

Event control action rules for Ampersand

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

Supervised by: Dr. Wolfram Kahl



Department of Computing and Software
McMaster University
Ontario, Canada
April 22, 2016

Abstract

Ampersand Tarski is a tool used to produce functional software documents based on business process requirements. At times, logical discrepancies arise when system changes occur which violate the restrictions set forth by the user. When a system violation occurs, one of two things can happen: the change that is meant to take place is adjusted so it no longer violates the restrictions or the changes are discarded. The purpose of Event condition action rules for Ampersand (EFA) was to replace the exec-engine that is currently used to deal with violations; unlike the exec-engine, EFA is automated and provides proof of correctness embedded in the code, it able to type SQL statements and assure no "dead-ends" occur when queries are executed.

Contents

1	Introduction	3
1.1	Ampersand	3
1.2	Purpose of the project	4
1.3	Document Guide	4
2	The Stakeholders and the intended audience	7
2.1	Naming Conventions and Terminology	7
2.2	Implementation Issues	8
3	System Architecture and Module Hierarchy	9
3.1	System Architecture	9
3.2	External Libraries	10
3.3	A Description of Haskell-Like Syntax	10
3.4	A Description of Module Diagram Syntax	13
3.5	Module Hierarchy	14
3.5.1	TypedSQL	17
3.5.2	TypedSQLCombinators	18
3.5.3	Equality	19
3.5.4	PrettySQL	20
3.5.5	Proof Utils	20
3.5.6	Singletons	23
3.6	Key Algorithm	23
3.7	Communication Protocol	23
4	Testing	24
4.1	QuickCheck	24
4.2	Test suite	24

1 Introduction

This document is meant as a guide for EFA that includes the motivations taken from a business perspective, the mathematical and software foundations that resulted in the logical flow of EFA's design, and the testing that took place to assure EFA's functionality and correctness.

Currently, Ampersand is readily accessible to the public through Github and it is equipped with the ability to assess logical discrepancies on sets of data based on user-specified restrictions. Logical discrepancies arise when system changes occur which violate the restrictions set forth by the user. When a system violation occurs, one of two things can happen: the change that is meant to take place is adjusted so it no longer violates the restrictions or the changes are discarded. Ampersand is used to manipulate data and generate prototypes, although there is a debugger, certain errors still slip through. When the system rules are changed by the user, all data which are inconsistent with the new system must be eliminated or rehabilitated so it can be returned back into the system. Data inconsistencies are persistent bugs that can distort the product that Ampersand seeks to provide.

These data inconsistencies are corrected through ECA rules which use process algebra (PA) to correct or discard data using violations. EFA is used to translate these ECA rules, execute SQL queries to correct violations and safeguards the database from illegal transactions.

1.1 Ampersand

A large part of designing software systems is requirements engineering. One of the greatest challenges of requirements engineering is translating from business requirements to a functional specification. Business requirements are informal, with the intention of being easily understood by humans; however, functional specifications are written in a formal language that is unambiguous and precise. It is meant to capture attributes of the information system that is desired by multiple parties. 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 [JG:: cite ampersand book]. 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 aids by reducing the manual

labour required to fix invariants and assures that the database is free of errors.

Ampersand is a software tool that translates natural language into

Ampersand in practice

The correct installation and usage of the Ampersand tools and the additional tools requires some IT awareness, especially in case some unexpected errors occur when the Ampersand tool uses the additional tools. To get used to the specific syntax of how business rules are defined in the ADL language requires some training. Besides the base syntax, a specific patterns hub is available, to support new and seasoned ADL users, which can be used for training, inspiration and re-usage.

[YS: *Merging in my section here*]

1.2 Purpose of the project

[JG: *Need an overview on what EFA is, what it does*] [YS: insert: Based on today's meeting. Merging in the purpose of the project here]

1.3 Document Guide

This document acts as a comprehensive guide to EFA, the first section covers the basics of Ampersand and the purpose of the project. The second section provides an overview of EFA's requirements specifications and various naming conventions and terminology used throughout the course of this guide. The third section focuses on design implementation and provides information on modules from a top down basis. It also explains the key algorithm used to translate businesses process into eca rules which EFA then uses to produce SQL queries. Much of the proof of correctness is embedded in the code that is explained in this section. Lastly, section four mostly focuses on testing.

