Event control action rules for Ampersand

Yuriy Toporovskyy (toporoy)
Yash Sapra (sapray)
Jaeden Guo (guoy34)

Supervised by: Dr. Wolfram Kahl

Department of Computing and Software
McMaster University
Ontario, Canada
April 25, 2016

Abstract

Ampersand Tarski is a tool used to produce functional software documents based on business process requirements. At times, atomic state changes cannot be accepted as transactions since they violate system invariants specified in the requirements. When a system invariant violation occurs, one of two things can happen: the change that is meant to take place is adjusted (where it no longer violates the rules of the system) or the change is discarded. The ultimate purpose of Event-Condition-Action rules for Ampersand (EFA) is to replace Ampersand's "Exec-engine" which is currently used to fix system violations. The EFA project aims to ultimately make the Exec-engine superfluous by correctly translating the Event-Condition-Action (ECA) rules and implementing them as SQL queries so that system violations may be automatically corrected. EFA is automated and requires no additional work from the user. In addition, the generated SQL produced by EFA is, by construction, syntax-correct and type-correct. Furthermore, EFA has a high degree of modularity as it only relies on Ampersand's essential components.

Contents

Chapter 1

Introduction

This document is a guide for the EFA project and includes

- an brief introduction to Ampersand
- the software requirements used to implement EFA
- development and design of EFA, describing each module of EFA
- a description of software testing done on the system.

1.1 Document Guide

The first chapter covers the basics of Ampersand, the purpose of the project, the intended audience (i.e. our stakeholders), tools used to run Ampersand and terminology which will be used throughout this guide.

The second chapter provides an overview of EFA's requirements specifications; it is divided into Ampersand system requirements and the EFA project requirements.

The third chapter details the module system of EFA, as well as the design principles which guided said module system. EFA, as well as the core Ampersand system, is currently in active development where changes occur frequently. Commonly accepted practice for this situation is to decompose modules based on the principle of abstraction, where unnecessary information in hidden for the benefit of designers and maintainers(DP84, Par72).

The fourth section covers testing, specifically, it covers property testing using QuickCheck and testing MySQL queries using WorkBench. Lastly, the Appendix covers a variety of information such as tips for setting up Ampersand, literate source code, and notes on issues that came up during the course of EFA's implementation.

1.2 Ampersand

The motivation behind Ampersand comes from Requirements Engineering, which is a large part of designing software systems. One of the greatest challenges of Requirements Engineering is translating informal business requirements into formal functional specifications. business requirements contain ambiguity because they are written in a natural language with the intention of being easily understood by humans; however, functional specifications must be written in a formal language that is unambiguous and precise. Typically, this translation of business requirements to a formal specification is done by a requirements engineer, which can be prone to human error.

Ampersand offers an alternative solution which translates the glsnatural language used in business process into requirement specifications (JWM10). Ampersand is a software system which offers many tools that aid their clients in the translation process from business requirement to formal requirement specifications. EFA is implemented as an internal component of Ampersand that is meant to automate the correction of system invariants. EFA helps reduce the amount of manual labour required by the user by executing SQL queries to fix data violations.

Given a set of data and a set of rules used to operate on the data, Ampersand is able to determine whether there are discrepancies between the two sets. Data discrepancies are violations or illegal transactions that occur in the Ampersand system when a set of data fails to follow the rules given by the user. When a system violation occurs, one of two things can happen: the change that is meant to take place is adjusted (where it no longer violates the rules the user provides) or the change is discarded. EFA enhances the Ampersand system by translating ECA rules and executing them as SQL queries to correct system violations and safeguard the database from transactions that would violate the business rules.

1.3 Purpose of the project

Ampersand follows a *rule based* design principle. Rules are integral to an organization and these are based on some principles and guidelines set by the organization. Ampersand uses an ECA (Event-Condition-Action) approach to make sure all rules are satisfied. An ideal information infrastructure supports employees and other stakeholders to maintain the rules of the business. To maintain a rule means to prevent or correct all violations that might occur due to any external or internal factor.

A large portion of the Ampersand system is already in place; the primary focus

of this project was to augment Ampersand with increased capabilities for automation. The module "Automatically Fix Violations" in Figure 1.1 represents the EFA project and where it fits in the current version of Ampersand. Ampersand relies on the AMMBR (Jooa) algorithm to help fix these data violations. The role of AMMBR in Ampersand is to generate functions which can restore violations in the generated prototype for the given information system. In AMMBR, human involvement is only limited to representing rules (in the ADL files). Before our current project was started, there was no way of actually applying the AMMBR-generated ECA rules to rule-violating database states. There was therefore also no means to test whether AMMBR is producing appropriate ECA rules.

The current state of Ampersand uses, as a work-around, the so-called "ExecEngine". This requires the author of the ADL file to write PHP code operating on strings in order to tell Ampersand how violations should be fixed. This required the author of the ADL file to know PHP and have knowledge of the internals of Ampersand, essentially making information hiding impossible in Ampersand itself if one wanted to use the ExecEngine.

The EFA project, an extension to the Ampersand system, allows the user of Ampersand to generate a prototype for their information system in which invariants are automatically maintained by code generated by EFA, which is derived from the specification given by the user.

The ECA rules, which act as an input to EFA, are translated into human-readable SQL. This can later be viewed in the command line using the `--print-eca-info` flag. The generated SQL queries allow the correctness of AMMBR to be checked by running code generated by AMMBR on actual databases. Therefore, EFA will be an essential tool for testing AMMBR.

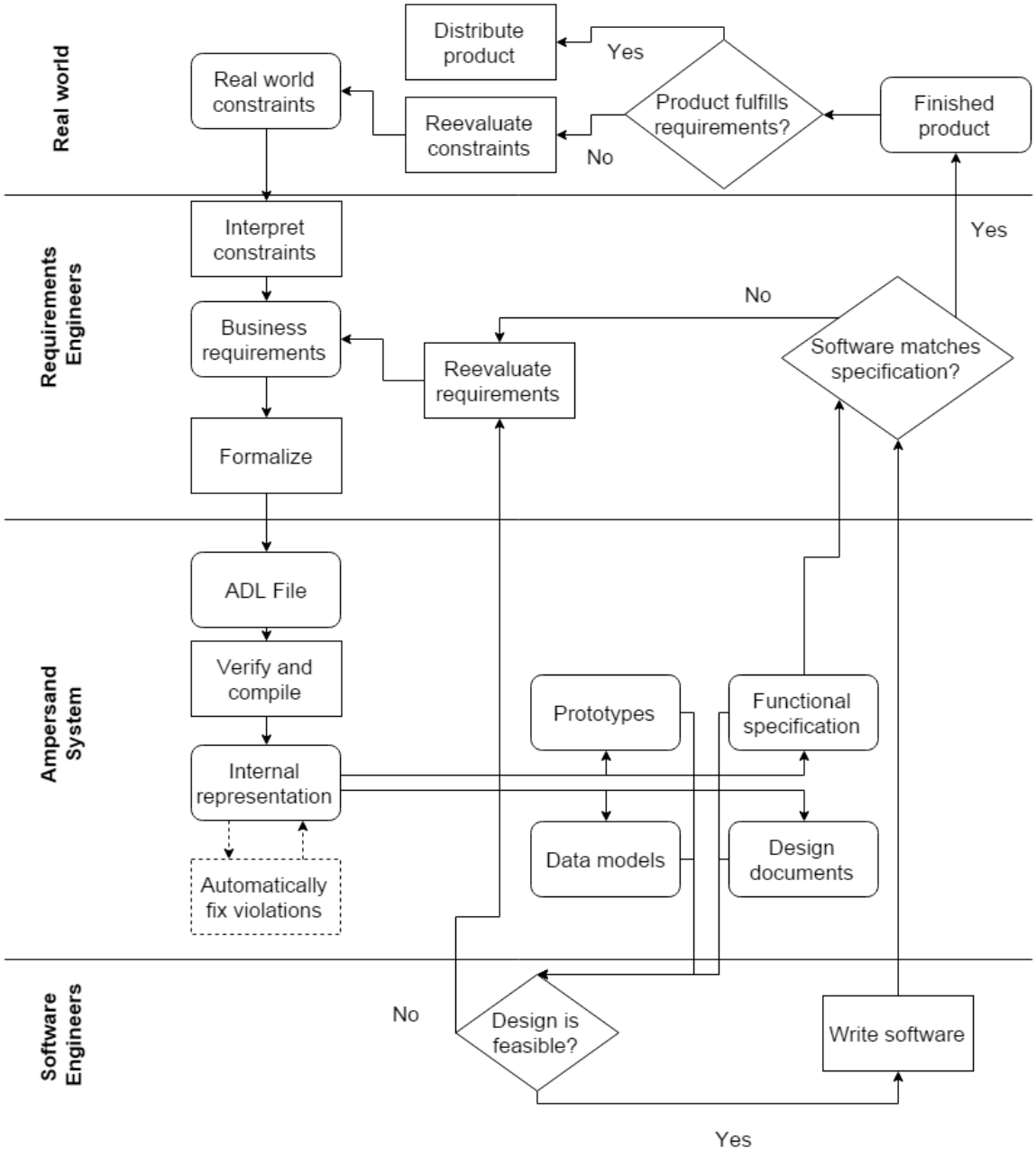

Figure 1.1: Business process diagram representing EFA project represented as a dashed box

1.4 The Stakeholders and the intended audience

The stake holders of Ampersand are:

Ampersand Designers Responsible for maintaining and developing Ampersand. The overall goal of Ampersand is to automate a large part of the software requirements and design process, and EFA fits this goal by greatly increasing the amount of automation in the system, and therefore decreasing the burden on the user. Furthermore, EFA replaces the ExecEngine, which allows more of Ampersand to be abstracted and simplifies the maintenance and design of the Ampersand code.

