unsafe key'"> (
        "description" :: text <= .["description"]
    )
```

Note that a key in a table schema may be composed of several key columns, which must be encoded as a single string. Simply concatenating these values would not work because, for example, the indices { `"aa"`, `"bb"` } and { `"aab"`, `"b"` } would both result in the collection key `"aabb"`. Instead, we take care to encode multi-column keys using a separator with escaping.

4.4.4 Reference operations

As described in appendix A, a *reference* in Alan points to an entry in a collection (identified by a reference path). Statically guaranteeing that a reference is valid is in general quite difficult, which is why all but one of the operations are marked as unsafe.

```
"machines" :: collection <= map table "machines":
    encode key -> (
        "description" :: text <= .["description"]
    )
"default machine" :: reference <"machines"> <= key <!> .["default machine id"]
```

There are three operation types for references:

- `key<!>`
- `index<!>`
- `entry`

The `key<!>` operation simply takes a text value, and uses this to index into the referenced collection. If the lookup fails, it causes an import failure.

```
"machines" :: collection <= map (table "machines" filter .["active"] == true):
    encode key -> (
        "description" :: text <= .["description"]
    )
"customers" :: collection <= map table "customers":
    encode key -> (
        "full name" :: text <= .["full name"]
    )
"products" :: collection <= map table "products":
    encode key -> (
        "machine" :: reference <../"machines"> <= index <!> (. -> "machine")
        "customer" :: reference <../"customers"> <= entry (. -> "customer")
    )
```

The other two operations are a little more subtle. An `index<!>` operation takes as input a row, which is then used to index into the **rowset** of the collection that is referenced. If we look at the example above, each "product" has a reference to the "machines" collection. We use the `(. -> "machine")` link as a lookup into the collection's row set. To get a collection key we can just run the collection's key expression on that row.

This operation is *still* unsafe, because we have no guarantee that the row is actually part of the row set! In the example above, if it turns out that the machine does not fulfill the filter `(.["active"] == true)`, then it cannot be an element of the "machines" collection.

In fact, using our type system there is only one case where we know for certain that we have a valid reference. This is the case where the link chain of the collection **rowset** contains just the base table. In other words, the referenced collection imports the entire table, and thus any row we obtain that originates from the same table must be part of the collection. This is a rather common use case, and thus we have added a special operation for it: the `entry` operation.

> **Constraint:** Given: a reference property r referencing a collection c, where c maps over an input of type $\textbf{rowset}[\,\vec{p}, \twoheadrightarrow s\,]$. An `entry` operation on r takes an input of type $\textbf{rowset}[\,\vec{q}, \rightarrow t\,]$. This operation is valid **iff** $s = t$ and $\vec{p} = [\,]$ (i.e. the type of the input to c is precisely $\textbf{rowset}[\twoheadrightarrow t\,]$).

4.4.5 Collection-of operations

The collection-of property type in Alan provides a similar challenge as reference properties. Again we have a path to some other collection, only this time we want to select a *subset* of that collection, rather than just a single entry. There are two operations available, one unsafe and one safe:

- `intersect<!>`
- `subset`

```
"operations" :: collection <= map table "operations":
    encode key -> (
        "tools" :: collection <= map table "tools":
            encode key -> (
                "description" :: text <= .["description"]
            )
    )
"products" :: collection <= map table "products":
    encode key -> (
        "product operations" :: collection of <../"operations">
            <= subset (. -> "operations")
        "tools" :: collection of <"product operations"/"tools">
            <= intersect <!> (. -> "tools")
    )
```

Analogous to `index` for references, the `intersect` operation takes as input a set of rows of the same table as the referenced collection, and uses these to fill the collection-of. We call the operation "intersect" because we are comparing two subsets of the same table, where neither is guaranteed to be a subset of the other.

The `subset` operation (like `entry` for references) is the safe counterpart, which only works if the referenced collection uses the complete contents of a table.

> **Constraint:** Given: a collection-of property c' referencing a collection c, where c maps over an input of type $\textbf{rowset}[\,\vec{p}, \twoheadrightarrow s\,]$. A `subset` operation on c' takes an input of type $\textbf{rowset}[\,\vec{q}, \twoheadrightarrow t\,]$. This operation is valid **iff** $s = t$ and $\vec{p} = [\,]$ (i.e. the type of the input to c is precisely $\textbf{rowset}[\twoheadrightarrow t\,]$).

4.4.6 Components

The last property type in Alan that we have yet to cover are *components*. Recall that components are named type fragments, which may be instantiated in multiple locations (reuse), or even within itself (recursive structures).

```
(
    "products" :: collection <= map table "products": encode key -> (
        "operations" :: component "operation list" <= . -> "first operation"
    )
)

component "operation list" (
    "description" :: text <= .["description"]
    "has tail" :: option <= settle "next operation" = . -> "successor"
        => "yes" :: (
            "tail" :: component "operation list" <= $"next operation"
        )
        none => "no" :: ()
    )
)
```

A component takes an expression of type **rowset**$[\vec{p}, \rightarrow t]$ (i.e. a single row), which then gets passed to the component as its context. There is a risk here, in that if the links we follow end up going in a cycle, the entire construction process will loop forever. We provide a few constructs to prevent such a scenario, which we detail in the following section.

4.5 Graph Constraints

Some of the more interesting data structures we may encounter come in the form of *graphs*. For example, table 4.1 shows a simple "bill of materials" hierarchy for the production of a bicycle. Each row is a part, and dependencies between parts are represented by a foreign key to the parent part. There should one part – the end product – which does not have a parent, denoted by a NULL value.

Bill of Materials		
part id	**parent id**	**description**
40-001	NULL	Bicycle
50-001	40-001	Bicycle frame
50-002	40-001	Saddle
60-001	50-002	Saddle support
60-002	50-002	MS Bolt
60-003	50-002	M10 Square nut
...	...	...

Table 4.1: An example of a graph structure encoded as a table.

If designed properly, the table should have a foreign key constraint on the "parent id" column, referencing back to the same table. Since this is a foreign key on a nullable column, we can import it as an **optional** link. We might define a schema for this table as follows:

```
table "bill of materials" <= map external "bill of materials"
    column * "part id" :: text <= .["part id"]
    column "parent id" :: text ? <= .["parent id"]
    column "description" :: text <= .["description"]
    link "parent part": single "bill of materials" ?
        <= { "part id" <= .["parent id"] }
    link "subparts": many "bill of materials"
        <= reverse "parent part"
```

The above setup is a common way to encode graphs in tabular format. The problem with this approach is that we have no guarantee about the form of the graph. We cannot, for example, ensure that the graph is connected, that there are no cycles, or that the graph is a tree.

To allow a programmer to express the assumption that something is a tree, we have added the `root` operator. This operator is part of the expression language, and has the following typing rule:

$$\frac{\Gamma \vdash e : \textbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad l \in \text{links}(t)}{\Gamma \vdash (\texttt{root}\ e\ \texttt{following}\ l) : \textbf{rowset}[\twoheadrightarrow t, \rightarrow t\,]}$$

The operator takes a table, and returns the root of the tree by following the given link (parent link). It fails unless all of the following conditions hold:

1. The graph is connected
2. The graph is acyclic
3. There is exactly one node without a parent link: the root[5]

A construction of the *bill of materials* table using this operator is shown below. Once we have the root, we can then recursively build up the tree by following the reverse `subparts` link, with the knowledge that we (1) cover the entire tree, and (2) terminate.

```
(
    "bill of materials" :: component "BOM tree"
        <= root (table "bill of materials") following "parent part"
            <! "Bill of materials should form a tree" >
)

component "BOM tree" (
    "subparts" :: collection <= map . -> "subparts": encode key -> (
        "part" :: component "BOM tree" <= .
    )
)
```

We can do something similar for lists (linear graphs). The extra requirement we need is that each node only has exactly *one* successor (or none if it is the last). We have already seen a way to model this kind of relation: give the reverse link multiplicity single. This allows us to assume a one-to-one relation between parent/child.

```
// Distillation
table "steps" <= map external "steps"
    column * "step id" :: integer <= .["step id"]
    column "description" :: text <= .["description"]
    column "previous step id" :: integer ?
    link "previous step": single "steps" ?
        <= { "step id" <= .["previous step id"] }
    link "next step": single "steps" ! ?
        <= reverse "previous step"
```

[5] The third condition is necessary, because a “parent” link implies a direction, and thus we cannot root the tree on any arbitrary node. Technically, we are dealing with the directed graph equivalent of a tree, sometimes called an *arborescence*.

```
// Construction
(
    "steps" :: component "step list"
        <= root (table "steps") following "previous step"
            <! "Steps should have a unique root" >
)

component "step list" (
    "description" :: text <= .["description"]
    "has tail" :: option <= settle "next step" = . -> "next step"
        => "yes" :: (
            "tail" :: component "step list" <= $"next step"
                <! "Non-final step should have exactly one successor" >
        )
        none => "no" :: ()
    )
)
```

Chapter 5

Implementation

We have built an implementation of Square, to serve as a proof of concept, but also to support our evaluation effort in the following chapters. This implementation consists of two main parts:

- The **language definition**
- The **runtime engine**

We make use of M-industries' language development platform. This platform makes it easy to build a compiler from a language definition, and a C++ API to access a compiled data structure (AST) for the runtime engine. The platform is proprietary, but the architecture we describe in the following sections can be implemented using any comparable toolchain. In section 5.4 we cover the tracing algorithm that we use to generate error reports.

5.1 Language Definition

The language definition consists of:

- An abstract syntax definition (written in Alan)
- A set of semantic constraints on the abstract syntax (e.g. for type checking)
- A grammar definition

The abstract syntax of the language is defined as an Alan model. Every possible Square AST (Abstract Syntax Tree) is an instance of this Alan model. The type system (and all other static checks) are implemented as additional constraints on the model. The constraints allow us to invalidate any ASTs which we deem semantically invalid (cf. the typing rules and constraints listed in chapter 4).

The abstract syntax model is accompanied by a grammar, which annotates each part of the model with a corresponding concrete syntax rule. A version of this grammar – rewritten in a more standard notation – is given in appendix B. Altogether, the language definition consists of about ~3000 lines of code.

Feeding the language definition to M-industries' *Alan compiler* allows us to parse and validate Square programs. The result of such a compilation step is an abstract syntax tree, which we can then feed to the Square engine in order to execute it.

5.2 Engine

The Square runtime engine is an application written in C++. The engine is configured by an import definition (a Square AST). Executing the engine takes raw data as input, and transforms it according to the import definition. An overview of the architecture is given in figure 5.1.

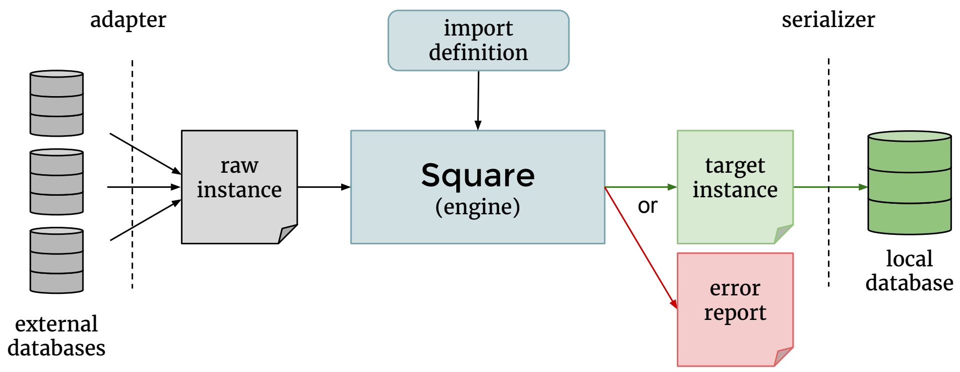

Figure 5.1: High-level architecture of the Square engine.

The output of the engine is either (1) a valid instance of the target schema, or (2) an error report, in case of one or more failed assumptions. Integration with external source and target database systems is done through *adapters* and *serializers*, respectively.

5.2.1 Adapters

The raw data set that the engine takes as input must contain the actual data, but also some metadata (schema information like data types and integrity constraints). This is because we use the schema information in our type system (cf. chapter 4). We use a custom in-memory database engine to store and query this raw data. An *adapter* is necessary to convert from a particular database format to this representation. We currently provide two adapters:

- CSV adapter
- SQL adapter (through JDBC)

The CSV adapter is very basic, taking a set of CSV files (one table per file), using the line number as the primary key, and annotating each column as a non-nullable text column. Since a CSV file does not conform to any schema, this is the best we can do.

The SQL adapter uses Java's JDBC interface, which means we can read from any DBMS that has a JDBC connector available.

The SQL adapter doubles as a *query generator*. We can use it to generate an SQL query from a distillation definition, to get exactly the data we need. This is necessary, because some of the tables we encounter contain on the order of millions of records, whereas a distillation may filter this down to just a few thousands. Filtering the data remotely through SQL is important to get a real-time performance.

5.2.2 Serializers

Once we have a target instance, we want to load it into the local database. A *serializer* is needed to export to a particular format. The most natural data formats for Alan are (semi-)structured formats, so we provide serializers for the following common representations:

- JSON
- XML

Currently, we do not support any kind of diffing mechanism. In other words, each time you run the importer you get the entire target data set as output. In the future, we would like to implement a smarter kind of loader that takes the existing state of the local database into account. Doing so would require specialized loaders for compatible database systems (e.g. XML databases, or JSON-based systems like MongoDB).

5.3 Mechanics

5.3.1 Distillation

The raw data is obtained through the following C++ function interface, which is to be implemented by any adapter:

```
Database adapt(string file_path);
```

We designed a simple in-memory database engine for relational data, with support for basic indexing and querying. The data structure that we store in memory conforms to the hierarchy in figure 5.2.

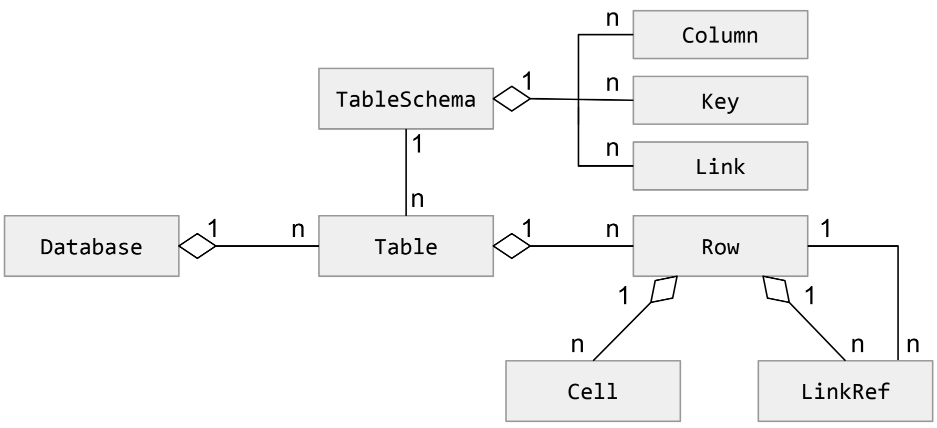

Figure 5.2: Database class diagram (UML).

The distillation process takes one instance of database structure (the *raw* database), and transforms it to another instance (the *source* database). This transformation is guided by the AST of the distillation definition. The function signature for this procedure is:

```
Database distill(AST::Distillation definition, Database raw);
```

5.3.2 Expression engine

There is an isolated, reusable module for expression evaluation. This module exports an evaluation function with the following signature:

```
Value evaluate(AST::Expression, Scope);
```

The scope object provides the scope constants, including the context ("."). It returns an instance of `Value`, which is the main interface for expression values, as shown in figure 5.3.

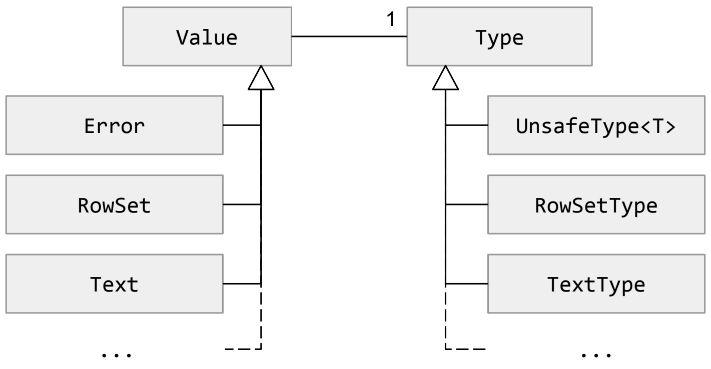

Figure 5.3: Expression value hierarchy.

5.3.3 Construction

Finally, the construction step transforms a source database into a target instance:

```
Target construct(AST::Construction definition, Database source);
```

Similar to the `distill` function, we traverse the AST node of the construction definition, building up the target instance piece by piece as we go. Whenever the AST contains an expression to be evaluated, we invoke the previously mentioned expression engine with the proper scope.

The `Target` data structure is an in-memory representation of an Alan instance. To output this data, we pass it to the chosen serializer. Each serializer implements the function interface:

```
string serialize(Target instance);
```

5.4 Tracing

In the context of model transformation languages, *traceability* is the ability to maintain a relation between target elements and their corresponding source elements [43, 44]. Traceability information allows us to analyze a model transformation; in particular, we can use it to derive debugging information in the case of a failure.

Traceability information generally takes on the form of a graph, where each transformation rule results in a link between the correponding inputs and outputs.

Let's take a look at a (contrived) example. Below, we set a target property `"number of products"` by writing an assumption that an unsafe constant `"parsed quantity"` may be safely "unwrapped" to an integer.

