Module Guide for ECA Rules for Ampersand

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

February 24th, 2016

Table 1: Revision History

Author	Date	Comments
Yash Sapra	24 / 02 / 2016	Initial draft
Jaeden Guo	27/ 02 / 2016	Update and merge
Yash Sapra	28/ 02 / 2016	Modifications to several sections
Yuriy Toporovskyy	28/ 02 / 2016	Addition of UML diagrams
Yuriy Toporovskyy	29/ 02 / 2016	Major changes (diagrams, explanations, organization)
Yash Sapra	29/ 02 / 2016	Proof reading
Yuriy Toporovskyy	29/ 02 / 2016	Formatting and references

Contents

1 Introduction

1.1 Description

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

Our design follows the principles laid out by (DP84), which can be summarized as follows:

- Unnecessary design details are omitted for simplicity.
- Each module is broken down based on hierarchy with no overlap of functionality.
- All our modules are *Open Modules*, that is, they are available for extension in the future.
- Reference materials are provided for external libraries but details of their implementation are not included here.

The language of implementation is Haskell. The primary reason for using Haskell is that the existing Ampersand system is largely written in Haskell. However, we leverage the full power of the Haskell type system in order to encode as many invariants in Haskell as possible. In particular, we use many modern features of the Glasgow Haskell Compiler (GHC) in order to do so. However, the particular features, as well as how and where they are used, are considered an implementation detail that is not relevant to the design of EFA.

1.2 Scope

The purpose of this document is to outline the implementation details of the EFA project described in the Problem Statement. EFA is responsible for generating SQL Statements from ECA rules that will be used to fixed any violated invariants in the Ampersand prototype. The document will serve as a referral document for future software development in the Ampersand project.

1.2.1 Intended Audience

This document is designed for:

New project members: 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.

Maintainers & Designers: The structure of this module guide will help maintainers rationalize where changes should be made in order to accomplish their intended purpose. Furthermore, the design document will act as a guide to EFA for future designers of Ampersand.

2 Anticipated and Unlikely Changes

2.1 Anticipated Changes

It is likely that EFA will require changes to the front-end interface. This addition may include a protocol that will connect the front-end interface to back-end functions, which will give the user more control. In addition, ECA rules are not static and may change over time, if changes do take place those changes will need to be incorporated into EFA's future releases.

Thus far anticipated changes include:

AC1: New front-end interface.

AC2: Addition or elimination of ECA rules.

AC3: The algorithm used for EFA.

AC4: The format of output.

AC5: The format of input parameters.

AC6: Integration of front-end interface to back-end modules.

AC7: Testing for individual modules and internal systems.

2.2 Unlikely Changes

These unlikely changes include the things that will remain unchanged in the system, and also changes that would not affect EFA.

UC1: There will always be a source of input data external to the software.

UC2: Results will always be provably correct.

UC3: The implementation language must be the same as that which is used for building the Ampersand system.

UC4: The format of initial input data and associated markers for data association.

UC5: Type of output data will always be a SQL query.

3 System Architecture

This section provides an overview of the module design. 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 3.2.

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 3.1.

3.1 Haskell-like Syntax Description

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 dependantly typed functional language, and since a large part of our work deals with "faking" dependant types, the syntax of Agda is conducive to easy communication of our module system. The principle of faking dependant types in Haskell is detailed in Hasochism (LM13) (a portmanteau of Haskell and masochism, because purportedly wanting to fake dependant 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.

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 $\mathrm{X} : \mathrm{A} \to \mathrm{B} \to \ldots \to \mathrm{Type}$ to denote a regular data type, and $\mathrm{Y : A \to B \to \ldots \to Kind}$ to denote a datatype which is used exclusively as a kind.

Dependant types

The syntax used to denote a "fake" dependant type in our model is the same as used to denote a real dependant type in Agda. $\mathrm{(x : A) \to B}$ 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 $\mathrm{(x : A) \to B}$ in our model is `forall (x :: A) . SingT x -> B`. `SingT :: A -> Type` denotes the singleton type for the kind A, which is inhabitted by precisely one value for each type which inhabits A. The role and use of singleton types is detailed further on, in section 3.3.7.