The Customer The requirements engineers using Ampersand to generate prototypes will benefit from the EFA project. This will decreases the amount of time Ampersand users spend manually inserting PHP code to restore system invariants. The prototypes generated by Ampersand with the inclusion of EFA will be less error prone due fewer occurences of violations exposed to the user.

This document is intended to help introduce Ampersand users to EFA (ECA rules for Ampersand) – it provides a basic structure that allows individuals to quickly access the information they seek. It is also intended to describe the design, implementation, requirements and testing of EFA for the Ampersand developers.

1.4.1 Ampersand In Practice

The Ampersand system is implemented in Haskell and relies on external software tools to help generate a prototype and its supporting documents. On MS Windows, there is an executable available, but on Linux and Mac operating systems it requires installing "stackage" (The Haskell Tool Stack) to build Ampersand from source. Instructions on how to install the tools and components listed in this section can be found in the appendix.

Glasgow Haskell Compiler

Ampersand requires the Glasgow Haskell Compiler (GHC) (GHC), at least version 7.10.3 to compiler Ampersand source code.

Ampersand's GitHub

Github hosts the most current version of the Ampersand system. Github is used to maintain consistency between the main Ampersand branch and this project. The most recent version of Ampersand can be found at (Joob).

Graphviz

Graphviz (Ell) is an open source graph visualization software that Ampersand uses to generate artifacts; specifically it is used to create graphics for the prototype's requirement documentation. Graphviz is able to take descriptions of graphs in simple text and create diagrams from them.

The Cabal System

The Cabal System (Cab) is used for building and packaging Haskell libraries and programs. Cabal describes what a Haskell package is, how these packages interact with the language, and what must be implemented to support the packages that are used. It is part of a larger infrastructure used to distribute, organize, and catalog Haskell programs and their associated libraries. The Cabal build system also uses the Hackage (hac) Haskell source repository.

The Haskell Tool Stack

The Haskell Tool Stack (Cab) is used to install the most recent version of Ampersand from source code, and various Haskell packages (e.g., QuickCheck).

QuickCheck

QuickCheck (Cla) is used to test properties of functions used in EFA's modules. QuickCheck is a library for random testing of program properties; a programmer simply provides a specification of properties which functions should satisfy, and QuickCheck generates a large number of random cases to test these properties. QuickCheck comes with a manual on how properties can be defined and used.

MySQL WorkBench

MySQL WorkBench (wor) is used to test the SQL queries generated by EFA. MySQL WorkBench is a graphical tool for MySQL databases and comes with a built-

in editor; in this project it is used to manually test queries. Through various stages in development it is used for data modeling.

1.5 Naming Conventions and Terminology

Ampersand Ampersand refers to the software tool described in this document, as well as the method of generating functional specification from formalized business requirements.. 1, 3, 4, 7, 8

business requirements Requirements which exist due to some real world constraints (i.e. financial, logistic, physical or safety constraints).. 3

ECA The rule structure used for data bases and commonly used in market-ready business rule engines. ECA rules are used in Ampersand to describe how a database should be modified in response to a system constraint becoming untrue.. 1, 4

functional specification A formal document which details the operation, capabilities, and appearance of a software system.. 3

natural language Language written in a manner similar to that of human communication; language intended to be interpreted and understood by humans, as opposed to machines.. 3

prototype Ampersand generates a prototype for the user that provides a front-end interface that connects to a back-end database.. 7

Requirements Engineering The process of translating business requirements into a functional specification.. 3

Chapter 2

Functional Requirements

This chapter details the functional requirements of EFA, and in particular the origin of each requirement as well as its priority and relevant test cases.

In the Ampersand system, each transformation which is done on the user specification of their information system has a reasonable argument of correctness. While a large part of ensuring this soundness falls to dynamic testing, many components can be verified by static program analysis, e.g. by the GHC typechecker and compiler. This requirement for EFA is the same as for Ampersand - as much information as possible should be modelled by the Haskell typechecker, which we assume to be sound. Therefore, typechecked programs are also assumed to be correct with respect to those semantics which have been encoded on the type level.

The Ampersand system also generates a large number and large variety of software design and requirements engineering artifacts, including documents in various markup formats, graphics and charts, and a prototype implementing the business logic encoding in the ADL file. This large variety of functions must all work independently of each other, and in particular not interfere with each others' operation, which is made very easy by the stateless nature of Haskell. However, it is still possible to write Haskell code with very complex dependencies and code structure, making it very difficult to maintain, upgrade and debug. EFA is required to be easy to maintain, and therefore to integrate with the core Ampersand code base easily.

The functional requirements of Ampersand can therefore by summarized as correctness and modularity.

2.1 System Requirements

Requirement	S1
Description	Create pure functions with no unintended side effects
Rationale	The use of a functional programing languages requires that this program be a pure function and does not have side effects, however certain portions of the code requires the execution of side effects to match the behaviour presented by external programs. In these specific instances, the side effects are an intended behaviour.
Originator	Stakeholder/Developer
Test Case	Desired results can be confirmed as they will be reflected in changes that take place in the Ampersand database.
Customer Satisfaction	5 - Highest
Priority	5 - Highest

Requirement	S2
Description	Added modules must fit within Ampersand's current framework
Rationale	Ampersand is a huge system that has weekly additions to prevent conflict and breaking of existing packages/modules, an effort should be made to minimize external dependencies. As EFA will be an internal component of Ampersand, if a package that EFA depends on to function properly is no longer maintained and breaks, it will in turn break Ampersand.
Originator	Ampersand Creators (i.e. our client)
Test case	Added modules are tested with cabal build inside of the Ampersand system as an internal component (i.e. System testing)
Customer Satisfaction	4 - High
Priority	4 - High

Requirement	S3
Description	All code must be maintainable.

Requirement	S3
Rationale	For a system such as Ampersand to be maintainable, all code for each of its components must be well documented so it may be easily understood by those that were not a part of its original development.
Originator	Ampersand Creators (i.e. our client)
Test case	A literate program that produces a LaTeX document.
Customer Satisfaction	4 - High
Priority	4 - High

2.2 Project Requirements

Requirement	P1
Description	Generated SQL queries must preserve the semantics of ECA rules.
Rationale	The translation would otherwise not be correct, as the rules would be meaningless if their semantics are lost.
Originator	Ampersand Creators
Test Cases	Internal structure of ECA rules can be compared to SQL queries through a series of datatype tests, each of which will result in a traceable result or error message
Priority	4 - High

Requirement	P2
Description	Provable Correctness: Haskell like other functional programming languages have a strong type system which can be used for machine-checked proofs.

Requirement	P2
Rationale	Curry-Howard correspondence which states that the return type of the function is analogous to a logical theorem, that is subject to the hypothesis corresponding to the types of the argument values that are passed to the function and thus the program used to compute that function is analogous to a proof of that theorem.
Originator	Ampersand Creators
Test Cases	Using QuickCheck to test function properties.
Priority	4 - High

Chapter 3

Non-functional Requirements

This chapter briefly describes the various non-functional requirements associated with the EFA project.

3.1 Look and Feel Requirements

- The product shall comply with Ampersand standards
- The product shall appear readable for Ampersand contributors

3.2 Usability and Humanity Requirements

- *Efficiency of use*
  - EFA is easy to use for any Ampersand user.
  - The user can use EFA with accuracy, as they are provided with the option to use EFA's automated service through the users interface.
- *Ease of remembering*
  - EFA does not require the user to memorize protocols
- *Error rates*
  - EFA eliminates ECA rule violations caused by the user, EFA's implementation is provably correct through Haskell type-checking system.

- *Feedback*
  - User receive feed-back concerning the ECA rule violations that have been resolved.
- *Overall satisfaction in using the product*
  - Users can be confident in using EFA as it is well tested and provides accurate results which the user can confirm.

3.2.1 Personalization and Internationalization Requirements

EFA is available only in English but can be adapted for other languages in later development versions.

3.2.2 Learning Requirements

EFA has a shallow learning curve, more time is spent learning the Ampersand system. No training is necessary to use EFA as long as the user can operate Ampersand.

3.3 Understandability and Politeness Requirements

Conceptually EFA is easy to understand and users will intuitively know what this product does for them. To further understandability, EFA feed back uses natural language that is familiar to the user and easy to understand. In addition, EFA hides all details of its construction from the user and provides the user.

3.3.1 Accessibility Requirements

EFA is unable to individually confirm to disability requirements as it is an internal component of Ampersand.

3.4 Performance Requirements

The performance requirements are those requirements related to the actual execution of the Ampersand tool with EFA.

3.4.1 Speed and latency Requirements

The use of EFA should not result in a noticeable time delay. EFA shall not take more than 3 seconds to complete in the worst case scenario.

3.4.2 Safety-Critical Requirements

EFA will not expose sensitive information in the Ampersand system to outside sources or create new vulnerabilities.

3.4.3 Precision or Accuracy Requirements

SQL queries generated by EFA is correct 100% of the time with full coverage of all ECA rules that apply.

3.4.4 Reliability and Availability Requirements

EFA is available for use anytime Ampersand is used. In the event that Ampersand fails, then EFA will also be unavailable as it depends on Ampersand generated data taken from the user.

3.4.5 Robustness or Fault-Tolerance Requirements

In the absence of ECA rule violations, EFA will continue to function and be on stand-by until a rule violation is triggered.

3.4.6 Capacity Requirements

Ampersand runs on individual machines; EFA as an internal component of Ampersand will be able to deal with any amount of data that Ampersand can handle.

