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 24, 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. However, the Exec-engine relies on manual formulation of repair actions in an error-prone setup that creates PHP-embedded SQL commands from string fragments, as opposed to using the automatically-generated Event-Condition-Action (ECA) rules that Ampersand already produces. The EFA project aims to ultimately make the Exec-engine superfluous by correctly translating the 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- and type-correct. Furthermore, EFA has a high degree of modularity as it only relies on Ampersand's essential components.

Contents

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, a section on software development, which describes what each module of EFA contains, and lastly, a section on software testing.

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(4, 15).

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 and notes on issues that came up during the course of EFA's implementation. [**WK:** *I hope there will also be literate source code in the appendix.*]

1.2 Ampersand

[**WK:** *TeX source becomes more maintainable if you break lines at the ends of grammatical units, in particular of sentences.*]

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 natural language used in business process into requirement specifications (**?**) [**WK:** *This is not recognisable as a reference. Choose a different bibliographystle?*]. 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 [**WK: replace:** if with: whether] there are discrepancies between the two sets. Discrepancies often occur when changes are made to the set of data Ampersand is given or to the set of rules that as associated with them [**WK:** *Changes to the set of rules? "Regenerage database"!*]. 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 [**WK: replace:** wrongful transactions with: transactions that would violate the business rules].

1.2.1 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 [**WK: replace:** windows with: MS Windows], there is an executable available, but on Linux and Mac operating systems it requires installing [**WK: ?: replace:** stack with: "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.

The Haskell Platform

Ampersand's most current version requires the Haskell Platform 7.10 . It is used to compile 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 Ampersand Tarski's Github [**WK: insert:** .] [**WK:** *Not optically recognisable as link in the PDF — this is a global problem.*]

Graphviz

Graphviz is an open source graph visualization software that Ampersand uses to generate artifacts [**WK: replace:** , with: ;] 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

[**WK:** *This is part of the Haskell Platform.*]
The Cabal System 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. (2)

The Haskell Tool Stack

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

QuickCheck

QuickCheck 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 [**WK: insert:** ,] 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 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[**WK: replace:** , with: ;] in this project it is used to manually test queries. Through various stages in development it is used for data modeling.

1.3 Naming Conventions and Terminology

ECA Stands for Event-Condition Action. The rule structure used for data bases and commonly used in [**WK: replace:** market ready with: 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.

ADL Stands for "Abstract Data Language" ((8, 13))[**WK:** *Look at what LaTeX produces here!*]. From a given set of formally defined business requirements, Ampersand generates a functional specification consisting of a data model, a service catalog, a formal specification of the services, and a function point analysis. An ADL script acts as an input for Ampersand. An ADL file consists of a plain ASCII text file.

Ampersand Ampersand is the name of this project[**WK:** *'this project"???*]. It is used to refer to both the method of generating functional specification from formalized business requirements, and the software tool which implements this method.

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

Business rules See *Business Requirements.*

EFA Stands for "ECA (see above) for Ampersand". This term is used to refer to the contribution of this project.

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

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.

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

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

1.4 Purpose of the project

$\boxed{\text{[\textbf{YS:}}}$ *Added the "AMMBR section" based on comments from Dr. Kahl. He has verified the section, I'll follow up with a proof read and other section of this document* $\boxed{]}$

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 represents the EFA project and where it fits in the current version of Ampersand. $\boxed{\text{[\textbf{JG:}}}$ *Could you mention what "this problem" is? When you read it, 'this' seems to refer back to something specific and I'm not sure what, it seems confusing* $\boxed{]}$ This problem is addressed in Ampersand using AMMBR (7). The role of the AMMBR method in Ampersand automatically generate correction methods that help maintain these business rules. In AMMBR, human involvement is only limited to representing rules (in the ADL files). While the Ampersand software still remains in development phase, there is no way of checking the correctness of the AMMBR method at a low level. Previously, the Exec Engine was used to maintain these business rules, that required direct injection of SQL hand-written as PHP strings.

The EFA project, an extension to the Ampersand system, allows us to evaluate the correctness of the Action according to the ECA rules. This will allow us to automatically rehabilitate existing system data while maintaining the information system according to user specifications. Furthermore, EFA permits us to automate the restoration of system invariants, and create a program which helps Ampersand to make the most efficient choice regarding how to proceed on a case-by-case basis. The ECA rules, which act as an input to the EFA project, are translated into human-readable SQL. This can later be viewed in the command line using the `‘‘--print-eca-info’’` flag, which is written out in one of the artifacts. The generated SQL queries, not only allow the correctness of AMMBR to be checked, but also help in the identification of patterns and unreachable states. Therefore, EFA will be an essential tool for testing the AMMBR algorithm.