```
// Construction
"number of products" :: integer
    <= $"parsed quantity" <!"Quantity should be integer">
```

In our scenario, the assumption fails, and we want to print a useful debug message. The first thing to note is that the actual error did not occur at this location. We simply *use* the result, but did not cause the error. To find the actual reason the error occurred, we need to take a step back to see the definition of the constant `"parsed quantity"`.

```
// Construction
"parsed quantity" = parse ((. -> "order")["quantity"] ++ "00") :: integer
```

The constant is defined as the result of a parse operation, from text to integer. This must be the operation that failed: `parse` could not convert its input. We now know *what* failed, however, this does not give us a great amount of debug information. We really want to find out *why* the error occurred.

```
((. -> "order")["quantity"] ++ "00")
```

In order for `parse` to fail, the input to this operator must have been a text value with an incorrect format: it was not in the form of an integer. Analyzing the expression, the blame lies with the combination of the two arguments of the concat (++) operator. The second argument is a literal value, and therefore not of further interest. The first argument on the other hand, is a cell in a table. Rather than looking at how we got to this cell (namely, the `"order"` link), we instead want to know how the *value* of this cell was obtained. For that, we need to look at the distillation. Let's say the relevant table definition looks as follows:

```
// Distillation
table "orders" <= map external "EXT.ORDERS"
    column "quantity" :: text <= .["raw A"] ++ .["raw B"]
```

Continuing our way up the chain as before, we finally reach our "raw" values: values which are part of the external database. In this case, these are the columns `"raw A"` and `"raw B"`. The graph of operations from the raw values to the error that we have just built up is called a *trace*. Figure 5.4 illustrates this graph for the case where the two raw columns are instantiated to `"4"` and `"a"`, respectively.

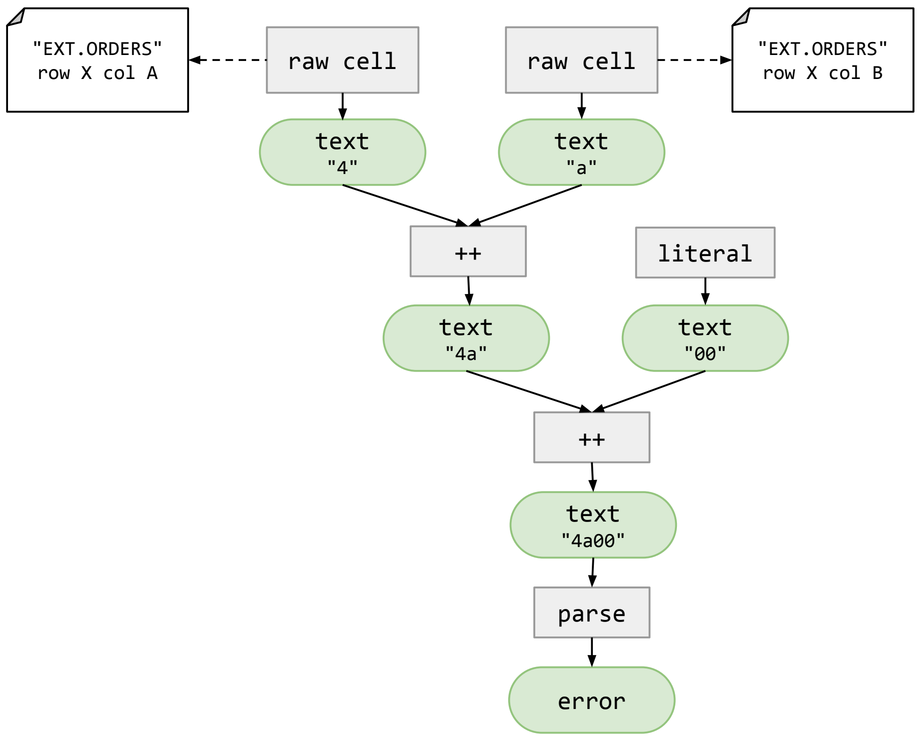

Figure 5.4: Example trace.

We now have all the information we need to provide a detailed error report. In particular, we know *where* the failure occurred (at the point of the assumption marker), *what* the error is (the failed unsafe operator), and *why* it occurred (the trace).

To prevent information overload, we have chosen to only show the list of raw values from the trace in the error report. An example of such a report may look as follows:

```
Failure at <"orders"/"number of products">:
    Assumption failed: "Quantity should be integer"
    Reason: Unable to parse "4a00" as integer.
    Trace:
        - table "EXT.ORDERS" row { "ORDERID" <= "42" }, column "raw A"
        - table "EXT.ORDERS" row { "ORDERID" <= "42" }, column "raw B"
```

In the future, we may expand the debugging facilities, possibly including more detailed visualization of a trace (including all the intermediate steps).

Chapter 6

Case Studies

In order to test Square in a practical setting, and compare it to current approaches, we performed three case studies. Each case study is based on an actual client project from M-industries. Client names have been anonymized, and certain parts of the projects have been simplified for presentation purposes. The three case studies are:

1. Alpha
2. Beta
3. Gamma

We chose these projects over others, because they each pose different design challenges. Gamma, in particular, was chosen because of its relative complexity, and the fact that it was built using an SQL-based import stack.

During this study, we followed the following approach. For each case, we first analyzed the characteristics of the project. We looked at the current import solution (if any). We interviewed three different team members (one per case), in order to discover how the developers experienced writing and maintaining their current importer. In particular, we asked the developers to describe any import failures that occurred, and what the cause was.

In the following sections, we will show how we can design import definitions in Square that match the project requirements. Along the way, we detail important design decisions, and describe any difficulties encountered. Our goal is to show how Square can be applied in a practical setting, as well as demonstrate important quality criteria like usefulness and usability.

6.1 Case Study – Alpha

Alpha is a manufacturer of security products, whose products include home security items and bicycle locks. The company runs multiple production facilities spread throughout Europe. Daily operations for each of these facilities follows the typical manufacturing process: there is a constant stream of production orders coming from Alpha's customers, and a supply chain that delivers the required materials. Alpha's production operators work to fulfill each order by following a pre-specified list of product operations.

M-industries was hired by Alpha to develop a software application that can analyze the performance of their manufacturing process. This application should import the production data in real time, crunch the numbers, then report the results to Alpha's operators and managers using on-site terminal displays.

The performance metrics generated by this application are based on the standard *OEE* (Overall Equipment Effectiveness) heuristic [45]. OEE looks at three factors, the *availability*, *performance*, and *quality* of the manufacturing operation, and combines these into an overall measure of effectiveness (e.g. an operation might be "95% effective" at some particular point in time). Calculating these metrics requires detailed information about the production – machine schedules, operation times, defective units, etc.

It is important to note that the development of this application started while this thesis was ongoing, and in fact uses a version of Square as its import system.[1] Thus, while we cannot use this case study to compare Square to historical results, we will nonetheless use this project to illustrate Square in a practical setting.

6.1.1 Characteristics

Alpha uses an IBM AS/400 system, running a DB2 database. M-industries has read-only access to this database through a remote connection. The OEE application itself runs on a separate system, which displays its output on a terminal at the Alpha production site.

The application was built around a data model of Alpha's production process, written in Alan. Such a model is made by analyzing the company, through communication with an Alpha representative, along with any available technical documents. The model only captures the concepts needed by the application (i.e. what information is required to produce the desired output). Since the model for Alpha is fairly small, we will go through it here in its entirety.

```
(
    // Company assets
    "machines" :: collection (
        "description" :: text
    )
    "tools" :: collection (
```

[1] We say "version of", because M-industries used their own customized implementation to accomodate their own development needs.

```
        "description" :: text
    )
    "tool sets" :: collection (
        // A subset of tools from the above collection
        "tools" :: collection of <../"tools"> ()
    )
```

Alpha's basic assets are its machines and tools. A peculiar concept in Alpha's infrastructure is the existance of "tool sets", which are groups of tools that may be treated as though they were an individual tool.

```
    // Product specifications
    "products" :: collection (
        "description" :: text
        "type" :: option (
            "material" :: ()
            "intermediate" :: ()
            "end product" :: ()
            "unknown" :: ()
        )
        "operations" :: collection (
            "description" :: text
            "has default machine" :: option (
                "no" :: ()
                "yes" :: (
                    "machine" :: reference <../../../"machines">
                )
            )
            "uses tools" :: option (
                "none" :: ()
                "single" :: (
                    "tool" :: reference <../../../"tools">
                )
                "set" :: (
                    "tool set" :: reference <../../../"tool sets">
                )
            )
        )
    )
```

Products capture any item that may be part of manufacturing, including intermediate products and end products (Alpha's product line). In Alpha's database, supply materials are also included under this entity type. The "operations" subcollection specifies the tasks that an operator needs to perform to produce a product of this type. Notice that "operations" does not have an ordering (recall that a `collection` in Alan is unordered), this is because Alpha does not specify a fixed ordering of operations upfront. There is some flexibility to choose the order of operations at production time.

```
    "orders" :: collection (
        "product" :: reference <../"products">
        "status" :: option (
            "pending" :: ()
            "finished" :: ()
        )
        "operations" :: collection of <"product"/"operations"> (
            "description" :: text
            "quantity" :: integer
            "week number" :: integer
            // We may specify a machine, or we may just stick to the default
            "machine" :: option (
                "default" :: ()
                "override" :: (
                    "machine" :: reference <../../../../"machines">
                )
            )
        )
    )
)
```

Finally, *orders* contains incoming orders for an instance of a particular product. The `operations` subcollection may override certain product operations. For example, an order may specify that a particular operation should be performed on a different machine than the default.

6.1.2 Design

The full import definition for Alpha can be found in appendix D.1. We will highlight some of the more interesting design choices.

Tools

The following definition imports the "TOOL" table from the "A" (Alpha) database in DB2. Note that DB2 on AS/400 does not support identifiers greater than 10 characters, which means the external table and column names have been forcefully shortened to the point where they are barely readable.

```
    table "tools" <= map external "A.TOOL"
        column * "tool id" :: text <= .["TOOL#"]
        column "description" :: text <= .["TOOLDSC"]
```

Tool sets are stored in a separate table, where the set ID and tool ID form a compound primary key. This construction looks similar to a join table, although there is no actual "tool set" table that forms the other end of the join. For our purposes however, because we want to import tool sets in to their own collection, we need to aggregate the tool sets themselves. This is accomplished through the additional `tool sets` table shown below.

```
// Table which associates tools with tool sets
table "tools x tool sets" <= map external "A.TOOLSET"
    column * "tool id" :: text <= .["TOOL#"]
    column * "tool set id" :: text <= .["TOOLSET"]
    link "tool": single "tools"
        <= { "tool id" <= .["tool id"] }
    link "tool set": single "tool sets"
        <= { "tool set id" <= .["tool set id"] }

// Aggregate all individual tool sets from the "tools x tool sets" table
table "tool sets" <= aggregate table "tools x tools set"
    column "tool set id" * <= .["tool set id"]
    link "tool entries": many "tools x tool sets"
        <= reverse "tool set"
```

Machine references

There are a few columns that reference other tables without an explicit foreign key. For example, operations have a reference to the `machines` table. This field is non-nullable, but may be empty (""), indicating the lack of a machine. We would like to differentiate between explicitly empty values, and references that do not resolve. To achieve this, we introduced the following construct:

```
// Distillation
table "product operations" <= map external "A.ROUTE"
    //...
    column "machine id" :: text ? <= optional .["MACH#"] on (. != "")
    link "machine": single "machines" ! ?
        <= { "machine id" <= .["machine id"] }

// Construction
"has default machine" :: option <= settle "machine" = . -> "machine"
    => "yes" :: (
        "machine" :: reference <../../../"machines"> <= $"machine"
            <!"Reference to machine should be valid">
    )
    none => "no" :: ()
```

The machine ID is imported as an optional value, where the value only exists if the value is not the empty string. Because the external table has no foreign key, the link `machine` is of type `optional<unsafe<integer>>`. In the construction, we can then settle the optional value to find out whether there is a default machine. If so, we resolve the machine reference using an assumption marker, because we assume that the provided machine reference is always correct (our sample data did not contain any counterexamples).

Collection subset

Production orders in Alpha contain a reference to a product. Let's call this product *P*. There is a complex relation between the operations of *P*, and the operations specified in the order. Recall that in our data model, we modeled an order's operations as a `collection of`, which references *P*'s operation collection. In other words, in other for an instance to conform to this model, the set of order operations should form a *subset* of *P*'s collection of operations.

```
"orders" :: collection <= map table "orders": encode key -> (
    "product" :: reference <../"products"> <= . -> "product"

    "operations" :: collection of <"product"/"operations"> <= subset
        <!"Order operations should form a subset of the product's operations">
        . -> "order operations" -> "operation":
            encode key <!"References to product operations should be distinct">
            -> with .^ (
                "description" :: text <= .["description"]
                //...
            )
)
```

We chose to approach this problem as follows: we fill the order operations collection using the `subset` operation. This operation takes a rowset of the same table as the original collection. However, because we cannot guarantee that an order operation does not inexplicably refer to a product operation of a *different* product, we must mark this operation as unsafe.

In addition, we have no guarantee that `. -> "product operations" -> "operation"` forms a proper set. In other words, two order operations might refer to the same product operation. For this reason, the compiler cannot guarantee the safety of the collection's `encode key` expression.

6.1.3 Discussion

We had originally made a mistake in our model of Alpha (and therefore also in the import definition). Based on Alpha's database schema, we had assumed that a product operation could only refer to tools that are part of the `tools` table. The corresponding construction looked like the following:

```
"products" :: collection <= map table "products": encode key -> (
    "operations" :: collection <= map . -> "operations": encode key -> (
        "uses tool" :: option <= settle "tool" = . -> "tool"
            => "yes" :: (
                "tool" :: reference <../../../"tools"> <= $"tool"
                    <! "Reference to tool should be valid" >
            )
            none => "no" :: ()
    )
)
```

Testing this import definition on our sample data set resulted in the following error report (cf. the tracing algorithm in section 5.4):

```
Failure at <"products"/"operations"/"uses tool"/"yes"/"tool">:
    Assumption failed: "Reference to tool should be valid"
    Reason: Could not resolve link "tool" on table "product operations".
            No row in table "tools" with index:
                { "tool id" <= "A543" }
    Trace:
        - table "A.ROUTE" row { "PROD#" <= "P42" }, column "TOOL#"
```

After discussion with Alpha, it turns out that our model was incorrect. The "tool" reference could be either a tool *or* it could be a tool set. The logic was supposed to be as follows: *if* the tool id refers to a valid entry in `tool set`, use that, *otherwise*, look in the `tools` table. In other words, their schema really looked like this:

```
table "product operations" <= map external "A.ROUTE"
    //...