The syntax $\mathrm{\forall\ (x : A) \to B}$ 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: $\mathrm{\forall\ (x : A) \to P\ x}$ and $\mathrm{\forall\ x \to P\ x}$ 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 $\mathrm{Show\ a \to a \to String}$. In certain casees, 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.

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.

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 … = ….

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.

Existential quantification

The type ∃ (x : A) (P x) indicates that there exists some x of kind A which satisfies the predicate P. Unfortunately, Haskell does not have first class existential quantification. It must be encoded in one of two ways:

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

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

3.2 Module Diagram Syntax Description

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 pacakge, 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 existance of EFA), are indicated in green. The module heirarchy 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 1.

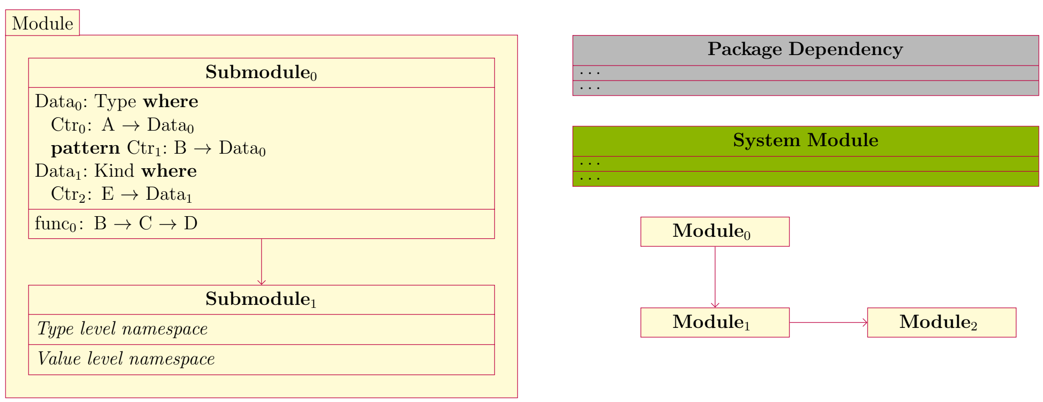

Figure 1: Example of module diagram syntax

3.3 Module Hierarchy

3.3.1 Coarse module hierarchy

This section contains a hierarchal breakdown of each module, as well as a brief explanation of each modules' elements.

The module heiarchy of EFA as a whole is given in figure 2. Note that every module which is part of EFA depends on the Haskell `base` package (which is the core libraries of Haskell). Also note that for the `base` package, we only include primitive definitions (i.e. those not defined in real Haskell) which may be difficult to track down in the documentation. The kinds $\mathbb{N}$ and Symbol correspond to type level natural number and string literals, respectively. The kind Constraint is the kind of class and equality constraints, for example, things like Show x and Int $\sim$ Bool. Note that `Show` itself does *not* have kind Constraint – its kind is Type $\rightarrow$ Constraint. The detailed