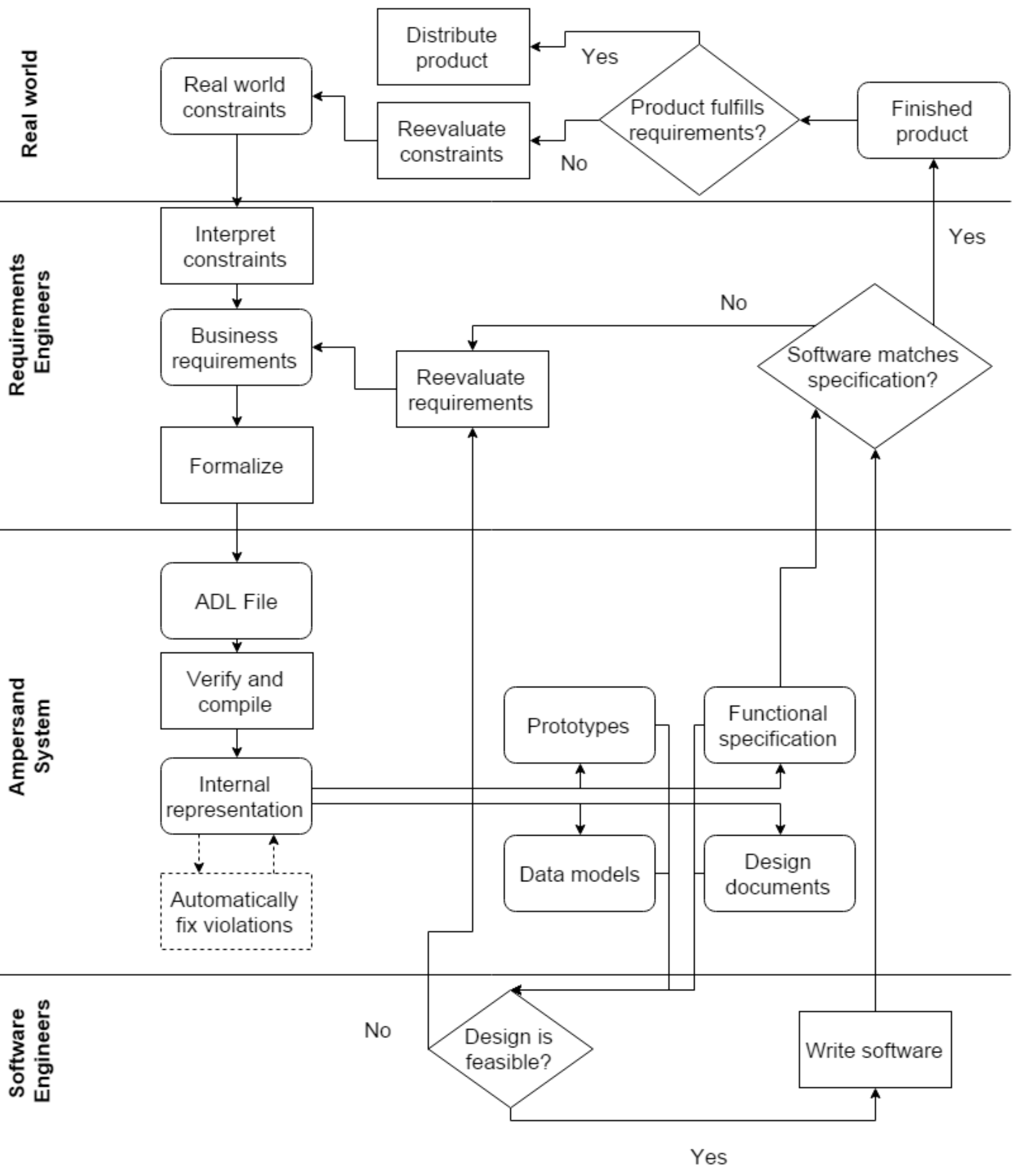


Figure 1: Business process diagram representing EFA project represented as a dashed box

1.5 The Stakeholders and the intended audience

The stake holders of Ampersand are:

- **Ampersand Designers**: Responsible for maintaining and developing Ampersand.
- **The Customer**: The end users of Ampersand will benefit from the EFA project. This will decreases the amount of time Ampersand users spend manually inserting PHP code to restore system invariants.

This document is designed to help introduce new Ampersand users to EFA (ECA rules for Ampersand). It provides a basic structure that allows individuals to quickly access the information they seek.

2 Functional Requirements

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

Requirement	P2
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	P3
Description	Generated SQL queries must be correctly implemented.
Rationale	Ampersand uses a MySQL database, for queries to be recognized and executed they must be error free.
Originator	Ampersand Creators
Test Cases	Using MySQL WorkBench queries are manually executed and checked for errors.
Priority	4 - High

3 System Architecture and Module Hierarchy

3.1 System Architecture

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

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

3.2 External Libraries

No addition dependencies are required outside of those that Ampersand already uses. The EFA project depends on the following Libraries:

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 (9), 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. (19)

wl-pprint
The wl-pprint library(10) is a pretty printer based on the pretty printing com-

binators. EFA uses this library in combination with the simple-sql-pretty to output the SQL statements in a human readable format.

3.3 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 (12). 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 (11) (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 (20). 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 $\text{X} : \text{A} \to \text{B} \to \ldots \to \text{Type}$ to denote a regular data type, and $\text{Y} : \text{A} \to \text{B} \to \ldots \to \text{Kind}$ to denote a datatype which is used exclusively as a kind.

Description of Dependant 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. $(\text{x} : \text{A}) \to \text{B}$ 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 $(\mathrm{x} : \mathrm{A}) \rightarrow \mathrm{B}$ in our model is `forall (x :: A) . SingT x -> B`. `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 3.5.6.

The syntax $\forall\ (\mathrm{x} : \mathrm{A}) \rightarrow \mathrm{B}$ 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: $\forall\ (\mathrm{x} : \mathrm{A}) \rightarrow \mathrm{P\ x}$ and $\forall\ \mathrm{x} \rightarrow \mathrm{P\ x}$ 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 $\mathrm{Show\ a} \rightarrow \mathrm{a} \rightarrow \mathrm{String}$. 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 $\Rightarrow$ syntax.