3.4.7 Scalability and Extensibility Requirements

EFA is capable of handling large volumes of data, and the translation of ECA rules to SQL queries requires a standard amount of time. The number of ECA rules is expected to expand as Ampersand etches closer to completion.

3.5 Operational and Environmental Requirements

Any system that is currently running Ampersand will be able to run this product under a new verion and thus no new requirements have been introduced. .

3.6 Maintainability and Support Requirements

All code submitted for this project must be maintainable, which mean it is well documented and comes with mathematical proof. EFA must make sure that each specification/error is traceable.

3.7 Security and Integrity Requirements

Access to the database and software which supports the various functions of Ampersand are run locally and subjected to the security system the user has in place on his or her work station.

3.8 Legal Requirements

The implementation must eventually be included in Ampersand, which is licensed under GPL3. To comply with this license, all of the implementation code must be either written by us so we may license it under GPL, or must already be licensed under GPL, or a compatible license, by its original author. We do not plan to use existing code, other than as a reference.

Chapter 4

System Architecture and Module Hierarchy

This section provides an overview of system architecture and module hierarchy. The initial section introduces term and tools used in the making of each EFA module. The module design is detailed with UML-like class diagrams. However, UML class diagrams are typically used to describe the module systems of object-oriented programs, as opposed to functional programs. Many of the components of the traditional UML class diagram are inapplicable to functional programs; therefore, we detail our modifications to the UML class diagram syntax in section 4.3.

Furthermore, the syntax used to describe types and data declarations is not actual Haskell syntax. The syntax shares many similarities, but several changes to the syntax are made in this document in order to present the module hierarchy in a clear manner. These changes are also detailed, in section 4.2.

4.1 External Libraries

No additional dependencies are required outside of those that Ampersand already uses. EFA directly uses the following dependencies of Ampersand:

Ampersand Core Libraries The EFA project depends on the Ampersand software for the definition of core Data Structures, (i.e., FSpec, which contains the definition of the underlying ECA rules). EFA also maintains the relational schema of the input, and hence, imports Ampersand's existing functions to fetch the table declarations while generating SQL Statements for the ECA rules. AMMBR (Joo07), which is the key algorithm responsible for translating

business requirements into ECA rules is an integral part of Ampersand.

simple-sql-parser EFA's pretty printer depends directly on this library for formatting and printing SQL statements. The SQL statement syntax defined here is built on top of the existing expression syntax defined in this package. This package is the one used by the core Ampersand system, so our use of it facilitates interaction and integration with Ampersand. (Whe15)

wl-pprint The wl-pprint library(Lei15) is a pretty printer based on the pretty printing combinators. EFA uses this library in combination with the simple-sql-pretty to output the SQL statements in a human readable format.

deepseq The deepseq library provides a type class which implements a function for reducing values to normal formal, similarly to the built in **seq** function.

4.2 A Description of Haskell-Like Syntax

This section details the syntax used to describe the module system of Ampersand. This syntax largely borrows from actual Haskell syntax, and from the Agda programming language (NDA16). Agda is a dependently typed functional language, and since a large part of our work deals with "faking" dependent types, the syntax of Agda is conducive to easy communication of our module system. The principle of faking dependent types in Haskell is detailed in Hasochism (LM13) (a portmanteau of Haskell and masochism, because purportedly wanting to fake dependent types in Haskell is masochism). While the implementation has since been refined many times over, the general approach is still the same, and will not be detailed here. While the changes made to the Haskell syntax are reasonably complex, the ensuing module description becomes vastly simplified. This section is meant to be used as a reference — in many cases, the meaning of a type is self-evident.

Description of Types and Kinds

In the way that a type classifies a set of values, a kind classifies a set of types. Haskell permits one to define algebraic data types, which are then "promoted" to the kind level (YWC$^+$12). This permits the type constructor of the datatype to be used as a kind constructor, and for the value constructors to be used as type constructors. In every case in our system, when we define a datatype and use the promoted version, we never use the *unpromoted* version. That is, we define types which are never used

as types, only as kinds, and constructors which are never used as value constructors, only type constructors. We write X : A → B → λldots$ → Type to denote a regular data type, and Y : A → B → λldots$ → Kind to denote a datatype which is used exclusively as a kind.

Description of Dependent types

The syntax used to denote a "fake" dependent type in our model is the same as used to denote a real dependent type in Agda. (x : A) → B is the function from x to some value of type B, where B can mention x. This nearly looks like a real Haskell type — in Haskell, the syntax would be `forall (x :: A) . B`. However, the semantics of these two types are vastly different - the former can pattern match on the value of x, while the latter cannot.

In certain cases, it may be elucidating to see the *real* Haskell type of an entity (function, datatype, etc.). To differentiate the two, they are typeset differently, as in this example.

The real type of a function whose type is given as (x : A) → B in our model is `forall (x :: A) . SingT x -> B`. The type constructor `SingT :: A -> Type` denotes the singleton type for the kind A, which is inhabited by precisely one value for each type which inhabits A. The role and use of singleton types is detailed further on, in section 4.4.6.

The syntax ∀ (x : A) → B is used to denote the regular Haskell type `forall (x :: A) . B`. As is customary in Haskell, the quantification may be dropped when the kind A is clear from the context: ∀ (x : A) → P x and ∀ x → P x denote the type `forall (x :: A) . P x`.

Constraints

The Haskell syntax `A -> B` denotes a function from A to B. However, we use the arrow to additionally denote constraints. For example, the function `Show a => a -> String` would be written simply as Show a → a → String . In certain cases, a constraint is intended to be used only in an implicit fashion (i.e. as an actual constraint), in which case the constraint is written with the typical ⇒ syntax.

Existential quantification

The type ∃ (x : A) (P x) indicates that there exists some x of kind A which satisfies the predicate P. Unfortunately, Haskell does not have first class existential

quantification. It must be encoded in one of two ways:

- With a function (by DeMorgan's law):
  `(forall (x ::  A) . P x -> r) -> r`
- With a datatype:
  `data Exists p where Exists ::  p x -> Exists p`

Which form is used is decided based on the circumstances in which the function will most likely be used, since whether one form is more convenient than the other depends largely on the intended use. However, these two forms are completely interchangeable (albeit with some syntactic noise) so the syntax presented here does not distinguish between the two.

Types, kinds, and type synonyms

Type synonyms are written in the model as Ty : K = X, where Ty is the name of the type synonym, K is its kind, and X its implementation. This is to differentiate from type families, which are written as Ty : K **where** Ty ... = ...

Overloading

Haskell supports overloaded function names through type classes. When we use a type class to simply overload a function name, we simply write the function name multiple times with different types. The motivation for this is that often the real type will be exceeding complex, because it must be so to get good type inference.

Omitted implementations

When the implementation of a type synonym, or any other entity, is omitted, it is replaced by "...". This is to differentiate from a declaration of the form Ty : Type, which is an abstract type whose constructors cannot be accessed. Furthermore, types may have pattern-match-only constructors; that is, constructors which can only be used in the context of a pattern match, and not to construct a value of that type. This is denoted by the syntax "**pattern** Ctr : Ty". It is not a simple matter of convention - the use of this constructor in expressions will be strictly forbidden by Haskell.

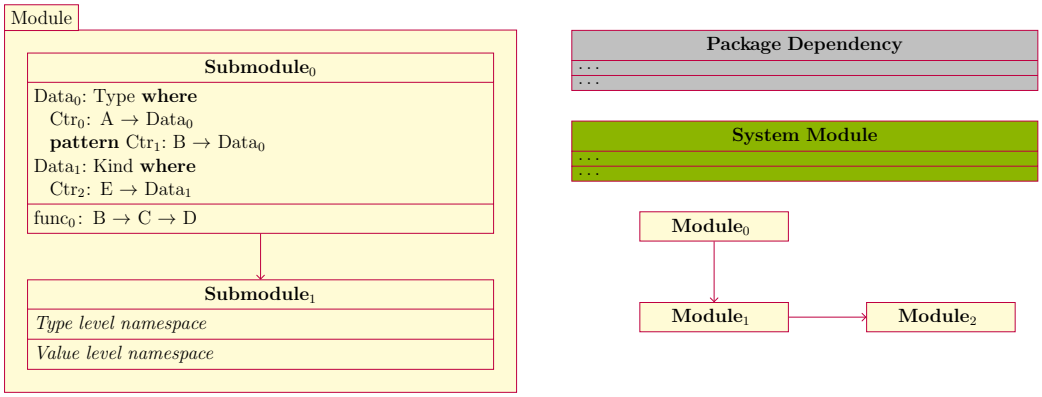

Figure 4.1: Example of module diagram syntax

4.3 A Description of Module Diagram Syntax

The module hierarchy is broken down into multiple levels to better describe the system. A coarse module hierarchy is given, and each module is further broken into submodules. A dependency between two modules A and B indicates that each submodule in A depends on all of B. There is no necessity to break down modules into submodule, if they do not have any interesting submodule structure. Arrows between modules and submodules denote a dependency.

External dependencies, which are modules which come from an external package, are indicated in grey. System modules, which are modules part of Ampersand, but not written specifically for EFA (or, on which EFA depends, but few or no changes have been made from the original module before the existence of EFA), are indicated in green. The module hierarchy of these modules is not described here; they are included simply to indicate which symbols are imported from these modules. An example of the syntax is found in figure 4.1.