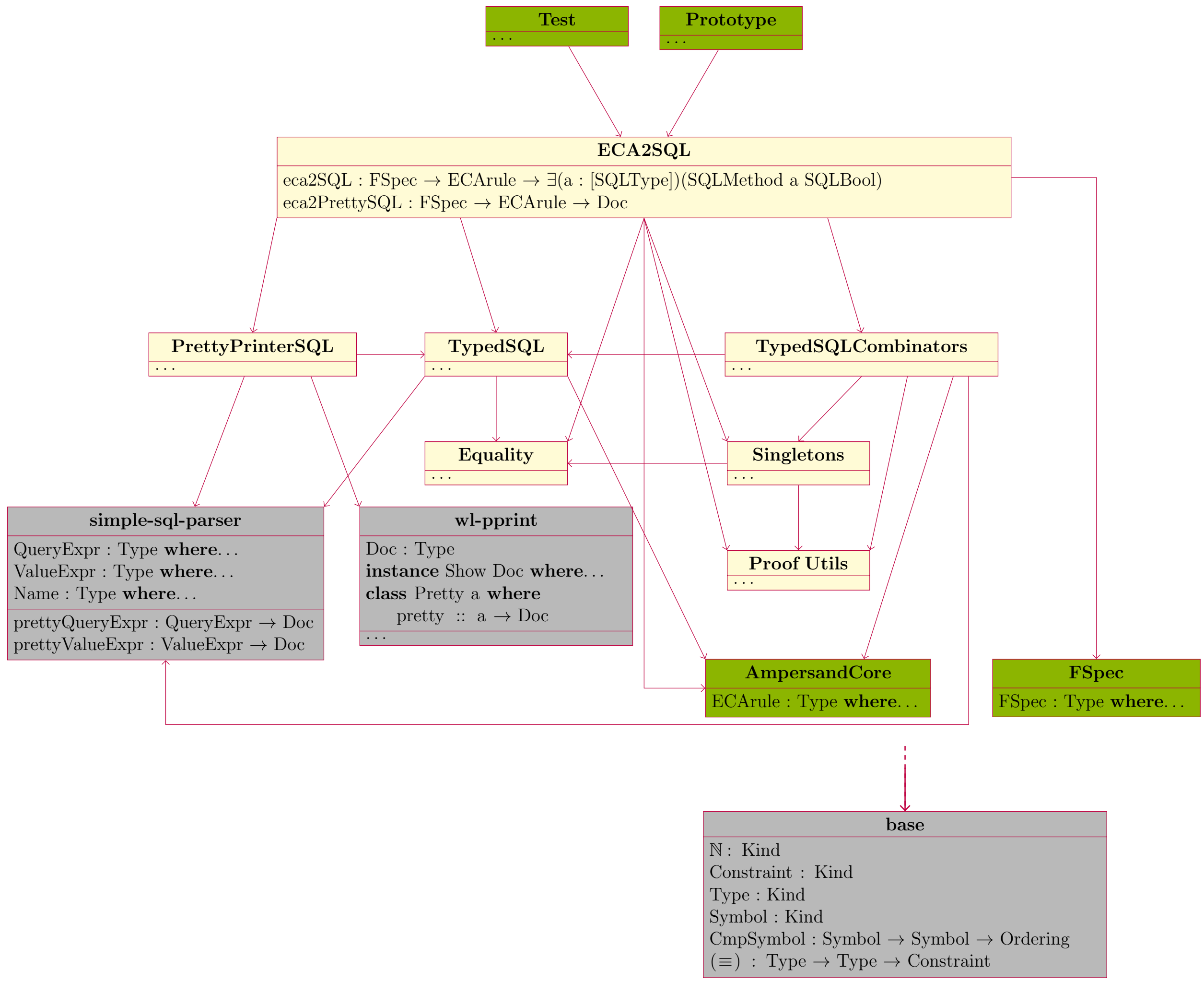

Figure 2: Module diagram for EFA as a whole

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 unhandled errors or infinite loops. Therefore, no additional information past the type of the function is required to deduce the inputs and ouputs of the function – they are precisely the inputs and outputs of the type.

3.3.2 TypedSQL

The module hierarchy for the TypedSQL module is shown in figure 3. The submodules of TypedSQL are quite large, however, the majority of the definitions within are type and kind definitions, which correspond precisely to entities defined by MySQL. The only exception is SQLRel, which distinguishes relations from scalar types - MySQL does not make this 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, 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 statment, will be rejected), these programs can be written in an equivalent way in our language without any loss of generality.

3.3.3 TypedSQLCombinators

The module TypedSQLCombinators, whose members are given in figure 4, implements a subset of primitive SQL functions on top of the TypedSQL expression type. The data type PrimSQLFunction encodes the specification of each function; the type and semantics of each function is that of the corresponding function in MySQL (refer to the MySQL manual (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.

3.3.4 Equality

The Equality module (7) 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 similair 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 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 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 trivallyTrue. 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.

3.3.5 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.

3.3.6 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 inhabitted 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 imporant 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, respecively, 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.

3.3.7 Singletons

The Singletons module (figure 5) 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 %$\equiv$. 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.

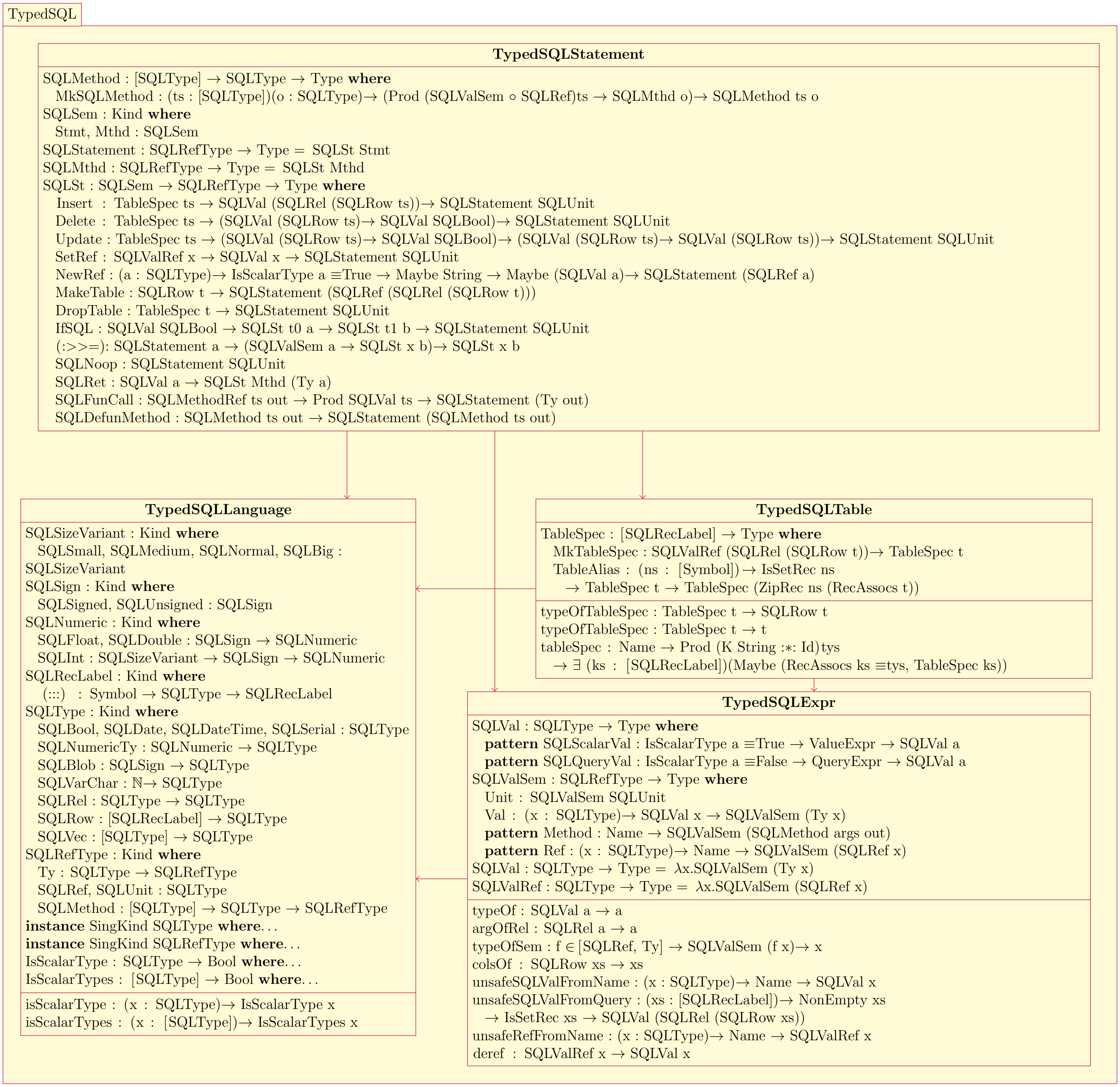

Figure 3: Module diagram for TypedSQL

TypedSQLCombinators

PrimSQLFunction : [SQLType] $\rightarrow$ SQLType $\rightarrow$ 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 $\rightarrow$ PrimSQLFunction [SQLRel a] a
  Alias : (RecAssocs ts : RecAssocs ts') $\rightarrow$ PrimSQLFunction [SQLRel (SQLRow ts)] (SQLRel (SQLRow ts'))

primSQL : $\forall$(args : [SQLType])(out : SQLType)$\rightarrow$ PrimSQLFunction args out $\rightarrow$ Prod SQLVal args $\rightarrow$ SQLVal out
(!) : (xs : [SQLType])(i : Symbol)$\rightarrow$ LookupRec xs i $\equiv$ r $\rightarrow$ SQLVal (SQLRow xs)$\rightarrow$ SingT i $\rightarrow$ SQLVal r
(!) : (xs : [SQLType])(i : Symbol)$\rightarrow$ LookupRec xs i $\equiv$ r $\rightarrow$ SQLVal (SQLRel (SQLRow xs))$\rightarrow$ SingT i $\rightarrow$ SQLVal r
(!) : (xs : [SQLType])(i : $\mathbb{N}$) $\rightarrow$ LookupIx t i $\equiv$ r $\rightarrow$ SQLVal (SQLVec t)$\rightarrow$ SingT i $\rightarrow$ SQLVal r

Figure 4: Module diagram for TypedSQLCombinators

Singletons

SingT : k $\rightarrow$ Type
class SingKind (k : Kind) **where**...

(%$\equiv$) : $\forall$ (x : k) (y : k) $\rightarrow$ SingKind k $\Rightarrow$ x $\rightarrow$ y $\rightarrow$ DecEq x y

Figure 5: Module diagram for Singletons

PrettySQL

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**...

Figure 6: Module diagram for PrettySQL

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 7: Module diagram for Equality