Existential quantification

The type $\exists\ (\mathrm{x} : \mathrm{A})\ (\mathrm{P\ x})$ 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 $\mathrm{Ty} : \mathrm{K} = \mathrm{X}$, 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 $\mathrm{Ty} : \mathrm{K}$ **where** $\mathrm{Ty} \ldots = \ldots$.

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 $\mathrm{Ty} : \mathrm{Type}$, 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". Furthermore, it is not a simple matter of convention - the use of this constructor in expressions will be strictly forbidden by Haskell.

- 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`

Choice of form is 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.

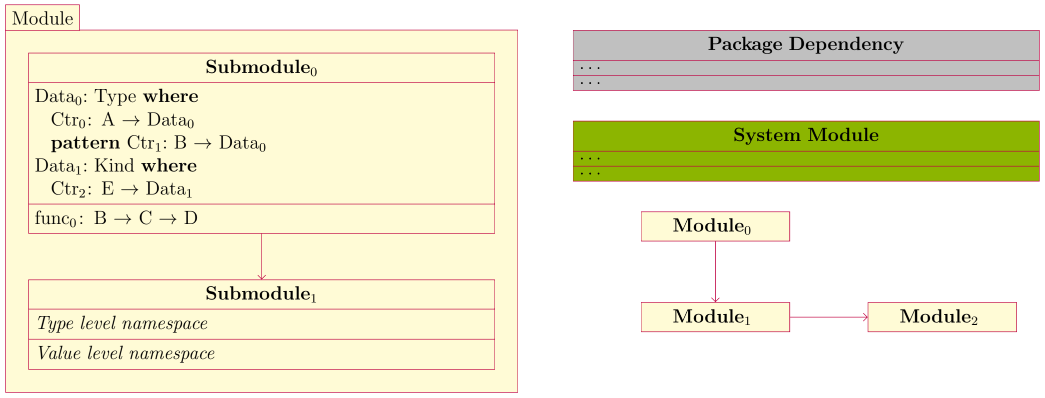


Figure 2: Example of module diagram syntax

3.4 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

3.5 Module Hierarchy

This section contains a hierarchal 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 3. 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 (13). 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 is 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.