4.4 Module Hierarchy

This section contains a hierarchical breakdown of each module, as well as a brief explanation of each modules' elements. The module hierarchy of EFA as a whole is given in figure 4.2. Note that every module which is part of EFA depends on the Haskell `base` package (which is the core libraries of Haskell). Also note that for the `base` package, we only include primitive definitions (i.e. those not defined in real Haskell) which may be difficult to track down in the documentation. The kinds $\mathbb{N}$

and Symbol correspond to type level natural number and string literals, respectively. The kind Constraint is the kind of class and equality constraints, for example, things like Show x and Int $\sim$ Bool. Note that `Show` itself does *not* have kind Constraint – its kind is Type $\rightarrow$ Constraint. The detailed semantics of these primitive entities can be found in the GHC user guide (GHC). While many modern features of GHC are used in the actual implementation, they are not mentioned in, nor required to understand, the module description.

The primary interface to EFA is the function eca2PrettySQL, which takes an FSpec (the abstract syntax of Ampersand) and an ECA rule, and returns the pretty printed SQL code for that rule. Also note that while the dependencies within EFA modules are relatively complex, they depend on the rest of the Ampersand system in a simple manner. The modules Test and Prototype implement the testing framework and the prototype generation, respectively; these modules depend directly on only one module from EFA, namely ECA2SQL. Similarly, the majority of EFA itself does not depend directly on Ampersand modules outside of EFA. This makes EFA very resilient to changes in the core Ampersand system; in order to update EFA to work with a modification to Ampersand, only one EFA module – ECA2SQL – will generally need to be modified.

All functions named in the module hierarchy are total - they do not throw exceptions, or produce errors which are not handled or infinite loops. Therefore, no additional information past the type of the function is required to deduce the inputs and outputs of the function – they are precisely the inputs and outputs of the type.

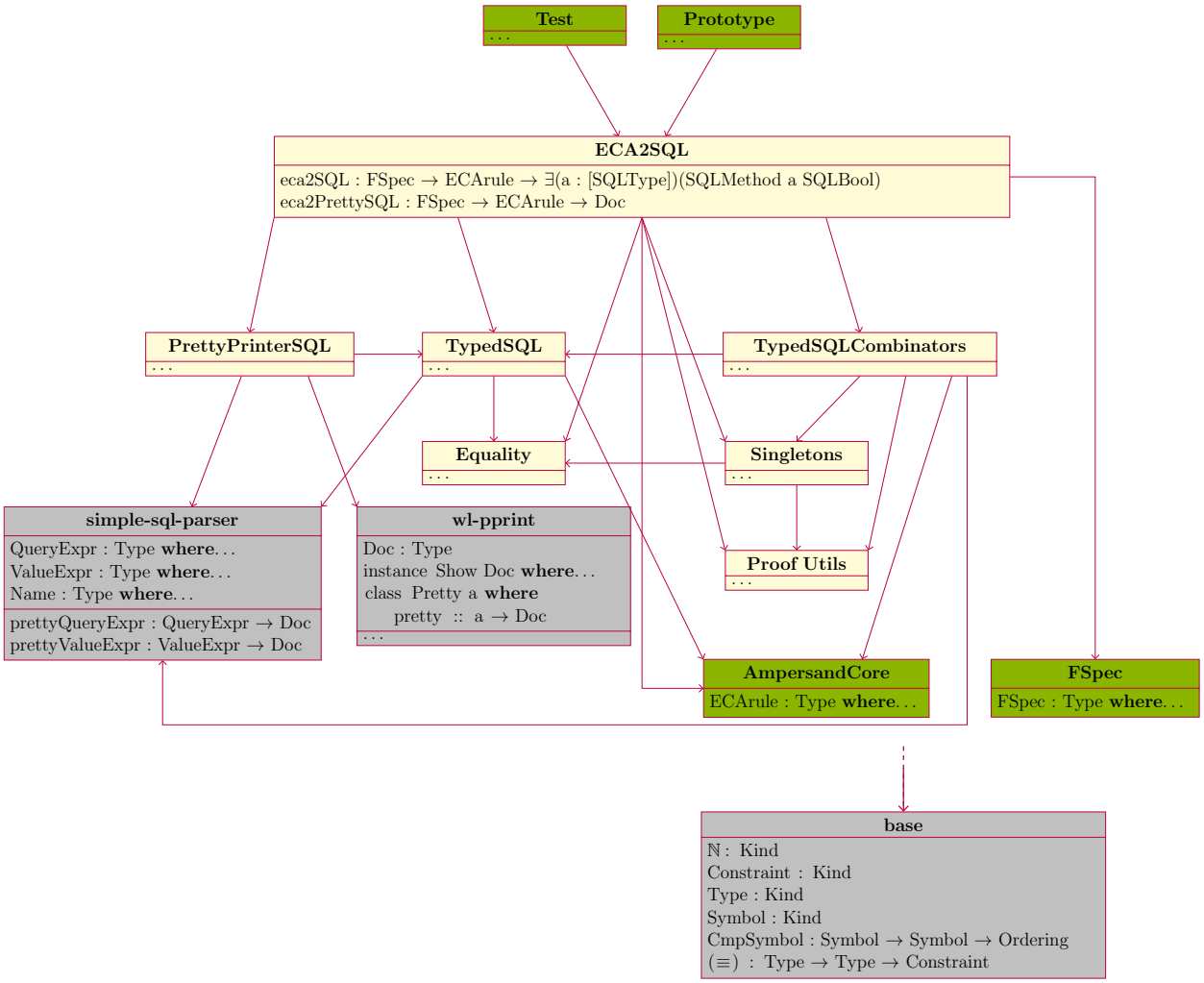

Figure 4.2: Module diagram for EFA as a whole

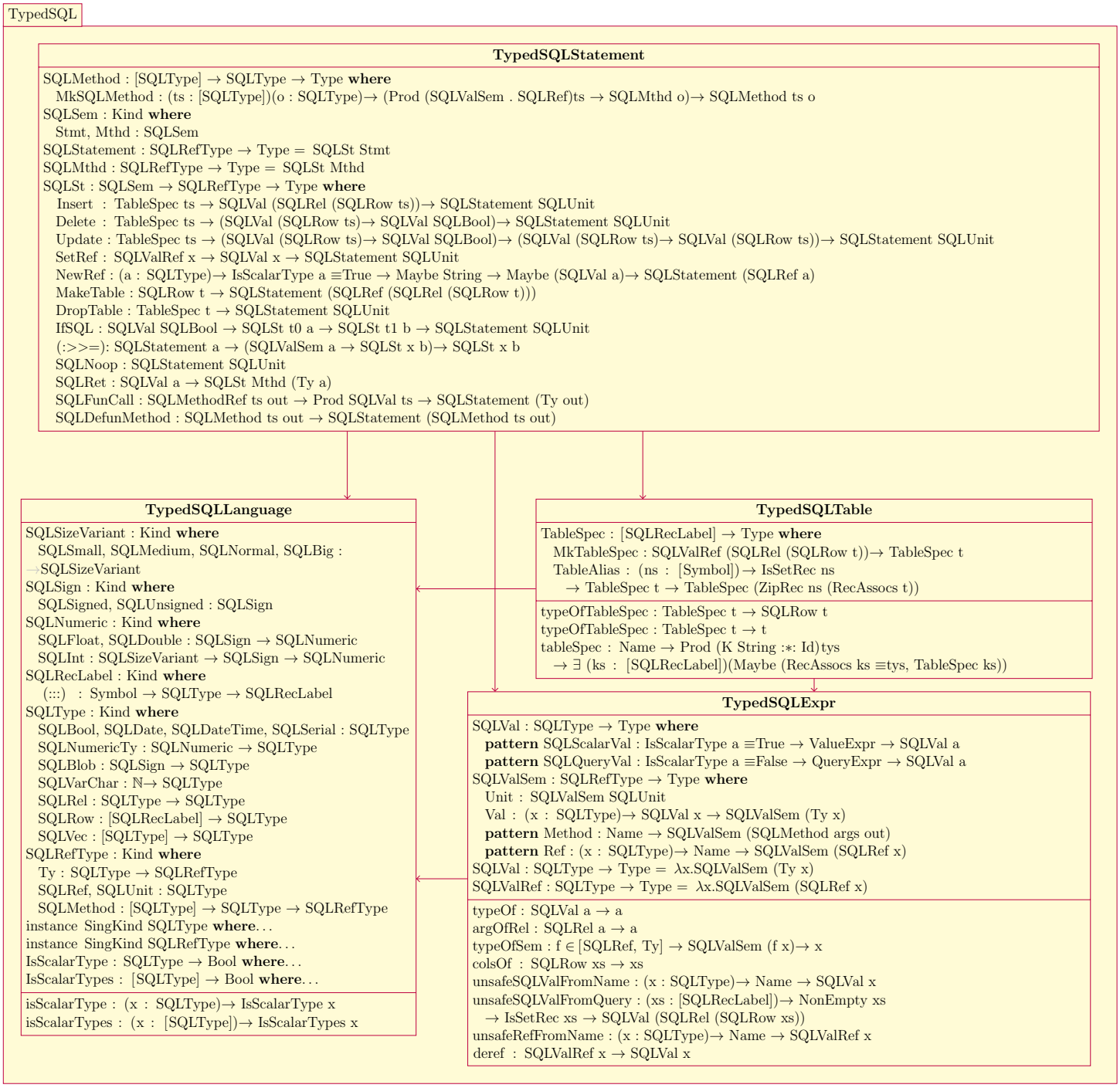

Figure 4.3: Module diagram for TypedSQL

4.4.1 TypedSQL

The module hierarchy for the TypedSQL module is shown in figure 4.3. The submodules of TypedSQL are quite large, however, the majority of the definitions within are type and kind definitions, which correspond precisely to entities defined by MySQL. The only exception is SQLRel, which distinguishes relations from scalar types - MySQL does not make this distinction. Only a few helper functions are defined in these modules – namely, only things which form the core interface to TypedSQL and in particular, SQLStatement. These functions cannot be defined outside of the module, usually because they use an abstract constructor. By making the core interface to TypedSQL very small, maintaining the TypedSQL language definition separately from the implementation of EFA is simplified. All of the data types in TypedSQL are correct by construction, with the exception of the functions explicitly labeled "unsafe". These functions (unsafeSQLValFromName, unsafeSQL-ValFromQuery, and unsafeRefFromName) are required only when implementing a new SQL primitive on top of the SQL language - they are not intended for regular use.