[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*]

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. This problem is addressed in Ampersand using AMMBR (Joo). 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.

With the extension of Ampersand by the EFA project, we allow means of checking the correctness of the Action associated with an ECA rule. This will allow us to automatically rehabilitate existing system data while maintaining the information system according to user specifications. Furthermore, EFA allows us to automatically restore system invariants and create a program that allows Ampersand to make the most efficient choice regarding how it wishes JG:to: insert: p] proceed based on each individual case. The ECA rules, which act as an input to the EFA project are translated to human-readable SQL. This can later be viewed in the command line using the ‘‘`--print~eca~info`’’ flag written out in one of the artifacts. The generated SQL, not only allows for checking the correctness of AMMBR but also helps in identifying patterns and unreachable states of the method. EFA project will also be utilized to complete the AMMBR algorithm and also to test out future modification to the AMMBR algorithm. Since AMMBR is an essential element of the Ampersand software, EFA will complement modifications to this method.

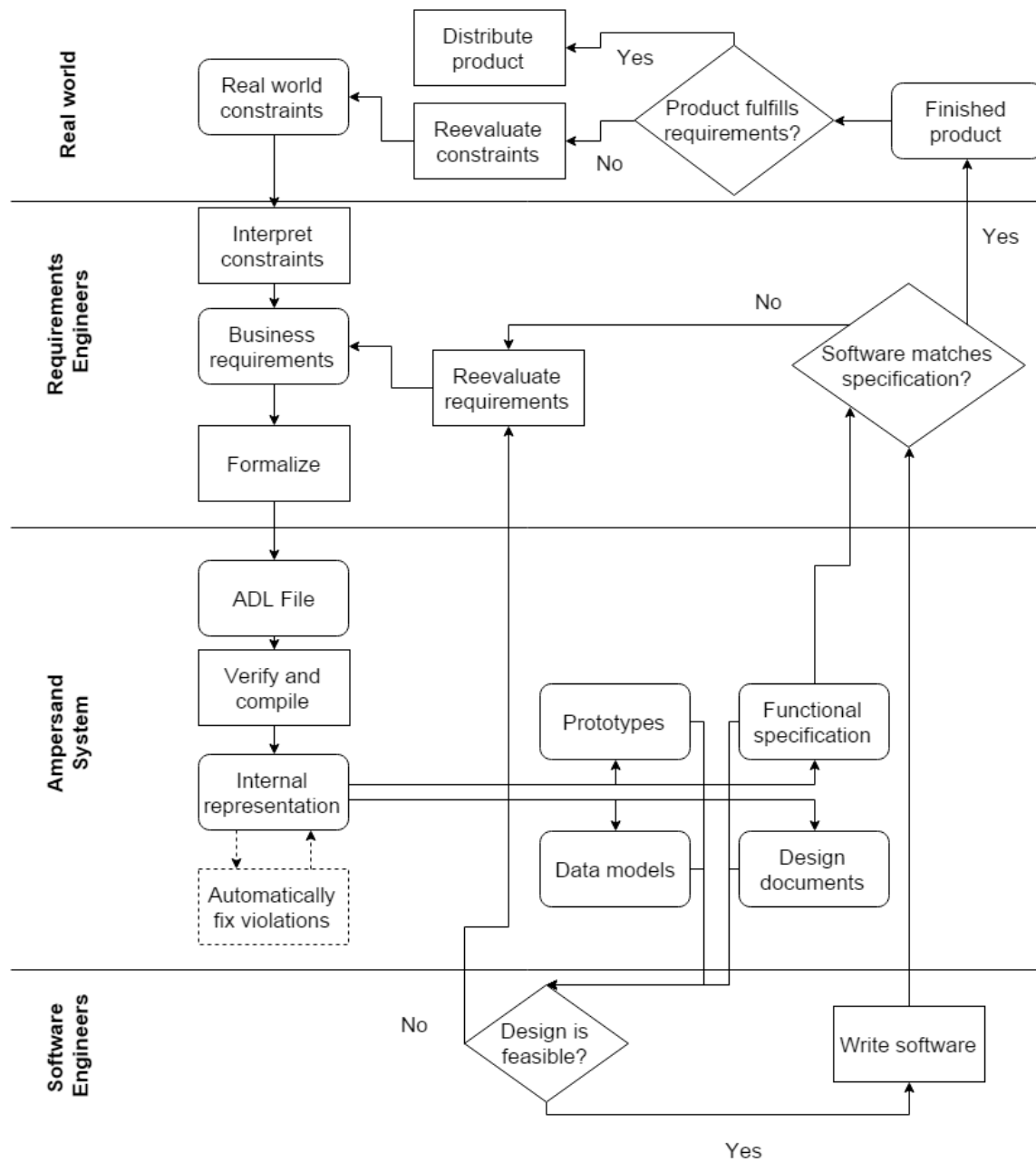


Figure 1: Business process diagram representing EFA project represented as a dashed box

2 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 decrease 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.

2.1 Naming Conventions and Terminology

ECA Stands for Event-Condition Action. 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.

ADL Stands for “Abstract Data Language” ((Joo07a, 13)). 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. 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.