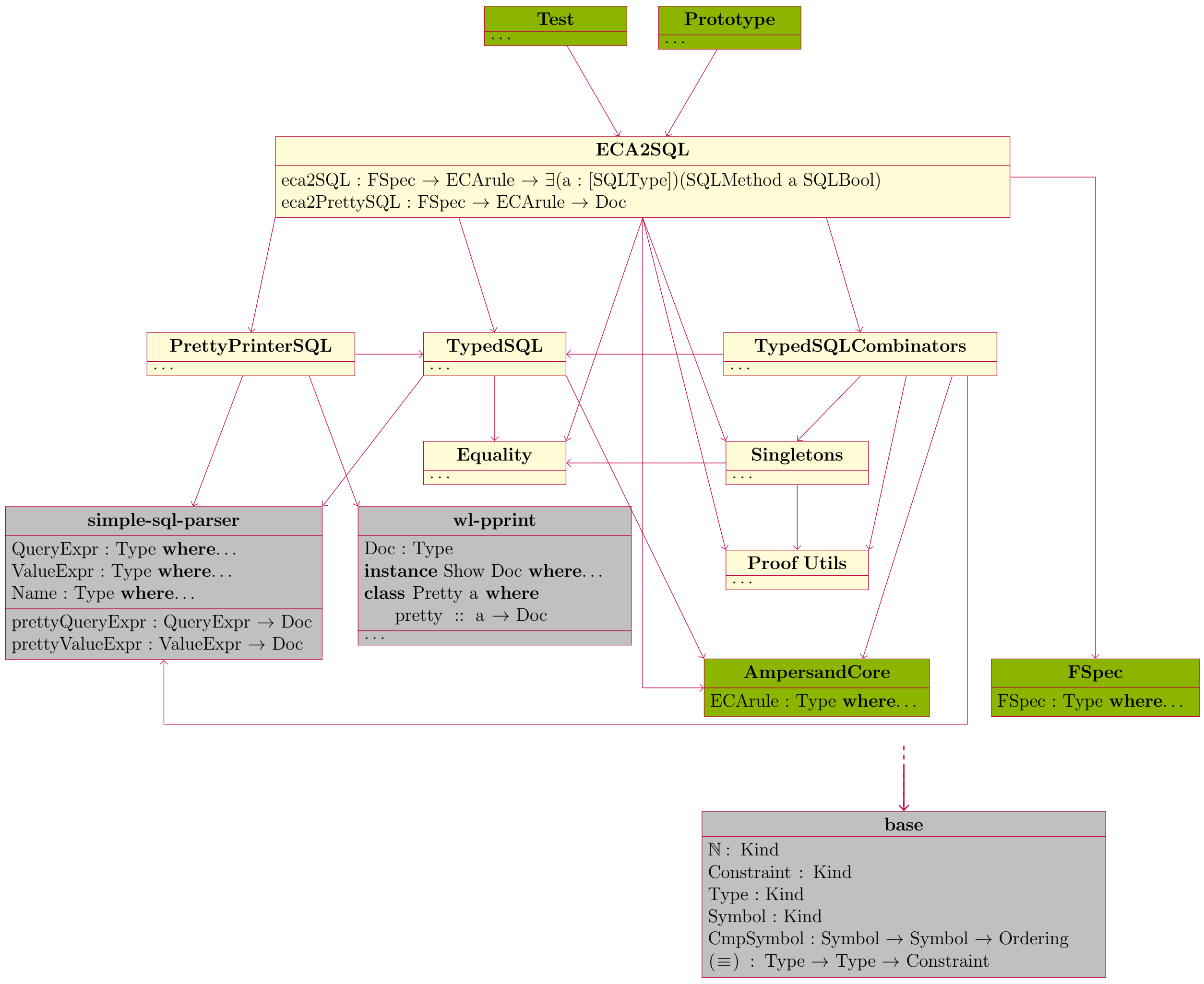
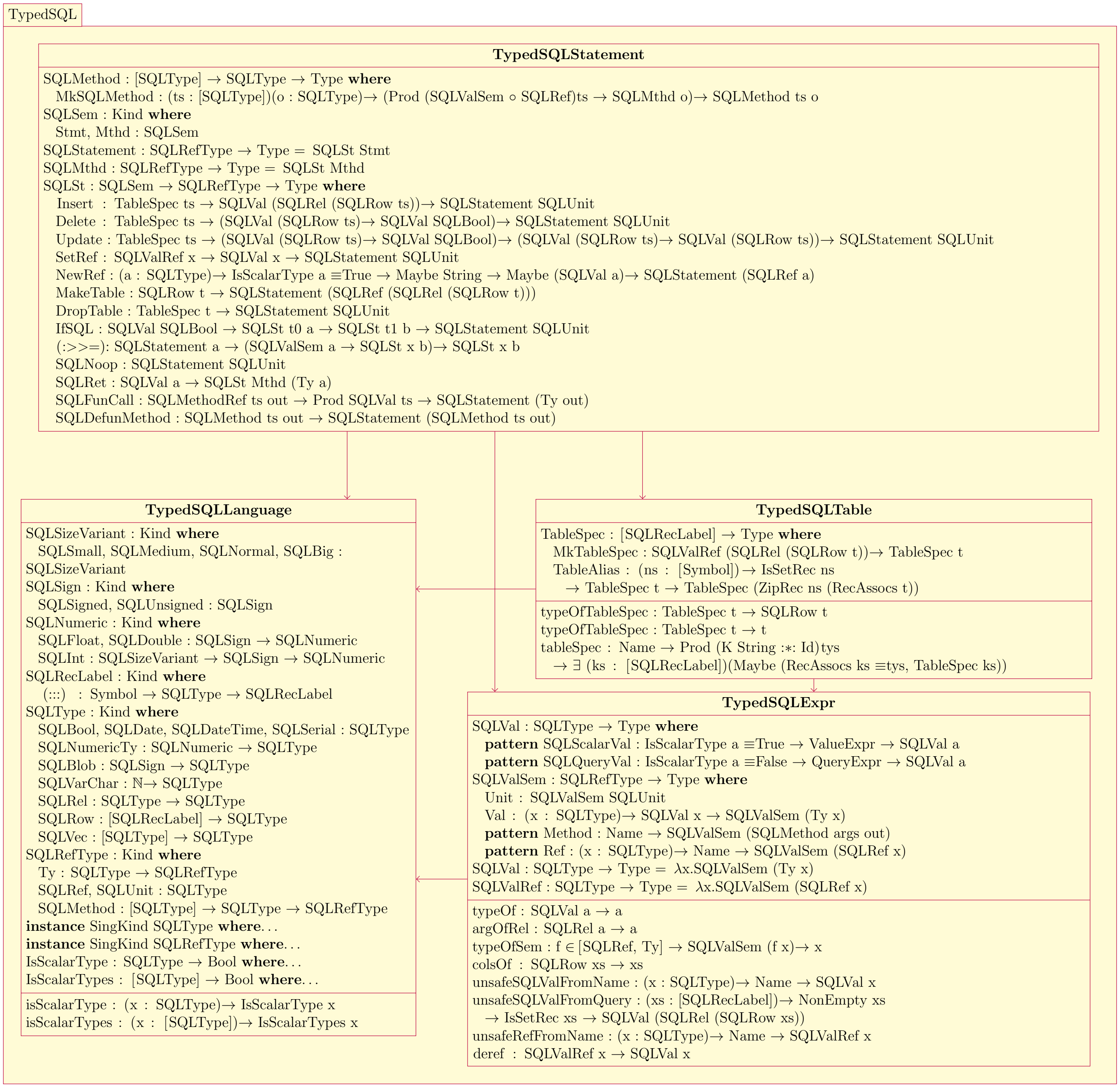

Test
. . .
Prototype
. . .
ECA2SQL
eca2SQL : FSpec → ECArule → ∃(a : [SQLType])(SQLMethod a SQLBool)
eca2PrettySQL : FSpec → ECArule → Doc
PrettyPrinterSQL
. . .
TypedSQL
. . .
TypedSQLCombinators
. . .
Equality
. . .
Singletons
. . .
Proof Utils
. . .
simple-sql-parser
QueryExpr : Type where. . .
ValueExpr : Type where. . .
Name : Type where. . .
prettyQueryExpr : QueryExpr → Doc
prettyValueExpr : ValueExpr → Doc
wl-pprint
Doc : Type
instance Show Doc where. . .
class Pretty a where
pretty :: a → Doc
. . .
AmpersandCore
ECArule : Type where. . .
FSpec
FSpec : Type where. . .
base
ℕ : Kind
Constraint : Kind
Type : Kind
Symbol : Kind
CmpSymbol : Symbol → Symbol → Ordering
(≡) : Type → Type → Constraint