The TypedSQLLanguage module models the SQL type language in Haskell with a series of kind declarations. The language being modeled is only a subset of the SQL type language, corresponding approximately to the subset which the core Ampersand system already uses. The meaning of each Haskell type corresponds exactly to the appropriate MySQL type, which are detailed in the MySQL manual (myS). Similarly, the constructors of SQLSt all correspond to different varieties of SQL statements – for the majority of constructors, there is a one-to-one correspondence between the semantics of the constructor, and the semantics of the SQL statement with the same name. The exceptions are:

`SQLNoop` MySQL does not have a primitive no-op statement

`SQLDefunMethod` MySQL does not allow defining procedures within procedures; this constructor denotes that a method "defined" within another statement must first be loaded as a MySQL Stored Procedure (myS).

`:>>=` This constructor corresponds to sequencing statements. This constructor embeds scope checking of MySQL statements in the Haskell compiler – ill-formed statements containing variables which are not defined (i.e. not in scope) will be rejected by the Haskell compiler.

The SQLSt data type also distinguishes between two varieties of statements: SQLStatement and SQLMthd. The former is the type of regular statements, while

the latter is the type of "almost" complete methods - methods whose formal parameters have not yet been bound. This is done in order to statically guarantee that a SQL method always returns a value. Due to the type of :>>=, this also rules out SQL programs which contain dead code – no code can follow SQLRet, which is always guaranteed to return from the function. While this does rule out some valid programs (for example, an if statement in which both branches end with a return, but there is no return following the if statement, will be rejected), these programs can be written in an equivalent way in our language without any loss of generality.

4.4.2 TypedSQLCombinators

The module TypedSQLCombinators, whose members are given in figure 4.4, implements a subset of primitive SQL functions on top of the TypedSQL expression type. The data type PrimSQLFunction encodes the specification of each function; the type and semantics of each function is that of the corresponding function in MySQL (refer to the MySQL manual (myS) for details on each function). The only exception is the Alias function, which is a primitive syntactic constructor (not a named entity) in MySQL - rows can be aliased with a select statement. Aliasing a row means to change the name of each association in the row, but not the shape of the row (i.e. the types of each element of the row, as well as their ordering). The single function primSQL implements all of the primitive SQL functions. It takes as an argument a specification of the primitive function, a tuple of arguments of the correspond types, and returns a SQL value, again of the corresponding type.

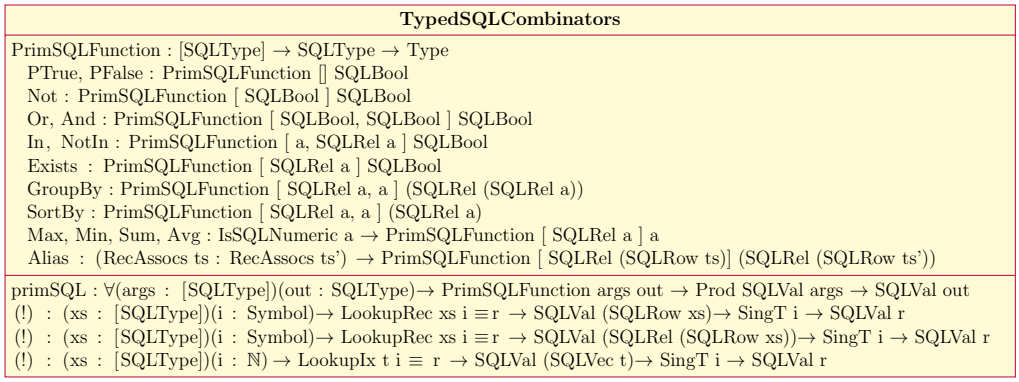

Figure 4.4: Module diagram for TypedSQLCombinators

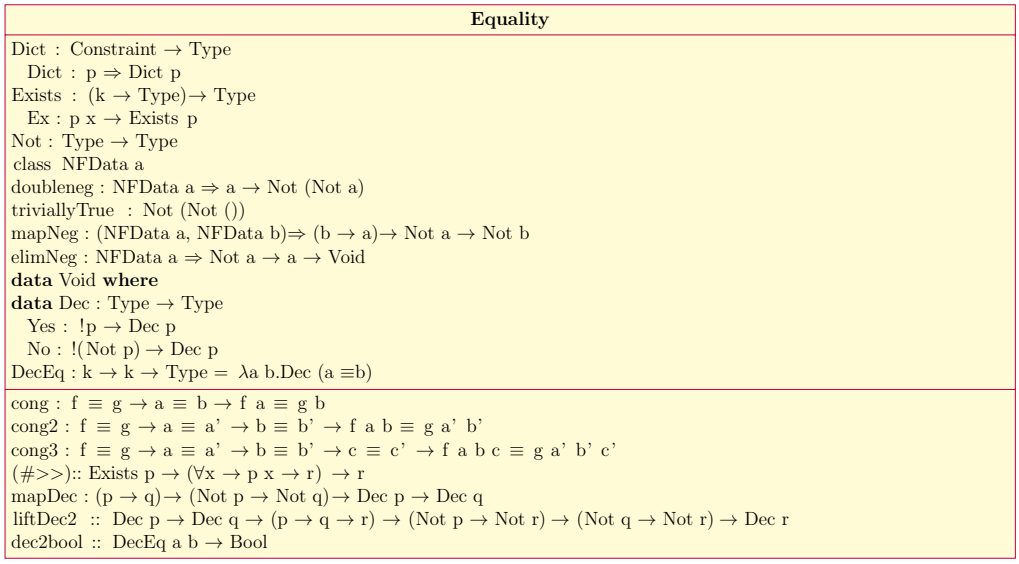

Figure 4.5: Module diagram for Equality

4.4.3 Equality

The Equality module (4.5) defines several utilities for working with proof-like values, including the existential quantification data type, proofs of congruence of propositional equality of various arities – cong, cong2, cong3 – and the Dec type, which encodes the concept of a decidable proposition. The most important element of this module, however is the abstract Not type. We must prove certain things about our program to the Haskell type system. For example, if one attempts to construct a scalar SQLVal for some SQLType S, one must first prove that that type is a scalar type. The main use of this is decidable equality, which is similar to regular equality, but additionally to giving a "yes" or "no" answer, it also stores a *proof* of that answer.

The view of propositions as types comes from the Curry-Howard isomorphism (Wad15); however, this is not quite sound in Haskell, because every type is inhabited by $\bot$, which corresponds to `undefined`, an exception, or non-termination. Due to laziness, an unevaluated $\bot$ can be silently ignored. At worst, this corresponds to a sound use of unsafeCoerce leaking into the "outside world", that is, allowing a user to accidently expose the use of an unsafeCoerce. One can usually work around this by evaluating all proof-like values to normal form before working with them (this is accomplished by the NFData class, which stands for normal form data). The normal form of most datatypes contains precisely one "type" of bottom – namely the value

$\bot$, as opposed to $\bot$ wrapped in a constructor, for example Just $\bot$. This bottom can then be removed with the Haskell primitive **seq**, producing a value which can soundly be used as a proof.

A problem arises when we consider the negation of a proposition. $\neg p$ is typically encoded in Haskell as p $\rightarrow \bot$, where the type $\bot$ can be represented by any uninhabited type, typically calledVoid. However, the normal form of a function can contain any number of $\bot$ hidden deep inside the function, because evaluating a function to normal form only evaluates up to the outermost binder. To prevent any unsoundness which this might cause, values of type Not p are reduced to normal form as they are built, starting with a canonical value which is known to be in normal form - the value triviallyTrue . This is the role of NFData in the type signature of mapNeg. As mentioned previously, the type Not is abstract, so the provided interface, which is known to be sound, is the only way to construct and eliminate values of type Not p.

4.4.4 PrettySQL

The module PrettySQL defines pretty printers for each of the types corresponding to SQL entities, including SQL types, SQL values, SQL references and methods, and SQL statements. These pretty printers produce a value of type Doc (which comes from the wl-pprint package), which is like a string , but contains the layout and indentation of all lines of the document, allowing for easy composition of Doc values into larger documents, without worrying about layout. The SQL entities are pretty printed in a human readable format, complete with SQL comments which indicate the origin and motive of generated code.

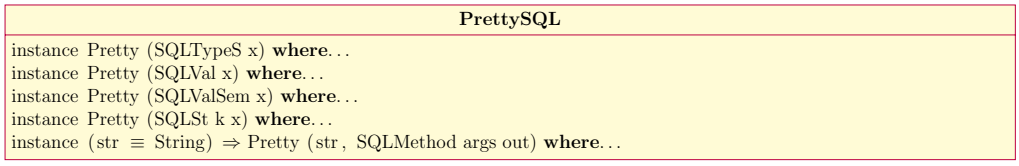

Figure 4.6: Module diagram for PrettySQL

4.4.5 Proof Utils

The ProofUtils modules, shown in figure 4.7, provides various utilities for proving compile time invariants, including the definitions of various predicates used in other modules, as well as the value-level functions which prove or disprove those predicates. Generally speaking, a predicate is a type level function which returns

either a true or false value, or has kind Constraint, in which case truth corresponds to a satisfied constraint, while false to an unsatisfied one. Many predicates have value level witnesses as well; these are datatypes which are inhabited if and only if the predicate is true. Therefore, pattern matching on the predicate witness can be used to recover a proof of the predicate at run time.