2.2 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 (JSE15).
- **Functional dependencies are *NOT* equivalent to type families.** For example, given:

```
class ooo ⇒ 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 (sof).

- **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" (Wad15).
- **Certain invariants are difficult to state and prove.** The solution implemented for this problem was to vastly simplify 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.

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 additional 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 (Joo07b), 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. (Whe)

wl-pprint

The wl-pprint library(Lei) 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.

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 (NU). 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 $X : A \rightarrow B \rightarrow \dots \rightarrow \text{Type}$ to denote a regular data type, and $Y : A \rightarrow B \rightarrow \dots \rightarrow \text{Kind}$ 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. $(x : A) \rightarrow B$ 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 $(x : A) \rightarrow B$ 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 $\forall (x : A) \rightarrow B$ 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: $\forall (x : A) \rightarrow P\ x$ and $\forall x \rightarrow P\ x$ 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 $\exists (x : A) (P\ x)$ 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 exceedingly 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`”. 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.

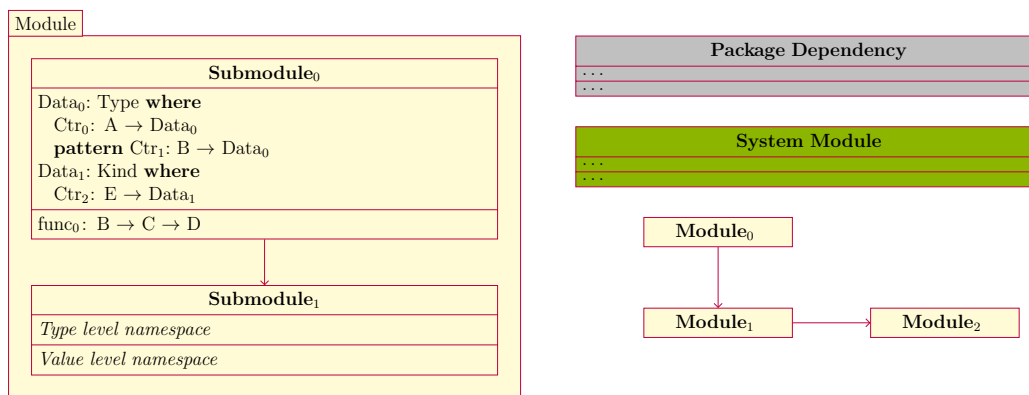


Figure 2: Example of module diagram syntax