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

3.4 Key Algorithm

The key algorithm for the EFA project is AMMBR (Joo07). 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.

3.5 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.

3.6 Error handling

Most of the error handling is done by the underlying Ampersand software. Any human errors (syntactic or semantical) in the input ADL file are handled during the generation of ECA rules. The resulting ECA rules are fed into the EFA project and are determined to be provably correct. The language of implementation (i.e. Haskell), guarantees type level correctness of the EFA project at compile time.

If any error occurs during runtime, the state of the database is checked before and after the SQL statement has ran. In case an inconsistency is found in the data, the SQL rollback command is issued and will change the database to its previous state. In such a scenario, the user will be notified about the event and can take the necessary action to fix the issue.

3.7 External Libraries

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 (Joo07), which is the key algorithm responsible for translating business requirements into ECA rules is an integral part of Ampersand.

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

3.8 Traceability Matrix

Note : The traceability matrix is based on test plan submitted by the EFA team after removing test cases 11,12,13 which are not feasible at this point. These test cases will be removed from the Test plan in the next update.

Table 2: Traceability Matrix for the EFA Project

		Requirements					
		F1	F3	F4	F6	N1	N2
Test Cases	T1	*					
	T2	*					
	T3	*					
	T4	*					
	T5	*	*				
	T6		*				
	T7			*			
	T8				*	*	
	T9						*
	T10				*		

References

[Cor] Oracle Corporation. *MySQL 5.7 Reference Manual.*

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

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

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

[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.

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

[SJ02] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002.

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

[Wad15] Philip Wadler. Propositions as types. *Communications of the 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.