The function of the most important predicates and types is briefly summarized as follows.

`Prod` The type Prod f xs represents the n-ary product of the type level list xs, with each $x \in xs$ being mapped to the type f x. This type is accompanied by the functions prod2sing, sing2prod which convert between singletons and products; the functions foldrProd, foldlProd, foldrProd', which are all eliminators for Prod (several eliminators are needed because the most general eliminator is not well typed in Haskell).

`Sum` The same as above, but the n-ary sum as opposed to product.

`All` The constraint All c xs holds if and only if c x holds for all $x \in xs$.

`Elem, IsElem` IsElem x xs holds precisely when $x \in xs$. Elem is the witness of IsElem.

`AppliedTo, Ap, :.:, Cmp, :*:, K, Id, Flip` Categorical data types which encode a generalized view of algebraic data types. This approach is largely standardized (and is only replicated here to avoid incurring a large dependency) – for more information, see (Swi08).

`RecAssocs, RecLabels, ZipRec` Type level functions for working with type level records. A record in this context is a list of types of some kind, each associated with a unique string label. RecAssocs, RecLabels retrieve the associations and labels of a record, respectively, while ZipRec constructs such a record from the associations and labels. It is the case that ZipRec (RecAssocs x)(RecLabels $\rightarrow$x) $\equiv$ x for all x .

`IsSetRec, SetRec` The predicate IsSetRec x holds precisely if x is a valid record type, whose labels are all unique. SetRec is the witness for IsSetRec.

`IsNotElem, NotElem` The predicate IsNotElem x xs holds precisely if $\neg x \in xs$. NotElem is the witnss for IsNotElem.

`&&,And` Binary and n-ary boolean conjunction, with the usual semantics.

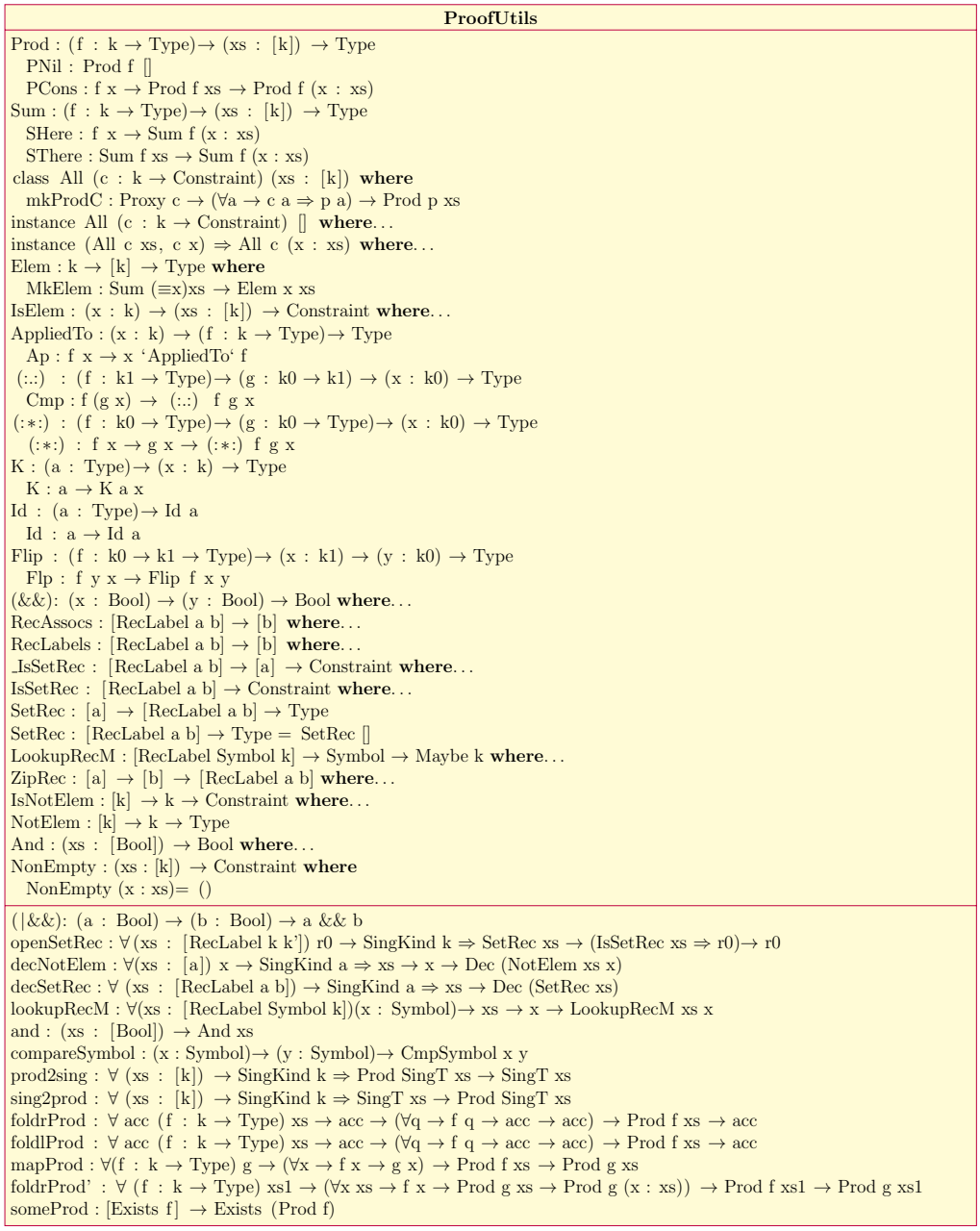

Figure 4.7: Module diagram for ProofUtils

4.4.6 Singletons

The Singletons module (figure 4.8) is not fully detailed here; rather, a vastly simplified version is presented. The SingT type denotes a generic singleton for any kind which implements SingKind – then, the main operation of interest on singletons is decidable equality, which is realized by the function %≡. The detailed implementation is omitted because the singletons approach in Haskell is well known and well documented (EW12). We re-implement singletons instead of using the well established `singletons` library because, while this library is very well written, it relies very heavily on Template Haskell (SPJ02), which is essentially string-based metaprogramming. Template Haskell is extremely error prone and very difficult to maintain. As one of our primary goals is long-term maintainability, and Template Haskell changes, sometimes drastically, with every new release of GHC, including it in this project was deemed not worth the headache. Therefore, we have reimplemented singletons without Template Haskell, at the cost of having to write slightly more boilerplate.

Singletons
SingT : k → Type class SingKind (k : Kind) **where**...
(%≡) : ∀ (x : k) (y : k) → SingKind k ⇒ x → y → DecEq x y

Figure 4.8: Module diagram for Singletons

4.5 Key Algorithm

This section contains the literate Haskell documentation of the portion of the EFA source code which implements the key algorithm of EFA.

The eca2SQL function implements the primary algorithm of EFA. This function takes as input an ECA rule and the specification of the information system from which that rule came, and returns a SQLMethod which takes some (unknown) amount of parameters.

```
eca2SQL :: FSpec → ECArule → (∀ (k :: [SQLType]) . SQLMethod k 'SQLBool → r) → r
eca2SQL fSpec@FSpec{plugInfos=_plugInfos} (ECA (On _insDel _ruleDecl) delta action
→_) q =
  unsafeDeclToRow fSpec _ruleDecl $ λargsT →
  q $ MkSQLMethod (prod2sing argsT) $ λargs →
```

The produced AST always begins with a method constructor which gives access to the formal parameters of the function, as a Haskell function. These formal parameters are converted into a SQL expression, and substituted into relation algebra expressions for the delta of the rule.

```
let expr2SQL' =
      FSpec.expr2SQL' (λcase
        d | d ≡ delta → Just (prod2VectorQuery args)
        _ → Nothing
        ) fSpec
```

This is a helper function which produces the abstract SQL for a single insertion or deletion. This function essentially just looks up the shape of the table to be modified (inserted into or deleted from) and computes which fields of that table correspond to the desired relation algebra expression.

```
unsafeMkInsDelAtom :: Declaration → InsDel → Expression → SQLStatement
→'SQLUnit
unsafeMkInsDelAtom decl = go (getDeclarationTableInfo fSpec decl)
 where
```

If the source and the target of the field are the same, we are working with the identity relation. We have to rename them first for someTableSpec to succeed.

```
go (plug, srcAtt', tgtAtt') =
 let (srcAtt, tgtAtt) =
       if srcAtt' ≡ tgtAtt'
       then ( srcAtt' { attName = "Src" ++ attName srcAtt' }
            , tgtAtt' { attName = "Tgt" ++ attName srcAtt' }
            )
       else (srcAtt', tgtAtt')

 in case plug of
      ScalarSQL{} → fatal 0 "ScalarSQL unexecpted here"
      _ →
```

This should not fail - the input list is clearly non-empty and the source and target names are different.

```
          case someTableSpec (fromString $ sqlname plug)
                 [ (fromString $ attName srcAtt, Ex (sing :: SQLTypeS
                 →'SQLAtom))
                 , (fromString $ attName tgtAtt, Ex (sing :: SQLTypeS
                 →'SQLAtom))
                 ] of
            Nothing →
              fatal 0 $ unwords
                [ "Could not construct table spec from attributesλn",
                show srcAtt, " and ", show tgtAtt ]
            Just (Ex (tbl :: TableSpec tbl)) → λact toInsDel →
```