Figure 3: Module diagram for EFA as a whole



```
SQLMethod : [SQLType] → SQLType → Type where
  MkSQLMethod : (ts : [SQLType])(o : SQLType)→ (Prod (SQLValSem ∘ SQLRef)ts → SQLMthd o)→ SQLMethod ts o
SQLSem : Kind where
  Stmt, Mthd : SQLSem
SQLStatement : SQLRefType → Type = SQLSt Stmt
SQLMthd : SQLRefType → Type = SQLSt Mthd
SQLSt : SQLSem → SQLRefType → Type where
  Insert : TableSpec ts → SQLVal (SQLRel (SQLRow ts))→ SQLStatement SQLUnit
  Delete : TableSpec ts → (SQLVal (SQLRow ts)→ SQLVal SQLBool)→ SQLStatement SQLUnit
  Update : TableSpec ts → (SQLVal (SQLRow ts)→ SQLVal SQLBool)→ (SQLVal (SQLRow ts)→ SQLVal (SQLRow ts))→ SQLStatement SQLUnit
  SetRef : SQLValRef x → SQLVal x → SQLStatement SQLUnit
  NewRef : (a : SQLType)→ IsScalarType a ≡True → Maybe String → Maybe (SQLVal a)→ SQLStatement (SQLRef a)
  MakeTable : SQLRow t → SQLStatement (SQLRef (SQLRel (SQLRow t)))
  DropTable : TableSpec t → SQLStatement SQLUnit
  IfSQL : SQLVal SQLBool → SQLSt t0 a → SQLSt t1 b → SQLStatement SQLUnit
  (:>>=): SQLStatement a → (SQLValSem a → SQLSt x b)→ SQLSt x b
  SQLNoop : SQLStatement SQLUnit
  SQLRet : SQLVal a → SQLSt Mthd (Ty a)
  SQLFunCall : SQLMethodRef ts out → Prod SQLVal ts → SQLStatement (Ty out)
  SQLDefunMethod : SQLMethod ts out → SQLStatement (SQLMethod ts out)
```

```
SQLSizeVariant : Kind where
  SQLSmall, SQLMedium, SQLNormal, SQLBig :
SQLSizeVariant
SQLSign : Kind where
  SQLSigned, SQLUnsigned : SQLSign
SQLNumeric : Kind where
  SQLFloat, SQLDouble : SQLSign → SQLNumeric
  SQLInt : SQLSizeVariant → SQLSign → SQLNumeric
SQLRecLabel : Kind where
  (:::) : Symbol → SQLType → SQLRecLabel
SQLType : Kind where
  SQLBool, SQLDate, SQLDateTime, SQLSerial : SQLType
  SQLNumericTy : SQLNumeric → SQLType
  SQLBlob : SQLSign → SQLType
  SQLVarChar : ℕ→ SQLType
  SQLRel : SQLType → SQLType
  SQLRow : [SQLRecLabel] → SQLType
  SQLVec : [SQLType] → SQLType
SQLRefType : Kind where
  Ty : SQLType → SQLRefType
  SQLRef, SQLUnit : SQLType
  SQLMethod : [SQLType] → SQLType → SQLRefType
instance SingKind SQLType where...
instance SingKind SQLRefType where...
IsScalarType : SQLType → Bool where...
IsScalarTypes : [SQLType] → Bool where...
---
isScalarType : (x : SQLType)→ IsScalarType x
isScalarTypes : (x : [SQLType])→ IsScalarTypes x
```

```
TableSpec : [SQLRecLabel] → Type where
  MkTableSpec : SQLValRef (SQLRel (SQLRow t))→ TableSpec t
  TableAlias : (ns : [Symbol])→ IsSetRec ns
    → TableSpec t → TableSpec (ZipRec ns (RecAssocs t))
---
typeOfTableSpec : TableSpec t → SQLRow t
typeOfTableSpec : TableSpec t → t
tableSpec : Name → Prod (K String :∗: Id)tys
  → ∃ (ks : [SQLRecLabel])(Maybe (RecAssocs ks ≡tys, TableSpec ks))
```

```
SQLVal : SQLType → Type where
  pattern SQLScalarVal : IsScalarType a ≡True → ValueExpr → SQLVal a
  pattern SQLQueryVal : IsScalarType a ≡False → QueryExpr → SQLVal a
SQLValSem : SQLRefType → Type where
  Unit : SQLValSem SQLUnit
  Val : (x : SQLType)→ SQLVal x → SQLValSem (Ty x)
  pattern Method : Name → SQLValSem (SQLMethod args out)
  pattern Ref : (x : SQLType)→ Name → SQLValSem (SQLRef x)
SQLVal : SQLType → Type = λx.SQLValSem (Ty x)
SQLValRef : SQLType → Type = λx.SQLValSem (SQLRef x)
---
typeOf : SQLVal a → a
argOfRel : SQLRel a → a
typeOfSem : f ∈ [SQLRef, Ty] → SQLValSem (f x)→ x
colsOf : SQLRow xs → xs
unsafeSQLValFromName : (x : SQLType)→ Name → SQLVal x
unsafeSQLValFromQuery : (xs : [SQLRecLabel])→ NonEmpty xs
  → IsSetRec xs → SQLVal (SQLRel (SQLRow xs))
unsafeRefFromName : (x : SQLType)→ Name → SQLValRef x
deref : SQLValRef x → SQLVal x
```

Figure 4: Module diagram for TypedSQL

3.5.1 TypedSQL

The module hierarchy for the TypedSQL module is shown in figure 4. 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 distinguishment. 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 (3). 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 (3).

`:>>=` 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.