    // This column ambiguously refers to either a tool *or* a tool set
    column "tool or tool set id" :: text <= .["TOOL#"]

    link "tool": single "tools" ! ?
        <= { "tool id" <= .["tool or tool set id"] }
    link "tool set": single "tool set" ! ?
        <= { "tool set id" <= .["tool or tool set id"] }
```

Referring to two tables ambiguously is considered poor design in relational databases. To tackle this problem, we decided to introduce two separate links to each of the two tables. Both of these links are necessarily unsafe. During construction, we can then try to resolve each of the links in the right order.

6.2 Case Study – Beta

Beta specializes in the production of roll formed metal profiles. *Roll forming* is a production method where thin sheets of metal are bent into complex shapes with a fixed cross-section (“profiles”). These profiles are used as components in various industries, like the automotive industry, construction, and furniture.

The application that M-industries develops for Beta is a complaint management system. Customers may file complaints about Beta’s products when the deliverable does not match the expected quality. The complaint manager keeps track of all of Beta’s complaints and their eventual resolution. Various *key performance indicators* (KPIs) are calculated from this data, which help to inform Beta’s management about the quality of their customer support. Complaints are added directly to the application, but it still needs an importer in order to get an up to date view of Beta’s production operations.

6.2.1 Characteristics

Beta is part of a larger conglomerate of industrial companies, due to an acquisition. This conglomerate has a shared IT infrastructure, including a massive DB2 database containing ~20,000 tables. Among these tables is a single “employees” table, containing all of the employees (past and present) of the entire conglomorate. Extracting just the information relevant to Beta from this database is a challenge in and of itself.

One table in particular is interesting – the “articles” table. This table contains specifications for all sorts of items used in the production process, including materials and profiles. Profiles are grouped into profile types, but this information must be deduced by looking at the initial part of the article ID. We consider this to be a poor relational database design, and the challenge is therefore to dissect this table into a proper hierarchy.

Another interesting aspect of the database design is the tendency to split up tables vertically into a primary table and supplementary tables. These are tables that share a primary key, but where the supplementary tables seem to have been added as an afterthought to provide extra information. Ideally there would be a strict one-to-one relation between the primary and supplementary tables, but this is not something that is guaranteed by the database schema.

The importer for Beta was initially planned to be developed using a system we will call M-industries’ “SQL stack”, a toolchain where all import logic is written using handwritten SQL queries. The result of these queries are then trivially mapped to collection types in the Alan model. The project switched to Square halfway during development.

6.2.2 Design

The import definition for Beta is listed in appendix D.2. We will again go over the more interesting aspects of this design.

Customer groups

Customers of Beta are grouped into customer groups. Each customer should belong to exactly one customer group. The way this has been represented in Beta's database is as follows: customer information is stored in its own table (we will denote this the "primary" table). The group ID of the customer is then stored in a separate table (the "supplementary" table). The two tables are associated throught their shared primary key.

```
table "customers" <= map external "B.VIBQREP"
    column * "customer id" :: text <= .["BQIMNB"]
    column "name" :: text <= .["BQMICD"]
    link "customer group entry": single "customers with group" !
        <= { "customer id" <= .["customer id"] }

table "customers with group" <= map external "B.VIBHREP"
    column * "customer id" :: text <= .["BHIMNB"]
    column "customer group id" :: text <= .["BHS0NA"]
    link "customer": single "customers" !
        <= { "customer id" <= .["customer id"] }
    link "customer group": single "customer groups"
        <= { "customer group id" <= .["customer group id"] }

table "customer groups" <= aggregate table "customers with group"
    column * "customer group id" :: text <= .["customer group id"]
    link "group entries": many "customers with group"
        <= reverse "customer group"
```

We considered a few different ways to model this. For example, we could introduce a special kind of join that enforces a one-to-one relationship. In the end though, we settled for the simpler solution above, where each table links to the other using an unsafe link. We assume that both links always resolve.

Articles

Articles are specifications for all sorts of items used at Beta. The main information is listed in the table "articles". Additional information is again given through a supplementary table.

```
table "articles" <= map external "B.VIAAREP"
    column * "article id" :: text <= .["AAAATX"]
    column "article type" :: text <= .["AADATX"]
    column "description" :: text <= .["AAEVTX"]
    column "customer id" :: text <= .["AGAKCD"]
    link "supplement": single "articles supplement" !
        <= { "article id" <= .["article id"] }

table "articles supplement" <= map external "B.LAST_ART_OV3"
```

```
    column * "article id" :: text <= .["AGAECD"]
    column "quantity per package" :: integer <= .["AGANNB"]
```

More specific types of articles are then extracted from this table, like materials and profiles. In order to get the profile *type*, we need a slightly more complicated operation: get the initial part of the article id as the type specifier, then aggregate on this column.

```
table "profiles" <= map table "articles"
        filter (.["article type"] == "profile") // Filter out just the profiles
    column * "profile id" :: text <= .["article id"]
    // The first 4 characters of the profile ID indicates its "profile type"
    column "type" :: text <= substring .["profile id"] from 0 to 4
    link "article": single "articles"
        <= { "article id" <= "profile id" }
    link "profile type": single "profile types"
        <= { "profile type id" <= .["type"] }

// Aggregate just the profile types
table "profile types" <= aggregate table "profiles"
    column * "profile type id" <= .["type"]
    link "profiles": many "profiles"
        <= reverse "profile type"
```

Product operations

Unlike in Alpha, the product operation list at Beta contains an ordering in the database. The target schema includes a linked list structure (using a recursive Alan *component*) to capture this information.

```
"profile types" :: collection <= map table "profile types": encode key -> (
    "profiles" :: collection <= map . -> "profiles": encode key -> (
        // (...)
        "operations" :: component "profile operation list"
            <= root . -> "operations"
                <!"Operations list should have a unique root">
    )
)
component "profile operation list" (
    "machine group" :: reference </"machine groups">
        <= entry . -> "machine group"
    "operation" :: reference </"operations">
        <= entry . -> "operation"
    "has tail" :: option <= settle "next operation" = . -> "next operation"
        => "yes" :: (
            "tail" :: component "profile operation list" <= $"next operation"
                <!"Non-final operation should have exactly one successor">
        )
        none => "no" :: ()
    )
)
```

6.2.3 Discussion

The most interesting thing about this case study is the recursive component structure. It seemed like quite a challenge to “decode” a linked list of components from a table in a simple, elegant way. In fact, we considered whether we should not just drop this part altogether in the target and use a simple collection.

We designed the graph constraints in section 4.5 based on this use case. There turned out to be a surprisingly elegant way to integrate these constraints with the link feature that we already had, by adding a single `root` operator.

Running the transformation on Beta’s dataset allowed us to check whether there were any spurious references. If there were, it would also inform us exactly which graph property was violated (either the lack of a unique root, or an operation with zero or multiple successors). We discovered that Beta’s database was in order: it passed all checks on the first try.

6.3 Case Study – Gamma

Gamma is a leading aluminium extrusion manufacturer, operating in many countries throughout the world. The company creates aluminium components used in a wide range of products, using a process called *extrusion*. In an extrusion process, material is pressed through a die in order to create objects with a fixed cross-sectional profile (e.g. beams, tracks, frames).

At Gamma, this manufacturing process is managed by a software system called EIS (Extrusion Information System), which oversees the machines that actually produce the aluminium profiles. However, the output of such a machine is not yet a finished product, and a large number of machines are required to do post-processing in a separate fabrication step.

The application that M-industries develops is an extension to EIS called PMI (Product Manager and Interfacing), that takes care of this post-process fabrication. All of the information that PMI needs comes from a remote relational database, which is periodically queried and mapped to an Alan model.

6.3.1 Characteristics

PMI is a large, long running project, and has some of the highest complexity amongst all projects in M-industries’ development history. To import the data, the project used a custom import system that worked by writing a number of SQL queries, one for each `collection` property in the target schema. A converter would translate these query results to an Alan instance, by trivial mapping from table to collection, and column to subproperty. Each property type had their own special encoding that the converter could handle.

We analyzed each of the existing queries, and created an equivalent design written in Square. We will compare the two in the discussion at the end of this section.

6.3.2 Design

Alternate customer key

The `customers` table has a primary key consisting of a unique customer ID. In addition, each customer also has a “short name” that is commonly used as a human readable identifier.

```
table "customers" <= map external "C.EIA4REP"
    column * "customer id" :: text <= .["A4A2NC"]
    column "short name" :: text <= .["A4B4CD"]
```

For presentational purposes in their application, M-industries wants to use the short name as the identifier instead of the customer ID. This should be possible, because there should never be two customers with the same short name. However, this uniqueness constraint is not enforced (i.e. there is no key constraint in the schema).

We have chosen to solve this design challenge by introducing an *unsafe key* definition for this column. The intuition here is that we feel this key is “missing”, and we will introduce it regardless. We *assume* that we always get a unique record by accessing this key, but doing so is naturally unsafe.

```
key "short name": ! { "short name" }
```

In the construction, the resulting definitions looks as follows. We use the short name as our key, and the resulting entry context (`.`) is thus of type `unsafe<row>`. To prevent having to unwrap this unsafe context for each property, we use a `with` expression to write a single assumption marker at the collection entry level.

```
"customers" :: collection <= map table "customers": encode key "short name"
    -> with . <!"Short name should be unique"> (
        "branch code" :: text <= .["branch code"]
        //...
    )
```

Production articles

At Gamma, there is a certain flexibility for where a particular article may be produced. A given article may be produced at any amount of different locations, and at each such location there are different production specifications. In the Gamma schema, this is modeled as a many-to-many relation, as follows:

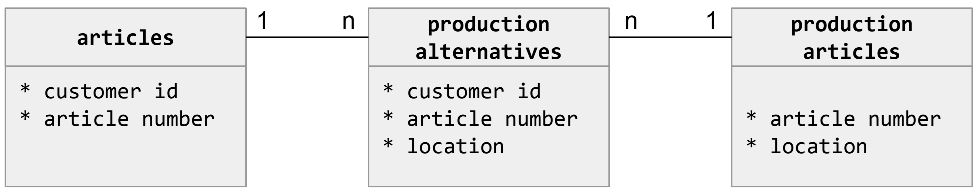

Figure 6.1: Tables related to production, with their primary key columns.

For this particular application, we are *only* interested in articles produced at a particular location, namely the site in Hungary. Note that when we fix the location column in the alternatives table, we end up with exactly the articles that are produced in Hungary, along with singular links to the corresponding article information *and* its production information in the production articles table. This allows us to squash this many-to-many relation to a flat list of articles we are interested in:

```
table "articles in hungary" <= map table "production alternatives"
        filter .["location"] == "HUN" // Just the articles produced in Hungary
    column * "customer id" :: text <= .["customer id"]
    column * "customer article number" :: text <= .["customer article number"]
    column * "location" :: text <= .["location"]
    column "quantity in stock" :: integer
        <= (. -> "production article")["quantity in stock"]
    column "material source" :: text
        <= (. -> "production article")["material source"]
```

6.3.3 Discussion

As mentioned in section 6.3.1, the Gamma project initially used a custom SQL-based importer. We interviewed the main M-industries developer in charge of this project, in order to compare the two methods.

We discovered that the SQL stack would result in failures on a regular basis. For example, it turns out that the external schema lacked a foreign key constraint on a certain column. Articles were expected to have a reference to a “product group”, but no foreign key constraint was specified. For a while, this would not cause any problems. However, at a certain point, an illegal reference occurred in the articles table.

As a result, the LEFT JOIN operation in the SQL import query would resolve to NULL. This resulted in an import error, which subsequently caused the entire application to halt. Debugging this error turned out to be difficult, because the importer was unable to specify what exactly went wrong beyond a basic “unexpected NULL value” error message.

In the Square version, the equivalent definition looks as follows:

```
table "articles" <= map external "C.EIIQREP"
    //...
    column "product group id" :: text <= .["IQPLKD"]
    link "product group": single "product groups" !
        <= { "product group id" <= .["product group id"] }
```

The presence of the unsafe type (!) on the link is mandated by the lack of a foreign key on the "IQPLKD" column. When we use the link, we have to add an assumption marker:

```
"article types" :: collection <= map table "articles in hungary": encode key -> (
    "product group" :: reference <../"product groups">
        <= index (. -> "product group")
```