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 **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 ~ Bool`. Note that **Show** itself does *not* have kind **Constraint** – its kind is `Type → Constraint`. The detailed semantics of these primitive entities can be found in the GHC user guide (otUoG). 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.

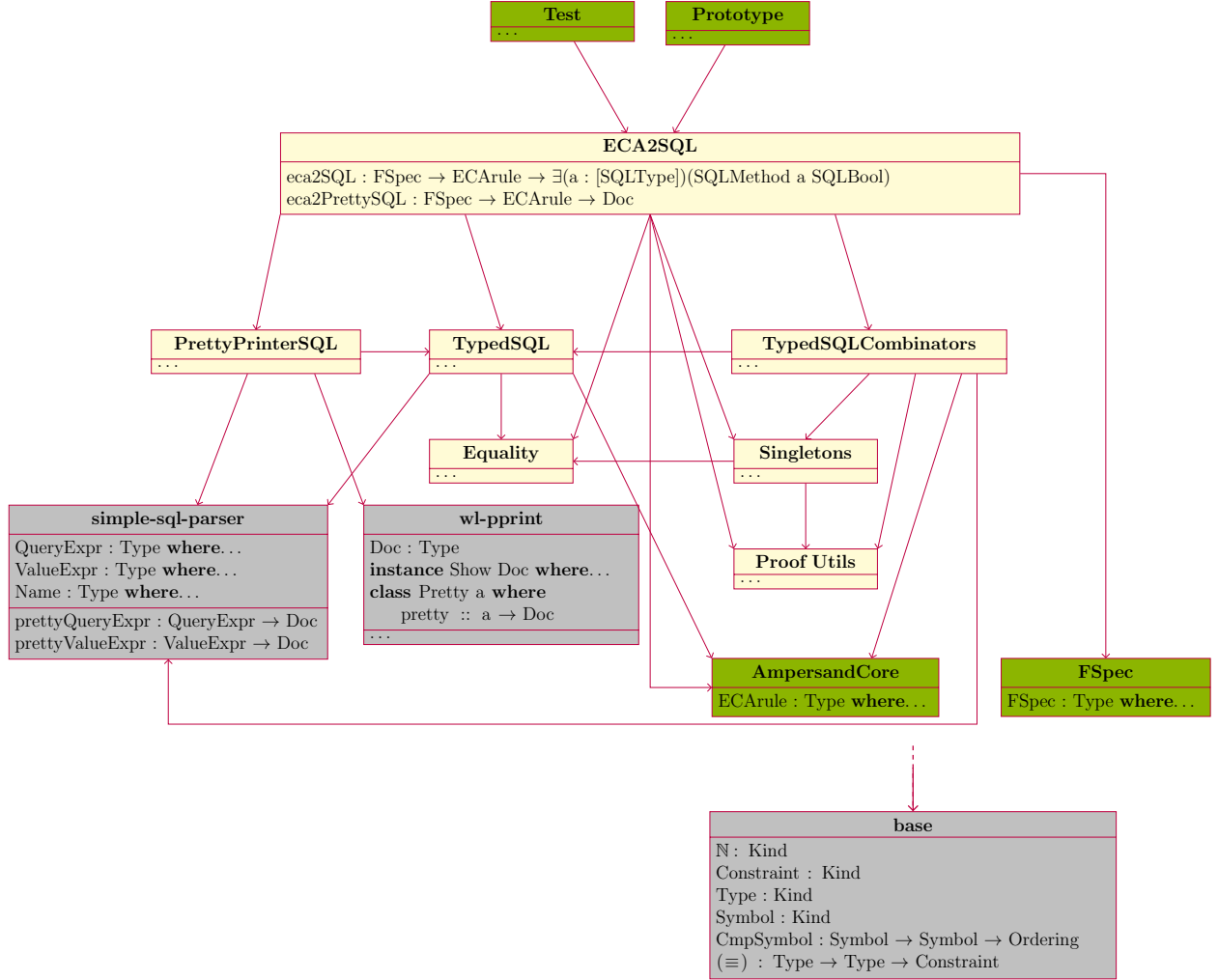


Figure 3: Module diagram for EFA as a whole

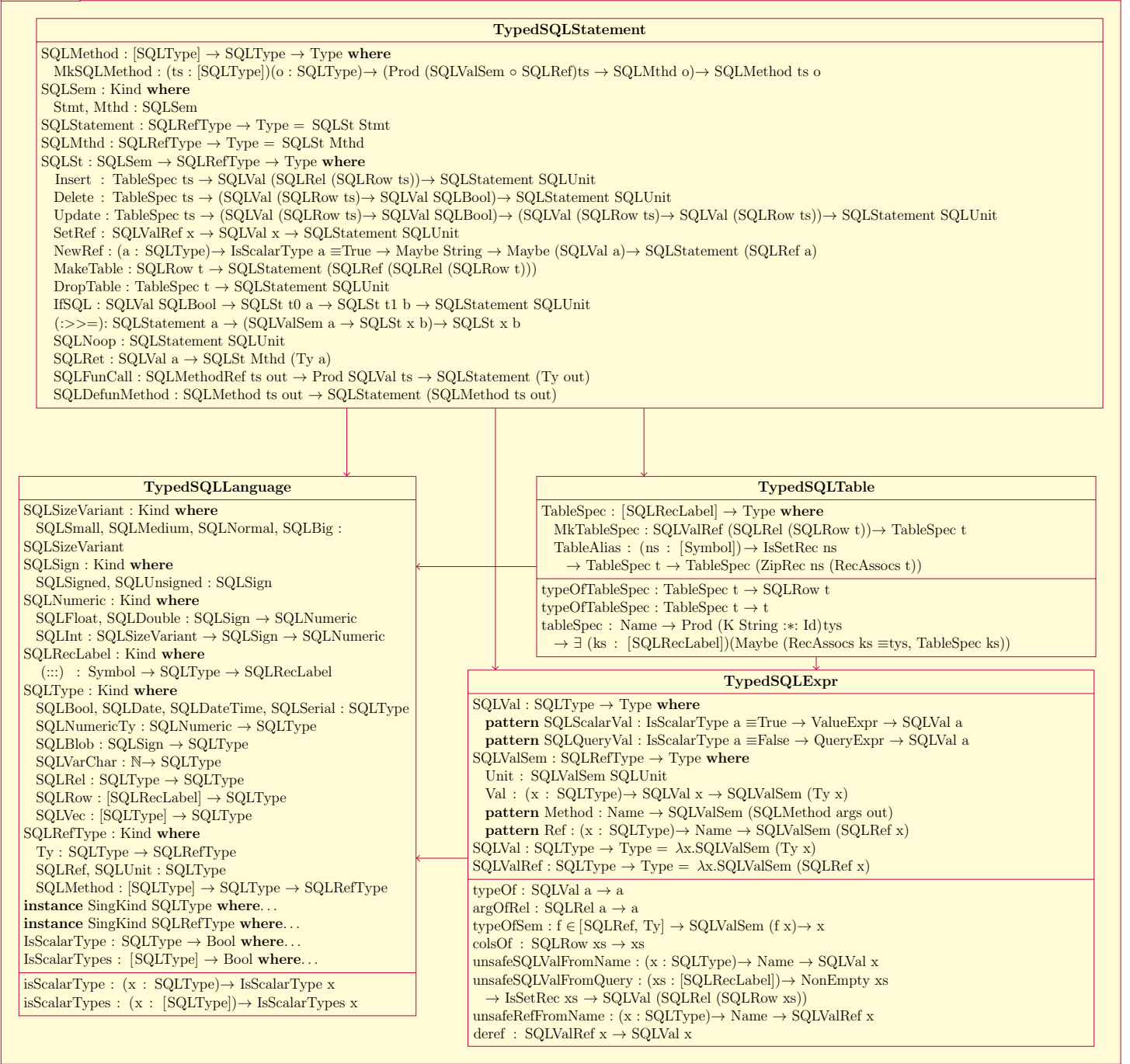


Figure 4: Module diagram for TypedSQL

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 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, unsafeSQLValFromQuery, 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 (Cor). 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 (Cor).

:>>= 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 (Cor) 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
<code>PrimSQLFunction : [SQLType] → SQLType → Type</code> <code>PTrue, PFalse : PrimSQLFunction [] SQLBool</code> <code>Not : PrimSQLFunction [SQLBool] SQLBool</code> <code>Or, And : PrimSQLFunction [SQLBool, SQLBool] SQLBool</code> <code>In, NotIn : PrimSQLFunction [a, SQLRel a] SQLBool</code> <code>Exists : PrimSQLFunction [SQLRel a] SQLBool</code> <code>GroupBy : PrimSQLFunction [SQLRel a, a] (SQLRel (SQLRel a))</code> <code>SortBy : PrimSQLFunction [SQLRel a, a] (SQLRel a)</code> <code>Max, Min, Sum, Avg : IsSQLNumeric a → PrimSQLFunction [SQLRel a] a</code> <code>Alias : (RecAssocs ts : RecAssocs ts') → PrimSQLFunction [SQLRel (SQLRow ts)] (SQLRel (SQLRow ts'))</code>
<code>primSQL : ∀(args : [SQLType])(out : SQLType)→ PrimSQLFunction args out → Prod SQLVal args → SQLVal out</code> <code>(!) : (xs : [SQLType])(i : Symbol)→ LookupRec xs i ≡ r → SQLVal (SQLRow xs)→ SingT i → SQLVal r</code> <code>(!) : (xs : [SQLType])(i : Symbol)→ LookupRec xs i ≡ r → SQLVal (SQLRel (SQLRow xs))→ SingT i → SQLVal r</code> <code>(!) : (xs : [SQLType])(i : N) → LookupIx t i ≡ r → SQLVal (SQLVec t)→ SingT i → SQLVal r</code>

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 NFDData a doubleNeg : NFDData a ⇒ a → Not (Not a) triviallyTrue : Not (Not ()) mapNeg : (NFDData a, NFDData b) ⇒ (b → a) → Not a → Not b elimNeg : NFDData 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 (Wad15); 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 accidentally 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 `NFDData` 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 called `Void`. 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
<pre>instance Pretty (SQLTypeS x) where... instance Pretty (SQLVal x) where... instance Pretty (SQLValSem x) where... instance Pretty (SQLSt k x) where... instance (str \equiv String) \Rightarrow Pretty (str, SQLMethod args out) where...</pre>

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 (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 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 witness for `IsNotElem`.

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

ProofUtils
<pre> 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) </pre>

Figure 8: Module diagram for ProofUtils

3.5.6 Singletons

The Singletons module (figure 9) is not fully detailed here; instead, a vastly simplified version of the Singletons module 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 (SJ02), 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
<code>SingT : k → Type</code> <code>class SingKind (k : Kind) where...</code>
<code>(%≡) : ∀ (x : k) (y : k) → SingKind k ⇒ x → y → DecEq x y</code>

Figure 9: Module diagram for Singletons

3.6 Key Algorithm

The key algorithm for the EFA project is AMMBR (Joo07b). 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(Ora). 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