3.5.2 TypedSQLCombinators

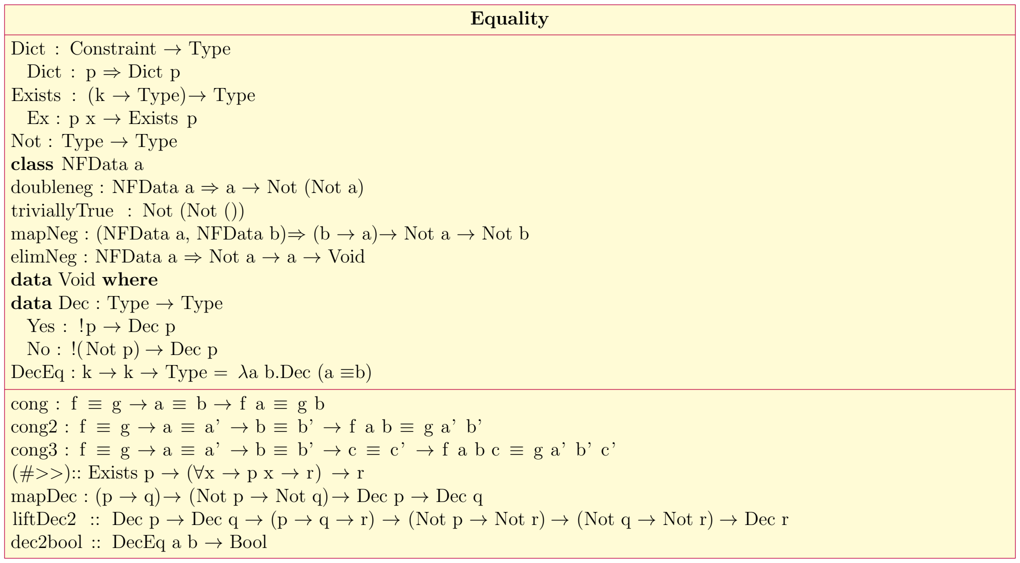
The module TypedSQLCombinators, whose members are given in figure 5, implements a subset of primitive SQL functions on top of the TypedSQL expression type. The data type PrimSQLFunction encodes the specification of each function; the type and semantics of each function is that of the corresponding function in MySQL (refer to the MySQL manual (3) 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.

TypedSQLCombinators

```
PrimSQLFunction : [SQLType] → SQLType → Type
  PTrue, PFalse : PrimSQLFunction [] SQLBool
  Not : PrimSQLFunction [ SQLBool ] SQLBool
  Or, And : PrimSQLFunction [ SQLBool, SQLBool ] SQLBool
  In, NotIn : PrimSQLFunction [ a, SQLRel a ] SQLBool
  Exists : PrimSQLFunction [ SQLRel a ] SQLBool
  GroupBy : PrimSQLFunction [ SQLRel a, a ] (SQLRel (SQLRel a))
  SortBy : PrimSQLFunction [ SQLRel a, a ] (SQLRel a)
  Max, Min, Sum, Avg : IsSQLNumeric a → PrimSQLFunction [ SQLRel a ] a
  Alias : (RecAssocs ts : RecAssocs ts') → PrimSQLFunction [ SQLRel (SQLRow ts)] (SQLRel (SQLRow ts'))
```

```
primSQL : ∀(args : [SQLType])(out : SQLType)→ PrimSQLFunction args out → Prod SQLVal args → SQLVal out
(!) : (xs : [SQLType])(i : Symbol)→ LookupRec xs i ≡ r → SQLVal (SQLRow xs)→ SingT i → SQLVal r
(!) : (xs : [SQLType])(i : Symbol)→ LookupRec xs i ≡ r → SQLVal (SQLRel (SQLRow xs))→ SingT i → SQLVal r
(!) : (xs : [SQLType])(i : ℕ) → LookupIx t i ≡ r → SQLVal (SQLVec t)→ SingT i → SQLVal r
```

Figure 5: Module diagram for TypedSQLCombinators



```
Equality
Dict : Constraint → Type
  Dict : p ⇒ Dict p
Exists : (k → Type)→ Type
  Ex : p x → Exists p
Not : Type → Type
class NFData a
doubleneg : NFData a ⇒ a → Not (Not a)
triviallyTrue : Not (Not ())
mapNeg : (NFData a, NFData b)⇒ (b → a)→ Not a → Not b
elimNeg : NFData a ⇒ Not a → a → Void
data Void where
data Dec : Type → Type
  Yes : !p → Dec p
  No : !(Not p) → Dec p
DecEq : k → k → Type = λa b.Dec (a ≡b)
cong : f ≡ g → a ≡ b → f a ≡ g b
cong2 : f ≡ g → a ≡ a' → b ≡ b' → f a b ≡ g a' b'
cong3 : f ≡ g → a ≡ a' → b ≡ b' → c ≡ c' → f a b c ≡ g a' b' c'
(#>>):: Exists p → (∀x → p x → r) → r
mapDec : (p → q)→ (Not p → Not q)→ Dec p → Dec q
liftDec2 :: Dec p → Dec q → (p → q → r) → (Not p → Not r) → (Not q → Not r) → Dec r
dec2bool :: DecEq a b → Bool
```

Figure 6: Module diagram for Equality

3.5.3 Equality

The Equality module (6) 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 (18); however, this is not quite sound in Haskell, because every type is inhabited by $\perp$, which corresponds to `undefined`, an exception, or non-termination. Due to laziness, an unevaluated $\perp$ 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

$\perp$, as opposed to $\perp$ wrapped in a constructor, for example Just $\perp$. 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 $\rightarrow \perp$, where the type $\perp$ can be represented by any uninhabited type, typically calledVoid. However, the normal form of a function can contain any number of $\perp$ 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.

3.5.4 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.

PrettySQL

```
instance Pretty (SQLTypeS x) where...
instance Pretty (SQLVal x) where...
instance Pretty (SQLValSem x) where...
instance Pretty (SQLSt k x) where...
instance (str ≡ String) ⇒ Pretty (str , SQLMethod args out) where...
```

Figure 7: Module diagram for PrettySQL

3.5.5 Proof Utils

The ProofUtils modules, shown in figure 8 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 (17).

`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 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 boolen conjunction, with the usual semantics.

```
ProofUtils
Prod : (f : k → Type)→ (xs : [k]) → Type
  PNil : Prod f []
  PCons : f x → Prod f xs → Prod f (x : xs)
Sum : (f : k → Type)→ (xs : [k]) → Type
  SHere : f x → Sum f (x : xs)
  SThere : Sum f xs → Sum f (x : xs)
class All (c : k → Constraint) (xs : [k]) where
  mkProdC : Proxy c → (∀a → c a ⇒ p a) → Prod p xs
instance All (c : k → Constraint) [] where. . .
instance (All c xs, c x) ⇒ All c (x : xs) where. . .
Elem : k → [k] → Type where
  MkElem : Sum (≡x)xs → Elem x xs
IsElem : (x : k) → (xs : [k]) → Constraint where. . .
AppliedTo : (x : k) → (f : k → Type)→ Type
  Ap : f x → x ‘AppliedTo‘ f
(:∘ :) : (f : k1 → Type)→ (g : k0 → k1) → (x : k0) → Type
  Cmp : f (g x) → (:∘ :) f g x
(:∗:) : (f : k0 → Type)→ (g : k0 → Type)→ (x : k0) → Type
  (:∗:) : f x → g x → (:∗:) f g x
K : (a : Type)→ (x : k) → Type
  K : a → K a x
Id : (a : Type)→ Id a
  Id : a → Id a
Flip : (f : k0 → k1 → Type)→ (x : k1) → (y : k0) → Type
  Flp : f y x → Flip f x y
(&&): (x : Bool) → (y : Bool) → Bool where. . .
RecAssocs : [RecLabel a b] → [b] where. . .
RecLabels : [RecLabel a b] → [b] where. . .
_IsSetRec : [RecLabel a b] → [a] → Constraint where. . .
IsSetRec : [RecLabel a b] → Constraint where. . .
SetRec : [a] → [RecLabel a b] → Type
SetRec : [RecLabel a b] → Type = SetRec []
LookupRecM : [RecLabel Symbol k] → Symbol → Maybe k where. . .
ZipRec : [a] → [b] → [RecLabel a b] where. . .
IsNotElem : [k] → k → Constraint where. . .
NotElem : [k] → k → Type
And : (xs : [Bool]) → Bool where. . .
NonEmpty : (xs : [k]) → Constraint where
  NonEmpty (x : xs)= ()
(|&&): (a : Bool) → (b : Bool) → a && b
openSetRec : ∀(xs : [RecLabel k k’]) r0 → SingKind k ⇒ SetRec xs → (IsSetRec xs ⇒ r0)→ r0
decNotElem : ∀(xs : [a]) x → SingKind a ⇒ xs → x → Dec (NotElem xs x)
decSetRec : ∀ (xs : [RecLabel a b]) → SingKind a ⇒ xs → Dec (SetRec xs)
lookupRecM : ∀(xs : [RecLabel Symbol k])(x : Symbol)→ xs → x → LookupRecM xs x
and : (xs : [Bool]) → And xs
compareSymbol : (x : Symbol)→ (y : Symbol)→ CmpSymbol x y
prod2sing : ∀ (xs : [k]) → SingKind k ⇒ Prod SingT xs → SingT xs
sing2prod : ∀ (xs : [k]) → SingKind k ⇒ SingT xs → Prod SingT xs
foldrProd : ∀ acc (f : k → Type) xs → acc → (∀q → f q → acc → acc) → Prod f xs → acc
foldlProd : ∀ acc (f : k → Type) xs → acc → (∀q → f q → acc → acc) → Prod f xs → acc
mapProd : ∀(f : k → Type) g → (∀x → f x → g x) → Prod f xs → Prod g xs
foldrProd’ : ∀ (f : k → Type) xs1 → (∀x xs → f x → Prod g xs → Prod g (x : xs)) → Prod f xs1 → Prod g xs1
someProd : [Exists f] → Exists (Prod f)
```

Figure 8: Module diagram for ProofUtils

3.5.6 Singletons

The Singletons module (figure 9) 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 (5). 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 (16), 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 9: Module diagram for Singletons

3.6 Key Algorithm

The key algorithm for the EFA project is AMMBR (9). AMMBR is a method that allow organizations to build information systems that comply to their business requirements in a provable manner. This algorithm is implemented in Ampersand and is responsible for translating the business requirements into ECA rules. These ECA rules contain information on how to fix any data violation and are translated into SQL queries in our EFA project.

3.7 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.

- **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(14). 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.

4 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.

4.1 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 3.5

Module	Function	Test Description

Singletons	singKindWitness1	For every pair of types and their type representation, an isomorphism exists, where an isomorphism is a morphism $f: \ a \to b$ 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 val2sing	sing2val takes a singleton and returns a value, while val2sing does the opposite, composing these two functions together should return the same output as the given input. $val2sing\ x(sing2val\ x) \equiv\ x$ $sing2val\ x(val2sing\ x) \equiv\ x$
Singletons	% ==	Tests for equality by checking for transitivity, symmetry and reflexivity. Transitive relation: $\forall\ a,\ b,\ c \in X\ :\ (aRb \wedge bRc) \Rightarrow\ aRc$ Reflexive relation: $\forall a \in\ X(aRa)$ Symmetry relation: $\forall a, b, \in X\ \ (aRb \Rightarrow bRa)$
Utils	prod2sing sing2prod	prod2sing is the opposite of sing2prod using this logic composing the two functions should return the same output irregardless of order. $prod2sing\ x(sing2prod\ x) \equiv\ x$ $sing2prod\ x(prod2sing\ x) \equiv\ x$
Utils	prod2sing	prod2sing is the opposite of sing2prod using this

	sing2prod	logic composing the two functions should return the same output irregardless of order. $prod2sing\ x(sing2prod\ x) \equiv\ x$ $sing2prod\ x(prod2sing\ x) \equiv\ x$
Utils	prod2sing sing2prod	prod2sing is the opposite of sing2prod using this logic composing the two functions should return the same output irregardless of order. $prod2sing\ x(sing2prod\ x) \equiv\ x$ $sing2prod\ x(prod2sing\ x) \equiv\ x$
Utils	foldrProd foldlProd zipProd mapProd	These functions can be tested using conventional methods as they are analogous to the built-in haskell functions.

4.2 System Testing Using Test suite

The test suite runs by using the cabal system and running `‘‘cabal configure --enable-tests’’` this test suite was built specifically for EFA and more details are provided in the literal haskell script located in EFA's github repository.

4.3 MySQL WorkBench for Query Testing

References

[1] Equality from a functional dependency.

[2] Hackage.

[3] Corporation, O. *MySQL 5.7 Reference Manual*.

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

[5] Eisenberg, R. A., and Weirich, S. Dependently typed programming with singletons. *SIGPLAN Not. 47*, 12 (Sept. 2012), 117–130.

[6] Jan Stolarek, S. J., and Eisenberg, R. Injective type families for haskell. *Microsoft Research 0*, 0 (2015), 118–128.

[7] Joosten, S. Deriving functional specications from business requirements with ampersand.

[8] Joosten, S. Deriving Functional Specifications from Business Requirements with Ampersand. *CiteSeer* (2007).

[9] Joosten, S. Deriving Functional Specifications from Business Requirements with Ampersand. *CiteSeer* (2007).

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

[11] Lindley, S., and McBride, C. Hasochism: The pleasure and pain of dependently typed haskell programming. *SIGPLAN Not. 48*, 12 (Sept. 2013), 81–92.

[12] Norell U., Danielsson N. A., A. A. The Agda Wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`. Accessed: 2016-02-28.

[13] of the University of Glasgow, T. U. C. *The Glorious Glasgow Haskell Compilation System User's Guide.* Accessed: 2016-02-29.

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

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

[16] Sheard, T., and Jones, S. P. Template meta-programming for haskell. *SIGPLAN Not. 37*, 12 (Dec. 2002), 60–75.

[17] Swierstra, W. Data types la carte. *Cambridge University Press* (2008).

[18] Wadler, P. Propositions as types. *Communications of the AC 58*, 12 (2015), 7584.

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

[20] Yorgey, B. A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., and Magalhães, J. P. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (New York, NY, USA, 2012), TLDI '12, ACM, pp. 53–66.

Appendix

This section contains notes on installing and using MySQL WorkBench to testing MySQL queries, as it is the only piece of software that is not covered in the Ampersand installation guide. In addition, there is a brief section on some implementation issues that were encountered during the course of implementing EFA.

SQL testing with MySQL server

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. The list is provided as a comprehensive overview of the thought process that went into the implementation of EFA, and the reasoning behind the design decisions which were made.

- **No kind equality.** Singletons cannot be implemented as widely as we would like them to be; essentially such a type is impossible:

  ```
  data X (a::k) where
  XF :: X a → X f → X (f a)
  ```

- **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 (6).

- **Functional dependencies are *NOT* equivalent to type families.** For example, given:

  ```
  class ∘∘∘ ⇒ C a b c| a → b

  you cannot write

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

  This is because it is simply not true, no solution for this is currently known (1).

- **Types as propositions.** This is assuming that one exists; simply put, it describes a correspondence between a given logic and a given programming language, where we have "simplification of proofs as evaluation of programs" (18).

- **Certain invariants are difficult to state and prove.** The solution implemented for this problem was to vastly simply term language and to use informal correctness reasoning.

- **Singletons have a lot of boilerplate code.** Each model requires a constant amount of boilerplate code for each constructor.

- **Defining a typed term from untyped ones.** This had to be done in a semi-safe way; the solution taken was to use abstraction barriers cautiously to construct unsafe data types.

- **The complex relationship between SQL schemata and declarations.** This is specific to the long tables used in Ampersand's database. Database schemata must be converted into a type level representations along with all necessary proofs. The solution implemented for this was to under specify the result type of the existentially quantified function. Rather than fabricating a proof using a function P and a valid input x, a decision procedure for P is written and the functions which require such a proof will check that it is true, or throw an error. These functions are only used internally – they are not part of the interface; such functions would only be called when the input is known to satisfy P. This is something that the programmer is able to reason about but not the compiler.