```
               <!"Each article should belong to a product group">
)
```

Thus, when we run test the same error using Square, the system will inform us exactly *why* the error occurred, and on which row(s).

Going back to the original developers of the Gamma project, they responded positively. Initially, the idea of having to add these kind of markers throughout the code was thought to be strange. However, it did not take long for the developers to get used to the idea, and the increased quality of the feedback in case of errors was very much welcomed.

Chapter 7

Evaluation

7.1 Evaluation Setup

7.1.1 Main requirements

The engineering challenge we set for ourselves back in chapter 1 was to design a specialized language for continuous data importing. This language must:

- Provide natural abstraction mechanisms to represent the source and target systems.
- Allow us to express a transformation from the source to the target, where the language statically guarantees that the transformation always results in either (1) a valid instance of the target, or (2) an error report that tells us exactly where the programmer made a wrong assumption.

These are the basic requirements for functional *correctness*. In addition, we formulated a few core goals that the language should fulfill:

1. *Transparency*. The transformation language should provide high-level, specialized abstractions closely matching the problem domain (i.e. the language should be highly declarative). There should be low barriers for both parties to introspect and validate the semantics of the transformation definition.
2. *Debugability*. The language should facilitate easy debugging of import failures, when they occur. Good support for tracing is a must.
3. *Expressivity*. The language should allow us to express a wide range of transformation operations, as long as it does not detract too much from the other two goals.

The first two goals focus on the prevention of failures beforehand, and resolving of failures after they occur, respectively. The last goal states that the language should be widely applicable. There is a trade-off between the first two goals and the third. As explained in chapter 1, we prioritize transparency and debugability over expressivity, as long as we can handle most common use cases.

7.1.2 Quality criteria

In addition to our own specific requirements, we want to evaluate how well Square performs when compared to model transformation approaches in general. We will use the taxonomy of model transformation languages defined by Mens et al. [7].

The taxonomy of Mens at al. consists of five main "questions" that identify important aspects of a model transformation language. In particular, the taxonomy classifies model transformation languages according to the following main aspects:

1. Characteristics of the source and target
2. Characteristics of the transformation
3. Functional requirements
4. Non-functional requirements
5. Mechanics

7.1.3 Evaluation method

We provide an evaluation of Square according to the requirements and quality criteria mentioned above. This is a personal analysis, supported by several information sources: the factual descriptions of Square as given in chapters 3, 4, and 5, the practical case studies of chapter 6, and the interviews performed with M-industries' developers who have worked on the projects that the case studies were based on.

7.2 Requirements

7.2.1 Correctness

By definition, a transformation is considered **correct**, if, for every possible source instance, the transformation produces either (1) a valid target instance, or (2) an error report in case of one or more failures.

For Square, this means that every potential target instance resulting from a construction step *must* be a valid instance of the target Alan model. The static semantics discussed in section 4.4 were designed such that:

1. The structure of the construction follows the structure of the target schema.
2. Referential integrity (`reference` and `collection-of` properties) is properly maintained by the relevant construction operations.

To *prove* that each possible output of our transformation conforms to the target schema, we would need the full operational semantics of the language. In this thesis report, we provide the typing rules and other static constraints that – as we reason in section 4.4 – *should* result in a valid instance. However, we do not prove this formally.

We have a reasonable level of trust that the system does indeed always produce valid instances, due to the fact that we tested the system "in the wild", on a large amount of data from different projects. But again, we have no formal proof of this fact.

7.2.2 Goals

Square is less expressive than a general-purpose solution, but we even fall short when compared to many comparable languages like SQL. We explicitly chose to sacrifice expressivity wherever it conflicted too much with our other goals. However, we would like Square to be **expressive enough** for most common patterns we find in real-world projects.

In our case studies, we demonstrated that Square is expressive enough to write import definitions for three different projects. The question of how *representative* these examples are is discussed in our threats to validity in section 7.4. We believe that Square is expressive enough for most simple problems, and easily extended in many others (e.g. addition of new operators). The case studies and its continued use in production systems provide some evidence for this.

Square focuses heavily on the **debugability** of the language in case of import failures. The main mechanisms are the type system – limiting potential failures to a particular language construct (assumptions) – and tracing. Error reports use the assumption marker as well as traces to help programmers identify the cause of failures. The traceability features of Square were described in section 5.4, and we gave a practical example of its usage following an actual import error in case study Beta.

The **transparency** of the language is somewhat subjective, and therefore difficult to quantify. In case study Gamma, we compared developer's experiences in using Square versus using an equivalent written in SQL. For this specific case at least, the developers judged Square to be more readable and easier to analyze and debug than the SQL version.

7.3 Quality Analysis

The following is based on the taxonomy of Mens et al. [7]. We answer each question posed in their taxonomy in the context of our approach.

1. *What needs to be transformed into what?*

   Square is a **model transformation** language. It transforms relational data to a **single target** data model. The language ostensibly supports **multiple sources**, although within the context of the transformation definition itself the programmer is presented with a single "source" abstraction (with different namespaces). Multiple sources must be consolidated into a single internal format through the use of an adapter.

   The transformation is classified as a **migration**, meaning it is an **exogenous**, **horizontal** transformation (different metamodels, working at the same level of abstraction). The technical spaces we use for the source and target are the **relational model**, and **Alan**, respectively. The transformation operates partly in each of the technical spaces (distillation versus construction).

2. *What are the important characteristics of a model transformation?*

A Square transformation is a manual transformation, of a fairly high level of complexity. Although ideally the transformation simply translates similar concepts from one metamodel to another, we need to be able to perform fairly complex operations (breaking up and combining tables, transforming values) in order to have the source model match up with the target.

3. *What are the success criteria for a transformation language/tool?*

   Mens et al. suggest a number of functional requirements that may contribute to the success of a transformation tool. Below, we give each criterion a score from (1) to (5). These scores should be interpreted as follows:

   - (1) Feature nonexistent.
   - (2) Very limited functionality.
   - (3) Partial functionality.
   - (4) Decent functionality.
   - (5) Feature fully present without limitation.

   Next to the score are relevant chapters (if any) that the reader may refer to.

   - **Ability to create/read/update/delete transformations**

(1)					×	(5)

     As stated by Mens et al., the ability to manage transformations is a trivial property of transformation languages (because we are just dealing with text). This criterion is relevant only for certain kinds of tools.
   - **Ability to suggest when to apply transformations**

N/A

     Only applicable to interactive tools.
   - **Ability to customize or reuse transformations**

(1)	×					(5)

     All language constructs are predefined, and we do not allow customization or extension of these constructs.
   - **Ability to guarantee correctness of the transformations**

(1)					×	(5)	(ch. 6)

     Assuming our language is correct, we always guarantee that the output is a valid instance. Whether the language is correct or not is a separate matter, see the discussion in the previous section.
   - **Ability to deal with incomplete or inconsistent models**

(1)	×					(5)	(ch. 5)

     No mechanism for consistency management is provided.
   - **Ability to group, compose and decompose transformations**

(1)		×				(5)	(ch. 4)

     Partially available using scope constants, but very limited.

- **Ability to test, validate and verify transformations**

(1)		×				(5)

  Testing transformation is only possible by executing them directly. There is no test infrastructure available, and (because this is a wholly new tool) no existing framework can be applied to it.

- **Ability to specify generic and higher-order transformations**

(1)	×					(5)

  Transformation rules are not first-class citizens in our language.

- **Ability to specify bidirectional transformations**

(1)	×					(5)

  Square only works in one direction; target instances cannot be transformed back into its source.

- **Support for traceability and change propagation**

(1)				×		(5)

  (ch. 6)

  We provide fairly elaborate support for tracing in the implementation. However, we lack a change propagation mechanism.

4. *What are the quality requirements for a transformation language/tool?*

   Besides the functional requirements above, Mens et al. also stipulate a number of non-functional requirements. The scores below should be interpreted as a scale from low quality to high quality.

- **Usability and usefulness**

(1)				×		(5)

  (ch. 6)

  We use the case studies to demonstrate both concrete use cases and practical developer experience.

- **Conciseness**

(1)				×		(5)

  (ch. 6)

  We place a high focus on simplicity and conciseness of the syntax. Many high-level operations are available, as opposed to a large number of simple ones.

- **Scalability**

(1)				×		(5)

  (ch. 6)

  The case studies have been tested with large sets of data (on the order of GB), from real company databases with a long operational history.

- **Mathematical properties**

(1)		×				(5)

  (ch. 4)

  We do not give a formal proof of the correctness of the language. We do strive to guarantee a number of properties of the transformation. For example, the expression language was built such that an expression always terminates (although a transformation as a whole may still loop through misuse of component structures).

- **Acceptability by user community**

(1)	×					(5)

At the moment, there is no user community.

- **Standardization**

(1)		×				(5)

Relevant standards are SQL, and Alan, for the source and target formats respectively. The SQL adapter that extracts data from SQL databases uses the standard JDBC interface, but this is fairly trivial. Alan is not a (public) standard.

5. *Which mechanisms can be used for model transformation?*

   Square focuses heavily on **functional programming** mechanisms. All operations are pure (side-effect free) functions, and operations routinely take other expressions as input (similar to higher-order functions). The language tries to be as declarative as possible.

This overview shows how Square lacks a lot of functionality that might be expected from a more general, powerful transformation language. Instead, we focus on a few key aspects (transparency, static guarantees, traceability).

7.4 Threats to Validity

The conclusions that we have presented in this chapter are subject to a number of possible threats to validity. In this section, we treat each of the different categories of validity one by one.

7.4.1 Internal

The results of this evaluation were made through personal analysis. Although we reference several sources of information (e.g. developer interviews), there is an undeniable risk of bias from the author to good scores. We have tried to limit these threats by motivating each score individually and citing relevant sources.

7.4.2 External

In our analysis, we depend heavily on the results of the case studies we performed. We try to generalize from these three cases. We have tried to choose projects which are representative for different classes of problems. However, because this thesis project was performed at one company (M-industries), working in a specific niche (industrial companies), our sample set is necessarily limited.

There is an additional threat, in that some of the case studies that we used for the evaluation were also used as inspiration to design the language in the first place. This causes a kind of "overfitting" effect, where it is natural for the language to perform well on the cases for which it was designed.

7.4.3 Construct validity

We have several conclusions based on the results of interviews with developers of M-industries. There is a possible bias here, because of familiarity of these developers with the author of the experiment. In addition, some of these developers were asked for advice during the design on the language, which makes it natural for these developers to be pleased with the results.

7.4.4 Reliability

If we were to perform this evaluation again at some later point, we expect the same results. However, there are some threats. For one, the tool that we have created is fairly new. Following some time, after extensive use, limitations may be found. Other, modern tools created after ours may expose lacking features which were not considered before.

Chapter 8

Related Work

At its core, data importing is just a conversion between data formats. This is not a new idea. On the contrary, transformations from one model of data to another are arguably one of the oldest and most fundamental ideas in software engineering (colorfully illustrated by Favre in the series "From Ancient Egypt to Model Driven Engineering" [46, 47]).

We will not claim that the work presented in this thesis report is a wholly new invention, rather we argue that we use a novel combination of techniques that work well for the specific problem at hand. By focusing heavily on transparency and static guarantees, we make a strong trade-off of safety over expressivity, which differentiates Square from other approaches.

8.1 ETL Methods

As touched upon in chapter 2, data warehousing faces many similar challenges. And because this is a relatively mature field with great interest from industry, there is a lot of research into good methods for ETL. Note that these methods generally integrate different relational databases into another relational database, rather than a structured data model, and thus is mostly just relevant for what we have termed *distillation*.

Calvanese et al. [48] present a framework for data integration, where the data warehouse itself is written as a view over the source data. Unlike the typical views in relational databases, the views in this framework are written as queries on a conceptual model – expressed as an Entity-Relation diagram. The developer can then express *assertions* between intermediate entities of the model.

The concept of *assertions* looks similar to the concept of *assumptions* presented in this thesis report, but the two are markedly different. We use the work of Calvanese et al. [48] as an example here, but the distinction applies to a larger category of methods (notably, those using OCL constraints on UML models).

An *assertion* provides a runtime check to ensure that some set of data is valid. In other words, we start with a more lenient type system, and then add constraints to limit the allowed set of values using ad-hoc expressions.

Our approach on the other hand, starts with a type system that allows only values where the source data itself guarantees it to be valid, and we can express additional (non-statically verifyable) operations only if the programmer tells the system that it is an assumption on their part (i.e. a risk). To put it simply, assertions are used to nail an unsafe transformation down, assumptions are required to open a safe transformation up.

In [31], Vassiliadis et al. present a conceptual modeling framework for ETL. This framework uses a custom transformation language which is specifically made for the kinds of patterns found in ETL. This results in a succinct, easy to understand transformation for most applications found in the data warehousing domain. This is similar to our own approach, where we designed the language specifically for the kinds of patterns we saw in real-world projects. In fact, we took inspiration from the model of building up a transformation through a graph (see e.g. the tracing algorithm in section 5.4). Unlike Square however, the model of Vassiliadis et al. use the same kind of assertion system as discussed above.

8.2 Relational – Structured Data Mapping

In going from a relational database schema to an Alan model, we need to map concepts from the former to the concepts from the latter. We can do this manually (i.e. on the individual schema level), or automatically (i.e. on the level of the schema language). The latter corresponds to the notion of a technical space projector [8], as discussed in chapter 2. Semi-automated methods also exist, by providing defaults but allowing customizations.

Due to SQL's ubiquity as a database language, pretty much every technical space has been projected to, and – more relevant for our purposes – *from* the SQL data model. Notable examples include generating program structures for in-memory manipulation of data, as is done in Object-Relational Mapping [49], LINQ [50], or in functional languages through by mapping algebraic data types [51]. Other examples are found in document formats, especially in the XML world, where it is not uncommon to generate XML Schemas from relational ones [52].

All of these mappings serve to automatically generate a new schema from a relational one. This is not particularly useful for our purposes, because both the source and target schema are already given, and thus we have no control over the schema we map to. We could, however, implement some semi-automatic operations in cases where the schemas happen to conform to certain patterns.

We identified a few of these patterns when designing Square. For example, we can easily map graphs encoded as tables to components in Alan (cf. 4.5). Another possibility (currently not implemented) would be to examine how the concept of an `option` type in Alan is commonly encoded in a relational schema, and support specific language operations for those encodings. In designing Square, we took inspiration from common conversion schemes in XML and OOP tools, where applicable to Alan. Other (Alan-specific) constructs like `collection-of` do not have easy correspondences (as far as we could find), and so had to be designed from scratch.

8.3 Model Transformations

The work described above can be mostly generalized to the concept of a *model transformation*. Model transformations are fundamental to model-driven engineering, and hence have been studied extensively in this field. The prime example of a model-driven engineering framework is OMG/MDA [10]. In MDA, the standard model transformation language is QVT [21], although other MOF-compatible languages have been proposed, like the ATLAS Transformation Language (ATL) [53].

These model transformation languages work by expressing a set of rules from one metamodel to another. A sort of pattern matching is used to match source elements, and transform them to a target element. The language ensures that any result is valid according to the target metamodel.

Model transformation languages, like QVT and ATL are very powerful. The differences lie mostly in the fact that Square does not need to be anywhere near as general or flexible, instead we can design the language specifically for the problem domain. Instead of pattern matching (which we can be difficult to reason about), we specify the target instance directly in terms of the target schema. Additionally, the concept of assumptions does not have a direct analog in any generalized model transformation we know of.

8.4 Static Guarantees

From the start, we focused heavily on error prevention and error reporting. To achieve this, we used mostly standard techniques from type theory and compiler construction (semantic constraints). We took inspiration from functional languages and the lambda calculus that underlies them for our typing setup, as covered in any introductory text (e.g. [42]).

One novelty we introduced was the concept of a *link chain*. This was a natural consequence of the need to constrain operations on sets of rows, beyond just their row type. The idea is similar to a *dependent type system* [54], which allows more expressive typing rules that depend not just on types but terms as well.

Ultimately, we chose to implement link chains without introducing the full power of dependent types in our type system. However, further work on Square would naturally work in this direction. A dependent type system would provide an elegant solution to provide even more safety than we have now. For example, it would allow us to make division safe, by disallowing division by zero on the type level.

Chapter 9

Conclusion

9.1 Research Questions

In chapter 1, we formulated a number of research questions. We can now revisit these questions one by one.

RQ1 What are the characteristics of current methods? In particular, what kind of static guarantees do these methods give us about runtime failures?

Data importing requires us to convert from a source schema to a target schema, where both schemas are given beforehand. This is fairly general idea, and we see this problem in a number of different fields in different incarnations (cf. chapter 2). Typically, import methods require some kind of manual (or semi-automated) transformation definition, which maps the source schema to the target schema. Some kind of validation mechanism is usually provided to invalid certain values (cf. chapter 8).

Some methods (especially in ETL, e.g. [31]) do not give any static guarantees about arbitrary inputs. Other methods (especially in model-driven engineering, e.g. QVT [21]) take a more formal model transformation approach, where the system guarantees a valid output by specialized transformation rules. Most of these transformation languages operate through a pattern matching mechanism.

RQ2 What kind of data import problems typically occur in industrial projects?

In chapter 6, we presented three case studies based on real-world industrial projects. We identified a number of design problems in the source database schemas, such as ambiguous references, entities that have been split up in unnatural ways across tables, graph structures encoded as flat tables, etc.

Of course, the case studies we have studied cannot represent the full spectrum of problems that a developer may encounter (see the threats to validity in section 7.4).

RQ3 What kind of language constructs do we need to be able to implement common use cases in way that is transparent and easy to debug?

In chapters 3 and 4, we have presented Square, a language to define import definitions, where we can clean up a relational schema using a distillation step, followed by a mapping to an Alan model in a subsequent construction step. The case studies demonstrate how this language is expressive enough to tackle the design problems we encountered. In addition, in the evaluation (chapter 7), we argue that the language performs well in terms of transparency and debugability (although again, we refer to the threats to validity).

9.2 Conclusions

It is commonly said that "assumption is the mother of all screw-ups". This certainly seems to be the case in data migration, where many system failures stem from making the wrong assumptions about the incoming data. Rather than trying to prevent assumptions altogether, we take the practical stance and say that uncertainty is an unavoidable part of integrating real-world systems together. The import language we have built attempts to make the transformation logic – including all of the assumptions embedded within it – as clear and explicit as possible.

There are two main contributions that we think are important. The first is the idea of making a language where we can express assumptions as a first-class language construct. Doing so not only helps document these decisions, but also opens opportunities to *use* this information in automated tooling. One example is in tracing, where we can display debug information in terms of failed assumption, which should help debugging by locating faults faster.

The second main contribution is in compile-time guarantees. Static type checking helps discover bugs before we even run the program. By including very fine-grained information in the type system, like the chain of foreign key relations (links) that leads to a particular value, we can perform an extensive number of compile-time checks. A central theme in Square is the presence of two (or more) operations to do the same thing: one that assumes nothing of the data, and which is possibly "unsafe" in the general case. And one which is "safe", as long as the programmer can prove its safety to the type checker. This particular approach is not something we have encountered in comparable systems, and it is one that has worked out well in our practical tests.

To evaluate the language, we looked at three case studies, where we designed an import definition and then tested the definition on real-world data sets. Testing a language based on a few examples is inherently limited, but should demonstrate the expressivity and usefulness of our work.

9.3 Future Work

From the start, we intentionally kept the scope of this thesis limited to fit the project schedule. The main concession we made was to focus only on *data importing*, i.e. unidirectional data migration between two different technical spaces. Another use case that was discussed early on was data model evolution, i.e. data migration from one version of a data model to the next. Adapting our current framework to support this feature would require a different kind of distillation phase, but otherwise could work with the same basic type system.

A fundamental principle of the current proposal is that we draw from the relational database schema to get integrity constraints on the source data. However, there may be more sources of constraints that we do not take into account. For example, the external party could use a constraint language like OCL, or perhaps there is a software system that checks validation rules before each change to the database. Further research could investigate how we can make better use of these kinds of sources of business rules.

One piece of “low-hanging fruit” is support for SQL check constraints, as discussed in section 3.1.2. Supporting this feature would require the ability to replicate the constraint expression in the distillation language, and then we would need to be able to verify that this constraint matches the one in the original schema exactly.

Chapter 10

Discussion

The subject of this thesis was very much motivated by practical, industrial concerns. Over the course of many software projects, with different characteristics, and different clients, time and time again the biggest source of runtime failures turned out to be the import chain. An importer is simple to write, but it turned out to be rather difficult to get it *robust*. The ideas that accumulated from this experience resulted in what would become Square. After the initial idea was sketched out, we experimented with an implementation, and through experimenting and testing we settled on the design presented here.

In hindsight, the lack of a theoretical foundation from the start made it more difficult to write this thesis report than need be. In a sense, we had to build the theory around the implementation, rather than the other way around. It also shows in the evaluation, which (being founded entirely on a limited set of case studies), was difficult to really nail down.

When we implemented the language, we created two artifacts: a language definition, and an engine. The language definition encompasses an abstract syntax, a description of a type system, semantic constraints, and a formal grammar. These are all fairly formal, exact descriptions. The engine on the other hand, was written in C++, and we made mostly ad-hoc design decisions to get a piece of working software.

This is reflected in the thesis report, where we have an elaborate description of the static semantics of the language, but where the operational semantics are underspecified. This is a weakness of our work, because a proper operational semantics would have allowed us to create a formal proof of correctness.

If we were to start over, with the knowledge and experience we have now, one thing that we would have done differently is start with a better theoretical basis. On the other hand, we think the focus on practicality and pragmatism makes the end result interesting. An adventage of this approach is that we our end result is significantly “battle-hardened”, by being constantly exposed to real-world challenges. We have confidence that the design we have created is useful to a large number of software projects, and hope to see Square (or any derivative design) adopted in the wider software development community.

Bibliography

[1] M. L. Markus, "Paradigm shifts – E-business and business/systems integration," *Communications of the Association for Information Systems*, vol. 4, no. 1, p. 10, 2000.

[2] M. A. Hernández and S. J. Stolfo, "Real-world data is dirty: Data cleansing and the merge/purge problem," *Data mining and knowledge discovery*, vol. 2, no. 1, pp. 9–37, 1998.

[3] S. Chaudhuri and U. Dayal, "An overview of data warehousing and OLAP technology," *ACM Sigmod record*, vol. 26, no. 1, pp. 65–74, 1997.

[4] P. Vassiliadis, "A survey of extract-transform-load technology," *International Journal of Data Warehousing and Mining (IJDWM)*, vol. 5, no. 3, pp. 1–27, 2009.

[5] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, pp. 1–17, USA, 2003.

[6] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.

[7] T. Mens, K. Czarnecki, P. Van Gorp, J. Bezivin, and R. Heckel, "A taxonomy of model transformations," *Language Engineering for Model-Driven Software Development*, vol. 4101, 2004.

[8] J. Bézivin, I. Kurtev, *et al.*, "Model-based technology integration with the technical space concept," in *Metainformatics Symposium*, vol. 20, pp. 44–49, 2005.

[9] I. Kurtev, J. Bézivin, and M. Akşit, "Technological spaces: An initial appraisal," 2002.

[10] J. Bézivin and O. Gerbé, "Towards a precise definition of the OMG/MDA framework," in *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pp. 273–280, IEEE, 2001.

[11] C. J. Date and H. Darwen, *A guide to the SQL Standard: a user's guide to the standard relational language SQL*, vol. 55822. Addison-Wesley Longman, 1993.

[12] P. Gulutzan and T. Pelzer, *SQL-99 complete, really.* CMP books, 1999.

[13] J. B. Warmer and A. G. Kleppe, "The Object Constraint Language: Precise modeling with UML (Addison-Wesley Object Technology Series)," 1998.

[14] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu, "XQuery 1.0: An XML query language," 2002.

[15] J. Clark *et al.*, "XSL transformations (XSLT)," *World Wide Web Consortium (W3C). URL http://www. w3. org/TR/xslt*, p. 103, 1999.

[16] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[17] E. F. Codd, *Relational completeness of data base sublanguages.* IBM Corporation, 1972.

[18] M. Y. Vardi, "The complexity of relational query languages," in *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pp. 137–146, ACM, 1982.

[19] L. Libkin, "Expressive power of SQL," in *Database Theory–ICDT 2001*, pp. 1–21, Springer, 2001.

[20] J. Cohen, "Constraint logic programming languages," *Communications of the ACM*, vol. 33, no. 7, pp. 52–68, 1990.

[21] T. Gardner, C. Griffin, J. Koehler, and R. Hauser, "A review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards the final Standard," in *MetaModelling for MDA Workshop*, vol. 13, p. 41, Citeseer, 2003.

[22] E. Visser, "Program Transformation with Stratego/XT," in *Domain-Specific Program Generation*, pp. 216–238, Springer, 2004.

[23] S. Sendall and W. Kozaczynski, "Model transformation the heart and soul of model-driven software development," tech. rep., 2003.

[24] C. Atkinson and T. Kühne, "Model-driven development: a metamodeling foundation," *Software, IEEE*, vol. 20, no. 5, pp. 36–41, 2003.

[25] D. C. Schmidt, "Model-driven engineering," *COMPUTER-IEEE COMPUTER SOCIETY-*, vol. 39, no. 2, p. 25, 2006.

[26] L. C. Kats and E. Visser, "The spoofax language workbench: rules for declarative specification of languages and IDEs," in *ACM Sigplan Notices*, vol. 45, pp. 444–463, ACM, 2010.

[27] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, *et al.*, "The state of the art in language workbenches," in *Software Language Engineering*, pp. 197–217, Springer, 2013.

[28] R. Y. Wang and D. M. Strong, “Beyond accuracy: What data quality means to data consumers,” *Journal of management information systems*, pp. 5–33, 1996.

[29] E. Rahm and H. H. Do, “Data cleaning: Problems and current approaches,” *IEEE Data Eng. Bull.*, vol. 23, no. 4, pp. 3–13, 2000.

[30] D. M. Strong, Y. W. Lee, and R. Y. Wang, “Data quality in context,” *Communications of the ACM*, vol. 40, no. 5, pp. 103–110, 1997.

[31] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos, “Conceptual modeling for ETL processes,” in *Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP*, pp. 14–21, ACM, 2002.

[32] J. Trujillo and S. Luján-Mora, “A UML based approach for modeling ETL processes in data warehouses,” in *Conceptual Modeling-ER 2003*, pp. 307–320, Springer, 2003.

[33] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos, “Modeling ETL activities as graphs.,” in *Design and Management of Data Warehouses (DMDW)*, vol. 58, pp. 52–61, 2002.

[34] D. Skoutas and A. Simitsis, “Designing ETL processes using semantic web technologies,” in *Proceedings of the 9th ACM international workshop on Data warehousing and OLAP*, pp. 67–74, ACM, 2006.

[35] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer, “Wrangler: Interactive visual specification of data transformation scripts,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 3363–3372, ACM, 2011.

[36] R. Verborgh and M. De Wilde, *Using OpenRefine*. Packt Publishing Ltd, 2013.

[37] T. Dasu and T. Johnson, *Exploratory data mining and data cleaning*, vol. 479. John Wiley & Sons, 2003.

[38] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, “Duplicate record detection: A survey,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 19, no. 1, pp. 1–16, 2007.

[39] A. E. Monge, “Matching algorithms within a duplicate detection system,” *IEEE Data Eng. Bull.*, vol. 23, no. 4, pp. 14–20, 2000.

[40] T. Steel Jr, “ANSI/X3/SPARC Study Group on Data Base Management Systems Interim Report,” *ACM SIGMOD FDT*, vol. 7, no. 2, 1975.

[41] M. West, *Developing high quality data models*. Elsevier, 2011.

[42] H. Geuvers, “Introduction to type theory,” in *Language Engineering and Rigorous Software Development*, pp. 1–56, Springer, 2009.

[43] F. Jouault, “Loosely coupled traceability for atl,” in *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany*, vol. 91, p. 2, Citeseer, 2005.

[44] K. Czarnecki and S. Helsen, “Feature-based survey of model transformation approaches,” *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.

[45] Õ. Ljungberg, “Measurement of overall equipment effectiveness as a basis for TPM activities,” *International Journal of Operations & Production Management*, vol. 18, no. 5, pp. 495–507, 1998.

[46] J.-M. Favre, “Foundations of model (driven)(reverse) engineering: Models–episode I: stories of the fidus papyrus and of the solarus,” in *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2005.

[47] J.-M. Favre, “Foundations of meta-pyramids: Languages vs. metamodels–episode II: Story of thotus the baboon1,” in *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2005.

[48] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati, “A Principled Approach to Data Integration and Reconciliation in Data Warehousing.,” in *DMDW*, vol. 99, p. 16, 1999.

[49] E. J. O’Neil, “Object/relational mapping 2008: hibernate and the entity data model (edm),” in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1351–1356, ACM, 2008.

[50] E. Meijer, B. Beckman, and G. Bierman, “LINQ: Reconciling object, relations and XML in the .NET framework,” in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pp. 706–706, ACM, 2006.

[51] G. Giorgidze, T. Grust, A. Ulrich, and J. Weijers, “Algebraic data types for language-integrated queries,” in *Proceedings of the 2013 workshop on Data driven functional programming*, pp. 5–10, ACM, 2013.

[52] J. Fong, F. Pang, and C. Bloor, “Converting relational database into xml document,” in *Database and Expert Systems Applications, 2001. Proceedings. 12th International Workshop on*, pp. 61–65, IEEE, 2001.

[53] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, “ATL: A model transformation tool,” *Science of computer programming*, vol. 72, no. 1, pp. 31–39, 2008.

[54] U. Norell, *Towards a practical programming language based on dependent type theory*, vol. 32. Citeseer, 2007.

Appendix A

Alan

A.1 Concepts

The most basic concept in Alan is the *type fragment* (or just “fragment”), denoted by parentheses. A type fragment specifies a set of named properties. Type fragments themselves are not named, but they may be nested within properties, thus building up a “schema tree”.

```
// Type fragment
(
    "description" :: text // Property
)
```

Alan offers the following property types:

• `text`	Textual value (a string).
• `integer`	Arbitrary integer value.
• `decimal`	Arbitrary decimal value.
• `option`	A choice out of several pre-defined options.
• `collection`	Mapping from keys (strings) to entries.
• `collection-of`	Collection where the key set is a subset of another collection.
• `reference`	Reference to a single entry in a collection.
• `component`	An instance of a named component type.

A.2 Primitives

Alan includes a few primitive types, which serve as the leaves of the tree. The basic types are `text`, `integer`, and `decimal`.

```
// Alan
(
  "description" :: text
  "version" :: integer
)
```

```
// Instance (JSON)
{
  "description": "Basic example",
  "version": 8
}
```

The snippet on the left shows the schema, and on the right is an example of a valid instance. Throughout this report we will use JSON as a serialization format for Alan instances.

A.3 Collections

A collection is a mapping from keys to entries. Each key is a string (which must be unique within the collection), and each entry is described by the specified entry fragment.

```
//...
"product types" :: collection (
  "name" :: text
)
```

```
//...
"product types": {
  "P-278": {
    "name": "Aluminium profile"
  },
  //...
}
```

Collections may be nested under other collections. This can be used to model a one-to-many relation between the parent and the child collection.

A.4 Options

An option type specifies a set of choices that may be selected from. Each choice is itself a type fragment. Options are used to model simple enumerated types (like booleans), but can also include complex type fragments, making it a useful tool to model conditional logic — in the sense that we can include properties that exist *only if* we are in a particular state.

```
//...
"product types" :: collection (
  "requires setup" :: option (
    "yes" :: ( )
    "no" :: ( )
  )
  "type" :: option (
    "intermediate" :: ( )
    "final" :: (
        "label" :: text
    )
  )
)
```

```
//...
"product types": {
  "278": {
    "requires setup": [ "no", {} ],
    "type": [ "final", {
      "label": "..."
    } ]
  }
}
```

A.5 References

References allow us to select a specific entry in a collection. A reference requires a relative path from the current location in the schema tree to the collection, denoted by angle brackets (<...>).

```
//...
"orders" :: collection (
  "product" :: reference <../"product types">
)
```

```
//...
"orders": {
  "ORDER-560": {
    "product": "P-278"
  }
}
```

We can also include a reference path on a collection, to create a “mapped collection” where the keys refer to entries in the referenced collection.

```
"products" :: collection (
  "operations" :: collection (
    "description" :: text
  )
)
"orders" :: collection (
  "product" :: reference <../"products">
  // Select a set of operations to be performed for this specific order
  "required operations" :: collection of <"product"/"operations"> (
    // Specify the priority for each operation
    "priority" :: integer
  )
)
```

```
{
  "products": {
    "P-278": {
      "operations": {
        "1": { "description": "Use machine M with materials X, Y, Z." }
      }
    }
  }
  "orders": {
    "ORDER-560": {
      "product": "P-278"
      "required operations" :: {
        "1": { "priority": 1 }
      }
    }
  }
}
```

In this fairly complicated example, we see that we can use mapped collections to *select* a particular subset of entries from another collection, and optionally *add* more information. This concept allows us to model a variety of structures, such as many-to-many relations.

A.6 Components

Components allow us to specify named types, which can then be shared in multiple locations. They can also be used to model recursive stuctures.

```
(
    "my list" :: component "list"
)

component "list" (
    "type" :: option (
        "nil" :: ()
        "cons" :: (
            "head" :: integer
            "tail" :: component "list"
        )
    )
)
```

Code Snippet 8: Linked list.

```
(
    "my tree" :: component "tree"
)

component "tree" (
    "children" :: collection (
        "child" :: component "tree"
    )
)
```

Code Snippet 9: Tree structure.

Infinitely recursive structures are technically allowed in Alan. However, any instance that conforms to such a model must itself be infinite, which means no (finite) data set will ever conform.

A.7 Conformance

We can define what it means for an instance to "conform" to an Alan model a little more precisely. As above, we will use JSON as the serialization format for instances. Conformance requires two things:

1. The instance should be *well-formed*, i.e. the structure should match the structure of the model.
2. Referential integrity should hold for properties which refer to other properties (references and collection-of).

Given (part of) an Alan model m, and a JSON value i, we recursively define the well-formedness function wf as:

```
$\mathrm{wf}(m, i) = \mathrm{true}$ iff one of the following holds:
    – $m$ is a fragment, and $i$ is a JSON object of the form
        {
            <n>: <v>
            such that $\mathrm{wf}(p, v)$ is true,
            for all properties $p$ with name $n$ in the fragment $m$
        }
    – $m$ is a text property, and $i$ is a JSON string
    – $m$ is an integer property, and $i$ is a JSON integer
        (i.e. a JSON number without a fractional part)
    – $m$ is a decimal property, and $i$ is a JSON number
    – $m$ is an option property, and $i$ is a JSON array of the form
        [ <name(c)>, <v> ]
        such that $\mathrm{wf}(\mathrm{choice\_fragment}(c), v)$ is true,
        for some choice $c$ in the option $m$
    – $m$ is a collection(-of) property, and $i$ is a JSON object of the form
        {
            <k>: <v>
            such that $\mathrm{wf}(\mathrm{entry\_fragment}(m), v)$ is true,
            for any number of distinct JSON keys $k$
        }
    – $m$ is a reference property, and $i$ is a JSON string
    – $m$ is a component property, and
        $\mathrm{wf}(\mathrm{component\_fragment}(m), i)$ is true
```

Here, the various $*\mathrm{fragment}$ functions refer to the subfragments defined as part of the property types.

To conform, an instance must be not just well-defined, but also must have referential integrity. Thus, for any model m and instance i, we define the function $\mathrm{conforms}$ as:

$\mathrm{conforms}(m, i) = \mathrm{true}$ iff $\mathrm{wf}(m, i)$ and all of the following hold:
– all reference properties in i must refer to a valid key in the referenced collection
– for all collection-of properties in i, each key must refer to a valid key in the referenced collection

Appendix B

Syntax

B.1 Lexical Syntax

The lexical syntax is shared across the languages in the remaining sections. The definitions below are given using regular expression syntax. Note that strings and identifiers share the same syntax. This is due to a limitation in the Alan-based parser we used for the project.

```
WHITESPACE ::= \s+|//.*\n
    String ::= ".*"
   Integer ::= -?[0-9]+
   Decimal ::= -?[0-9]+\.[0-9]+
        ID ::= ".*"
```

B.2 Alan

```
       Choice ::= ID "::" Fragment
PathComponent ::= ".." | ID
         Path ::= "<" [ "/" ] [ PathComponent { "/" PathComponent } ] ">"
     Property ::= ID "::" (
                      "text" | "integer" | "decimal"
                      | "option" "(" Choice { Choice } ")"
                      | "collection" Fragment
                      | "reference" Path
                      | "collection" "of" Path Fragment
                      | "component" ID
                  )
   Components ::= { "component" ID Fragment }
     Fragment ::= "(" { Property } ")"
       Schema ::= Fragment Components
```

B.3 Expression Sublanguage

The expression sublanguage is used by the construction and distillation languages. First, some basic definitions:

$$\begin{aligned}
\text{EnumSet} &::= \texttt{"\{"}\ [\ \text{String}\ \{\ \texttt{","}\ \text{String}\ \}\ ]\ \texttt{"\}"} \\
\text{PrimType} &::= \texttt{"text"} \mid \texttt{"integer"} \mid \texttt{"decimal"} \mid \texttt{"boolean"} \\
&\quad\mid \texttt{"date"} \mid \texttt{"datetime"} \mid \texttt{"enum"}\ \text{EnumSet} \\
\text{TextLit} &::= \text{String} \\
\text{IntegerLit} &::= \text{Integer} \\
\text{DecimalLit} &::= \text{Decimal} \\
\text{BooleanLit} &::= \texttt{"true"} \mid \texttt{"false"} \\
\text{Context} &::= \texttt{"."} \\
\text{ConstantRef} &::= \texttt{"\$"}\ \text{ID} \\
\text{Assumption} &::= \texttt{"<!"}\ [\ \text{String}\ ]\ \texttt{">"}
\end{aligned}$$

The following gives the most basic kinds of expressions, occurring at the leaves of expression trees.

$$\begin{aligned}
\text{AtomicExp} ::= &\ \texttt{"error"}\ \texttt{"::"}\ \text{PrimType} \\
&\mid\ \text{TextLit}\ \mid\ \text{IntegerLit}\ \mid\ \text{DecimalLit}\ \mid\ \text{BooleanLit} \\
&\mid\ \text{ConstantRef} \\
&\mid\ \text{Context} \\
&\mid\ \texttt{"table"}\ \text{ID}\ \mid\ \texttt{"external"}\ \text{ID}
\end{aligned}$$

Finally, the following defines the expression syntax, with operators given in order from lowest precedence to highest.

```
   ConstantDef ::= ID "=" Exp
ConstantsScope ::= "{" [ ConstantDef { "," ConstantDef } ] "}"
           Exp ::= Exp "where" ConstantsScope
                 | "ceil" Exp
                 | "floor" Exp
                 | "decimalize" Exp
                 | "not" Exp
                 | "count" Exp
                 | "single" Exp
                 | "concat" Exp "over" Exp "separated" "by" Exp
                 | ("min" | "max" | "avg" | "sum") Exp "over" Exp
                 | "if" Exp "then" Exp "else" Exp
                 | "substring" Exp "from" Exp "to" Exp
                 | "parse" Exp "::" PrimType
                 | "serialize" Exp
                 | "optional" Exp "on" Exp
                 | "root" Exp "following" ID
                 | "try" Exp "catch" Exp
                 | "settle" Exp "otherwise" Exp
                 | Exp "filter" Exp
                 | Exp "union" Exp
                 | Exp Assumption
                 | Exp "and" Exp
                 | Exp "or" Exp
                 | Exp ("==" | "!=" | "<" | ">") Exp
                 | Exp ("++") Exp
                 | Exp ("+" | "-") Exp
                 | Exp ("*" | "/") Exp
                 | Exp "[" ID "]"
                 | Exp ("->" ID | "^")
                 | AtomicExp
                 | "(" Exp ")"
```

B.4 Construction Language

The construction syntax is similar to that of Alan, the main addition being import operations. Note that we use the Exp nonterminal from the previous section.

```
       Choice ::= ID "::" Fragment
PathComponent ::= ".." | ID
         Path ::= "<" PathComponent { "/" PathComponent } ">"
     Property ::= ID "::" (
                       ( "text" | "integer" | "decimal" ) "<=" Exp
                       | "option" "<=" OptionExp
                       | "collection" "<=" CollectionExp
                       | "reference" Path "<=" ReferenceExp
                       | "collection" "of" Path "<=" CollectionOfExp
                       | "component" ID "<=" ComponentExp
                  )
   Components ::= { "component" ID Fragment }
     Fragment ::= [ "with" Exp ] "(" { Property } ")"
 Construction ::= Fragment Components
```

The import operations for the various types of properties are given by the following syntax:

```
     Assumption ::= "<!" [ String ] ">"
      OptionExp ::= "error" Assumption
                  | Choice
                  | "case" { Exp "=>" OptionExp } "otherwise" "=>" OptionExp
                  | "settle" ID "=" Exp "=>" OptionExp "none" "=>" OptionExp
                  | "try" ID "=" Exp "=>" OptionExp "catch" "=>" OptionExp
         KeyExp ::= "key" Assumption Exp
                  | "encode" "key" [ ID ]
  CollectionExp ::= "map" Exp ":" KeyExp "->" Fragment
   ReferenceExp ::= "key" Assumption Exp
                  | "index" Assumption Exp
                  | "entry" Exp
CollectionOfExp ::= "intersect" Assumption Exp
                  | "subset" Exp
   ComponentExp ::= Exp
```

B.5 Distillation Language

```
     EnumSet ::= "{" [ String { "," String } ] "}"
    PrimType ::= "text" | "integer" | "decimal" | "boolean"
               | "date" | "datetime" | "enum" EnumSet
      Column ::= "column" [ "*" ] ID "::" PrimType [ "!" ] [ "?" ] "<=" Exp
         Key ::= "key" ID ":" [ "!" ] "{" ID { "," ID } "}"
        Link ::= "link" ID ":" ( "single" | "many" ) ID [ "!" ] [ "?" ] "<=" (
                     "{" ID "<=" ID { "," ID "<=" ID } "}"
                     | "reverse" ID
                 )
 TableSchema ::= "table" ID
                 "<=" ( "map" | "aggregate" ) Exp
                 { Column }
                 { Key }
                 { Link }
Distillation ::= { TableSchema }
```

Appendix C

Typing Rules

C.1 Primitive Type Operators

$$\frac{\Gamma \vdash e_1,\, e_2 : \textbf{integer}}{\Gamma \vdash (e_1 \texttt{ + } e_2) : \textbf{integer}} \qquad \frac{\Gamma \vdash e_1,\, e_2 : \textbf{integer}}{\Gamma \vdash (e_1 \texttt{ - } e_2) : \textbf{integer}}$$

$$\frac{\Gamma \vdash e_1,\, e_2 : \textbf{integer}}{\Gamma \vdash (e_1 * e_2) : \textbf{integer}} \qquad \frac{\Gamma \vdash e_1,\, e_2 : \textbf{integer}}{\Gamma \vdash (e_1 \texttt{ / } e_2) : \textbf{unsafe}\langle\textbf{decimal}\rangle}\ (1)$$

$$\frac{\Gamma \vdash e_1,\, e_2 : \textbf{decimal}}{\Gamma \vdash (e_1 \texttt{ + } e_2) : \textbf{decimal}} \qquad \frac{\Gamma \vdash e_1,\, e_2 : \textbf{decimal}}{\Gamma \vdash (e_1 \texttt{ - } e_2) : \textbf{decimal}}$$

$$\frac{\Gamma \vdash e_1,\, e_2 : \textbf{decimal}}{\Gamma \vdash (e_1 * e_2) : \textbf{decimal}} \qquad \frac{\Gamma \vdash e_1,\, e_2 : \textbf{decimal}}{\Gamma \vdash (e_1 \texttt{ / } e_2) : \textbf{unsafe}\langle\textbf{decimal}\rangle}\ (1)$$

$$\frac{\Gamma \vdash e_1,\, e_2 : \textbf{text}}{\Gamma \vdash (e_1 \texttt{ ++ } e_2) : \textbf{text}} \qquad \frac{\Gamma \vdash e_1 : \textbf{text} \quad \Gamma \vdash e_2,\, e_3 : \textbf{integer}}{\Gamma \vdash (\texttt{substring } e_1 \texttt{ from } e_2 \texttt{ to } e_3) : \textbf{unsafe}\langle\textbf{text}\rangle}\ (2)$$

$$\frac{\Gamma \vdash e_1,\, e_2 : \textbf{boolean}}{\Gamma \vdash (e_1 \texttt{ and } e_2) : \textbf{boolean}} \qquad \frac{\Gamma \vdash e_1,\, e_2 : \textbf{boolean}}{\Gamma \vdash (e_1 \texttt{ or } e_2) : \textbf{boolean}}$$

$$\frac{\Gamma \vdash e : \textbf{boolean}}{\Gamma \vdash (\texttt{not } e) : \textbf{boolean}} \qquad \frac{\Gamma \vdash e_1 : \textbf{boolean} \quad \Gamma \vdash e_2,\, e_3 : \tau}{\Gamma \vdash (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) : \tau}$$

$$\frac{\Gamma \vdash e_1,\, e_2 : \tau \quad \tau \text{ is primitive}}{\Gamma \vdash (e_1 \texttt{ == } e_2) : \textbf{boolean}} \qquad \frac{\Gamma \vdash e_1,\, e_2 : \tau \quad \tau \text{ is primitive}}{\Gamma \vdash (e_1 \texttt{ != } e_2) : \textbf{boolean}}$$

$$\frac{\Gamma \vdash (e_1,\, e_2 : \textbf{integer} \lor e_1,\, e_2 : \textbf{decimal})}{\Gamma \vdash (e_1 \texttt{ < } e_2) : \textbf{boolean}} \qquad \frac{\Gamma \vdash (e_1,\, e_2 : \textbf{integer} \lor e_1,\, e_2 : \textbf{decimal})}{\Gamma \vdash (e_1 \texttt{ > } e_2) : \textbf{boolean}}$$

[1] Unsafe due to possibility of division by zero. Necessary because the type checker cannot differentiate between zero and non-zero denominator values.

[2] Unsafe due to possibility of an index being out of range.

C.2 Type Conversions

$$\frac{\Gamma \vdash e : \mathbf{text} \quad \tau \text{ is primitive}}{\Gamma \vdash (\texttt{parse}\ e :: \tau) : \mathbf{unsafe}\langle\tau\rangle} \qquad \frac{\Gamma \vdash e : \tau \quad \tau \text{ is primitive}}{\Gamma \vdash (\texttt{serialize}\ e) : \mathbf{text}}$$

[3]

$$\frac{\Gamma \vdash e : \mathbf{decimal}}{\Gamma \vdash (\texttt{floor}\ e) : \mathbf{integer}} \qquad \frac{\Gamma \vdash e : \mathbf{decimal}}{\Gamma \vdash (\texttt{ceil}\ e) : \mathbf{integer}}$$

$$\frac{\Gamma \vdash e : \mathbf{integer}}{\Gamma \vdash (\texttt{decimalize}\ e) : \mathbf{decimal}}$$

C.3 Optional and Unsafe

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma,\ \texttt{.} : \tau \vdash e_2 : \mathbf{boolean}}{\Gamma \vdash (\texttt{optional}\ e_1\ \texttt{on}\ e_2) : \mathbf{optional}\langle\tau\rangle} \qquad \frac{\Gamma \vdash e_1 : \mathbf{optional}\langle\tau\rangle \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\texttt{settle}\ e_1\ \texttt{otherwise}\ e_2) : \tau}$$

$$\frac{}{\Gamma \vdash (\texttt{error} :: \tau) : \mathbf{unsafe}\langle\tau\rangle} \qquad \frac{\Gamma \vdash e_1 : \mathbf{unsafe}\langle\tau\rangle \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\texttt{try}\ e_1\ \texttt{catch}\ e_2) : \tau}$$

$$\frac{e : \mathbf{unsafe}\langle\tau\rangle}{\Gamma \vdash (e\ \texttt{<!>}) : \tau}$$

C.4 Scoping

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash \texttt{\$}x : \tau} \qquad \frac{\texttt{.} : \tau \in \Gamma}{\Gamma \vdash \texttt{.} : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1,\ \ldots,\ e_n : \tau_n \quad \Gamma,\ x_1 : \tau_1,\ \ldots,\ x_n : \tau_n \vdash e_0 : \tau_0}{\Gamma \vdash (e_0\ \texttt{where}\ \{\ x_1 = e_1,\ \ldots,\ x_n = e_n\ \}) : \tau_0}$$

C.5 Row Set Operators

$$\frac{t \text{ is a table in the source schema}}{\Gamma \vdash \texttt{table}\ t : \mathbf{rowset}[\twoheadrightarrow t\,]} \qquad \frac{t \text{ is a table in the raw schema}}{\Gamma \vdash \texttt{external}\ t : \mathbf{rowset}[\twoheadrightarrow t\,]}$$

[3] Unsafe due to possibility of a parse error.

$$\frac{\Gamma \vdash e_1 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad \Gamma \vdash e_2 : \mathbf{rowset}[\,\vec{q}, \twoheadrightarrow s\,] \quad \mathrm{columns}(t) = \mathrm{columns}(s)}{\Gamma \vdash e_1 \ \texttt{union}\ e_2 : \mathbf{rowset}[\,\twoheadrightarrow u\,]}$$

$$\text{where } u = (\,\mathrm{columns}(t),\ \mathrm{links}(t) \cap \mathrm{links}(s),\ \varnothing\,) \in D_{\mathrm{tables}}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad \Gamma,\ \textbf{.} : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \vdash e_2 : \mathbf{boolean}}{\Gamma \vdash e_1 \ \texttt{filter}\ e_2 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t, \twoheadrightarrow t\,]}$$ [4]

$$\frac{\Gamma \vdash e : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \quad (\,c,\ \tau\,) \in \mathrm{columns}(t)}{\Gamma \vdash e\texttt{[}c\texttt{]} : \tau}$$

$$\frac{\Gamma \vdash e : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \quad (\,l,\ (\,s,\ \mathrm{single},\ \tau\,)\,) \in \mathrm{links}(t)}{\Gamma \vdash e \ \texttt{->}\ l : \tau\langle \mathbf{rowset}[\,\vec{p}, \rightarrow t, \rightarrow s\,]\rangle}$$

$$\frac{\Gamma \vdash e : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \quad (\,l,\ (\,s,\ \mathrm{many},\ \tau\,)\,) \in \mathrm{links}(t)}{\Gamma \vdash e \ \texttt{->}\ l : \tau\langle \mathbf{rowset}[\,\vec{p}, \rightarrow t, \twoheadrightarrow s\,]\rangle}$$

$$\frac{\Gamma \vdash e : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,]}{\Gamma \vdash e\ \texttt{\^{}} : \mathbf{rowset}[\,\vec{p}\,]}$$

C.6 Aggregation Operators

$$\frac{\Gamma \vdash e_0 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,]}{\Gamma \vdash (\texttt{count}\ e_0) : \mathbf{integer}}$$

$$\frac{\Gamma \vdash e_0 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,]}{\Gamma \vdash (\texttt{single}\ e_0) : \mathbf{unsafe}\langle \mathbf{rowset}[\,\vec{p}, \rightarrow t\,]\rangle}$$

$$\frac{\Gamma \vdash e_0 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad \Gamma,\ \textbf{.} : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \vdash e_1 : \tau}{\Gamma \vdash (\texttt{shared}\ e_1\ \texttt{over}\ e_0) : \mathbf{unsafe}\langle \tau \rangle}$$ [5]

[4] Note that we treat a filter as a kind of "link" to itself, to represent the fact that we have a subset. Hence the addition of *t* to the link chain in the result.

[5] Unsafe, only succeeds if all values are equal.

$$\frac{\Gamma \vdash e_0 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad \Gamma, \textbf{.} : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \vdash e_1 : \mathbf{integer}}{\Gamma \vdash (\texttt{min}\ e_1\ \texttt{over}\ e_0) : \mathbf{unsafe}\langle\mathbf{integer}\rangle}\ {}^{(6)}$$

$$\frac{\Gamma \vdash e_0 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad \Gamma, \textbf{.} : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \vdash e_1 : \mathbf{integer}}{\Gamma \vdash (\texttt{max}\ e_1\ \texttt{over}\ e_0) : \mathbf{unsafe}\langle\mathbf{integer}\rangle}\ {}^{(6)}$$

$$\frac{\Gamma \vdash e_0 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad \Gamma, \textbf{.} : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \vdash e_1 : \mathbf{integer}}{\Gamma \vdash (\texttt{avg}\ e_1\ \texttt{over}\ e_0) : \mathbf{unsafe}\langle\mathbf{decimal}\rangle}\ {}^{(6)}$$

$$\frac{\Gamma \vdash e_0 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad \Gamma, \textbf{.} : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \vdash e_1 : \mathbf{integer}}{\Gamma \vdash (\texttt{sum}\ e_1\ \texttt{over}\ e_0) : \mathbf{integer}}$$

$$\frac{\Gamma \vdash e_0 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad \Gamma, \textbf{.} : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \vdash e_1 : \mathbf{decimal}}{\Gamma \vdash (\texttt{min}\ e_1\ \texttt{over}\ e_0) : \mathbf{unsafe}\langle\mathbf{decimal}\rangle}\ {}^{(6)}$$

$$\frac{\Gamma \vdash e_0 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad \Gamma, \textbf{.} : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \vdash e_1 : \mathbf{decimal}}{\Gamma \vdash (\texttt{max}\ e_1\ \texttt{over}\ e_0) : \mathbf{unsafe}\langle\mathbf{decimal}\rangle}\ {}^{(6)}$$

$$\frac{\Gamma \vdash e_0 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad \Gamma, \textbf{.} : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \vdash e_1 : \mathbf{decimal}}{\Gamma \vdash (\texttt{avg}\ e_1\ \texttt{over}\ e_0) : \mathbf{unsafe}\langle\mathbf{decimal}\rangle}\ {}^{(6)}$$

$$\frac{\Gamma \vdash e_0 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad \Gamma, \textbf{.} : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \vdash e_1 : \mathbf{decimal}}{\Gamma \vdash (\texttt{sum}\ e_1\ \texttt{over}\ e_0) : \mathbf{decimal}}$$

$$\frac{\Gamma \vdash e_0 : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad \Gamma, \textbf{.} : \mathbf{rowset}[\,\vec{p}, \rightarrow t\,] \vdash e_1 : \mathbf{integer} \quad \Gamma \vdash e_2 : \mathbf{text}}{\Gamma \vdash (\texttt{concat}\ e_1\ \texttt{over}\ e_0\ \texttt{separated by}\ e_2) : \mathbf{text}}$$

C.7 Graph Operators

$$\frac{\Gamma \vdash e : \mathbf{rowset}[\,\vec{p}, \twoheadrightarrow t\,] \quad l \in \mathrm{links}(t)}{\Gamma \vdash (\texttt{root}\ e\ \texttt{following}\ l) : \mathbf{rowset}[\,\twoheadrightarrow t, \rightarrow t\,]}\ {}^{(7)}$$

[6] Unsafe due to the possibility of an empty set as input.
[7] Fails if the table is not a valid tree encoding.

Appendix D

Case Studies

This appendix lists the full import definitions for the three case studies performed as part of this thesis. These import definitions were developed for actual projects currently in production. The company names have been anonymized, and identifiers have been translated to English.

D.1 Alpha

D.1.1 Distillation

```
table "machines" <= map external "A.MACHINE"
    column * "machine id" :: text <= .["MACH#"]
    column "description" :: text <= .["MACHDSC"]

table "tools" <= map external "A.TOOL"
    column * "tool id" :: text <= .["TOOL#"]
    column "description" :: text <= .["TOOLDSC"]

// Table which associates tools with tool sets
table "tools x tool sets" <= map external "A.TOOLSET"
    column * "tool id" :: text <= .["TOOL#"]
    column * "tool set id" :: text <= .["TOOLSET"]
    link "tool": single "tools"
        <= { "tool id" <= .["tool id"] }
    link "tool set": single "tool sets"
        <= { "tool set id" <= .["tool set id"] }

// Aggregate all individual tool sets from the "tools x tool sets" table
table "tool sets" <= aggregate table "tools x tools set"
    column "tool set id" * <= .["tool set id"]
    link "tool entries": many "tools x tool sets"
        <= reverse "tool set"

// Product specifications
table "products" <= map external "A.PRODUCT"
```

```
    column * "product id" :: text <= .["PROD#"]
    column "description" :: text <= .["PRODDSC"]
    column "type" :: text <= .["PRODTYP"]
    link "operations": many "product operations"
        <= reverse "product"

// Operations that should be performed to make a particular product
table "product operations" <= map external "A.ROUTE"
    column * "product id" :: text <= .["PROD#"]
    column * "operation id" :: text <= .["RTOP"]
    column "description" :: text <= .["RTOPDSC"]
    column "standard time" :: float <= .["RTNORM"]
    column "machine id" :: text ? <= optional .["MACH#"] on (. != "")
    // This column ambiguously refers to either:
    // - no tool (empty string)
    // - a single tool (reference to the tools table)
    // - a tool set (reference to the tool set table)
    column "tool or tool set id" :: text ? <= optional .["TOOL#"] on (. != "")

    link "product": single "products"
        <= { "product id" <= .["product id"] }
    // Note: this link is optional (because the machine ID may be empty), but it is
    // also unsafe (because there is no foreign key constraint on the external table).
    link "machine": single "machines" ! ?
        <= { "machine id" <= .["machine id"] }
    // Create links for both possibilities of "tool or tool set id"
    link "tool": single "tools" ! ?
        <= { "tool id" <= .["tool or tool set id"] }
    link "tool set": single "tool set" ! ?
        <= { "tool set id" <= .["tool or tool set id"] }

// Orders for an instance of a particular product
table "orders" <= map external "A.PRPO"
    column * "order id" :: text <= .["WPONR#"]
    column "product id" :: text <= .["PROD#"]
    link "product": single "products"
        <= { "product id" <= .["product id"] }
    link "order operations": many "order operations"
        <= reverse "order"

// Operations belonging to an order
table "order operations" <= map external "A.WPOROUTE"
    column * "order id" :: text <= .["WPONR#"]
    column * "operation id" :: text <= .["WPRBEW"]
    column "description" :: text <= .["WPRBWD"]
    column "machine id" :: text ? <= optional .["WPRMCH"] on (. != "")
    column "quantity" :: text <= .["WPRQNT"]
    column "week number" :: text <= .["WPRBWK"]
    column "status" :: text <= .["WPRAFG"]

    link "order": single "orders"
        <= { "order id" <= .["order id"] }
    // The corresponding product operation that this operation is based on
    link "product operation": single "product operations"
        <= {
            "product id" <= (. -> "order")["product id"],
            "operation id" <= .["operation id"]
        }
    link "machine": single "machines" ! ?
        <= { "machine id" <= .["machine id"] }
```

D.1.2 Construction

```
(
    "machines" :: collection <= map table "machines": encode key -> (
        "description" :: text <= .["description"]
    )
    "tools" :: collection <= map table "tools": encode key -> (
        "description" :: text <= .["description"]
    )
    "tool sets" :: collection <= map table "tool sets": encode key -> (
        "tools" :: collection of <../"tools"> <= subset . -> "tool entries" -> "tool"
    )
    "products" :: collection <= map table "products": encode key -> (
        "description" :: text <= .["description"]
        "type" :: option <= case
            (.["type"] == "06") => "material" :: ()
            (.["type"] == "09") => "intermediate" :: ()
            (.["type"] == "15") => "end product" :: ()
            otherwise => "unknown" :: ()
        "operations" :: collection <= map . -> "operations": encode key -> (
            "description" :: text <= .["description"]
            // Switch on the optionality of the "default machine" link.
            "has default machine" :: option <= settle "machine" = . -> "machine"
                => "yes" :: (
                    "machine" :: reference <../../../"machines"> <= index $"machine"
                        <!"Reference to machine should be valid">
                )
                none => "no" :: ()
            // Try both of the "tool"/"tool set" links.
            "uses tools" :: option <= settle "unresolved tool set" = . -> "tool set"
                => try "tool set" = "unresolved tool set"
                    => "set" :: (
                        "tool set" :: reference <../../../"tool sets"> <= entry $"tool set"
                    )
                    catch => settle "tool" = . -> "tool"
                        => "single" :: (
                            "tool" :: reference <../../../"tools"> <= index $"tool"
                                <!"Reference to tool should be valid">
                        )
                        none => "none" :: ()
                none => "none" :: ()
        )
    )
    "orders" :: collection <= map table "orders": encode key -> (
        "product" :: reference <../"products"> <= entry . -> "product"
        "status" :: option <= case
            (.["status"] == "X") => "finished" :: ()
            otherwise => "pending" :: ()

        // Note: here we have two row sets which are both subsets of the
        // "product operations" table. We assume that the order operations
        // form a subset (i.e. the product should match between the two).
        "operations" :: collection of <"product"/"operations"> <= intersect
            <!"Order operations should form a subset of the product’s operations">
            . -> "order operations" -> "operation":
                encode key <!"References to product operations should be distinct">
                -> with .^ (
                    "description" :: text <= .["description"]
                    "quantity" :: integer <= parse .["quantity"] :: integer
                        <!"Quantity should be integer">
                    "week number" :: integer <= parse .["week number"] :: integer
                        <!"Week number should be integer">
                    "machine" :: option <=
                        settle "machine" = . -> "machine"
                            => "override" :: (
                                "machine" :: reference <../../../"machines">
                                    <= index $"machine"
                                        <!"Reference to machine should be valid">
                            )
                            none => "default" :: ()
                )
    )
)
```

D.2 Beta

D.2.1 Distillation

```
table "articles" <= map external "B.VIAAREP"
    column * "article id" :: text <= .["AAAATX"]
    column "article type" :: text <= .["AADATX"]
    column "description" :: text <= .["AAEVTX"]
    column "customer id" :: text <= .["AGAKCD"]
    link "customer": single "customers"
        <= { "customer id" <= .["customer id"] }
    link "supplement": single "articles supplement" !
        <= { "article id" <= .["article id"] }

// A table of additional article information, which is (ideally strictly)
// one-to-one with the "articles" table
table "articles extra" <= map external "B.LAST_ART_OV3"
    column * "article id" :: text <= .["AGAECD"]
    column "quantity per package" :: integer <= .["AGANNB"]

table "machine groups" <= map external "B.VIBSREP"
    column * "machine group id" :: text <= .["BS06CD"]
    column "description" :: text <= .["BS04CD"]
    column "department" :: text <= .["BSBHKO"]

table "tools" <= map external "B.VNDOREP"
    column * "tool id" :: text <= .["DOKQKD"]
    column "profile id" :: text <= .["DOKRKD"]
    // The first character of the tool ID denotes the type
    column "type" :: text <= substring .["DOKQKD"] from 0 to 1
    column "status" :: text <= .["DOTPST"]
    link "profile": single "profiles"
        <= { "profile id" <= .["profile id"] }

table "employees" <= map external "B.VIB7REP"
        filter (.["B7AWCD"] == "BETA") // Only employees working for Beta
    column * "employee id" :: text <= .["B7UQCD"]
    column "name" :: text <= .["B7URCD"]
    column "supervisor id" :: text ? <= .["B7E2KE"]
    column "access card id" :: text <= .["B7CBCO"]
    // Access cards are only distributed once (and thus are unique)
    key "access card": { "access card id" }
    link "supervisor": single "employees" ?
        <= { "employee id" <= .["supervisor id"] }

table "suppliers" <= map external "B.VICGREP"
    column * "supplier id" :: text <= .["CGWBCD"]
    column "name" :: text <= .["CGWDCD"]

table "materials" <= map table "articles"
        filter (.["article type"] == "material") // Filter out just the materials
    column * "material id" :: text <= .["AAAATX"]
    column "description" :: text <= .["AAEVTX"]
    column "type" :: text <= .["AADATX"]

table "material shipments" <= map external "B.VIAJREP"
    column * "shipment date" :: date <= .["AJALDT"]
    column * "sequence number" :: text <= .["AJBANB"]
    column "supplier id" :: text <= .["CHWBCD"]
    column "material id" :: text <= .["AJAATX"]
    column "quantity" :: integer <= .["AJCSTX"]
    link "supplier": single "suppliers"
        <= { "supplier id" <= .["supplier id"] }
    link "material": single "materials"
        <= { "material id" <= .["material id"] }

table "customers" <= map external "B.VIBQREP"
    column * "customer id" :: text <= .["BQIMNB"]
    column "name" :: text <= .["BQMICD"]
    link "customer group entry": single "customers with group" !
        <= { "customer id" <= .["customer id"] }

// Same primary key as the "customers" table, but here it lists the customer group.
// Note that not all customers are necessarily present in this table.
table "customers with group" <= map external "B.VIBHREP"
    column * "customer id" :: text <= .["BHIMNB"]
    column "customer group id" :: text <= .["BHS0NA"]
    link "customer": single "customers" !
        <= { "customer id" <= .["customer id"] }
    link "customer group": single "customer groups"
        <= { "customer group id" <= .["customer group id"] }

// Aggregate just the customer groups from the "customers with group" table
table "customer groups" <= aggregate table "customers with group"
    column * "customer group id" :: text <= .["customer group id"]
    link "group entries": many "customers with group"
        <= reverse "customer group"

table "operations" <= map external "B.VIBRREP"
    column * "operation code" :: text <= .["BRNMNB"]
    column "description" :: text <= .["BRO3CD"]

table "profiles" <= map table "articles"
        filter (.["article type"] == "profile") // Filter out just the profiles
    column * "profile id" :: text <= .["article id"]
    // The first 4 characters of the profile ID indicates its "profile type"
    column "type" :: text <= substring .["profile id"] from 0 to 4
    link "article": single "articles"
        <= { "article id" <= "profile id" }
    link "profile type": single "profile types"
        <= { "profile type id" <= .["type"] }
    link "operations": many "profile operations"
        <= reverse "profile"

// Aggregate just the profile types
table "profile types" <= aggregate table "profiles"
    column * "profile type id" <= .["type"]
    link "profiles": many "profiles"
        <= reverse "profile type"

table "profile operations" <= map external "B.VIB2REP"
    column * "profile id" :: text <= .["B2AATX"]
    column * "sequence number" :: text <= .["B2TYNB"]
    column "operation id" :: text <= .["B2NMNB"]
    column "machine group id" :: text <= .["B2O6CD"]
    column "description" :: text <= .["B2QZKD"]
    column "next sequence number" :: text ? <= .["B2STNR"]
    link "profile": single "profiles"
        <= { "profile id" <= .["profile id"] }
    link "operation": single "operations"
        <= { "operation code" <= .["operation id"] }
    link "machine group id": single "machine groups"
        <= { "machine group id" <= .["machine group id"] }
    link "next operation": single "profile operations" ?
        <= {
            "profile id" <= .["profile id"],
            "sequence number" <= .["next sequence number"]
        }

table "orders" <= map external "B.VICEREP"
    column * "order id" :: integer <= .["CEV0NB"]
    column "profile id" :: text <= .["CEAATX"]
    column "status" :: text <= .["CEHPST"]
    column "start date" :: date <= .["CEBMDT"]
    link "profile": single "profiles"
        <= { "profile id" <= .["profile id"] }
```

D.2.2 Construction

```
(
    // Assets
```

```
"machine groups" :: collection <= map table "machine groups": encode key -> (
    "description" :: text <= .["description"]
    "department" :: text <= .["department"]
)
"tools" :: collection <= map table "tools": encode key -> (
    "profile" :: reference <../"profiles"> <= . -> "profile"
    "type" :: option <= case
        (.["type"] == "1") => "stamp" :: ()
        (.["type"] == "2") => "roll form set" :: ()
        (.["type"] == "5") => "stencil" :: ()
        otherwise => "other" :: ()
)
"employees" :: collection <= map table "employees": encode key -> (
    "name" :: text <= .["name"]
    "has supervisor" :: option <= settle "supervisor" = . -> "supervisor"
        => "yes" :: (
            "supervisor" :: reference <../../"employees"> <= $"supervisor"
        )
        none => "no" :: ()
    )
)
// Because there is a 1:1 correspondence between employees and access cards,
// we can simply map over the employees table here.
"access cards" :: collection <= map table "employees": encode key "access card" -> (
    "employee" :: reference <../"employees"> <= .
)

// Supply
"suppliers" :: collection <= map table "suppliers": encode key -> (
    "name" :: text <= .["name"]
)
"materials" :: collection <= map table "materials": encode key -> (
    "description" :: text <= .["description"]
    "type" :: text <= .["type"]
)
"material shipments" :: collection from table "supply orders": encode key -> (
    "supplier" :: reference <../"suppliers"> <= . -> "supplier"
    "material" :: reference <../"materials"> <= . -> "material"
    "quantity" :: integer <= .["quantity"]
    "shipment date" :: date <= .["shipment date"]
)

// Demand
"customer groups" :: collection <= map table "customer groups": encode key -> ()
"customers" :: collection <= map table "customers": encode key -> (
    "name" :: text <= .["name"]
    "group" :: reference <../"customer groups"> <= entry
```

```
            (. -> "customer group entry" <!"There should be a corresponding group">)
                -> "customer group"
    )

    // Production
    "operations" :: collection <= map table "operations": encode key -> (
        "description" :: text <= .["description"]
    )
    "profile types" :: collection <= map table "profile types": encode key -> (
        "customer" :: reference <../"customers"> <= entry
            shared . -> "profiles" -> "article" -> "customer"
                <!"All profiles falling under a profile type should share
                    the same customer">
        // Instances of this profile type
        "profiles" :: collection <= map . -> "profiles": encode key -> (
            "description" :: text <= .["description"]
            "quantity per package" :: integer <= (
                    . -> "article" -> "supplement"
                    <!"Should be supplemental information for each article">
                )["quantity per package"]
            "operations" :: component "profile operation list"
                <= root . -> "operations"
                    <!"Operations list should have a unique root">
        )
    )
    "orders" :: collection from table "orders": encode key -> (
        "profile type" :: reference <../"profile types">
            <= entry . -> "profile" -> "profile type"
        "profile" :: reference <"profile type"/"profile variants">
            <= entry . -> "profile"
        "status" :: option <= case
            (.["status"] == "P") => "pending" :: (
                "scheduled date" :: date <= .["start date"]
            )
            (.["status"] == "D") => "done" :: ()
            otherwise => error <!"Status should be one of: pending, done">
    )
)

component "profile operation list" (
    "machine group" :: reference </"machine groups">
        <= entry . -> "machine group"
    "operation" :: reference </"operations">
        <= entry . -> "operation"
    "has tail" :: option <= settle "next operation" = . -> "next operation"
        => "yes" :: (
            "tail" :: component "profile operation list" <= $"next operation"
```

```
                <!"Non-final operation should have exactly one successor">
        )
        none => "no" :: ()
    )
)
```

D.3 Gamma

D.3.1 Distillation

```
table "employees" <= map external "C.EIADREP"
    column * "employee id" :: text <= .["ADACCD"]
    column "full name" :: text <= .["ADABTX"]

table "customers" <= map external "C.EIA4REP"
    column * "customer id" :: text <= .["A4A2NC"]
    column "short name" :: text <= .["A4B4CD"]
    column "full name" :: text <= .["A4B9NA"]
    column "responsible employee id" :: text <= .["A4ACCD"]
    column "branch code" :: text <= .["A4BCCD"]
    column "sales area" :: text <= .["A4IDST"]

    // Note: short name should be unique, although the external schema
    // does not guarantee it (thus, this definition is marked unsafe)
    key "short name": ! { "short name" }

    link "sales representative": single "employees"
        <= { "employee id" <= .["responsible employee id"] }
    link "locations": many "customer locations"
        <= reverse "customer"
    link "articles": many "articles"
        <= reverse "customer"

table "customer locations" <= map external "C.EIA7REP"
    column * "customer" :: text <= .["A7A2NC"]
    column * "location number" :: text <= .["A7AKNC"]
    column "country" :: text <= .["A7A8CD"]
    column "address type" :: text <= .["A7BEST"]
    column "name" :: text <= .["A7BHNA"]
    column "address" :: text <= .["A7H8NA"]
    link "customer": single "customers"
        <= { "customer id" <= .["customer id"] }

table "machines" <= map external "C.EIA6REP"
    column * "operation type" :: text <= .["A6ARCD"]
    // Machine number (unique within a particular operation type)
    column * "machine number" :: text <= .["A6BSCD"]
    column "description" :: text <= .["A6BLNA"]
    link "operation type": single "operation types"
        <= { "operation type" <= .["operation type"] }

table "treatments" <= map external "C.EIBAREP"
    column * "operation type" :: text <= .["BAARCD"]
    column * "treatment id" :: text <= .["BABUCD"]
    column "description" :: text <= .["BABQNA"]
    link "operation type": single "operation types"
        <= { "operation type" <= .["operation type"] }

table "operation types" <= map external "C.EIAKREP"
    column * "operation type" :: text <= .["AKARCD"]
    link "machines": many "machines"
        <= reverse "operation type"
    link "treatments": many "treatments"
        <= reverse "operation type"

table "product groups" <= map external "C.EIFNREP"
    column * "product group id" :: text <= .["FNAEST"]
    column "type" :: text <= .["FNJCCO"]

table "articles" <= map external "C.EIIQREP"
    // Note: there"s a "special" customer ID ("03408") which is used for stock
    // products rather than being a real customer
    column * "customer id" :: text <= .["IQA2NC"]
    column * "article number" :: text <= .["IQPJKD"]
    column "alloy code" :: text <= .["IQAZCD"]
    column "customer product" :: text <= .["IQLQNA"]
    column "negative length tolerance" :: decimal <= .["IQP6PQ"]
    column "positive length tolerance" :: decimal <= .["IQP7PQ"]
    column "product group id" :: text <= .["IQPLKD"]
    column "finished product length" :: decimal <= .["IQQKPQ"]
    column "status" :: text <= .["IQRASS"]
    column "description" :: text <= .["IQTTNA"]
    link "product group": single "product groups" !
        <= { "product group id" <= .["product group id"] }
    link "customer": single "customers"
        <= { "customer id" <= .["customer id"] }
    link "production alternatives": many "production alternatives"
        <= reverse "article"

// Note: one-to-one with "articles" once we fix this on a particular location
// (this is evident by comparing the primary keys of the two tables)
table "production alternatives" <= map external "C.EIISREP"
    column * "customer id" :: text <= .["ISA2NC"]
    column * "customer article number" :: text <= .["ISPJKD"]
    column * "location" :: text <= .["ISMYKD"]
    column "production article number" :: text <= .["ISPKKD"]
    // There is an alternative key, which uses the production article
    // number instead
    key "alternative by production article":
        {
            "customer id",
            "location",
            "production article number"
        }
    link "article": single "articles"
        <= {
            "customer id" <= .["customer id"],
            "article number" <= .["customer article number"]
        }
    link "production article": single "production articles"
        <= {
            "location" <= .["location"]
            "article number" <= .["production article number"]
        }

// Articles may be produced in different locations, this table provides
// production information for articles at specific locations
table "production articles" <= map external "C.EIIRREP"
    column * "location" :: text <= .["IRMYKD"]
    column * "article number" :: text <= .["IRPKKD"]
    column "surface quality code" :: text <= .["IRCNCD"]
    column "quantity in stock" :: integer <= .["IRCSQT"]
    column "anodizing tooling code" :: text <= .["IRPQKD"]
    column "operation sequence" :: text <= .["IRRBSS"]
    column "material source" :: text <= substring .["IRRBSS"] from 0 to 1

table "articles in hungary" <= map table "production alternatives"
        filter .["location"] == "HUN" // Filter just the articles produced in Hungary
    column * "customer id" :: text <= .["customer id"]
    column * "customer article number" :: text <= .["customer article number"]
    column * "location" :: text <= .["location"]
    column "quantity in stock" :: integer
        <= (. -> "production article")["quantity in stock"]
    column "material source" :: text
        <= (. -> "production article")["material source"]
    link "article": single "articles"
        <= {
            "customer id" <= .["customer id"],
            "article number" <= .["customer article number"]
        }

table "areas" <= map external "C.EIAQREP"
    column * "location" :: text <= .["AQMYKD"]
    column * "area" :: text <= .["AQA6CD"]
    link "location": single "locations"
        <= { "location" <= .["location"] }
```

```
table "locations" <= aggregate table "areas"
    column * "location" :: text <= .["location"]
    link "areas": many "areas":
        reverse "location"

table "orders" <= map external "C.EIBEREP"
    column * "order id" :: text <= .["BEA0NC"]
    column "customer id" :: text <= .["BEA2NC"]
    column "customer article number" :: text <= .["BEPJKD"]
    column "phase" :: text <= .["BEILST"]
    column "material source" :: substring .["BECAST"] from 0 to 1
    column "requested quantity" :: integer <= .["B2G3PN"]
    column "realized quantity" :: integer <= .["B2ZVPN"]
    column "delivery date" :: date <= .["KEILDT"]
    link "customer": single "customers"
        <= { "customer id" <= .["customer id"] }
    link "article": single "articles"
        <= {
            "customer id" <= .["customer id"],
            "article number" <= .["customer article number"]
        }
    link "order phase": single "order phases"
        <= { "phase" <= .["phase"] }

table "order phases" <= aggregate table "orders"
    column * "phase" :: text <= .["BEILST"]

table "order operations" <= map external "C.EIBPREP"
    column * "order id" :: text <= .["BPA0NC"]
    column * "operation type" :: text <= .["BPARCD"]
    column "customer id" :: text <= .["BPA2NC"]
    column "machine number" :: text <= .["BPBSCD"]
    column "latest possible starting date" :: date <= .["BPCFDT"]
    column "earliest possible starting date" :: date <= .["BPCHDT"]
    link "order": single "orders"
        <= { "order id" <= .["order id"] }
    link "operation type": single "operation types"
        <= { "operation type" <= .["operation type"] }
```

D.3.2 Construction

```
(
    // Demand

    "customers" :: collection <= map table "customers": encode key "short name"
        -> with . <!"Short name should be unique"> (
            "articles" :: collection <= map . -> "articles": encode key -> (
                "description" :: text <= .["description"]
            )
            "branch code" :: text <= .["branch code"]
            "sales area" :: text <= .["sales area"]
            // Note: this is not a reference, because we only need the name
            "sales representative" :: text <= (. -> "sales representative")["full name"]
            "shipping locations" :: collection <= map . -> "locations": encode key -> (
                "country" :: text <= .["country"]
                "address" :: text <= .["address"]
            )
        )

    // Product line

    "operation types" :: collection <= map table "operation types": encode key -> (
        // Machines used as part of operation
        "machines" :: collection <= map . -> "machines": encode key -> (
            "machine number" :: text <= .["machine number"]
            "description" :: text <= .["description"]
            "is press" :: option <= case
                (.["operation type"] == "P") => "yes" :: ()
                otherwise => "no" :: ()
        )
        // Treatments used as part of this operation
        "treatments" :: collection <= map . -> "treatments": encode key -> (
            "is repacking" :: option <= case
                (.["operation type"] == "U") => "yes" :: (
                    "description" :: text <= .["description"]
                )
                otherwise => "no" :: ()
        )
    )
    "product groups" :: collection <= map table "product groups": encode key -> (
        "type" :: text <= .["type"]
    )
    "article types" :: collection <= map table "articles in hungary": encode key -> (
        "product group" :: reference <../"product groups">
            <= index (. -> "product group")
                   <!"Each article should belong to a product group">
        "quantity in stock" :: integer <= .["quantity in stock"]
        "type" :: option <= case
            (.["customer id"] == "03408") => "stock article" :: ()
            otherwise => "customer article" :: (
                "customer" :: reference <../../"customers">
                "customer article" :: reference <"customer"/"articles">
            )
        // The type of source material for this article
```

```
        "material source" :: option <= case
            (.["material source"] == "R") => "stock" :: ()
            (.["material source"] == "P") => "extrusion" :: (
                // The operation of extrusion is always a press operation
                "operation type" :: reference <../../"operation types"> <= key "P"
                    <!"There should be an operation type with the key ’P’,
                        representing press operations">
            )
            otherwise => error <!"Unknown material source type">
        // The operation types to produce this article
        "operations" :: collection <= map . -> "operations": encode key -> (
            "operation type" :: reference <../../"operation types">
                <= entry . -> "operation"
            "type" :: option <= case
                (.["operation type"] == "Q") => "non fabrication" :: ()
                otherwise "fabrication" :: ()
            )
        )
    )

    // Production

    "order phases" :: collection <= map table "order phases": encode key -> ()
    "locations" :: collection <= map table "locations": encode key -> (
        "areas" :: collection <= map . -> "areas": encode key -> ()
    )
    "orders" :: collection <= map table "orders": encode key -> (
        "quantity" :: integer <= .["requested quantity"]
        "delivery date" :: text <= serialize .["delivery date"]

        "phase" :: reference <../"order phases"> <= entry . -> "order phase"
        "status" :: option <= case
            (.["phase"] == "99") => "cancelled" :: ()
            (.["phase"] == "98") => "finished" :: ()
            otherwise => "active" :: ()

        "article type" :: reference <../../"article types"> <= entry . -> "article"
        "material source" :: option <= case
            (.["material source" == "P"]) => "extrusion" :: ()
            (.["material source" == "R"]) => "stock" :: ()
            otherwise => error <!"Material source should be either P or R">
        "order operations" :: collection <= map table "order operations": encode key -> (
            "earliest possible start time" :: text
                <= serialize .["earliest possible start time"]
            "latest possible start date" :: text
                <= serialize .["latest possible start date"]
            "is packing operation" :: option <= case
```

```
                (.["operation type"] == "E") => "yes" :: ()
                otherwise => "no" :: ()
            )
            "operation type" :: reference <../"article types"/"operations">
                <= entry . -> "operation type"
        )
    )
)
```