Insertion or deletion can now be performed on the typed table specification. In the case of deletion, a SQL delete statement must be given a predicate for selecting rows to be deleted, but the PAclause contains an expression representing the rows to be deleted, so the desired predicate must be constructed use the TSQLCombinators module.

```
              case act of
                Ins →
```

```
    withSingT (typeOfTableSpec tbl) $ λ(singFromProxy →
    →SingT (WSQLRow{})) →
      Insert tbl (unsafeSQLValFromQuery $ expr2SQL' toInsDel)

  Del →
    let toDelExpr :: SQLVal ('SQLRel ('SQLRow '[ "src" :::
    →'SQLAtom, "tgt" ::: 'SQLAtom ] ))
        toDelExpr = unsafeSQLValFromQuery $ expr2SQL' toInsDel
        src = sing :: SingT "src"
        tgt = sing :: SingT "tgt"
        dom = toDelExpr T.! src
        cod = toDelExpr T.! tgt
```

The row is deleted if the source and target are in the domain and codomain, respectively, of the expression to delete.

```
        cond :: SQLVal ('SQLRow '[ "src" ::: 'SQLAtom, "tgt"
        →::: 'SQLAtom ] ) → SQLVal 'SQLBool
        cond = λtup → T.sql T.And (T.in_ (tup T.! src) dom)
        →(T.in_ (tup T.! tgt) cod)

        cond' :: SQLVal ('SQLRow tbl) → SQLVal 'SQLBool
        cond' = λ(SQLQueryVal x) → cond (SQLQueryVal x)

    in Delete tbl cond'
```

Helper functions for use in paClause2SQL below. If a PAclause has executed successfully, it sets the "done" variable to true. If it has not executed successfully, it does nothing.

```
done = λr → SetRef r T.true
notDone = const SQLNoop
```

The paClause2SQL function takes the clause to be converted and the variable reference which corresponds to the "done" variable for the given clause. The output is a statement which produces no value (or the unit value).

```
paClause2SQL :: PAclause → SQLValRef 'SQLBool → SQLStatement 'SQLUnit
```

A single deletion or insertion is handled by unsafeMkInsDelAtom above. This always succeeds, assuming that the given expression and table are indeed valid.

```
paClause2SQL (Do insDel' tgtTbl tgtExpr _motive) = λk →
  unsafeMkInsDelAtom tgtTbl insDel' tgtExpr:>>=λ_ →
  done k
```

A Nop and Blk clause will always succeed or fail, respectively. The simplifier which produces the PAclauses which are the input to this function should eliminate occurences of Blk.

```
paClause2SQL (Nop _motive) = done
paClause2SQL (Blk _motive) = notDone
```

The clause CHC ps indicates that precisely one of the clauses ps should be executed in order for this clause to succeed.

In order to produce SQL for such a clause, the paClause2SQL function is recursively applied to each subclause, with a fresh variable as the "done" variable – checkDone , whose value is initially false. If any clause succeeds, then the subsequent clauses are not called. After all clauses are executed, the original "done" variable – k – is set to true if checkDone is true.

The result for CHC ps where ps ranges from p1 to pN is a series of nested `IF` statements:

```
checkDone := false
< paClause2SQL p1 >
IF checkDone THEN (SELECT 0) ELSE
  < paClause2SQL2 p2 >
  IF checkDone THEN (SELECT 0) ELSE
    ...
      ELSE
        < paClause2SQL pN >
k := k OR checkDone
```

```
paClause2SQL (CHC ps _motive) = λk →
  NewRef sing (Just "checkDone") (Just T.false):>>=λcheckDone →
  let fin = SetRef k (T.sql T.Or (deref checkDone) (deref k)) in
  foldl (λdoPs p → paClause2SQL  p checkDone:>>=λ_ →
                   IfSQL (deref checkDone) SQLNoop doPs
        ) fin ps
```

The clause ALL ps indicates that precisely all of the clauses ps should be executed in order for this clause to succeed.

The generated SQL is very similair to the previous case in that it is composed of a series of nested `IF` statements. Again, a fresh "done" variable is created; in this case, before the execution of each clause, checkDone is set to false, and the clause must succeed and therefore set it to true. If after any one of the clauses, checkDone is not true, then this clause fails. It does so by setting the value of k to checkDone unconditionally after the execution of clauses until the first failing one, or until completion.

The resulting code is of the form

```
checkDone := false
< paClause2SQL p1 >
IF checkDone THEN
  checkDone := false
  < paClause2SQL2 p2 >
  IF checkDone THEN
```

```
    ...
      checkDone := false
      < paClause2SQL2 pN >
  ELSE (SELECT 0)
ELSE (SELECT 0)

k := checkDone
```

```
        paClause2SQL (ALL ps _motive) = λk →
          NewRef sing (Just "checkDone") Nothing:>>=λcheckDone →
          foldl (λdoPs p → SetRef checkDone T.false:>>=λ_ →
                           paClause2SQL p checkDone:>>=λ_ →
                           IfSQL (deref checkDone) doPs SQLNoop
                ) (SetRef k (deref checkDone)) ps
```

The clause GCH ps indicates that precisely one of the *guarded* clauses ps should be executed in order for this clause to succeed. A guarded clause is a PAclause with an expression, which may only be executed if the expression is not empty (or the negated version of the predicate, if the expression *is* empty).

```
        paClause2SQL (GCH ps _motive) = λk →
          NewRef sing (Just "checkDone") (Just T.false):>>=λcheckDone →
          let fin = SetRef k (T.sql T.Or (deref checkDone) (deref k)) in
          foldl (λdoPs (neg, gr, p) →
                   let nneg = case neg of { Ins → id; Del → T.sql T.Not }
```

The guard expression can have any type, but it must be a relation.

```
                       guardExpr = unsafeSQLValFromQuery $ expr2SQL' gr
                       guardExpr :: SQLVal ('SQLRel ('SQLRow '[ "dummy" :::
                       →'SQLAtom ]))
                   in
```

If the guard expression is non empty (optionally negated by nneg) then attempt to perform this clause, and if that fails, attempt the other clauses. If the guard predicate fails, then attempt the other clauses immediately.

```
                     IfSQL (nneg $ T.sql T.Exists guardExpr)
                       (paClause2SQL p checkDone:>>=λ_ →
                        IfSQL (deref checkDone) SQLNoop doPs
                       ) doPs
                ) fin ps

        paClause2SQL x = error $ "paClause2SQL: unsupported operation: " ++ show x
    in
```

Create a new reference for paClause2SQL, call it on the top-level PAclause and return the value of the reference at the end.

```
    NewRef sing (Just "checkDone") (Just T.false):>>=λcheckDone →
    paClause2SQL action checkDone:>>=λ_ →
    SQLRet (deref checkDone)
```

4.6 Communication Protocol

The EFA implementation needs to communicate with the front end to be able to run the generated SQL queries when a violation occurs.

- **Old communication protocol - PHP engine**
  In the existing version, Ampersand depends on PHP code to run the generated SQL on the database. However, this comes at the cost of human intervention, which results in manual maintenance when changes occur during development.

- **Current Ampersand development version:** All generated material is stored in JSON files — these could also be used to store EFA-generated SQL. It is then up to the front-end implementation to connect them to violations.

- **New Communication protocol - Stored Procedures**
  The developments teams of EFA has come to a conclusion that the best way of communicating with the front-end will be to use Stored Procedures(MyS16). These Stored Procedures provide the extra benefit of query optimization at compile time which results in better performance. While this is a suggested change, it will require changes to the existing Ampersand software in order for this idea to be successfully implemented. This anticipated change will be implemented in the near future.

Chapter 5

Testing

This section covers property testing using QuickCheck, system testing using a built-in test suite and MySQL WorkBench for SQL query testing. EFA's implementation uses dependent types which express application-specific program invariants that are beyond the scope of existing type systems. The tests that are implemented make two assumptions, the first being any element reused from the Ampersand system is already correct, and the second being that if the properties of a function are correctly implemented, when used in combination with assumption one that the outcome is correct.

5.1 Property Testing Using QuickCheck

Not all functions can be tested using QuickCheck due to their complexity (e.g. eca2SQL). All of EFA's modules rely on a few base modules, and testing the properties of the functions implemented in the base modules, by extension, tests all functions that rely on these core modules. These tests can be checked using stack and running ‘‘`quickCheck <function>`’’. The data types are presumed correct as they produce the correct results and are accepted by the Ampersand system. Furthermore, proof of EFA's TypedSQL was provided in software implementation section 4.4

Singletons.singKindWitness1

For every pair of types and their type representation, an isomorphism exists, where an isomorphism is a morphism $f : \ a \rightarrow b$ and there exists a morphism

$g : b \to a$ with $fg = 1_b$ and $gf = 1_a$. The default implementation uses unsafeCoerce and will only work if everything is correctly defined.

Singletons.sing2val and Singletons.val2sing

sing2val takes a singleton for some type and converts it to the value from which it was promoted (e.g. 'True becomes True). val2sing does the opposite, except that it must return its output existentially quantified. These functions should be witnesses to the isomorphism between singletons and their unlifted types, i.e it should be the case that sing2val . val2sing = id and val2sing . sing2val = id.

Singletons.(% ==)

This function implements equality between types by induction on their singletons. This function (when viewed as a binary relation) should be an equivalence, so therefore symmetry, transitive, and reflexive.

Transitivity $\forall\ a,\ b,\ c \in X : (aRb \wedge bRc) \Rightarrow\ aRc$

Reflexivity $\forall a \in\ X(aRa)$

Symmetry $\forall a, b, \in X\ (aRb \Rightarrow bRa)$

Utils.foldrProd

This function, and similarly Utils.foldlProd implement the natural fold over the Prod type, in the same way that the function foldr acts upon lists. This function must satisfy the property that it behaves the same as foldr does on lists. The function zipProd, mapProd, and foldlProd have similar properties.