[JG:] : *mention GHC type checking and prev assumption of correctness from datatype and properties already existing in Ampersand*]

4.1 QuickCheck

4.2 Test suite

References

- [Cor] Oracle Corporation. *MySQL 5.7 Reference Manual*.
- [EW12] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. *SIGPLAN Not.*, 47(12):117–130, September 2012.
- [Joo] Stef Joosten. Deriving functional specifications from business requirements with ampersand.
- [Joo07a] Stef Joosten. Deriving Functional Specifications from Business Requirements with Ampersand. *CiteSeer*, 2007.
- [Joo07b] Stef Joosten. Deriving Functional Specifications from Business Requirements with Ampersand. *CiteSeer*, 2007.
- [JSE15] Simon Jones Jan Stolarek and Richard Eisenberg. Injective type families for haskell. *Microsoft Research*, 0(0):118–128, 2015.
- [Lei] Daan Leijen. wl-pprint package. <https://hackage.haskell.org/package/wl-pprint-1.2>. 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.
- [NU] A. Andreas Norell U., Danielsson N. A. The Agda Wiki. <http://wiki.portal.chalmers.se/agda/pmwiki.php>. Accessed: 2016-02-28.
- [Ora] Oracle. Mysql reference manual. <https://dev.mysql.com/doc/refman/5.7/en/>. Accessed: 2016-02-28.
- [otUoG] The University Court of the University of Glasgow. *The Glorious Glasgow Haskell Compilation System User’s Guide*. Accessed: 2016-02-29.
- [SJ02] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002.
- [sof] Equality from a functional dependency.
- [Swi08] Wouter Swierstra. Data types la carte. *Cambridge University Press*, 2008.

- [Wad15] Philip Wadler. Propositions as types. *Communications of the AC*, 58(12):7584, 2015.
- [Whe] Jake Wheat. simple-sql-parser. <https://hackage.haskell.org/package/simple-sql-parser-0.4.1>. Accessed: 2016-02-28.
- [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.