5.1.1 System Testing Using Test suite

The test suite is built using the Cabal build system and can be enabled by running `cabal configure --enable-tests` and run by running the created test executable in the subfolder `dist/build/ampersand-test/ampersand-test` of the project root directory.

5.2 Manual Execution of EFA's SQL Queries

The queries produced by EFA were manually tested on a MySQL server; Ampersand relies on a MySQL database which can be installed as a part of xampp along with apache or simply on its own from the MySQL website (Cor). A simple guide is available on MySQL's website on how to install use WorkBench with xampp, if an MySQL server outside of xampp, it is highly advisable for these two components to be installed together. Furthermore, WorkBench (wor) can be found on MySQL's official website.

Xampp must be running Apache and MySQL for WorkBench to be able to connect to the database server.

Figure 5.1: 1. Shows that both and Apache and MySQL must be running at the same time. 2. The Admin button for mySQL opens phpAdmin which is an graphical interface for accessing the databases in an MySQL server. 3. Configuration button → services and ports → [MySQL] tab, check that the port is identical to the connected established for WorkBench. The standard is port 3306

Queries can be executed manually using WorkBench. Once Work Bench is running properly, it will very similar to screen shot provided.

Figure 5.2: The right-hand side shows the status of the server, and on the lower left-hand side under 'SCHEMAS', all the databases in current server is listed. Ampersand requires a local database and a password, both of which are 'ampersand'. The blue box in the top middle area provides information concerning the host.

Queries are executed by going to [File] → [New Query Tab], a new tab will open and one can manually type queries. The execution of these queries can be specified to the highlighted portion or everything. The pretty-printer outputs onto the command console what is given as output to the Ampersand its database.

Figure 5.3: This figure shows what the system sees in terms of specification, how ECA are structured, and how they are to be executed in SQL.

Appendix: Notes on Implementation Issues

This section contains a detailed list of concerns that came to light during the course of software implementation. Though some issues were addressed, others are left unresolved as there is no known solution for them. This list is meant to provide an overview of the various problems – and their solutions, if applicable – which were encountered during the development of EFA.

No kind equality One would naturally like to implement a singleton type as

```
data SingT (a::k) where
  SingApp :: SingT f → SingT a → SingT (f a)
```

in order to represent the singleton of the application of a type constructor. However, such a type, while possible to define, is essentially impossible to use, because the typechecker (as of GHC 7.10) does not know about the equality of kinds. Namely, when the typechecker sees an equality between *types*, it knows that the left-hand-side and right-hand-side are interchangeable in all contexts, but it does not derive the same information from an equality on *kinds*.

This feature is currently planned to be made available in GHC in version 8.0 (the next stable release) in the form of the the `TypeInType` extensions (Eis).

Non-injective type families Specifically, lack of injectivity in type functions means that GHC cannot infer an instantiation of a type variable that appears only under type families. Recently, Microsoft researchers have implemented a GHC modification that allows type functions to be annotated as injective and plan to make it available with the next stable release of the compiler (SJE15).

Functional dependencies are *not* equivalent to type families For example, given:

```
class ... ⇒ C a b |a → b
```

the typechecker knows that if the type a is known and there is a constraint C a b given, then the type b is known as well. In other words, the functional dependency ...| a → b asserts that the class C is actually a function.
However, you cannot write

```
injec :: (C a b, C a b') ⇒ b ≡ b'
```

as one would expect – this property is clearly true of a true math function – because of global incoherence caused by module boundaries. No solution is currently known (sof).

Exceeding complex type level invariants Certain invariants are difficult to state and prove, for example, the type of the `SELECT` SQL operator for the `TypedSQL` query language. The type for this function in particular was too difficult to even write down, let alone write an implementation which satisfied the invariants in the type. The solution to this problem is to use a vastly simplified term language, without losing the generality of expressing certain interesting expressions.

Singletons have a lot of boilerplate code The `singletons` package solves this problem by using Template Haskell to generate the boilerplate. However, TH is little more than string-based metaprogramming, so it is very clumsy to use, but more importantly nearly unmaintainable, as the internals of the Template Haskell library change with nearly every compiler version, and when code generated by TH breaks, it is very difficult to debug.

Our reimplemenation of the singleton pattern simply accepts the fate of having to write some boilerplate. However, the design of our implemenation ensures that for every type constructor, one only needs to write a constant amount of boilerplate, and then one can implement functions generically (for all singleton kinds) in terms of those functions. This is in constrast to the `singletons` library, which would require more boilerplate for every additional feature to be implemented on singleton types.

Defining a typed term language from an untyped one The TypedSQL query language encodes at the type level the type of the value which is represented by any given expression. However, this type is internally implemented as a wrapper around an abstract syntax tree which does *not* encode the type of the expression as a Haskell type. Users of the TypedSQL code must not be allowed to create typed expresions from untyped ones erroneously, that is, by assigning an incorrect type to the untyped expression.

The solution is abstraction barriers restricting access to the implementation of the TypedSQL query language. The approved way of constructing such expressions is through the use of the *typed* query functions provided by `TSQLCombinators`. However, in order to inject untyped expressions into typed ones when one is truly certain that the assigned type is correct, functions marked "unsafe" are still exported for the construction of typed expressions from untyped ones.

Relationship between SQL schemata and relation algebra Ampersand represents the underlying information system in terms of relation algebra expressions representing entities in the information system and relations between them. Ampersand maps relation algebra entities to SQL schemas in a such a way that there is minimal duplication of data. Therefore, the source and target of some relation may actually be columns of some larger table

For example the relations

```
r0 : A -> B [ FUN, INJ ]
r1 : A -> C [ FUN, INJ ]
```

can be modelled as a single table with three columns – A, B and C – where the first column contains an element a iff that element is a memeber of `I[A]`, the second and third columns contain a non-null value b or c iff $a\ r0\ b$ or $a\ r0\ c$ respectively.

Since the schemas of tables are encoded at the type level, these complex relations must also be computed at the type level.

The previously mentioned technique of dealing with exceeding complex type level invariants is also applied to this issue. Furthermore, in this case, we may also underspecify the result of the output type, when it cannot be fully determined (mainly due to limitations in the type) by leaving the output type existentially quantified, but allowing it to be accompanied by the proof of some interesting property.

Bibliography

[Cab] The Haskell Cabal guide. `https://www.haskell.org/cabal/`.

[Cla] Koen Claessen. Link to the QuickCheck library on hackage. `https://hackage.haskell.org/package/QuickCheck`.

[Cor] Oracle Corporation. MySQL Developer Zone. `http://dev.mysql.com/`.

[DP84] D.M. Weiss D.L. Parnas, P.C. Clements. The modular structure of complex systems. *Proceeding ICSE '84: Proceedings of the 7th international conference on Software engineering*, 1984.

[Eis] Richard Eisenberg. Adding kind equalities to GHC. `https://ghc.haskell.org/trac/ghc/wiki/DependentHaskell/Phase1`.

[Ell] John Ellson. `http://www.softpedia.com/get/Others/Miscellaneous/Graphviz.shtml`.

[EW12] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. *SIGPLAN Not.*, 47(12):117–130, September 2012.

[GHC] *The Glorious Glasgow Haskell Compilation System User's Guide*. Accessed: 2016-02-29.

[hac] Hackage. `https://hackage.haskell.org`.

[Jooa] Stef Joosten. Deriving functional specifications from business requirements with Ampersand. `http://icommas.ou.nl/wikiowi/images/e/e0/ampersand_draft_2007nov.pdf`.

[Joob] Stef Joosten. Link to the Ampersand Tarski's GitHub repository. `https://github.com/AmpersandTarski`.

[Joo07] Stef Joosten. Deriving Functional Specifications from Business Requirements with Ampersand. *CiteSeer*, 2007.

[JWM10] Stef Joosten, Lex Wedemeijer, and Gerard Michels. Rule based design. *Open Universiteit, Heerlen*, 2010.

[Lei15] Daan Leijen. wl-pprint package. `https://hackage.haskell.org/package/wl-pprint-1.2`, 2015. Accessed: 2016-02-28.

[LM13] Sam Lindley and Conor McBride. Hasochism: The pleasure and pain of dependently typed haskell programming. *SIGPLAN Not.*, 48(12):81–92, September 2013.

[myS] *MySQL 5.7 Reference Manual*.

[MyS16] Mysql reference manual. `https://dev.mysql.com/doc/refman/5.7/en/`, 2016. Accessed: 2016-02-28.

[NDA16] Ulf Norell, N. A. Danielsson, and Andreas Abel. The Agda Wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`, 2016. Accessed: 2016-02-28.

[Par72] D.L. Parnas. On the criteria to ne used in decomposing systems into modules. *Communications of the ACM*, 1972.

[SJE15] Jan Stolarek, Simon Jones, and Richard Eisenberg. Injective Type Families for Haskell. *Microsoft Research*, 0(0):118–128, sept 2015.

[sof] Equality from a functional dependency. `https://stackoverflow.com/questions/34645745#35413064`.

[SPJ02] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002.

[Swi08] Wouter Swierstra. Data types la carte. *Cambridge University Press*, 2008.

[Wad15] Philip Wadler. Propositions as types. *Communications of the ACM*, 58(12):7584, dec 2015.

[Whe15] Jake Wheat. simple-sql-parser. `https://hackage.haskell.org/package/simple-sql-parser-0.4.1`, 2015. Accessed: 2016-02-28.

[wor] MySQL Workbench reference manual. `https://dev.mysql.com/doc/workbench/en/`.

[YWC+12] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM.