

Bachelor thesis
University of Leipzig

CARBON
a Web application and a RESTful API for
argumentation

Jakob Runge
mam09crm@studserv.uni-leipzig.de
2145708

Supervisor: Prof. Dr. Gerhard Brewka
Assistance: Dipl-Ing Stefan Ellmauthaler, BSc.

June 16, 2014

Abstract

Jakob Runge

CARBON a Web application and a RESTful API for argumentation

This thesis documents the development of *Collaborative Argumentation Brought Online* (CARBON). *Collaborative Argumentation Brought Online* (CARBON) aims to make abstract dialectical frameworks (ADFs) available via HTTP by providing a RESTful API and a JavaScript heavy application, that allows to use ADFs in a wiki context on top of that API. The thesis summarizes basic concepts of abstract argumentation using examples of Dung argumentation frameworks (AFs), bipolar argumentation frameworks (BAFs) and abstract dialectical frameworks (ADFs). The advantages of using Haskell as a programming language for server side software are demonstrated by discussing central concepts of functional programming and how these influenced the design of our solutions and simplified the creation of a RESTful API. It is described, how argumentation can be embedded in a wiki, and how a mapping between wiki articles and statements can be established to enable users to create new content while still being able to work with ADFs. To simplify the creation of acceptance conditions, a custom approach to proof standards is presented that allows to translate a bipolar argumentation framework (BAF) with proof standards into a ADF.

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Problem description	2
1.3	Requirements	2
2	Background	5
2.1	Dung argumentation frameworks	5
2.1.1	Example	5
2.1.2	Extensions	6
2.2	Bipolar argumentation frameworks	7
2.2.1	Example	7
2.2.2	Extensions	8
2.3	Abstract dialectical frameworks	9
2.3.1	Propositional formula ADFs	9
2.3.2	Example	9
2.3.3	Bipolar abstract dialectical frameworks	10
2.3.4	Models	11
2.3.5	Proof standards	11
2.4	The Haskell programming language	11
2.4.1	Pure Code	12
2.4.2	Lazy evaluation	12
2.4.3	Static type system	13
2.4.4	Monads	14
2.4.5	Monoids	16
2.4.6	Generalized algebraic data types	17
2.5	Representational State Transfer	19
2.5.1	Cacheable	20
2.5.2	Stateless	20
2.5.3	Layered System	20
2.5.4	Code on demand	21
2.5.5	Uniform Interface	21

3	Embedding ADFs in a wiki context	23
3.1	Proof standards	24
3.1.1	Formalization of proof standards for CARBON	24
4	The RESTful API	27
4.1	Cacheable	27
4.2	Stateless	27
4.3	Layered System	28
4.4	Code on demand	28
4.5	Uniform Interface	28
5	The technology stack	29
5.1	Client side	30
5.2	Server side	32
6	Data stored by CARBON	33
6.1	User	34
6.2	Item	35
6.3	Description	36
6.4	Article	36
6.5	AcceptanceCondition	37
6.6	Relation	38
6.7	Discussion	39
6.8	ResultSet	40
6.9	Result	40
7	Implementing a DSL for the back end	43
7.1	Composing GADTs by using monads	43
7.2	Ram only	44
7.3	Flat files	44
7.4	MySQL	44
7.5	PostgreSQL	45
8	Algorithms developed for CARBON	47
8.1	Representing propositional formulas in CARBON	47
8.2	Calculating acceptance conditions	47
8.2.1	The work of autoCondition	48
8.2.2	How mkFormula proceeds	49
8.3	The function fitInstance	51
8.3.1	Adding missing nodes	51
8.3.2	Calculating possible relations	51
8.3.3	Adding and Removing relations	51
8.4	Evaluation of discussions	51
9	Discussion	53

10 Related work	55
10.1 The Argument Interchange Format	55
10.2 ArguBlogging	56
10.3 Argumentation in Haskell	56
10.4 ASPARTIX	56
10.5 TOAST	56
11 Future work	59
11.1 Additional features for user interaction	59
11.2 Integrating CARBON with the AIF	60
11.2.1 Integrating with argumentation in Haskell	60
11.2.2 Detection of relation types between arguments	60
11.3 Possible improvements to the implementation	60
11.3.1 Breaking the monolithic structure of CARBON	60
Acronyms	63
Bibliography	65
Appendix	69
A The database layout	69
B Predicate dependencies of Carbon.Data.Item	71
C Software libraries used to implement CARBON	73
C.1	73
C.2	73
C.3	73
C.4	74
C.5	74
C.6	74
C.7	74
C.8	74
C.9	74
C.10	75
C.11	75
C.12	75
C.13	75
C.14	75
C.15	76
C.16	76
C.17	76

D Screenshots of the web application**77**

CHAPTER 1

Introduction

To document the development of *Collaborative Argumentation Brought Online* (CARBON), this chapter starts with the motivation for our work, describing how a concrete problem emerges from that, and what the resulting requirements for CARBON are. In the following chapter we will describe the concept of abstract argumentation, by discussing Dung argumentation frameworks (AFs), bipolar argumentation frameworks (BAFs) and abstract dialectical frameworks (ADFs). We will also describe core concepts of the Haskell programming language, which lay the grounds upon which CARBON is designed and implemented. In addition we will recall the concept of Representational State Transfer (REST), which is a central part both, for the API provided by CARBON, and for the web application build on top of that API. Afterwards we describe how CARBON embeds abstract argumentation in a wiki, and we introduce a combination of BAFs and proof standards, to simplify the task of entering the necessary informations into the system. Following that we turn towards the design of our RESTful API and what the existing software parts are that our solution relies on. We describe in detail how data is stored and handled, and afterwards turn towards the implementation of a domain specific language (DSL) that allows us to work with and store this data independent from a certain database or storage layer. Towards the end of this thesis we discuss the implementation of elected algorithms involved in the solution, which include the calculation of acceptance conditions for individual statements and the integration of user defined ADFs into already existing frameworks. Finally the thesis will close with a discussion of related works and an outlook towards future work that could be based on CARBON. CARBON is open source software, and released under the GPL General Public License Version 3. The full source code can be found in a git repository at [sourceforge](https://sourceforge.net/projects/carbon-adf/)¹, and the version used with this thesis is tagged ‘v.1.0’. Of course CARBON is constructed upon several existing software libraries, all of which are open source as well. A complete overview of these libraries, their versions and licenses can be found in appendix C.

¹ <https://sourceforge.net/projects/carbon-adf/>

1.1 Motivation

Argumentation has been a growing topic in AI in the last decades, and with the realization of projects such as ArguBlogging and the growing specification and use of the Argument Interchange Format (AIF), argumentation becomes more important and interesting in a networked context. Motivated by this trend, CARBON delivers another web application and API to bring argumentation to the web. The software *Dialectical Models Encoding* (DIAMOND)¹, which is an implementation of ADFs, is a pure command line tool. Building a web application around DIAMOND would facilitate the visualization of ADFs as graphs and would enable users to discover the functionality of ADFs in a playful manner. We also consider abstract argumentation as a potentially helpful tool for decision making in collaborative environments, and therefore wanted to enable users of a typical collaborative platform, the wiki, to take advantages of abstract argumentation.

1.2 Problem description

Coming from the motivation, we can now formulate a full problem. We conclude that there is a need for more services that supply abstract argumentation on the web. Specifically it is helpful for DIAMOND to be integrated in a solution that can visualize ADFs as graphs and enables clients to explore these. As a collaborative platform the functionality of a wiki will need to be expanded, to integrate abstract argumentation. These objectives can be solved by creating a web server, that on the one hand supports the typical wiki tasks, and on the other hand understands abstract argumentation, specifically ADFs well enough to solve some helper tasks for diamond, such as to construct input data and work with the returned outputs.

1.3 Requirements

Based upon the problem description we can now derive the requirements that arise. To supply the wiki context it is necessary for the server, to store data in a permanent, reliable manner. Also articles in wikis are typically versionized which means, that the server must also support articles in different versions and enable clients to view different versions. To prevent abuse and track authorship, CARBON shall have a simple form of user management, that allows authentication by password as well as some form of registration with the system. This also means, that clients accessing the service will have different roles, of either not being logged in, or being authenticated against the system, so that they are known as distinct users. For administration cases the additional role of an admin user will need to be introduced. In this wiki context, abstract dialectical frameworks (ADFs) shall be embedded, which makes it necessary to invent a way of extending the typical wiki setup. To allow visualization of ADFs as graphs, CARBON must supply a way of drawing these, and it is necessary that CARBON has a uniform representation of the argumentation data. To work together with DIAMOND, CARBON needs to produce valid input data, and must parse the according output correctly. When bringing ADFs to the web, CARBON must not focus solely on the presentation in a browser, but shall also be usable as a service for other

¹ <https://isysrv.informatik.uni-leipzig.de/diamond>

software. Therefore it is necessary to deliver an API that can easily be understood, and shall, in particular, follow the Representational State Transfer (REST) style constraints described in the background section. Since there is a separation between the API and the presentation anyway, it makes sense to design the presentation layer as a web application, that shall be implemented in JavaScript. These requirements also advise that CARBON makes use of a configuration file, which is typical for server side applications, especially in the Linux world, and allows for more flexible setups. It would also be a useful feature, to design the server side program so that it works in a stateless which makes updates and restarts easy, and also makes it possible to scale with growing load, should the need arise, by running multiple instances of CARBON.

CHAPTER 2

Background

In this chapter we will focus on the preliminaries and the basics that form the foundation for the work on CARBON. We will start by recalling the definition of argumentation frameworks (AFs), to introduce bipolar argumentation frameworks (BAFs) on top of that, and then turn to the concept of abstract dialectical frameworks (ADFs), to explain how argumentation is processed within CARBON. In addition this section describes a selection of concepts popular in functional programming, namely the Haskell language, which has been used to implement CARBON, so that many of these concepts are involved in design decisions. At the end of this chapter, we recall the concept of Representational State Transfer (REST), which provides the basic guidelines for the API design of CARBON.

2.1 Dung argumentation frameworks

In the last decades, argumentation frameworks (AFs) have been a popular topic in AI and non monotonic reasoning. The basic idea behind argumentation frameworks is to abstract away from a concrete scenario towards graph structures from which additional knowledge can be evaluated that will also hold for the original scenarios. The typical argumentation frameworks (AFs) after Dung (1995) [Dun95; Rah09b, ch. 2] are comprised of a tuple $(\mathcal{A}, \mathcal{R})$, where \mathcal{A} denotes the set of arguments/statements, and $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{A}$ denotes the attack relations between these arguments. Thus, an argument $a \in \mathcal{A}$ attacks another argument $b \in \mathcal{A}$ iff $(a, b) \in \mathcal{R}$.

2.1.1 Example

As an example for Dung AFs, let us consider the popular party game Mafia, that was invented by Dimma Davidoff in 1986¹, but is believed to be known since before 1970².

Mafia has been described as ‘a game between informed minority and uninformed majority’[Yao08][ch. 1], and a single instance of the game consists of several rounds, where each round is divided into two phases, day and night. In the night phase, the informed minority chooses a player to be eliminated from the game, with the intend to have only players from the minority remain at the end of the game. In the day phase, all players discuss

¹ http://web.archive.org/web/19990302082118/http://members.theglobe.com/mafia_rules/

² <https://www.princeton.edu/~sucharit/~mafia/history.htm>

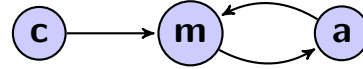
to decide on a player to be eliminated, but while the minority knows all its members, the majority doesn't, and so the minority can partake in the discussion to undermine the majority's intent. A complete description of the game rules can also be found at <http://www.eblong.com/zarf/werewolf.html>, where the Werewolf themed adaptation of the game is described. Let us now look at a concrete example:

The first night just passed, and Bob was eliminated by the minority. In the following day phase, Mallory starts the discussion by accusing Alice of being a member of the minority, because Alice was eyeballing other players just before the night. Alice, to her defense, says, that she way only eyeballing others to see if any player would behave suspicious. Carol says, that Mallory's argument is just a straw man to appear as a part of the majority, claiming that Mallory would belong to the minority.

To abstract from this example, we identify Mallory, Alice and Carol as our set of arguments $\mathcal{A} = \{m, a, c\}$, and their arguments against each other as attack relations $\mathcal{R} = \{(m, a), (a, m), (c, m)\}$.

$$\mathcal{F}_{AF} = (\{m, a, c\}, \{(m, a), (a, m), (c, m)\}) \quad (2.1)$$

(a) Dung argumentation framework



(b) The according graph

Figure 2.1: Representation of our example

2.1.2 Extensions

To obtain results from an argumentation framework, some notions are defined by Dung:

defense: A set of arguments $E \subseteq \mathcal{A}$ defends an argument $a \in \mathcal{A}$, iff $\forall (b, a) \in \mathcal{R}$ there is a $c \in E$, so that $(c, b) \in \mathcal{R}$.

acceptable: An argument $a \in \mathcal{A}$ is called acceptable with respect to $E \subseteq \mathcal{A}$, iff a is defended by E .

conflict-free: A set of arguments $E \subseteq \mathcal{A}$ is called conflict-free, iff $\forall a, b \in E$, $(a, b) \notin \mathcal{R}$. The set of conflict-free sets in our example would be $\{\emptyset, \{c\}, \{m\}, \{a\}, \{c, a\}\}$.

admissible: A set of arguments $E \subseteq \mathcal{A}$ is called admissible, iff it is conflict-free and $\forall a \in E$, a is acceptable with respect to E . For our example the set of admissible sets is $\{\emptyset, \{c\}, \{c, a\}\}$.

These notions are then used to define computable sets of arguments, called extensions. Some more popular extensions are:

complete: An extension E is called a complete extension of \mathcal{A} , iff E is admissible and $\nexists a \in \mathcal{A} \setminus E$, so that a is acceptable with respect to E . This means, that every acceptable argument with respect to E must be an element of E . For our example the complete extension is $\{\{c, a\}\}$.

preferred: An extension E is called a preferred extension of \mathcal{A} , iff E is maximal among the admissible sets extensions with respect to set inclusion. In the example this means, that the preferred extension turns out to be the same as the complete extension, $\{\{c,a\}\}$.

stable: An extension E is called a stable extension of \mathcal{A} , iff E is conflict-free and $\forall b \in \mathcal{A} \setminus E$ there is an $a \in E$, so that $(a,b) \in \mathcal{R}$. This means, that a stable extensions is defended against all arguments not in the extension. For the example the stable extension is $\{\emptyset, \{c,a\}\}$.

grounded: An extension E is called the grounded extension of \mathcal{A} , iff E is minimal among the complete extensions with respect to set inclusion. The grounded extension of our example is $\{\{c,a\}\}$.

2.2 Bipolar argumentation frameworks

The focus of Dung-style AFs on attack relations only has been perceived as insufficient, so that different approaches emerged that aim to enrich the original concept. BAFs add a second type of relations that express support between arguments. In the paper ‘On the bipolarity in argumentation frameworks’[Amg04], different applications for bipolar argumentation are mentioned, and the concept is motivated further by stating, that the ‘distinction between positive and negative preferences is supported by studies in cognitive psychology which have shown that these two types of preferences are independent and processed separately in the mind’.

In [Rah09b][ch. 4], the following definition for BAFs is given:

An abstract bipolar argumentation framework (BAF) $\langle \mathcal{A}, \mathcal{R}_{att}, \mathcal{R}_{sup} \rangle$ consists of: a set \mathcal{A} of arguments, a binary relation \mathcal{R}_{att} on \mathcal{A} called the attack relation and another binary relation \mathcal{R}_{sup} on \mathcal{A} called the support relation. These binary relations must verify the following consistency constraint: $\mathcal{R}_{att} \cap \mathcal{R}_{sup} = \emptyset$.

2.2.1 Example

Using this definition for BAFs, we can now translate our example for Dung AFs into a BAF. To do this, it becomes necessary to identify the arguments and their relations slightly different, or otherwise we would obtain a trivial BAF where $\mathcal{R}_{sup} = \emptyset$, and $\mathcal{A}, \mathcal{R}_{att}$ being directly translated from the Dung AF.

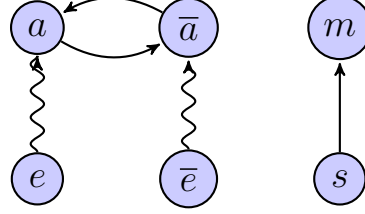
Table 2.1: Interpretation of our example for BAFs

Argument	Content
a	Alice belongs to the minority.
e	Alice was eyeballing others.
\bar{a}	Alice belongs to the majority.
\bar{e}	Alice has just looking for others.
m	Mallory belongs to the majority.
s	Mallory’s argument is a straw man.

Based on this new interpretation, we can now construct a BAF:

$$\mathcal{F}_{BAF} = \langle \{a, e, \bar{a}, \bar{e}, m, s\}, \{(a, \bar{a}), (\bar{a}, a), (s, m)\}, \{(e, a), (\bar{e}, \bar{a})\} \rangle \quad (2.2)$$

(a) bipolar argumentation framework



(b) The according graph

Figure 2.2: Representation of our example

2.2.2 Extensions

Following the definitions from [Rah09b][ch. 4] it now becomes necessary to redefine some notions and extensions. We now distinguish between different kinds of attacks in addition to the new notion of support:

support: Given $a, c \in \mathcal{A}$, a supports c , iff either $(a, c) \in \mathcal{R}_{sup}$ or $\exists b \in \mathcal{A}. (a, b) \in \mathcal{R}_{sup}$ and b supports c . The intuition behind support is, that there must be a path of support relations between a and c .

direct attack: Direct attacks work just like attacks in typical Dung AFs, where an argument $a \in \mathcal{A}$ attacks an argument $b \in \mathcal{A}$, iff $(a, b) \in \mathcal{R}_{att}$.

support attack: Given $a, b, c \in \mathcal{A}$, a support attacks c , iff a supports b , and $(b, c) \in \mathcal{R}_{att}$.

set support: Given $\mathcal{S} \subseteq \mathcal{A}, b \in \mathcal{A}$, \mathcal{S} set supports b , iff $\exists a \in \mathcal{S}$, so that a supports b .

set attack: Given $\mathcal{S} \subseteq \mathcal{A}, b \in \mathcal{A}$, \mathcal{S} set attacks b , iff there $\exists a \in \mathcal{S}$, so that either $(a, b) \in \mathcal{R}_{att}$, or $\exists c \in \mathcal{A}$, so that a supports c and $(c, b) \in \mathcal{R}_{att}$.

defense: For BAFs there exist more elaborate concepts for defense as well, but for our overview of BAFs we'll stick to the notion from Dung AFs, where arguments are defended, iff all direct attacking arguments are attacked. For our example, this means, that a defends itself against $\{\bar{a}, \bar{e}\}$, even though \bar{e} set attacks a .

In contrast to [Rah09b][ch. 4], which describes different variants of admissible and preferred extensions, we will focus on single variants as examples to show that the extensions known from Dung AFs can also be generalized for BAFs.

+conflict-free: A set $\mathcal{S} \subseteq \mathcal{A}$ is called +conflict-free, iff $\nexists a, b \in \mathcal{S}$, so that a set attacks b . For our example, some +conflict-free sets are: $\{\emptyset, \{a\}, \{\bar{a}, \bar{e}, m\}\}$.

d-admissible: A set $\mathcal{S} \subseteq \mathcal{A}$ is d-admissible, iff \mathcal{S} is +conflict-free, and all elements of \mathcal{S} are defended. Some d-admissible sets would be: $\{\emptyset, \{a, e\}, \{\bar{a}, \bar{e}, s\}\}$.

d-preferred: A set \mathcal{S} is d-preferred, iff it is maximal with respect to set inclusion among the d-admissible sets. The d-preferred sets of \mathcal{F}_{BAF} are: $\{\{a, e, s\}, \{\bar{a}, \bar{e}, s\}\}$.

stable: A stable extension is given, iff a set \mathcal{S} is +conflict-free, and $\forall a \in \mathcal{S} \setminus \mathcal{A}$, \mathcal{S} set-attacks a . For \mathcal{F}_{BAF} , the stable sets are the same as the d-preferred ones.

2.3 Abstract dialectical frameworks

Similar to BAFs, abstract dialectical frameworks (ADFs) expand the concept of argumentation frameworks, but in contrast to BAFs a wider variety of relations between arguments are possible.

For instance, ADFs allow to express that an argument supports another one, that two arguments – none of which is strong enough individually – may jointly attack a third one, what the effects of combining attacking and supporting arguments are, and the like. [Bre13]

ADFs were first introduced in [Bre10] with the slogan ‘ abstract dialectical frameworks = dependency graphs + acceptance conditions ’. An ADF is a tuple $D = (S, L, C)$, where S is our set of statements (arguments), $L \subseteq S \times S$ are the dependency relations between the statements, and C are the acceptance conditions. Given a statement $a \in S$, we describe the parents of a as $par(a) = b \mid (b, a) \in L$. Based on the parents function, we can now define the set of acceptance conditions $C = \{C_s \mid s \in S\}$, where $C_s : 2^{par(s)} \rightarrow \{t, f\}$. This way, each statement $s \in S$ has an acceptance condition C_s , that maps every selection of parent statements $par(s)$ to a truth value¹.

2.3.1 Propositional formula ADFs

In [Ell12, p. 27], based on the observation, that the dependency relations and the acceptance conditions can be replaced by propositional formulas, a different definition of ADFs, the propositional formula ADFs (pForm-ADFs) are established.

A pForm-ADF is a pair $D = (S, AC)$, where

- S is a set of statements
- $AC = \{AC_s\}_{s \in S}$ is the set of acceptance conditions, where each statement has exactly one associated condition.

[Ell12, Def. 3.1.6]

Where, of course, AC_s is a propositional formula. CARBON makes use of pForm-ADFs exclusively, because DIAMOND has a native syntax for them, and they are simpler to generate for CARBON than other formats.

2.3.2 Example

We can now abstract our instance of the Mafia game as a pForm-ADF, and will enhance our interpretation from table 2.1 to do so.

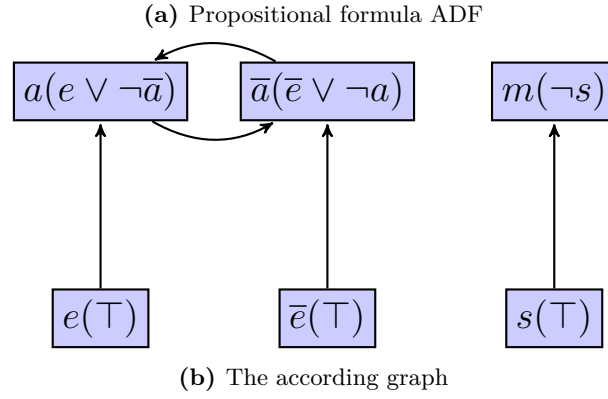
¹ In contrast to [Bre10], which used $\{in, out\}$, we adopt the notion of truth values used in [Bre13].

Table 2.2: Interpretation of our example for pForm-ADFs:

Argument	Propositional formula	Content
a	$e \vee \neg \bar{a}$	Alice belongs to the minority.
e	\top	Alice was eyeballing others.
\bar{e}	\top	Alice has just looking for others.
\bar{a}	$\bar{e} \vee \neg a$	Alice belongs to the majority.
m	$\neg s$	Mallory belongs to the majority.
s	\top	Mallory's argument is a straw man.

$$\mathcal{F}_{ADF} = (\{a, e, \bar{a}, \bar{e}, m, s\}, \quad (2.3)$$

$$\{AC_a = e \vee \neg \bar{a}, AC_e = \top, AC_{\bar{a}} = \bar{e} \vee \neg a, AC_{\bar{e}} = \top, AC_m = \neg s, AC_s = \top\} \quad (2.4)$$

**Figure 2.3:** Representation of our example

Note, that while $\{a\}$ was defended and +conflict-free in our BAF example, the according acceptance condition $AC_a(\emptyset) = f$ would not hold true without any parents.

2.3.3 Bipolar abstract dialectical frameworks

An interesting property of \mathcal{F}_{ADF} is, that all its relations can be categorized into either supporting or attacking relations. This is done by the following distinction:

Let $D = (S, L, C)$ be an ADF. A link $(r, s) \in L$ is

1. supporting iff for no $R \subseteq \text{par}(s)$ we have that $C_s(R) = in$ and $C_s(R \cup \{r\}) = out$,
2. attacking if for no $R \subseteq \text{par}(s)$ we have that $C_s(R) = out$ and $C_s(R \cup \{r\}) = in$.

[Bre10, Def. 5]¹

¹ Note that we use the current labelings with $\{t, f, u\}$ from [Bre13] rather than the legacy $\{in, out, udec\}$.

Following this definition, \mathcal{F}_{ADF} belongs to the subset of bipolar abstract dialectical frameworks (BADFs), and its notation is in the form of a *propositional formula* BADF (pForm-BADF). This also highlights the relation of ADFs to the formerly introduced BAFs.

2.3.4 Models

Since ADFs can also be used to represent normal logic programs, as displayed in [Bre10], there are no extensions for ADFs, but rather models with certain properties that preserve the same semantics. Given a subset $M \subseteq S$, S is called a model, iff $\forall s \in S. C_s(M \cap \text{par}(s)) = t$ holds. This means, that a model must satisfy the acceptance conditions of all statements in an ADF. A model $M \subseteq S$ is called free, iff $\forall s \in M. C_s(M \cap \text{par}(s)) = t$, that is, the acceptance conditions of all statements in the model must hold true, given a set of all statements in the model that are also parents. While [Bre10] originally defined some models only for BADFs, [Bre13] generalizes models from BADFs to ADFs, and corrects unintended results.

2.3.5 Proof standards

[Bre10] mentions, that the concept of proof standards is widely used in literature on legal reasoning, and also states, that ‘in everyday reasoning proof standards play an essential role: in situations involving risk we obviously apply higher standards than in cases where there is not much to lose.’ While [Bre10] defines formal constraints, that build on top of four different argument types introduced in [Far95], namely *valid*, *strong*, *credible* and *weak* arguments, CARBON takes a different approach, where the degree of certainty, that users believe a statement has, is basically encoded as a level $l \in \mathbb{N}$. The handling of proof standards in CARBON is further described in chapter 3, which explains, how argumentation, and proof standards in particular can be embedded in a web application.

2.4 The Haskell programming language

CARBON makes use of the Haskell programming language for its server part implementation. Therefore it makes sense to discuss some of the more special features that are either provided by functional programming languages in general, or by Haskell in special. A short introduction to Haskell is given by ‘Learn you a Haskell for Great Good’¹. For a more thorough book on Haskell ‘Real World Haskell’² is a good source. On the official website³, Haskell is described as follows:

Haskell is an advanced purely-functional programming language. An open-source product of more than twenty years of cutting-edge research, it allows rapid development of robust, concise, correct software. With strong support for integration with other languages, built-in concurrency and parallelism, debuggers, profilers, rich libraries and an active community, Haskell makes it easier to produce flexible, maintainable, high-quality software.

1 <http://learnyouahaskell.com/>

2 <http://realworldhaskell.org/>

3 <http://www.haskell.org/haskellwiki/Haskell>

When writing about Haskell this usually includes the Haskell Platform along with it, which is known as ‘a comprehensive, robust development environment for programming in Haskell’¹, that not only includes a big collection of useful libraries, but also the Haskell specific package and build tool Cabal and the Glasgow Haskell Compiler (GHC).

2.4.1 Pure Code

In contrast to imperative programming languages, which have the Turing Machine as their theoretical background, functional programming languages build on the concept of the lambda calculus and the mathematical notion of functions as their theoretical background. Coming from that mathematical background, it is a typical tendency for functional programming languages to avoid state or mutable data. The Haskell Wiki² describes this concept as follows:

Purely functional programs typically operate on immutable data. Instead of altering existing values, altered copies are created and the original is preserved. Since the unchanged parts of the structure cannot be modified, they can often be shared between the old and new copies, which saves memory.

The concept of immutable data can also be described as referential transparency, which, due to the deterministic nature of pure code, allows for equational reasoning. An example is given by the following transformation:

$$g = h(f(x), f(x)) \equiv y = f(x), g = h(y, y) \quad (2.5)$$

The main difference between this mathematical notation, and the Haskell notation is, that function application in Haskell is written as $f x$ instead of $f(x)$, and that a Haskell program may possess different efficiency in both cases. Such transformation allow for easy refactoring, because pure code guarantees, that it is impossible to introduce any unwanted side effects this way. In addition to its benefit for refactoring this also makes it easier to reuse code trough out the program. Another example for such a refactoring transformation would be a pipeline of functions chained together:

$$P = f \cdot g \cdot h \cdot i \equiv x = g \cdot h, P = f \cdot x \cdot i \quad (2.6)$$

In this case, we can, due to the law of associativity for function composition, replace $g \cdot h$ by a more efficient function x , should the need arise, or to expand to functionality of such a pipeline, by grouping composed functions and replacing them with others.

2.4.2 Lazy evaluation

While most programming languages have a pattern of evaluation that executes one computation after another, in the order of the instructions in the program code, Haskell makes use of a strategy known as lazy evaluation. This means, that the calculation of computations is deferred as long as they are not required by other computations. Thereby it is possible to avoid

¹ <http://www.haskell.org/platform/contents.html>

² http://www.haskell.org/haskellwiki/Functional_programming#Immutable_data

unnecessary calculations, should they not be required by later code, and it also becomes possible to define infinite data structures, that are only evaluated as far as necessary. The downside of this approach is, that it becomes possible to introduce endless recursive loops unwillingly, and that debugging can be more difficult once it becomes necessary to track down an error that is only discovered upon evaluation of a bigger computation. An example of this is given by the following two programs, the first written in C, and the second in Haskell:

```
1 #include <stdio.h>
2
3 #define LIMIT 100
4
5 int main(){
6     int fibs[LIMIT];
7     fibs[0] = 0;
8     fibs[1] = 1;
9     for(int i = 2; i < LIMIT; i++)
10         fibs[i] = fibs[i-1] + fibs[i-2];
11     for(int i = 0; i < LIMIT; i++)
12         printf("Fib(%i)=%i\n", i, fibs[i]);
13     return 0;
14 }
```

Listing 2.1: Fibs.c

```
1 module Fibs where
2
3 limit = 100 :: Int
4
5 fibs = 0:1:zipWith (+) fibs (tail fibs)
6
7 outputs = zipWith (\i f -> "Fib(" ++ show i ++ ")=" ++ show f) [0..] fibs
8
9 main = mapM_ putStrLn $ take limit outputs
```

Listing 2.2: Fibs.hs

In the C program from listing 2.1, a fixed size array structure is computed for later use, and it is necessary to know the size of the array beforehand. In the Haskell program from listing 2.2 however, the list *fibs* is of potentially infinite length, which is the same for the list *outputs* of strings. The use of *for* loops in the C code is also a typical case of state as used in imperative languages, which is avoided in the Haskell code by using the infinite list *[0..]* to generate the indices necessary for the output.

2.4.3 Static type system

In contrast to other programming languages, Haskell has a static type system, that is based on System F [Gir03, p. 81]. This allows a Haskell Compiler to check data types at compile time and thereby eliminate whole classes of typical errors. For example the often problematic null pointer exception cannot be found in typical Haskell programs, as long

as no unsafe language extensions such as the foreign function interface (FFI)¹ are used. In addition to giving safety guarantees for the code, the type system delivers important information about the code, that make refactoring a lot simpler. Once code is changed, it will not compile until all types match again, which makes it exceptionally hard to miss parts that do not fit together well. Since Haskell has a clean separation between pure and impure code, looking at the type also gives quick information about how refactoring can be attempted, and what the code can possibly do.

2.4.4 Monads

Monads make it possible, to put computations into sequence and introduce side effects. With pure code being lazy evaluated, the order of evaluation is usually not decided by the declaration, but by the demands of the running program code. This turns out to be a problem when a program needs to perform operations that introduce side effects. Input and output operations, for example have side effects by their very nature. The order of read and write accesses to files is of vital importance for the outcome of the operation, and is by no means deterministic in the sense of pure code. To solve this problem, Haskell makes use of the concept of monads, that is introduced in the Haskell 2010 report as follows:

‘The term monad comes from a branch of mathematics known as category theory. From the perspective of a Haskell programmer, however, it is best to think of a monad as an abstract data type.’

[Mar10, p. 107]

When working with more complex programming languages, it is common practice, to make use of type variables² to abstract a special case and work with a more general case instead. A typical example for this are lists, where it is enough to work on a list structure without knowing the type contained in the list, in order to figure out its length. This is well demonstrated by the listing below, which hides the Haskell implementation of the *length* function in order to define it anew. Sure enough, running the main function prints out 11.

```
1 module List where
2
3 import Prelude hiding (length)
4
5 length :: [a] -> Int
6 length []      = 0
7 length (x:xs) = 1 + length xs
8
9 main = print $ length [5..15]
```

Listing 2.3: List.hs

With monads, the concept of type variables is taken one step further, from simple type variables to type constructors. The definition of the monad typeclass therefore looks as

1 The FFI allows connecting Haskell to libraries written in another language such as C.

2 This abstraction is also known as templates in C++, Java and similar languages.

follows:

```

1 class Monad m where
2   (>>=) :: m a -> (a -> m b) -> m b
3   return :: a -> m a

```

Listing 2.4: The Monad typeclass

From its definition, we only know, that every monad supports the two operations `>>=`, called ‘bind’, and `return`. The bind operation allows us to sequence two computations, by using the result *a* from a monad as the parameter to a function that produces a monad with a potentially different return type *b*. Using the `return` function, we can lift any value into the context of a monad. To understand this definition further, let us look at monads by the example of the state monad, that provides a notion of mutable state for pure code, which, by definition, must be deterministic.

The state monad: Since the problem of sequencing file access operations has already been mentioned, let us suppose that we have some code that needs to open a lot of different files in some order, and we would like to keep track of which files need to be closed again. In particular we want to close all files in reverse order as they were opened.

```

1 module FileState where
2
3 import Control.Monad
4
5 data FileState a = FS [FilePath] a
6
7 instance Monad FileState where
8   (FS fs x) >>= f = let (FS fs' y) = f x
9                       in FS (fs' ++ fs) y
10  return a = FS [] a

```

Listing 2.5: FileState.hs:1-10

In line 5, the `FileState` data type is defined, with its constructor `FS`, that carries a list of `FilePath`s and a type variable. As we can see from the `Monad` instance for `FileState`, the bind operator uses the given function *f* to produce an intermediate state, so that the two lists of `FilePath` can be concatenated and the resulting `FileState` is constructed. The `return` function builds on top of the idea, that pure code does not open any files, and therefore the `FileState` is constructed with an empty list.

```

12 printState :: FileState a -> IO ()
13 printState (FS [] _) = putStrLn "No files there to close."
14 printState (FS files _) = mapM_ putStrLn ("Closing files:" : files)
15
16 addState :: FilePath -> FileState ()
17 addState f = FS [f] ()

```

Listing 2.6: FileState.hs:12-17

`printState` and `addState` are two example functions for working with the state. `addState`

can be used to add a new file to the state, which corresponds to the idea of logging which files are open, and *printState* takes a *FileState* into the *IO* monad to list all of its files. In a real world program we could replace both these functions with more sophisticated structures, that would allow us to open files or connect through a network, knowing, that all these operations will be cleaned of.

```

19 main = printState $ do
20   -- We open a first file:
21   addState "/tmp/foo.log"
22   -- Some computations, and more files:
23   mapM_ (addState . ("/etc/" ++)) ["group", "passwd", "shadow"]

```

Listing 2.7: FileState.hs:19-23

With the *main* function we get an example of how using Haskell's *do* notation makes it possible to get blocks of instructions similar to imperative programming languages, with the difference, that we could implement the context ourselves. This introduction of a custom context by the programmer is also the reason, that monads have been described as ‘programmable semicolons’ [OSu08, ch. 14], because the introduced context is ‘executed’ just where the typical semicolons from C or Java would separate instructions. To finish this example, we can confirm the working of the code by its output:

```

$ runhaskell FileState.hs
Closing files:
/etc/shadow
/etc/passwd
/etc/group
/tmp/foo.log

```

Note that it the notion of a state monad is not inconsistent with the concept of pure code. The state monad merely simulates the existence of a state for pure code, but still depends on a given start state to deliver a result outside the monad. Given a defined start state, every computation inside the state monad is deterministic, and the same start state will always lead to the same result. As a last note, consider that the notion of a monad may even be generalized to a stack of monads where each monad may introduce a different effect for a computation, such as state, automatic checks for failures or the iteration over different structures. The generalization of a monad to a part of such a monad stack goes by the name of a monad transformer in the Haskell world.

2.4.5 Monoids

Just like in mathematics, in abstract algebra, Haskell also has monoids, and since the implementation of CARBON makes use of them in a bigger scale, it makes sense to discuss them here. In abstract algebra, a monoid is defined as a set \mathcal{S} and a binary operation \cdot , for which two axioms must hold:

1. The binary operation \cdot must obey the law of associativity for all elements $a, b, c \in \mathcal{S}$:

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c \tag{2.7}$$

2. There must be an identity element $e \in \mathcal{S}$, such that for all elements $a \in \mathcal{S}$:

$$e \cdot a = a \cdot e = a \tag{2.8}$$

In math a typical example for a monoid would be the set \mathbb{N}_0 with the binary operation $+$ and 0 as the identity element. In Haskell a typical example for a monoid would be a list structure, with the empty list being the identity element and the concatenation operator $++$ as the binary operation. In CARBON, the monoid structure is used for most data types, where an empty instance is defined and used, and the binary operation is understood as overwriting fields in a data structure. However it is important not to overwrite fields with empty content like that in the empty instance, so that the empty instance still works as a neutral element. In Haskell the monoid type class is defined in the module `Data.Monoid` and looks like this:

```
1 class Monoid a where
2   mempty  :: a
3   mappend :: a -> a -> a
```

Listing 2.8: `Data.Monoid`

2.4.6 Generalized algebraic data types

CARBON comes with its own embedded domain specific language (EDSL), that is used to model requests against a back end used for data storage. When writing domain specific languages (DSLs) in Haskell, the two typical ways are to either embed the language in a shallow way, by defining the necessary operators and functions directly in Haskell, or to embed it deeper by using generalized algebraic data types (GADTs)¹. The second option allows to perform transformations on the DSL at run time in addition to the usual type checking². To better understand GADTs, its helpful to recall what hands the algebraic nature to algebraic data types (ADTs). It is possible to understand types as sets of possible values/instances, the type `Bool` for example has two instances, `True` and `False`, and we could easily write `{True, False}` instead of `Bool`. If we now look at the `Either` type, which uses type variables, we can see that the number of instances grows:

```
1 data Either a b = Left  a
2                 | Right b
```

Listing 2.9: `Data.Either`

With the type signature of `Either Bool Bool` we have already got 4 possible instances, which is the reason why `Either` is also called a sum type. When it comes to tuples, which are also called product types, we easily obtain multiplication, where `(Either Bool Bool, Either Bool Bool)` has 16 possible instances. Coming from this, we can already see that algebraic data types (ADTs) are composite types. The Glasgow Haskell Compiler (GHC) user guide describes GADTs:

¹ In literature and other programming languages GADTs are also known as phantom types.
² <http://www.haskell.org/haskellwiki/EDSL>

‘Generalised Algebraic Data Types generalise ordinary algebraic data types by allowing constructors to have richer return types.’

[Tea13, Ch. 7.4.7]

As an example let us have a look at an example for a simple computation:

```

1 {-# LANGUAGE GADTs #-}
2 module GADTExample where
3
4 data Comp a where
5   ANumber :: Int -> Comp Int
6   AString :: String -> Comp String
7   NumberToString :: Comp Int -> Comp String
8   Add :: Comp Int -> Comp Int -> Comp Int
9   Concat :: Comp String -> Comp String -> Comp String
10  Fold :: (x -> x -> x) -> [Comp x] -> Comp x

```

Listing 2.10: GADTExample.hs

In this example the *Comp* data type is declared, which depends on a type variable *a*. The lines 5-10 each define one constructor for *Comp*, and except for the *Fold* constructor they all define the type variable *a*. The two constructors *ANumber* and *AString* both take a standard Haskell type to create a *Comp* of that type. The constructor *NumberToString* goes one step further by requiring a *Comp Int* as a parameter to construct a *Comp String*, which picks up the possibility for recursive types, that are also possible with simple ADTs. The constructors *Add* and *Concat* take this theme even a step further by taking two parameters, which must both be *Comps* of the same type. The most general constructor is *Fold*, which accepts a binary function $\mathcal{X} \times \mathcal{X} \rightarrow \mathcal{X}$, and a list of *Comps* of the same type as the functions parameters, to create a *Comp x* with a variable type. From looking at the constructors we can see, that there are only a few ways to build a *Comp* at all. Of course there are the possibilities of passing the empty list to the *Fold* constructor, or using infinite recursion with the *Add* constructor, but these will either lead to an infinite loop in the case of recursion, or, due to the simple type of *Fold*, lead to a run time exception¹. Beside this unintended possibilities, there are two simple ways to construct a *Comp*, which are using either *ANumber* or *AString*. Knowing this, we can already infer when *Add* or *Concat* can be used, or that there is a way to construct a *Comp String* from a *Comp Int*, but not the other way around. The names of our constructors already hint at the intended semantics for the simple DSL that is created with this example, but to be sure, let us look at the *compute* function, which works on *Comps* of any type:

```

12 compute :: Comp a -> a
13 compute (ANumber x)      = x
14 compute (AString s)     = s
15 compute (NumberToString n) = show $ compute n
16 compute (Add x y)       = compute x + compute y
17 compute (Concat x y)    = compute x ++ compute y

```

¹ Note that by making certain that *Fold* has at least a single value, this problem can be avoided.

```
18 compute (Fold f xs)          = foldl1 f $ map compute xs
```

Listing 2.11: GADTExample.hs

The *compute* function performs pattern matching on constructors, and from its type signature we can already see that it takes a *Comp a* as its input and produces a simple *a* from that. To do so, *compute* has two base cases, which match the constructors *ANumber* and *AString*. The other 4 possible cases of *compute* only evaluate a single step of the *Comp*, and rely on recursion to traverse and evaluate the tree structure of *Comp*. To round this example off, let us have a look at two examples for a *Comp String*, *task1* and *task2*:

```
20 task1 = Concat (AString "2 + 3 = ") (NumberToString (Add (ANumber 2) (ANumber 3))
)
```

```
21 task2 = Concat (AString "Sum [1..5] = ") (NumberToString (Fold (+) $ map ANumber
[1..5]))
```

```
22
23 main = mapM_ (putStrLn . compute) [task1, task2]
```

Listing 2.12: GADTExample.hs

For both tasks, the type annotations are omitted, but they can be inferred easily, because their topmost constructors are *Concat*. This also demonstrates how the Haskell type system is utilized to check the syntax of our DSL, which fosters confidence in the correctness of the code, since invalid task structures would throw compile time errors. The output of the whole example looks as expected:

```
$ runhaskell1 GADTExample.hs
2 + 3 = 5
Sum [1..5] = 15
```

The definition of the *Fold* constructor introduces even more flexibility in this DSL, because custom binary functions can be given to it, and we could easily replace the *+* in *task2* by a *** to get a product instead of a sum.

2.5 Representational State Transfer

The term REST was introduced by Roy T. Fielding in his dissertation, where he describes REST as an ‘architectural style for distributed hypermedia systems’ [Fie00b, Ch. 5]. An architecture satisfying the constraints given by this style is said to be RESTful. The definition of REST influenced the design of HTTP 1.1 as well as the notion of Uniform Resource Identifiers (URIs) [Ber96; Fie99]. Besides influencing the design of HTTP, the 2014 tutorial ‘RESTful Web Services: Principles, Patterns, Emerging Technologies’ gives an additional motivation to use RESTful architectures:

‘advocates of Representational State Transfer (REST) have come to believe that their ideas explaining why the World Wide Web works are just as applicable to solve enterprise application integration problems and to radically simplify the plumbing required to implement a Service-Oriented Architecture (SOA).’ [Pau14, p. 1]

Following this motivation, CARBON presents a RESTful API that makes it possible to easily build applications on top of it, which is also directly put to use by the web application part that is part of CARBON.

On page 85 [Fie00b], Fielding describes a constraint graph which has different constraints as its edges and architecture names as nodes, where the most constrained node is the REST architecture. We will not discuss the whole constraint graph here, but shall instead focus on some of the core constraints:

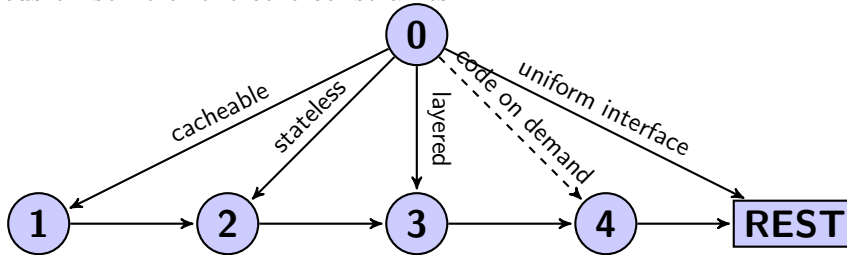


Figure 2.4: A simplified constraint graph

Just like the original constraint graph, the simplified version also starts with the null architecture, which has no constraints to it applied. For all other nodes, the constraints of all edges leading to them must be satisfied. This means, that an architecture can only be of type 3, iff it satisfies the layered, stateless and cacheable constraints. Taking this graph as a base, we can now recall some of the REST constraints in order:

2.5.1 Cacheable

In a RESTful architecture, responses must indicate whether they may be cached or not, so that caching is possible and can be used to reduce server load.

2.5.2 Stateless

The stateless constraint implies that no client dependent state may be stored on the server, and that all data about a client necessary to process a request is given in a single request. The session is entirely managed by the client except for authentication data, which may be stored on the server in a persistent manner. The stateless nature of a RESTful architecture makes it possible to scale the server-side easily and allows for intermediate cache servers. For example if all state used by a RESTful service is stored in a database, such a service can easily run in multiple instances operating on the same database, which distributes the load across several machines.

2.5.3 Layered System

In a layered system a client cannot necessarily tell whether the corresponding server is an endpoint or is build on top of other services. For example it is common practice, to setup a Varnish¹ cache server to encapsulate a slower running service, such as big PHP scripts to reduce load times[Wub09]. Another example of a layered system is an authentication server, that limits access to distinct services, but can appear as the same server to a client.

¹ <https://www.varnish-cache.org>

2.5.4 Code on demand

Code on demand ‘is only an optional constraint within REST’ [Fie00a, p. 123], and it allows a server to extend the clients functionality by supplying program code itself. A typical way to achieve this for web services is to embed JavaScript, which allows the client to have different behavior depending on the server. Code on demand also makes it easy to upgrade the functionality of clients, because updates need only be supplied on the server side, and clients can automatically request and use these.

2.5.5 Uniform Interface

The uniform interface is one of the most basic constraints for a RESTful architecture. It implies the distinction between clients and servers and thereby also causes a separation of concerns. HTTP is a typical example of such a uniform interface, where the *GET*, *POST*, *PUT* and *DELETE* methods can be used for Create, Read, Update and Delete (CRUD) semantics. In addition HTTP has a typical separation of concerns, where servers generate and distribute content and clients process it further, for example by rendering it on a screen in the case of a web browser or by filling databases with the retrieved data in the case of search engines. Typical hallmarks of a uniform interface in the REST sense are conceptual resources that are identified by single requests using Uniform Resource Identifiers (URIs) instead of working on whole databases, the manipulation of resources through their representations, which is easily performed in the case of HTTP by the use of the according methods and parameters, and messages being self-descriptive in the sense that they specify their syntax by the use of an Internet media type.

CHAPTER 3

Embedding ADFs in a wiki context

When discussing the task of bringing abstract argumentation to the web, we have to face the question of how this can be realized and in what kind of framework we want argumentation to be embedded. The example of ArguBlogging[Sna12a] shows, how argumentation can be added to existing web services, namely blogs, by the use of a Bookmarklet. A Bookmarklet is essentially a piece of JavaScript that can be run on a web page by clicking a Bookmark in the browser¹.

Instead of using blogs, as already hinted in the introduction, CARBON builds on the collaborative nature of wikis, and embeds argumentation into the concept. In contrast to ArguBlogging this is not realized via a Bookmarklet, but by implementing a completely new system. The basic building blocks of a wiki are single articles, whereas the abstract argumentation builds upon single statements. Therefore an obvious way to extend the wiki design with argumentation is to extend articles with information that is normally associated with statements. Since we choose ADFs for argumentation in CARBON, this strongly suggests, that articles need to have associated acceptance conditions. Acceptance conditions can be added to articles in two ways:

1. A proof standard for an article can be given, which is, together with attack and support relations from other articles, than used to construct an acceptance condition.
2. An acceptance condition can be specified as a propositional formula by a user. In this case CARBON can only infer which articles are related to a given article by its acceptance condition, but cannot figure out whether a given relation is of attacking or supporting nature. In particular [Ell12, p. 57] shows that the attack link decision problem is coNP-complete.

In addition to enhancing articles with acceptance conditions, CARBON provides a concept called discussions, that allows to group several articles in order to visualize them as a graph and calculate extensions for them. Discussions in CARBON are, however, not only

¹ A feature introduced in Netscape Navigator 4.0 in 1997. Compare <http://web.archive.org/web/20020611183734/http://developer.netscape.com/docs/manuals/communicator/jsguide/misc.htm#1005712>.

understood as a way for users to collect and visualize articles, but are meant to support a group of users in making a decision about a given subject. To make this possible, each discussion has a title and description of its own, and it is possible for users to vote among the extensions of the ADF described by a discussion in order to decide what they together consider as the best outcome when considering conflicting information. To keep users from modifying the contents of a discussion without end, it is possible to add a deadline to a discussion, so that the set of articles contained cannot be changed after a while, and the extensions can no longer change.

3.1 Proof standards

One way to add acceptance conditions to articles is, as described, for users to enter propositional formulas for each article. This method, however can prove cumbersome, because it implies that each time a new relation between two articles arises, a user would have to carefully edit the corresponding formula. To circumvent this problem, CARBON provides a notion of proof standards, together with attack and support relations. In CARBON, proof standards are used to label how certain the users are of the correctness of information given in an article. In addition to a proof standard, an article can be attacked or supported by various other articles. When we consider only a set of articles of a shared proof standard, the incoming attack and support relations for such an article work just like in bipolar argumentation frameworks[Rah09b, ch. 4], where it is enough if a single article has at least as many supports as attack with respect to a set, to be defended. However, once more than a single proof standard is considered, higher proof standards break lower ones. This means, that a single attack from a higher proof standard is enough to render all lower level supports meaningless, and a single support from a higher proof standard is enough to render attacks from lower level meaningless as well. Of course, the question might arise, why users would choose to specify attacks or supports from articles with a weaker proof standard to such with a higher one, but even though these relations do not have an immediate effect on the acceptance condition of an article, there are good reasons to do so:

1. Setting relations between articles makes it possible for CARBON to present related information to a client, which may be of interest even if the proof standard of such articles is a lower one.
2. The referenced article may be edited, and its proof standard may change in later versions. In such a case a different acceptance condition may be the result, if other relations are considered by the algorithm.

3.1.1 Formalization of proof standards for CARBON

Let us now formalize our concept of proof standards for CARBON. Since proof standards are understood as an ordered set of labels, we can easily assign them to a subset of \mathbb{N} . We

write a complete instance of such a framework F_{car} as follows:

$$F_{car} = (\mathcal{S}, \mathcal{R}_{att}, \mathcal{R}_{sup}, P) \quad (3.1)$$

$$\mathcal{R}_{att}, \mathcal{R}_{sup} \subseteq \mathcal{S} \times \mathcal{S} \quad (3.2)$$

$$P : \mathcal{S} \rightarrow \mathbb{N} \quad (3.3)$$

The typical set of statements is given as \mathcal{S} , with \mathcal{R}_{att} and \mathcal{R}_{sup} being the sets of attack and support relations. The function P is a mapping from the set of statements \mathcal{S} to the natural numbers \mathbb{N} , which represent our set of proof standards. To work with this representation, we need a translation from our representation given as F_{car} to a *propositional formula* BADF (pForm-BADF). To define this translation, let us first consider a simplified translation function \mathcal{T}_n , that only translates our representation for a single proof standard given as $n \in \mathbb{N}$:

$$\mathcal{T}_{n \in \mathbb{N}} : (\mathcal{S}, \mathcal{R}_{att}, \mathcal{R}_{sup}, P) \rightarrow (\mathcal{S}_n, \{\mathcal{C}_s | s \in \mathcal{S}_n\}) \quad (3.4)$$

The set of statements in the pForm-BADF created by \mathcal{T}_n can easily be calculated by restricting the original set to all elements that belong to a given proof standard n :

$$\mathcal{S}_n = \{s | s \in \mathcal{S}, P(s) = n\} \quad (3.5)$$

Now that the statements are filtered, we need to construct the acceptance condition \mathcal{C}_s for each statement $s \in \mathcal{S}_n$ by taking into account the attack and support relations. To do this, we define the function acc_s , which provides us with all possible subsets of \mathcal{S}_n , where the statement s can be accepted:

$$acc_s : \mathcal{S}_n \rightarrow \mathcal{A}, \mathcal{A} \subseteq 2^{\mathcal{S}_n} \quad (3.6)$$

$$acc_s(\mathcal{S}_n) = \{t | t \subseteq \mathcal{S}_n, |\{u | u \in t, (u, s) \in \mathcal{R}_{att}\}| \leq |\{u | u \in t, (u, s) \in \mathcal{R}_{sup}\}|\} \quad (3.7)$$

Of course, using acc_s , we can define a complementary function, ina_s , which delivers all sets for which we do not want s to be accepted:

$$ina_s(\mathcal{S}_n) = 2^{\mathcal{S}_n} \setminus acc_s(\mathcal{S}_n) \quad (3.8)$$

Given these two functions, we can now define the acceptance condition \mathcal{C}_s with more ease:

$$\mathcal{C}_s = \bigvee_{x \in acc_s(\mathcal{S}_n)} \left(\bigwedge_{y \in x} y \wedge \neg \bigwedge_{z \in 2^{\mathcal{S}_n} \setminus x} z \right) \vee \bigvee_{x \in ina_s(\mathcal{S}_n)} \left(\neg \bigwedge_{y \in x} y \right) \quad (3.9)$$

The intuition behind this formula is, that a statement is accepted, when either one of its acceptable sets is met, or none of the sets that render it unacceptable. Now, that the definition of \mathcal{T}_n is complete, we can generalize our translation further to make it

independent from a single proof standard.

$$\mathcal{T} : (\mathcal{S}, \mathcal{R}_{att}, \mathcal{R}_{sup}, P) \rightarrow (\mathcal{S}, \{\mathcal{C}_s | s \in \mathcal{S}\}) \quad (3.10)$$

The generalized translation function \mathcal{T} does not need to restrict the set of statements to a certain proof standard, like \mathcal{T}_n does, so that we can focus solely on the construction of the acceptance conditions \mathcal{C}_s . To enable us to define generalized acceptance conditions based on the ones specific for a single proof standard, we address the acceptance condition for a single standard by the function \mathcal{C}_n :

$$\mathcal{C}_n(s \in \mathcal{S}) = \begin{cases} \mathcal{X}_s & | \ (_, \mathcal{X}) = \mathcal{T}_n(\mathcal{S}, \mathcal{R}_{att}, \mathcal{R}_{sup}, P), P(s) = n \\ \perp & | \ \text{otherwise} \end{cases} \quad (3.11)$$

We then define the generalized acceptance conditions by reusing the previous definitions as follows:

$$\mathcal{C}_s = \bigvee_{n=1}^{\infty} \left((\mathcal{C}_n(s)) \wedge \bigwedge_{m=n+1}^{\infty} \neg \bigwedge_{z \in ina_s(\mathcal{S}_m)} z \right) \quad (3.12)$$

The intuition behind this definition is, that for each proof standard, we take the acceptance condition, and accept it unless we have a case, where a higher proof standard is not accepted. While we use ∞ in our definition for the generation of acceptance conditions, the actual application will always use a subset of \mathbb{N} to represent proof standards, so that the corresponding algorithm for this definition terminates. Note that our definition of $\mathcal{C}_n(s)$ provides \perp in cases where s does not belong to the proof standard n , which fits together with the disjunction as the top level operator of \mathcal{C}_s , because instead of making the whole formula true, it does not change the outcome.

CHAPTER 4

The RESTful API

In this section we will describe how the API that CARBON provides is structured, and in particular why we call it a RESTful API. To do this, we discuss the central style constraints displayed in figure 2.4 in the background section.

4.1 Cacheable

The cacheable constraint requires that the API provides possibilities for caching, which are in part already supported by HTTP. Of course, all static files included in CARBON allow caching. The static files can be found in the *files/* directory, and include:

<i>/</i>	the root document, that the web root maps to.
<i>files/img/</i>	the directory that contains all image files.
<i>files/css/</i>	the stylesheet directory.
<i>files/js/</i>	the whole code for the JavaScript web application.

In addition parts of the dynamic content can be cacheable, which namely include all versionized content that is not the newest version of a version chain. Since the main data representation that clients handle are *Items*, this affects the major part of all data. However, the newest version of an *Item* cannot be cached completely, as it still may get a deletion datum added, which is also the case if a newer version is written. The *User* data representation however cannot be cached, because it can be modified by clients at all time, and is not under version control. The root document is actually loaded into RAM once CARBON is started, or if any of the files it is composed of changes while the server program runs. Keeping the root document in RAM also increases the speed of delivering it to the client.

4.2 Stateless

The stateless constraint specifies, that all data necessary must be given along with the request, so that the server can operate in a stateless fashion. It is only allowed to keep state in files or databases, but not within the server program itself, and CARBON adheres this rule precisely. Indeed the deterministic nature of pure code in Haskell makes it quite simple to implement a stateless server. There is one special case for CARBON, when calculating extensions for an ADF by using DIAMOND, where the execution of DIAMOND can lead to long running requests, but this does not introduce extra state. Not only does

this constraint allow to distribute different CARBON instances over several servers that access a common database, but it also simplifies debugging, by allowing us to compile and restart the server without recreating several steps in the web application, because we could simply resend a certain HTTP request.

4.3 Layered System

Basing our API on top of HTTP obviously introduces a layered structure already, which is found in the distinction between clients and a server program. However, CARBON also aims to make it possible to easily build on top of it, by providing all dynamic data as JavaScript Object Notation (JSON) and therefore lowering the burden for other programs to use it. A different case of using HTTP in a layered system can be found in our development setup, where we used nginx¹ as a reverse proxy to add Transport Layer Security (TLS) to CARBON.

4.4 Code on demand

The code on demand constraint is satisfied despite its optional nature, because one of the requirements was to build a web application on top of CARBON, so that it became necessary to provide clients with JavaScript code.

4.5 Uniform Interface

To compose a uniform interface, CARBON delivers all data in JSON format, which is independent of its currently used back end. By using an Internet media type, and stating whether they are cacheable, resources and elements become self-descriptive. Elements deliver data on where to find related elements by the resources that belong to their respective Ids. Since changes to such elements are only done onto an element directly, the interface consists of a single layer, and the interface has a uniform structure in total.

¹ <http://nginx.org/en/>: 'nginx [engine x] is an HTTP and reverse proxy server, as well as a mail proxy server, written by Igor Sysoev.'

CHAPTER 5

The technology stack

From its technical design, CARBON is a system consisting of several parts. The main distinction lies between the client side, implemented as a web application, and the server side, which provides the RESTful API. The following diagram gives an overview about the software libraries used to implement CARBON, and where they are used within the system:

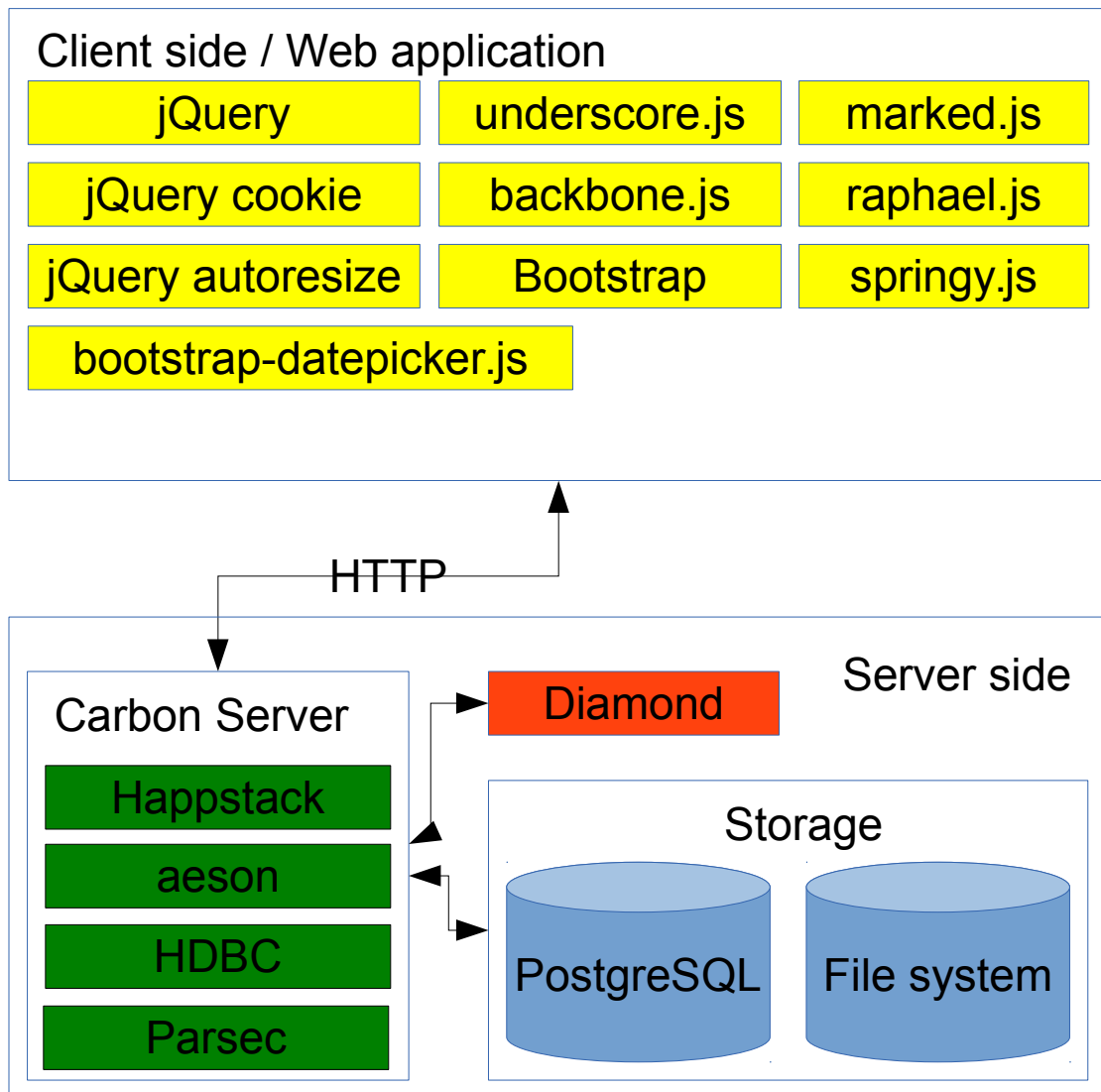


Figure 5.1: Diagram of the technology stack

5.1 Client side

Let us now focus on the client side implementation, the web application part of CARBON. The client side application is initially loaded in a browser by a client, and consists of a HTML document that contains bigger parts of JavaScript and CSS. The web application is designed to communicate with the server side via the RESTful API using only asynchronous requests once it is initially loaded. By using JavaScript Object Notation (JSON) as the exchange format between both sides, the client application profits from the JavaScript native representation, while only data but not information regarding its representation needs to be exchanged between server and client, thus keeping the network load low. The

web application is organized following the Model-View-* pattern, where the controller is omitted or, in parts, replaced by server functionality. This pattern is directly supported by the backbone.js (C.4) library, which supplies definitions for models, views, collections and a router. With modern web applications the router often becomes a central part that assists the web application in routing users through the application in a similar way to links on typical web sites, which also provides the possibility to make use of the browsers address bar and back/forward buttons from within the application. Just like backbone.js, bigger parts of the web application also rely on jQuery (C.1), which is a popular JavaScript library that aims to make tasks like manipulating the DOM and using AJAX requests easier. In particular, the client side implementation relies on the jQuery cookie (C.2) and jQuery autoresize (C.3) plugins, to ease the manipulation of cookies in the browser and to adjust the size of certain inputs with the `<textarea>` tag. The manipulation of cookies is of extra importance, because CARBON relies on them solely as the method of authentication. To further make the code shorter and more readable, the web application makes use of underscore.js (C.5), which is also a dependency for backbone.js. Underscore.js brings the map and fold/reduce style known from functional languages to JavaScript and provides many additional functions that are missing in the standard JavaScript environment, such as set operations and advanced sorting facilities. Since the main concern of the web application is providing a GUI and dealing with user interactions, CARBON also makes use of some libraries that are specifically built to aid the rendering of websites. One of the most popular examples is the Bootstrap (C.9) front-end framework, that provides different helpful building blocks together with a tested standard layout, so that the web application obtains a modern look, while the implementation must not worry about different browsers/devices and the resulting CSS problems. To extend Bootstrap even further, CARBON makes use of the bootstrap-datepicker.js (C.10) library, that provides a datepicker based on Bootstrap. The wiki context incorporated in CARBON makes it necessary that the system enables clients to write articles. For such articles it is typical that wikis do not require users to know HTML, but instead provide a simplified syntax that is then translated into HTML. CARBON solves this problem by the use of marked.js (C.6), which allows the web application to render content written in markdown syntax¹. Since the content is only rendered on the client side, and cannot contain HTML itself, it becomes fairly easy to prevent cross site scripting attacks by simply escaping the according tags, so that no HTML can be supplied by the server side for articles, and all article content is rendered by the client. Another benefit of rendering articles on the client side is, that users get a live presentation of their content while creating it, without bothering the server to compute their HTML. The probably most involved part of the GUI is the presentation of graphs for *Discussions*. To achieve this, the web application relies on springy.js (C.8), which is a force directed graph layout algorithm, that makes it possible to calculate a graph layout only from the graphs own structure, so that no node positions must be synchronized with the server and between different clients. Additional aid for drawing is given by raphael.js (C.7), which is a vector graphic library that makes it

¹ <http://daringfireball.net/projects/markdown/syntax>

easy to build and manipulate svg in the browser.

5.2 Server side

The server side of CARBON is written in the Haskell programming language, and has to cope with various tasks. To allow for interaction with the client side, the server must provide the RESTful API, and to do this, it relies on the Happstack project (C.12). The Happstack library provides a complete HTTP-Server together with functions for routing, query parameters, dealing with client responses, cookie handling and providing static files. In particular, Happstack provides a monad and a corresponding monad transformer that makes it easy to write code depending on client requests while handling the implicit state given by a configuration file and the database connection. To deal with the frequent encoding and decoding issues created by the usage of JSON in the RESTful API, CARBON relies on the library Aeson (C.13), which describes itself as ‘a JSON parsing and encoding library optimized for ease of use and high performance.’¹. The Haskell Database Connectivity (HDBC) library provides CARBON with the means to access the PostgreSQL (C.17) database that is used for persistent storage of mutable data. ‘HDBC provides an abstraction layer between Haskell programs and SQL relational databases’², which brings the benefit that it uses the static type system of Haskell to infer how different basic types must be escaped and parsed when converting between their Haskell and SQL representations. This feature makes sure that the common problem of SQL injection attack can only occur if there is a problem with the central escaping mechanism, and does not depend on code provided by programmers that only rely on the library. Of course, another central part for the server is the interaction with DIAMOND (C.16), which means, that the server needs methods to parse both, the input for DIAMOND and the output given by DIAMOND. To allow the creation of readable and fast parser code, CARBON makes use of Parsec (C.15), a parser combinator that is quite popular and has inspired similar implementations in different other programming languages, such as JavaScript, Ruby, Python or C++³. To coordinate concurrent requests, Happstack makes use of Software Transactional Memory (STM) [Dis06; Har05], and CARBON uses the same implementation to coordinate parallel executions of DIAMOND. Relying on Software Transactional Memory (STM) allows CARBON to benefit from implicit locking algorithms rather than having to declare custom mutexes to coordinate different concurrent threads. The STM implementation provided by Haskell makes it especially easy to write composable memory transactions by using the according monad.

1 <http://hackage.haskell.org/package/aeson-0.6.0.2>

2 <http://hackage.haskell.org/package/HDBC>

3 http://www.haskell.org/haskellwiki/Parsec#Parsec_clones_in_other_languages

CHAPTER 6

Data stored by CARBON

This chapter describes how different informations are represented and handled within our solution. The Unified Modeling Language (UML) class diagrams used are a direct translation of the representation implemented in Haskell on the server side. The web application uses a translation of the Haskell representation to JSON, and uses its own set of models and views on top on that. The database layout is derived from the Haskell implementation by the use of pgModeler¹. PgModeler also generated the Entity-relationship (ER) diagram that can be found in appendix A. Except for the *User* data type all of the following data types implement the Monoid type class described in the background section (2.4.5). This makes it easy for the server side part of CARBON to update and merge different instances of the same type. In particular, since the *Item* type honors the monoid instances of its encapsulated types, it also allows merging of complete *Item* tree structures. This mechanism proved to be particularly useful in conjunction with the CRUD semantics used with the RESTful API.

¹ The PostgreSQL Database Modeler, to be found at <http://www.pgmodeler.com.br/>.

6.1 User

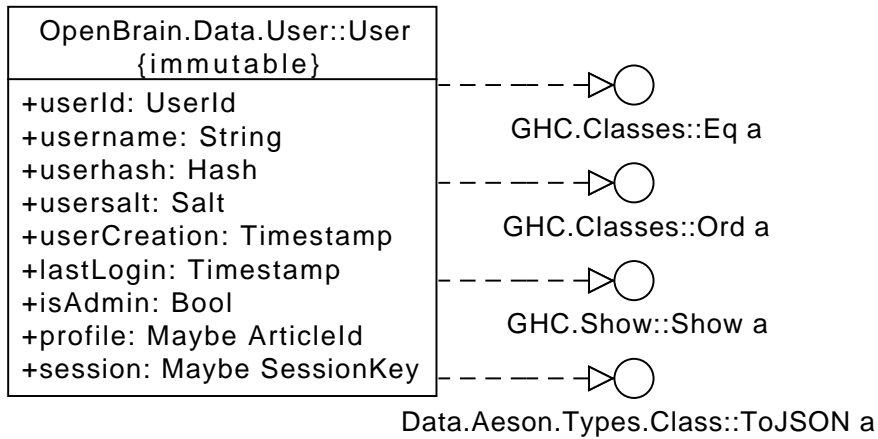


Figure 6.1: UML diagram of the User data type

In order to create or manipulate content within the system, a client needs to log in as a *User*. This allows for a simple form of access control, by distinguishing between simple users and admins. Having clients log in as users also allows to track which content is added or changed, and thereby introduce versioning of content. Authentication of clients is handled via username and password, where the passwords are saved as salted hashes¹ in the database. Users can have an article associated, which will be displayed as their profile in the web application. The session field is used only if a client is logged in, to check the validity of requests by comparing it with the clients cookie.

¹ Currently SHA2 512 bit hashes are used.

6.2 Item

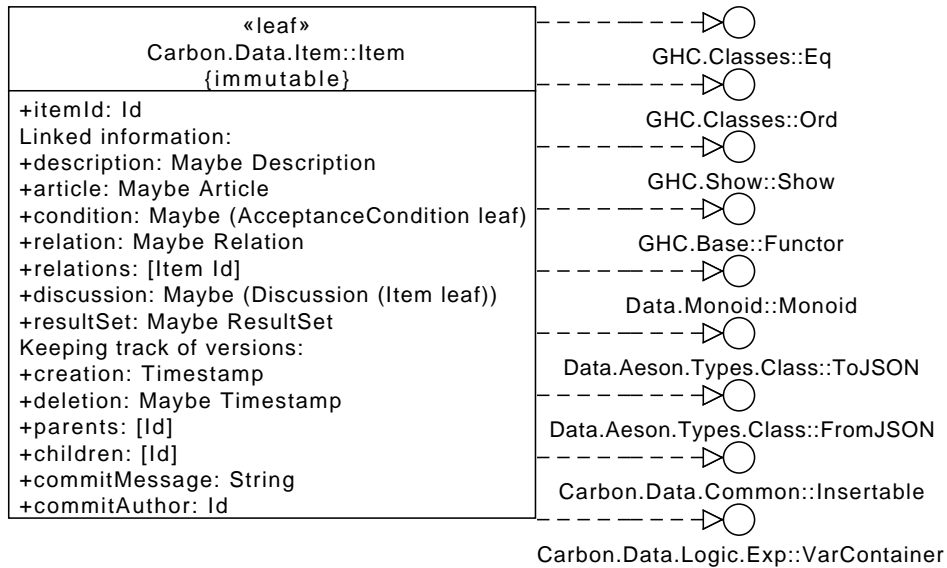


Figure 6.2: UML diagram of the *Item* data type

The *Item* is the most central data type of CARBON. Its structure is derived from two main requirements:

1. All data except for users shall have versioning.
2. Some notions like that of a *Description* or a *Relation* shall be applied to different data.

Usually an *Item* is defined by its linked information. For example an *Item* carrying an *Article* is required to also have a *Description* and is thereafter said to *be an Article*. A more thorough documentation of these predicates on *Items* can be found in appendix B. Besides deciding how a given *Item* can be used the predicates also allow to check if an *Item* is OK to be handled at all by means of the *itemIsSane* predicate. Besides its linked information, an *Item* also carries with it information used for versioning. The two obvious fields for this are *creation* and *deletion*, so that for each version of an *Item* it is known when the change occurred by its creation. Each time an *Item* gets modified, its *deletion* is set to the actual Timestamp and a new *Item* is created, that links to the new data while carrying the *commitAuthor* and a usually auto generated *commitMessage*. The *parents* and *children* fields are simply used to track the tree of versions that each *Item* will have.

6.3 Description

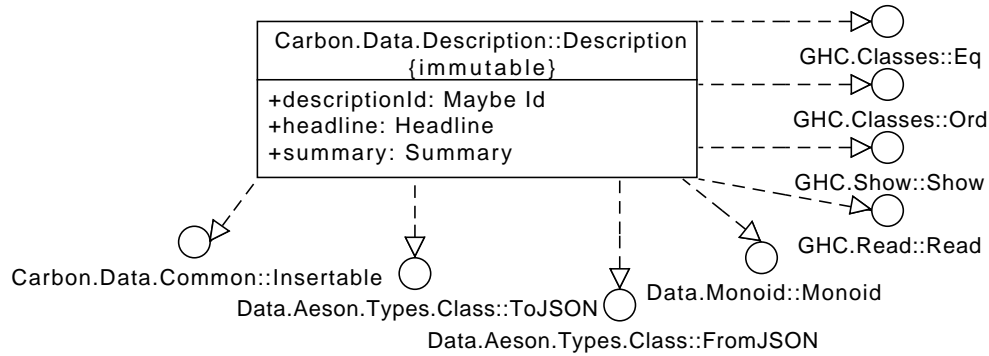


Figure 6.3: UML diagram of the *Description* data type

A *Description* is composed from a *headline* and a *summary*, both of which are strings. Together with the predicates in appendix B this implies, that all sane *Items* are *Descriptions* and thereby have a *summary* and a *headline*. This approach makes sure that valuable information about every item can be displayed to a client, be it an *Article*, a *Discussion* or a *Relation*.

6.4 Article

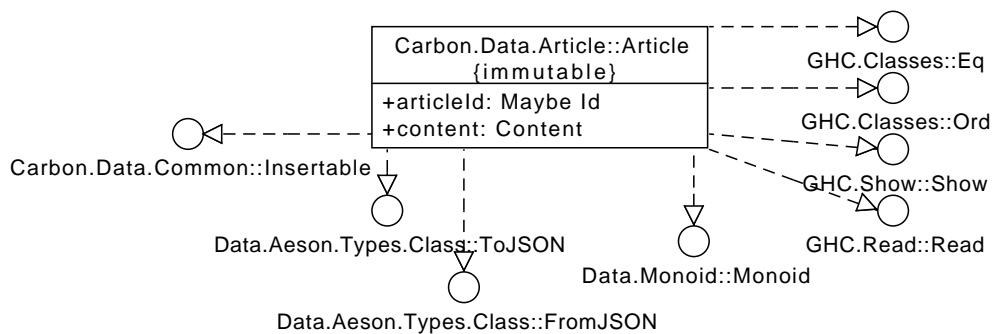


Figure 6.4: UML diagram of the *Article* data type

An *Article* is a single page of versionized content that can be edited by clients. The wiki like functionality of CARBON is build out of *Articles*, whereas *Articles* are used as single

arguments inside *Discussions*.

6.5 AcceptanceCondition

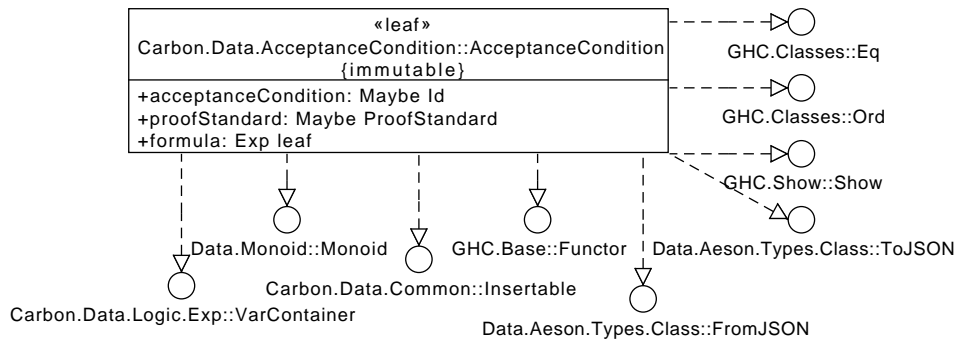


Figure 6.5: UML diagram of the *AcceptanceCondition* data type

Every argument in a *Discussion* must have an *AcceptanceCondition*, so that the *Discussion* can be transformed to an instance for DIAMOND. Whenever an *AcceptanceCondition* is necessary, there are two ways to retrieve it. Either a Client specifies a formula explicitly, or a proofstandard is set for an argument. In case of an explicit formula, the proofstandard is set to *Nothing*, to note that the formula is not inferred by a proofstandard. In case of a proofstandard, every change to the discussion an argument is contained in triggers a recalculation of the formula from the proofstandard. This (re-)calculation is described in detail in section 8.2. The proofstandard itself is basically an enum useful to decide if one argument can be trusted more than another.

6.6 Relation

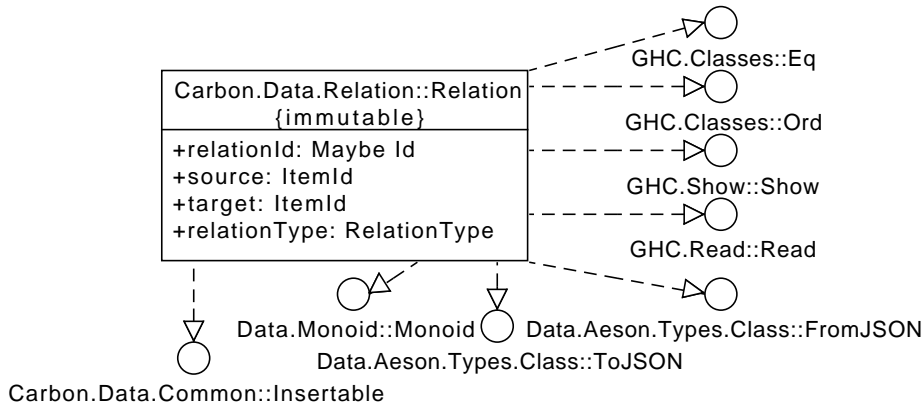


Figure 6.6: UML diagram of the *Relation* data type

A *Relation* represents a directed edge in an ADF, between two arguments. *Relations* can be created in two different fashions. Either a client indicates that a *Relation* of type attack or support exists between two arguments, or a custom type *Relation* is inferred from a custom formula being specified. Work on *Relations* is described in more detail in section 8.3. In addition to *AcceptanceConditions* it is necessary to have explicit *Relations*, so that they can be presented to a client and it is easier to manipulate them directly. CARBON distinguishes between three *RelationTypes*:

Attack: This is a typical Dung-style attack between two arguments.

Support: This indicates that one argument defends another. Basically the number of Supporting arguments must be higher than that of the attacking ones for an argument to be acceptable.

Custom: The relation between two arguments is set to custom whenever the existence of a *Relation* is implied by a custom formula in an *AcceptanceCondition*.

6.7 Discussion

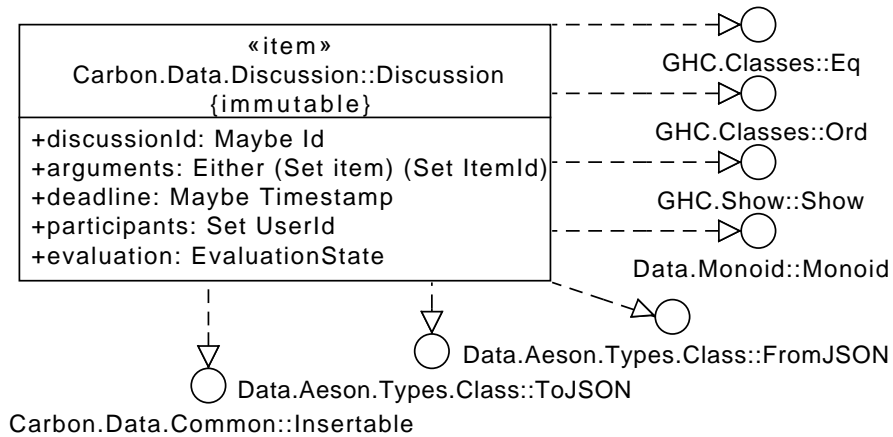
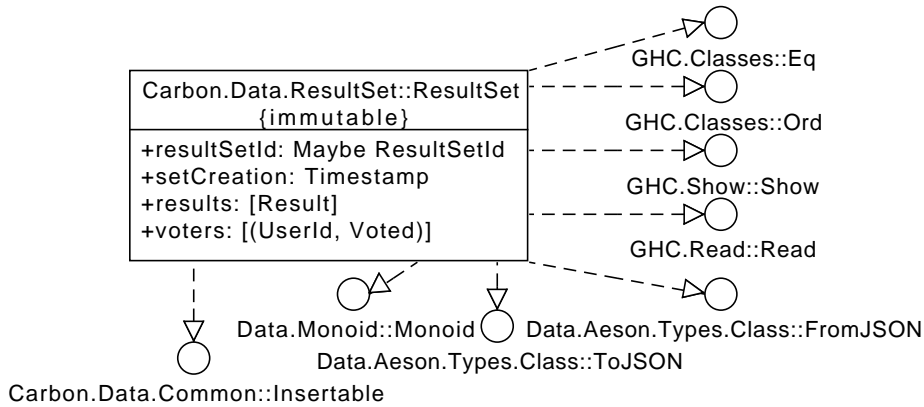


Figure 6.7: UML diagram of the *Discussion* data type

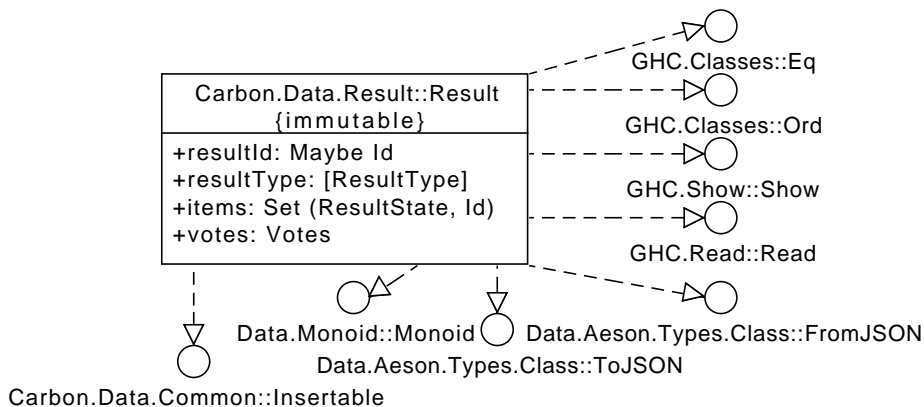
A *Discussion* collects a set of *Articles* as arguments, which can then be evaluated as an ADF by DIAMOND. Every *User* that manipulates a *Discussion* becomes a participant. Once a *Discussion* is evaluated, a *ResultSet* is added to its *Item*, so that all participants can vote on the outcomes and thereby decide on the outcome of a *Discussion*. A *Discussion* can also have a deadline, which is understood as a time after which modification of the *Discussion* is no longer possible. The motivation behind the deadline is to force users to stop changing the argumentation at once, so that final *Results* exist on which voting is possible. For more information on how evaluation of *Discussions* takes place see section 8.4.

6.8 ResultSet

Figure 6.8: UML diagram of the *ResultSet* data type

A *ResultSet* is produced every time a *Discussion* is evaluated with the help of DIAMOND. In addition to a collection of *Results*, the *ResultSet* carries a list of voters. The list of voters is the same as the list of participants in the *Discussion* at the time it was evaluated, but additionally tracks whether a voter has already voted or can still vote.

6.9 Result

Figure 6.9: UML diagram of the *Result* data type

Each *Result* captures a set of *ItemIds* together with the typical labels used by ADFs, $\{\text{In, Udec, Out}\}$, where the label is represented by the type *ResultState*. In addition to this set of items, a *Result* also has a list of *ResultTypes*, which note from what extensions in the ADF a *Result* comes. While it would be possible to compute with models a *Result* is involved with by using set inclusion and deduce this from the definition of the respective models, we choose another, more flexible method. Since CARBON already has all *Results* for each of their models loaded in RAM, it is a simple task to find identical *Results* and merge their model names. This approach is more flexible, should different or additional models be introduced later, due to the fact, that it only operates on model names rather than relying on detailed knowledge about individual models. A *Result* also carries a count of votes, to represent its popularity in comparison to other *Results* in the *ResultSet*.

CHAPTER 7

Implementing a DSL for the back end

During the development of CARBON different reasons arose to implement a DSL in order to separate the server-side back end from the RESTful API. For debugging, it is a lot easier to have a separation of concerns between the back end, which is only concerned with storing and providing data, and the rest of the server. In addition it as suspected, and later proved, to be helpful, if the back end could be exchanged easily should problems with an implementation arise. Lastly, the implementation of a specific DSL makes it possible to compose complex queries out of simpler ones, and execute such queries as complete ACID¹ transactions.

7.1 Composing GADTs by using monads

To embed the DSL in CARBON, we made use of GADTs as described in the background section, but while the tasks in listing 2.12 look similar to lisp code, we wanted to embed the DSL a bit further, and found, that the typical concept for code composition in Haskell, the monad, can be applied to GADTs, allowing us to write back end transactions in the typical do-notation style. To compose arbitrary queries in the DSL, we use two constructors:

```
1 Backendλ :: BackendDSL p -> (p -> BackendDSL r) -> BackendDSL r
2 Nop :: r -> BackendDSL r
```

Listing 7.1: BackendDSL composition constructors

By recalling the definition of the monad typeclass, given in listing 2.4, we see that the type signatures of these constructors are specializations of the according monad functions, which already produces the actual instance implementation:

```
1 instance Monad BackendDSL where
2   (>>=) = Backendλ
3   return = Nop
```

Listing 7.2: BackendDSL Monad

Let us now consider an example for the flexibility we gain from this approach:

¹ Atomicity, Consistency, Isolation, Durability (ACID) - common guarantees for database transactions.

```
5 foo, bar, baz :: BackendDSL Int -> BackendDSL Int
6 foo x = Backendλ x (Nop . (+ 6))
7 bar x = do
8   y <- x
9   return $ y + 6
10 baz = liftM (+ 6)
```

Listing 7.3: BackendDSL Example

The three functions *foo*, *bar*, *baz* share the same type signature and are different means of expressing the function $f(x) = x + 6$ in the back end DSL. While *foo* simply composes the existing constructors provided by the GADT, *bar* is a demonstration of the usual do-notation often encountered in Haskell code, and *baz* provides a short version that could be used to lift the pure function `+6` into any monad.

7.2 Ram only

The first approach to data storage in CARBON was to keep everything in volatile memory, and make use of Haskell's STM implementation to compose transactions and execute them on sets of variables directly. While this made it possible to get some fast test results from our first prototypes of Haskell server side implementations, it became a problem, once the need for persistent storage grew, and we had to change the implementation to get a back end that allows for program restarts.

7.3 Flat files

Coming from the ram only approach, the simplest thing appeared to implement a synchronization from ram to flat files. This was first implemented by reading some files into ram at start up, and writing all changes back into files, once its existing ram representation changed. However, as the server grew, we discovered, that a growing functionality of the back end was similar to transactions in a relational database, up to a point, that it made no longer sense to reimplement custom functions similar to typical SQL.

7.4 MySQL

The similarities to SQL motivated us to exchange the back end from flat files to a popular relational database, MySQL. In addition we already had useful experience from work in typical Linux, Apache, MySQL, PHP (LAMP) setups, which made MySQL the natural choice. For this implementation, we used the `libmysql` library (C.14), that makes use of the according C library for MySQL. Sadly, a harder problem arose, that neither the C library nor `libmysql` handled correctly: The Haskell run-time comes with lightweight threads, similar to the implementations of go or Erlang, that put a significantly lower burden on the system than standard POSIX threads do. These threads are used throughout the server code to handle client requests, and also query the database. To control these threads, the Haskell run-time makes use of signals that allow one thread to continue while another is blocked by IO operations. It appeared, that neither the C library nor `libmysql` mask these signals accordingly, so that the MySQL connection got terminated every time the run-time decided to switch between two threads. This problem was especially hard to reproduce,

because it does not occur in simpler test cases, where only one thread executes the same queries. To fix this problem, there were two possible solutions:

1. We could write a handler function to execute MySQL queries, that keeps the Haskell run-time from changing threads. This concept is called bound threads, and is sometimes used when working with the foreign function interface (FFI) to perform IO heavy work that needs fast reactions from the run-time. The problem with this approach is, that it introduces performance burdens for the server side code, and should not be something the server code has to deal with, as the masking of signals should either be done in the HDBC library, or in the C library. Newer versions of the HDBC library come with a function *withRTSignalsBlocked*¹, but also describe, that this is a problem stemming from the C library.
2. The alternative solution was to exchange MySQL for a different database. PostgreSQL was the candidate of choice at this point, and would only need slight adjustments, because it also uses SQL.

7.5 PostgreSQL

PostgreSQL is the current back end for CARBON, and we make use of pgModeler² to construct the ER diagram (A) and the database schema with all its constraints. It turned out that the HDBC library binding for PostgreSQL has no problems with signals, and that PostgreSQL comes with some additional features that could be put to use. Even though PostgreSQL is currently the only back end implemented in CARBON, there exist means to specify the back end that should be used in the config file, and it would be easy to add additional back ends. In particular, the back end system could be used to introduce proxy back ends, which could, for example, log transactions while forwarding to a different back end to perform the actual work.

¹ <http://hackage.haskell.org/package/HDBC-mysql-0.6.6.1/docs/Database-HDBC-MySQL.html>

² <http://www.pgmodeler.com.br/>

CHAPTER 8

Algorithms developed for CARBON

To realize CARBON it was necessary to develop specific algorithms for some problems, which are mostly concerned with logic necessary to handle pForm-ADFs. This section describes our custom algorithms and starts with our type declaration to represent propositional formulas. Afterwards the function `autoCondition` is described, which implements our translation \mathcal{T} to map from proof standards and bipolar relations to pForm-BADFs.

8.1 Representing propositional formulas in CARBON

Propositional formulas in CARBON are represented using the ADT `Exp a`, which carries a type variable to represent the type of its variables. Having an exchangeable variable type allows us to use database ids as well as strings for formulas, which makes it possible to work with ids in the algorithms, but still present string names to DIAMOND and the user.

```
22 data Exp a = Var    a
23             | And   (Exp a) (Exp a)
24             | Or    (Exp a) (Exp a)
25             | Neg   (Exp a)
26             | Const Bool
27             deriving (Eq, Ord)
```

Listing 8.1: Carbon.Data.Logic.Exp

Using `foldl`, we can generalize the `And` and `Or` constructors to work on lists:

```
63 and' = foldl And $ Const True
64 or'  = foldl Or  $ Const False
```

Listing 8.2: Carbon.Data.Logic.Exp

8.2 Calculating acceptance conditions

The calculation of acceptance conditions for arguments is performed by the two functions `autoCondition` and `mkFormula` in the module `Carbon.Backend.Logic`, where `autoCondition` makes sure that all arguments have acceptance conditions, and `mkFormula` is used to calculate the condition if a proof standard is given instead of a custom condition set by a client.

8.2.1 The work of autoCondition

In general autoCondition distinguishes three cases:

```

1 autoCondition :: Item Id -> BackendDSL (Item Id)
2 autoCondition item
3   | Maybe.isNothing $ condition item = do
4     let p = mempty :: ProofStandard
5         c = mempty :: AcceptanceCondition Id
6     autoCondition $ item <+ c <+ p

```

Listing 8.3: Carbon.Backend.Logic:autoCondition, case 1

In this case autoCondition was called with a newly created *Item*, that does not have an *AutoCondition*. Therefore an empty condition is created and added to the item, which is then used to call autoCondition again.

```

7   | Maybe.isJust . proofStandard $ getC item = do
8     let rels = Maybe.mapMaybe relation $ relations item
9         incomming = filter ((itemId item ==) . target) rels
10        sourceIds = map source incomming
11        (sourceErrors, sourceItems) <- liftM Either.partitionEithers $ mapM GetItem
12        sourceIds
13        unless (null sourceErrors) $ -- Exception for errors
14        let problem = "Could not fetch items in Carbon.Backend.Logic:autoCondition"
15            :sourceErrors
16            in error $ unlines problem
17        let getProofStandard = mkGetProofStandard sourceItems
18            newExp = mkFormula (getP item) incomming getProofStandard
19        return $ item <+ getC item <+ newExp

```

Listing 8.4: Carbon.Backend.Logic:autoCondition, case 2

The guard in line 7 makes sure that in this case the item has a *ProofStandard*. Therefore we can just gather a bit more information on the item from the database, so that mkFormula can be used to calculate a condition which is then written into the item.

```

18   | otherwise = do
19     let incomming = filter ((==) (itemId item) . target . getR) $ relations item
20         notCustom = filter ((/=) RelationCustom . relationType . getR) incomming
21         setCustom = map (\i -> i <+ getR i <+ RelationCustom) notCustom :: [Item
22         Id]
23     (errs, _) <- liftM Either.partitionEithers $ mapM SetItem setCustom
24     unless (null errs) $
25     let problem = "Could not set items in Carbon.Backend.Logic:autoCondition":
26         errs
27     in error $ unlines problem
28     return item

```

Listing 8.5: Carbon.Backend.Logic:autoCondition, case 3

If the *AcceptanceCondition* has no proof standard set, but a formula, it is clear, that the formula was set by a client. In this case autoCondition just makes sure, that all incoming

relations to the argument are of type *Custom*, so that the nature of the relations is visible to clients.

8.2.2 How mkFormula proceeds

```

1 mkFormula :: ProofStandard -> [Relation] -> (ItemId -> ProofStandard) -> Exp
  ItemId
2 mkFormula itemProofStandard incomming getProofStandard =
3   let incoming' = filter ((itemProofStandard <=) . getProofStandard . source)
      incomming
4     rFilter r = filter $ (r ==) . relationType
5     attacks = rFilter RelationAttack incoming'
6     supports = rFilter RelationSupport incoming'

```

Listing 8.6: Carbon.Backend.Logic:mkFormula

When looking at the type signature of `mkFormula` in line 1, we notice that it takes a *ProofStandard*, a list of *Relations*, and a function that maps *ItemIds* to *ProofStandards*. This is enough to calculate the acceptance condition for an *Item* based on its proof standard. In line 3, the list of incoming relations, *incoming* is filtered to create a list *incoming'* that only contains relations where the source has a higher or equal *ProofStandard* than the one given by *itemProofStandard*. This is useful, because arguments with lower proof standards cannot effect the acceptance of arguments with a higher proof standard, as they cannot attack it successfully, and their support will never outweigh an attack from a higher proof standard. The lines 4-6 are used to separate the incoming relations into attacking and supporting ones.

```

7   asPairs = do -- :: [(Set Relation, Set Relation)]
8     let hasP p = filter $ (p ==) . getProofStandard . source
9         p <- [minBound ..] :: [ProofStandard]
10    let as = hasP p attacks
11        ss = hasP p supports
12    return (Set.fromList as, Set.fromList ss)

```

Listing 8.7: Carbon.Backend.Logic:mkFormula

`asPairs` is constructed so that `mkFormula` can iterate over pairs of attacking and supporting relations, while taking for granted that they belong to the same *ProofStandard*.

```

13   conditions = do -- :: [Exp ItemId]
14     let body = map (Var . source) . Set.toList
15         nBody = map Neg . body

```

Listing 8.8: Carbon.Backend.Logic:mkFormula

The next task is to generate a list of all possible cases where we want to accept an argument. To help with this, we also define the two functions *body*, *nbody*, which aid building the positive and negative bodies.

```

16   (attackSet, supportSet):breakers <- List.init $ List.tails asPairs
17   let breakList = map (Neg . Var . source) . Set.toList . Set.unions $ map
fst breakers

```

```

18     guard . not $ Set.null attackSet && Set.null supportSet
19     attackSet' <- powerset' attackSet
20     guard . not $ Set.null attackSet'

```

Listing 8.9: Carbon.Backend.Logic:mkFormula

This part iterates all tuples of attacks and supports for each proof standard from line 16 on. Note that the list *breakers* is comprised of all such tuples from higher proof standards due to the way that *List.init . List.tails* works. *breakers* gets its name from the nature, that its elements are suited to deny an acceptance condition. To exploit this fact, the *breakList* is created from all attack relations in *breakers* in line 17. Afterwards we iterate over the power set of all attack relations, if there are attack and support relations given, while making sure that we skip empty elements of that power set.

```

13     let canDefend = Set.size attackSet' < Set.size supportSet
14     if canDefend
15         then do
16             supportSet' <- powerset' supportSet
17             guard $ Set.size supportSet' >= Set.size attackSet'
18             let notAttack = Set.difference attackSet attackSet'
19             if null breakList
20             then return . and' $ body attackSet' ++ body supportSet' ++ nBody
notAttack
21         else do
22             breaker <- breakList
23             return . and' $ body attackSet' ++ body supportSet' ++ nBody
notAttack ++ [breaker]
24         else if null breakList
25             then return . and' $ nBody attackSet'
26         else do
27             breaker <- breakList
28             return . and' $ nBody attackSet' ++ [breaker]

```

Listing 8.10: Carbon.Backend.Logic:mkFormula

Now *mkFormula* has to deal with 4 cases determined by two conditions. The first condition being whether it is possible to defend the *attackSet'*, and the second being if the *breakList* is empty. When the *attackSet'* can be defended, *mkFormula* generates all *supportSet'*s from the power set of *supportSet*, that have more elements than the current *attackSet'*. If the *breakList* is not empty, a single element from it is enough to deny the acceptance, so that for each of its entries a separate condition is generated.

```

1     in null conditions ? (Const True, simplify $ or' conditions)

```

Listing 8.11: Carbon.Backend.Logic:mkFormula

The last line of *mkFormula* simply checks if there are any conditions at all, which are than joined into a disjunction, or if the argument can just be accepted. The function *simplify* is

used to reduce some obvious cases like $and(c(t),x)$ to x .

8.3 The function fitInstance

CARBON allows a user to upload an ADF instance to a *Discussion*. The necessary adjustment to a *Discussion* that arises from such an upload is performed by the function `fitInstance` in the module *Carbon.Backend.Logic*. There are three basic changes that `fitInstance` has to deal with:

8.3.1 Adding missing nodes

The first step to perform in `fitInstance` is to add missing arguments. For the user all arguments are identified by their name, rather than their id in the database. Since names for arguments are saved as the headline in the *Description* of an *Item*, a set of current headlines, *args*, is computed. Now the sub function `addMissingNodes` uses *args* and the set of variable names *names(i)* to compute the set of missing arguments as $newHeads = names(i) \setminus args$, which are then added to the *Discussion*.

8.3.2 Calculating possible relations

The next step is to calculate the set of possible relations, *possibleRels*, in a *Discussion* so that by using the set difference with the current relations, the relations that should be added or removed can be computed. Since *Relations* link between Ids rather than headlines, a map from headlines to ids, the *hToIdMap* is created. With this map the possible relations can be computed by simply using each id in the map as the target of a relation and each variable in its acceptance condition as a source.

8.3.3 Adding and Removing relations

From the current *Discussion*, a set of current relations, *currentRels* can be projected, so that it is possible to calculate the sets of relations that should be added to the discussion as well as those that should be removed:

$$addRels = possibleRels \setminus currentRels \tag{8.1}$$

$$delRels = currentRels \setminus possibleRels \tag{8.2}$$

After adjusting the *Discussion* to the new relations, `fitInstance` finishes with saving the newly generated data.

8.4 Evaluation of discussions

The evaluation of *Discussions* is performed with the help of DIAMOND and implemented in the module *Carbon.Data.Logic.Evaluation*. The entry point for the evaluation module is the `run` function, which takes the *Config* loaded by CARBON, a *FilePath* and a *String* as the content that should be evaluated by DIAMOND. Evaluation happens in the following basic steps:

1. Writing the content for DIAMOND in a file at the given *FilePath*

2. Calling DIAMOND concurrently for each *ResultType* specified in the *Config*¹
3. Parsing the output of DIAMOND in each concurrent thread
4. Gathering the parsed outputs to return a combined result

For the concurrent evaluation, CARBON uses a shared state that is comprised of STM transactional variables. The shared state keeps track of the running threads and their results. Now the `run` function, after initializing the shared state and writing the content for DIAMOND to the given *FilePath*, starts threads for each *ResultType* specified in the configuration file. Afterwards the `run` function starts an additional `watchDog` thread, which sleeps for 10 seconds before stopping all threads that have not already finished. On the one hand the watchdog thread was necessary to ensure that CARBON can give a reply to the client starting the evaluation before a HTTP timeout, and on the other hand we have experienced cases where DIAMOND would not finish in reasonable time. After the watchdog is started by `run`, STM is used to wake up `run` as soon as all worker threads have finished their work, so that the results can be gathered and returned.

¹ The configuration file is defined in *Carbon.Config* and defines a mapping from each requested *ResultType* to a list of parameters for DIAMOND.

CHAPTER 9

Discussion

In this thesis we explained and documented the design and implementation of *Collaborative Argumentation Brought Online* (CARBON). In doing so we described the situation before our solution, which motivated our work. From this motivation, we drew our problem description, that identified a need for more services that supply abstract argumentation to the web. We also set out to create a web interface for DIAMOND, and to provide a RESTful API that would enable future creations on top of it. To achieve this we found a way of integrating ADFs in the concept of a wiki. We could make the concept of abstract argumentation even more accessible by introducing a conversion from BAFs with proof standards to ADFs, that makes it possible for users to only specify how certain some arguments are (proof standards), and if they support or attack each other, while CARBON can still work on the basis of ADFs. To realize our solution, we could draw on some specific features of the Haskell programming language. Namely we realized our own DSL, which made it possible to abstract from a concrete realization of data storage and still operate with composable transactions. In addition using Haskell also enforced an implementation style that fits a RESTful API very well by avoiding state where possible.

CHAPTER 10

Related work

While there are already a number of tools that allow to work with argumentation structures both via a polished GUI, and by integration into other programs, the number of applications that bring argumentation to the web is rather small. This can change through the introduction of the growing Argument Interchange Format (AIF) as well as through projects like ArguBlogging. While ArguBlogging builds an argumentative structure on top of blogs, and already delivers integration with a World Wide Argument Web (WWAW), CARBON is built on top of a wiki like structure, and integration into a WWAW or usage of an AIF are seen as possible future works.

10.1 The Argument Interchange Format

The paper ‘Towards an argument interchange format’[Che06] proposes a core ontology for an AIF that abides by the following principles:

Machine readable syntax: An AIF should focus on being machine readable rather than being used by humans.

Explicit, machine processable semantics: Different tools should be able to work with an AIF independent of special, tool dependent, implicit knowledge.

Unified abstract model, multiple reifications: The proposed model is abstract and lacks a realization. It is expected that multiple reifications can exist.

Extensibility: Only a core concept together with possibilities to extend it is defined.

This work is extended by ‘Laying the foundations for a World Wide Argument Web’[Rah07], which provides a reification that makes use of RDF and RDFS to represent argumentation structures in the semantic web. The article ‘AIF⁺: Dialogue in the Argument Interchange Format’[Ree08] extends the works on an AIF by means to represent dialogic argumentation, a feature that is used by ArguBlogging. Finally the article ‘The Argument Interchange Format’[Rah09a] summarizes a core of the AIF and gives an overview of different applications that are either already implementing the AIF or could benefit from its usage. The parallel between the development of an AIF and the construction of CARBON lies in the fact that both have a focus on interchanging argumentation structures in a networked context. This

implies that it could be of great benefit for CARBON to make it capable of using AIF, which is further discussed in section 11.

10.2 ArguBlogging

ArguBlogging, as described in the ‘Implementing ArguBlogging’[Sna12a] uses the AIF and allows users to capture the argumentative structure between different blog posts and their responses, so that this structure becomes more obvious than the implicit one given by hyperlinks. To achieve this, ArguBlogging provides a Bookmarklet¹, which enables users to select text on a website, mark if they agree or disagree with it, and post their reason to a blogging platform. ArguBlogging builds on the works of [Rah07; Ree08] to become part of a WWAW.

10.3 Argumentation in Haskell

‘Tools for the implementation of argumentation models’[Gij13a] presents, that ‘functional programming allows to realize structured argumentation models in such a way that the implementation is sufficiently close to the mathematical definitions to serve as specifications in their own right’[Gij13a, p 47] by using Haskell as the example. Additional upsides of using Haskell are the possibility of verifying (parts of) the implementation in a Theorem prover like Agda², or to deliver the documentation for the code with the code through the use of literal programming. In ‘Haskell gets Argumentative’[Gij13b] the example of Carneades is used for this approach, and by using a domain-specific language, the task of using Carneades becomes easier in a programming environment. This strong formalization can also be helpful to enable exchange between different programs, like the AIF, except, that it tackles the problem from a different angle, starting with a formal definition rather than an ontology.

10.4 ASPARTIX

The Answer Set Programming Argumentation Reasoning Tool (ASPARTIX) is ‘ a tool for computing acceptable extensions for a broad range of formalizations of Dung’s argumentation framework and generalizations thereof. ’[Egl08] It’s relation to AFs is therefore similar to that of DIAMOND towards ADFs. This implies, that CARBON could also have been build on top of ASPARTIX rather than DIAMOND. Though DIAMOND was the candidate of choice from our motivation on, it appears to be possible that CARBON could also make use of ASPARTIX in the future, should a situation arise, that demands more involvement with AFs rather than ADFs.

10.5 TOAST

The Online Argument Structures Tool (TOAST) presents itself through a web form³, and via an API, and is an implementation of the ASPIC+ framework, that was described in

1 www.bookmarklets.com/about/

2 Agda is a dependently typed functional programming language as well as a proof assistant. Its source can be found at <http://hackage.haskell.org/package/Agda>.

3 <http://www.arg.dundee.ac.uk/toast/>

[Pra10].

TOAST accepts a knowledge base and rule set with associated preference and contrariness information, and returns both textual and visual commentaries on the acceptability of arguments in the derived abstract framework. [Sna12b]

TOAST is an example of a service with a focus on a single task, and thereby leads to considerations of possible improvements on CARBON, as discussed in the following chapter (11.3.1), regarding the monolithic structure of CARBON.

CHAPTER 11

Future work

Based on our work on CARBON, different opportunities for future works become available and can be pursued. In this chapter, we will first describe possible improvements in the web application part, that sets the limits for the actions that users can perform. Afterwards, we focus on possible paths to increase the possibilities for using CARBON, by either integrating it with the AIF, or integrating it further with existing Haskell solutions in the field. Finally there are some improvements possible on the actual implementation, that also need consideration.

11.1 Additional features for user interaction

While the web application can react faster than individual page loads, and even comes with features like the presentation of argument graphs, that allows a specialized kind of interaction not possible with the standard HTML5 page elements, there is still room left for some improvements. Even though the web application displays graphs, there is currently no good way to save or print such graphs, as they are only created with the help of JavaScript, and are not in a format that would support such things. In addition CARBON makes it possible to have commit messages for changes to any data stored in the back end, but the web app currently does not take advantage of this, and instead uses fixed commit messages depending on the task performed rather than the users intent. An improvement upon this would be, to let the user simulate some changes in the web application only, and once the user wants to save these changes to the server, the web application should summarize all changes, let the user specify a commit message, and send them to the server in a single request. Finally the web application could support even more user interaction. While users can already engage in common discussions, it would be helpful, if they could also vote over acceptance conditions, proof standards or relations between their arguments in general. Currently CARBON relies on its users to figure out the correct structure of arguments and their relations by themselves, but it appears logical, that this can also be a topic of discussion between users. Since our solution also presents an API it is also feasible to create entirely new services that make us of it as seen fit.

11.2 Integrating CARBON with the AIF

As a web service and application with a focus on argumentation, it would be very interesting to have CARBON become part of a WWAW. To achieve this, it is necessary to find an adoption of the AIF for ADFs, and to implement it in CARBON or related tools. It would also be interesting to see if current AIF representations could be transferred into such an adoption and if this could be used to apply ADFs to a wider domain, while still preserving the original semantics.

11.2.1 Integrating with argumentation in Haskell

With works like [Gij13a; Gij13b] making use of the Haskell language with a special focus on argumentation, it appears an interesting target, to investigate works on argumentation based on a Haskell background further, and, if possible make our works on CARBON available to integrate with current solutions. This would strengthen the existing supply of Haskell code for argumentation, and, as shown by [Gij13b], could make argumentation in programming environments easier to use.

11.2.2 Detection of relation types between arguments

Currently, CARBON relies entirely on its users to specify the types relations between arguments have, and once an acceptance condition is set directly rather than by specifying a proof standard and attack or support relations, CARBON no longer has any information about these relation types. These relation types could be discovered by solving the attack link problem for given ADFs. But since [Ell12][p. 57] has shown that this problem is coNP-complete, we didn't approach this problem in our current implementation. It may, however, be possible, that the problem can be solved at for many or some of the ADFs entered in CARBON, so that it would be an interesting challenge to see what is possible within a reasonable effort.

11.3 Possible improvements to the implementation

With CARBON being close to 10.000 lines of code, different bugs are expected to show up, despite our choice for a programming language that helps avoiding such. These bugs will, of course, be a concern in the future, and shall be patched when they become apparent. Despite these errors, there are some parts about CARBON that can be improved. A small example is given by our Item data representation which currently implements versioning as a parent-child relation ship, that is stored in the database up to any possible length. It would be helpful, to implement a garbage collection that checks for such relations over a certain length or age, and removes items accordingly, so that the version history kept by CARBON can be configured to grow only as large as intended. A bigger example of a possible improvement is given in the following subsection:

11.3.1 Breaking the monolithic structure of CARBON

Despite its separation into a Web Application and a server side API, CARBON currently has to manage several different tasks. We consider it a good idea to split CARBON into several smaller parts that focus on different tasks but work together well. This would make upgrades easier and, because there would be several programs, rather than one, code could

be easier to maintain and reason about. In case of such a separation, all parts could make their methods available via a RESTful API for other tasks to use. Possible parts would be:

- A wiki that encapsulates the content and makes it possible to refer to specific arguments and outcomes of discussions.
- An argumentation engine, that makes it simple to identify central arguments from a wiki, and group such arguments according to topic. This part could also become a part of a WWAW through implementing the AIF. Having a central service for argumentation with ADFs would also make it possible to discover related arguments or topics, a feature, that is currently missing in CARBON.
- A discussion service, that allows clients to group arguments, and evaluate them just like discussions in CARBON currently are. Such a service would also handle the results and votes on them.
- A presentation service, that serves a web application, and has a focus on presentation features. This would make the presentation less entangled with the rest of CARBON, and could also serve as a model on how to build clients for such a family of services.
- A DIAMOND service, which only focuses on the evaluation of ADFs via DIAMOND. This would be mostly a simple job scheduler, but could also give detailed load statistics, which are currently obscured by CARBON. Since evaluation of bigger ADFs with DIAMOND could take time longer than the usual timeout, having such a service could also mitigate the current need for a watchdog function in CARBON, that makes sure DIAMOND terminates correctly, or is terminated after a timeout.

Acronyms

Acronyms

Notation	Description
ACID	Atomicity, Consistency, Isolation, Durability
ADF	abstract dialectical framework
ADT	algebraic data type
AF	argumentation framework
AIF	Argument Interchange Format
ASPARTIX	Answer Set Programming Argumentation Reasoning Tool
BADF	bipolar abstract dialectical framework
BAF	bipolar argumentation framework
CARBON	<i>Collaborative Argumentation Brought Online</i>
CRUD	Create, Read, Update and Delete
DIAMOND	<i>Dialectical Models Encoding</i>
DSL	domain specific language
EDSL	embedded domain specific language
ER	Entity-relationship
FFI	foreign function interface
GADT	generalized algebraic data type
GHC	Glasgow Haskell Compiler
HDBC	Haskell Database Connectivity
JSON	JavaScript Object Notation

Notation	Description
LAMP	Linux, Apache, MySQL, PHP
pForm-ADF	propositional formula ADF
pForm-BADF	<i>propositional formula</i> BADF
REST	Representational State Transfer
STM	Software Transactional Memory
TLS	Transport Layer Security
TOAST	The Online Argument Structures Tool
UML	Unified Modeling Language
URI	Uniform Resource Identifier
WWAW	World Wide Argument Web

Bibliography

- [Amg04] LEILA AMGOUD, CLAUDETTE CAYROL, and MARIE-CHRISTINE LAGASQUIE-SCHIEX: ‘On the bipolarity in argumentation frameworks’. In *NMR*. Ed. by JAMES P. DELGRANDE and TORSTEN SCHAUB. 2004: pp. 1–9 (cit. on p. 7).
- [Ber96] TIM BERNERS-LEE, ROY T. FIELDING, and HENRIK FRYSTYK NIELSEN: *RFC 1945 – Hypertext Transfer Protocol – HTTP/1.0*. <http://www.faqs.org/rfcs/rfc1945.html>. May 1996 (cit. on p. 19).
- [Bre13] GERHARD BREWKA, HANNES STRASS, STEFAN ELLMAUTHALER, JOHANNES PETER WALLNER, and STEFAN WOLTRAN: ‘Abstract Dialectical Frameworks Revisited’. In *IJCAI*. Ed. by FRANCESCA ROSSI. IJCAI/AAAI, 2013 (cit. on pp. 9–11).
- [Bre10] GERHARD BREWKA and STEFAN WOLTRAN: ‘Abstract Dialectical Frameworks’. In *KR*. Ed. by FANGZHEN LIN, ULRIKE SATTLER, and MIROSLAW TRUSZCZYNSKI. AAAI Press, 2010 (cit. on pp. 9–11).
- [Che06] CARLOS IVÁN CHESÑEVAR, JARRED MCGINNIS, SANJAY MODGIL, IYAD RAHWAN, CHRIS REED, GUILLERMO RICARDO SIMARI, MATTHEW SOUTH, GERARD VREESWIJK, and STEVEN WILLMOTT: ‘Towards an argument interchange format’. In *Knowledge Eng. Review* (2006), vol. 21(4): pp. 293–316 (cit. on p. 55).
- [Dis06] ANTHONY DISCOLO, TIM HARRIS, SIMON MARLOW, SIMON L. PEYTON JONES, and SATNAM SINGH: ‘Lock Free Data Structures Using STM in Haskell’. In *FLOPS*. Ed. by MASAMI HAGIYA and PHILIP WADLER. Vol. 3945. Lecture Notes in Computer Science. Springer, 2006: pp. 65–80 (cit. on p. 32).
- [Dun95] PHAN MINH DUNG: ‘On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and n-Person Games’. In *Artif. Intell.* (1995), vol. 77(2): pp. 321–358 (cit. on p. 5).
- [Egl08] UWE EGLY, SARAH ALICE GAGGL, and STEFAN WOLTRAN: ‘ASPARTIX: Implementing Argumentation Frameworks Using Answer-Set Programming’. In *ICLP*. Ed. by MARIA GARCIA de la BANDA and ENRICO PONTELLI. Vol. 5366. Lecture Notes in Computer Science. Springer, 2008: pp. 734–738 (cit. on p. 56).
- [Ell12] STEFAN ELLMAUTHALER: ‘Abstract Dialectical Frameworks: Properties, Complexity, and Implementation’. MA thesis. Technische Universität Wien, Institut für Informationssysteme, 2012 (cit. on pp. 9, 23, 60).

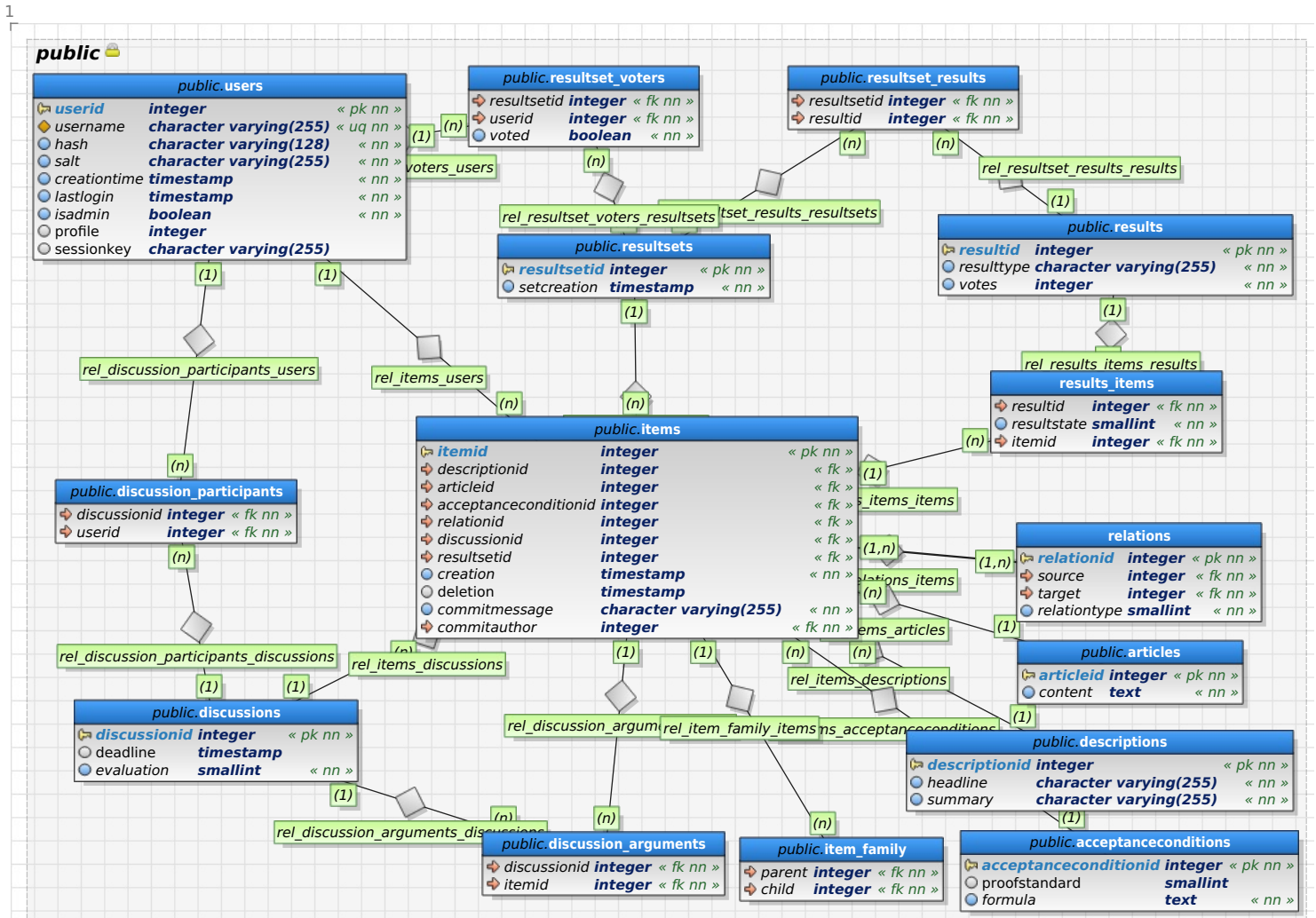
- [Far95] ARTHUR M. FARLEY and KATHLEEN FREEMAN: ‘Burden of Proof in Legal Argumentation’. In *ICAIL*. 1995: pp. 156–164 (cit. on p. 11).
- [Fie99] R. FIELDING, J. GETTYS, J. MOGUL, H. FRYSTYK, L. MASINTER, P. LEACH, and T. BERNERS-LEE: *RFC 2616, Hypertext Transfer Protocol – HTTP/1.1*. 1999 (cit. on p. 19).
- [Fie00a] ROY T. FIELDING and RICHARD N. TAYLOR: ‘Principled design of the modern Web architecture’. In *ICSE*. Ed. by CARLO GHEZZI, MEHDI JAZAYERI, and ALEXANDER L. WOLF. ACM, 2000: pp. 407–416 (cit. on p. 21).
- [Fie00b] ROY THOMAS FIELDING: ‘Architectural Styles and the Design of Network-based Software Architectures’. PhD thesis. University of California, Irvine, 2000 (cit. on pp. 19, 20).
- [Gij13a] BAS van GIJZEL: ‘Tools for the implementation of argumentation models’. In *2013 Imperial College Computing Student Workshop*. Ed. by ANDREW V. JONES and NICHOLAS NG. Vol. 35. OpenAccess Series in Informatics (OASISs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013: pp. 43–48 (cit. on pp. 56, 60).
- [Gij13b] BAS van GIJZEL and HENRIK NILSSON: ‘Haskell Gets Argumentative’. In *Trends in Functional Programming - 13th International Symposium, TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers*. Ed. by HANS-WOLFGANG LOIDL and RICARDO PEÑA. Springer, 2013: pp. 215–230 (cit. on pp. 56, 60).
- [Gir03] JEAN-YVES GIRARD, PAUL TAYLOR, and YVES LAFONT: *Proofs and Types*. New York, NY, USA: Cambridge University Press, 2003 (cit. on p. 13).
- [Har05] TIM HARRIS, SIMON MARLOW, SIMON L. PEYTON JONES, and MAURICE HERLIHY: ‘Composable memory transactions’. In *PPOPP*. 2005: pp. 48–60 (cit. on p. 32).
- [Lei01] DAAN LEIJEN and ERIK MEIJER: *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Tech. rep. UU-CS-2001-27. Department of Computer Science, Universiteit Utrecht, 2001 (cit. on p. 76).
- [Lip11] MIRAN LIPOVAČA: *Learn You a Haskell for Great Good!* 38 Ringold St., San Francisco, CA 94103 USA: No Starch Press, Inc., Apr. 2011.
- [Mar10] SIMON MARLOW: *Haskell 2010 Language Report*. 2010 (cit. on p. 14).
- [OSu08] BRYAN O’SULLIVAN, JOHN GOERZEN, and DON STEWART: *Real World Haskell*. 1st. O’Reilly Media, Inc., 2008 (cit. on p. 16).
- [Pau14] CESARE PAUTASSO: ‘RESTful Web Services: Principles, Patterns, Emerging Technologies’. In *Web Services Foundations*. 2014: pp. 31–51 (cit. on p. 19).
- [Pra10] HENRY PRAKKEN: ‘An abstract framework for argumentation with structured arguments’. In *Argument & Computation* (2010), vol. 1(2): pp. 93–124 (cit. on p. 57).

- [Rah09a] IYAD RAHWAN and CHRIS REED: ‘The Argument Interchange Format’. English. In *Argumentation in Artificial Intelligence*. Ed. by GUILLERMO SIMARI and IYAD RAHWAN. Springer US, 2009: pp. 383–402 (cit. on p. 55).
- [Rah09b] IYAD RAHWAN and GUILLERMO R. SIMARI: *Argumentation in Artificial Intelligence*. 1st. Springer Publishing Company, Incorporated, 2009 (cit. on pp. 5, 7, 8, 24).
- [Rah07] IYAD RAHWAN, FOUAD ZABLITH, and CHRIS REED: ‘Laying the foundations for a World Wide Argument Web’. In *Artif. Intell.* (2007), vol. 171(10-15): pp. 897–921 (cit. on pp. 55, 56).
- [Ree08] CHRIS REED, SIMON WELLS, JOSEPH DEVEREUX, and GLENN ROWE: ‘AIF+: Dialogue in the Argument Interchange Format’. In *COMMA*. Ed. by PHILIPPE BESNARD, SYLVIE DOUTRE, and ANTHONY HUNTER. Vol. 172. *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2008: pp. 311–323 (cit. on pp. 55, 56).
- [Sna12a] MARK SNAITH, FLORIS BEX, JOHN LAWRENCE, and CHRIS REED: ‘Implementing ArguBlogging’. In *COMMA*. Ed. by BART VERHEIJ, STEFAN SZEIDER, and STEFAN WOLTRAN. Vol. 245. *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2012: pp. 511–512 (cit. on pp. 23, 56).
- [Sna12b] MARK SNAITH and CHRIS REED: ‘TOAST: Online ASPIC⁺ implementation’. In *COMMA*. Ed. by BART VERHEIJ, STEFAN SZEIDER, and STEFAN WOLTRAN. Vol. 245. *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2012: pp. 509–510 (cit. on p. 57).
- [Tea13] THE GHC TEAM: *The Glorious Glasgow Haskell Compilation System User’s Guide*. 7.6.3. Apr. 2013 (cit. on p. 18).
- [Ver12] BART VERHEIJ, STEFAN SZEIDER, and STEFAN WOLTRAN, eds.: *Computational Models of Argument - Proceedings of COMMA 2012, Vienna, Austria, September 10-12, 2012*. Vol. 245. *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2012.
- [Wub09] ZEGAYE SEIFU WUBISHET: ‘Understanding the Nature and Production Model of Hybrid Free and Open Source Systems: The Case of Varnish’. In *HICSS*. IEEE Computer Society, 2009: pp. 1–11 (cit. on p. 20).
- [Yao08] E. YAO: ‘A Theoretical Study of Mafia Games’. In *ArXiv e-prints* (Apr. 2008), vol. (cit. on p. 5).

APPENDIX A

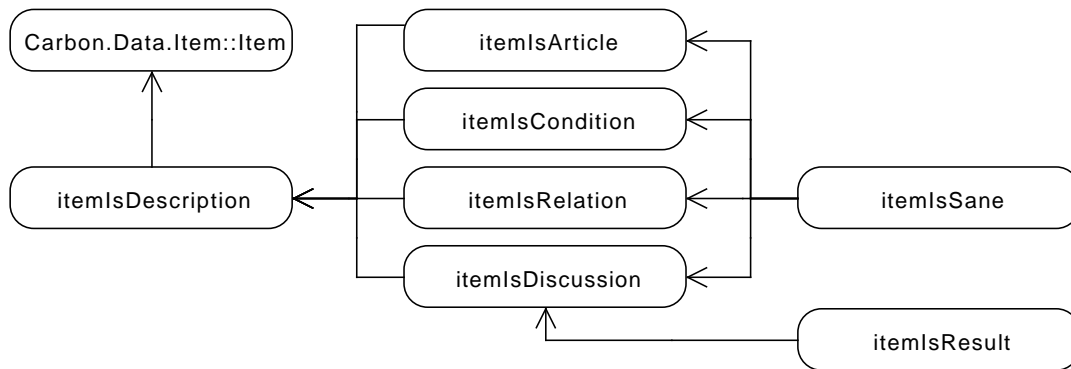
The database layout

The database layout as derived from the data described in chapter 6. The database layout was created with the software *pgModeler*, which also generated the following ER diagram:



APPENDIX B

Predicate dependencies of Carbon.Data.Item



Predicates that can be applied to the data type described in section 6.2. Arrows mean that it is necessary that the targeted predicate holds, for the source predicate to hold. If a predicate has more than one arrow to other items, these should be read as an or.

APPENDIX C

Software libraries used to implement CARBON

This appendix holds a list of software libraries used in CARBON together with short descriptions, so that it becomes easier to track which licenses are used for which parts of the software, and that the authors and sources of the respective libraries can be found.

C.1

Title:	jQuery JavaScript Library v1.9.1
Author:	jQuery Foundation, Inc. and other contributors
Year:	2013-02-04
URL:	http://jquery.com
License:	MIT

C.2

Title:	jQuery Cookie plugin
Author:	Klaus Hartl
Year:	2010
URL:	https://github.com/carhartl/jquery-cookie
License:	MIT

C.3

Title:	jQuery autoResize
Author:	James Padolsey
Year:	2011-09-24
URL:	https://github.com/alexbaradas/jQuery.fn.autoResize
License:	WTFPL - http://www.wtfpl.net/txt/copying/

C.4

Title: Backbone.js 1.0.0
Author: Jeremy Ashkenas, DocumentCloud,
and Investigative Reporters & Editors
Year: 2013
URL: <http://backbonejs.org>
License: MIT

C.5

Title: Underscore.js 1.4.4
Author: Jeremy Ashkenas, DocumentCloud Inc.
Year: 2013
URL: <http://underscorejs.org>
License: MIT

C.6

Title: marked - A markdown parser
Author: Christopher Jeffrey
Year: 2012
URL: <https://github.com/chjj/marked>
License: MIT

C.7

Title: Raphaël 2.1.2 - JavaScript Vector Library
Author: Dmitry Baranovskiy
Year: 2008
URL: [http://raphaeljs.com](http://dmitrybaranovskiy.github.io/raphael/)
License: MIT [http://raphaeljs.com/license.html](http://dmitrybaranovskiy.github.io/raphael/license.html)

C.8

Title: Springy v.2.3.0
- A force directed graph layout algorithm for JavaScript
Author: Dennis Hotson
Year: 2013
URL: [http://getspringy.com](http://dennisreid.com/springy/)
License: MIT

C.9

Title: Bootstrap V.2.3.2
Author: Twitter, Inc.
Year: 2012
URL: <http://getbootstrap.com/2.3.2/>
License: Apache license V.2.0

C.10

Title: | Datepicker for Bootstrap
Author: | Stefan Petre
Year: | 2012
URL: | <http://www.eyecon.ro/bootstrap-datepicker/>
License: | Apache license V.2.0

C.11

Title: | The Haskell Platform
Author: | The Haskell Community
Year: | 2013
URL: | <http://www.haskell.org/platform/>
License: | BSD

C.12

Title: | Happstack
Author: | SeeReason Partners, LLC
Year: | 2013
URL: | <http://happstack.com>
License: | BSD3

C.13

Title: | Aeson
Author: | Bryan O'Sullivan
Year: | 2011
URL: | <https://github.com/bos/aeson>
| <http://hackage.haskell.org/package/aeson>
License: | BSD3

C.14

Title: | HDBC
Author: | John Goerzen
Year: | 2011
URL: | <https://github.com/hdbc/hdbc>
| <http://hackage.haskell.org/package/HDBC>
License: | BSD3

C.15

Title: Parsec 3.0.0
Author: Daan Leijen, Erik Meijer[Lei01]
Year: 2001
URL: <http://legacy.cs.uu.nl/daan/parsec.html>
<http://www.haskell.org/haskellwiki/Parsec>
License: BSD style

C.16

Title: *Dialectical Models Encoding* (DIAMOND) 1.0.0
Author: Stefan Ellmauthaler, Joerg Puehrer, Hannes Straß
Year: 2014
URL: <https://isysrv.informatik.uni-leipzig.de/diamond>
<http://sourceforge.net/projects/diamond-adf/>
License: GNU GPL v.3

C.17

Title: PostgreSQL
Author: Michael Stonebraker
the PostgreSQL global Development Group
Year: 2014
URL: <http://www.postgresql.org>
License: PostgreSQL License (similar to MIT)

APPENDIX D

Screenshots of the web application

Figure D.1: A new article is created to reflect the statement that ‘Alice is part of the minority’.

Figure D.2: Viewing a single article in the web application. It is the one that was created in D.1.



#	Headline	Description	Creation
348	Alice belongs to the minority.	Mallory claims that it is a fact that Alice belongs to the minority.	2014-06-12 02:20:10.342944

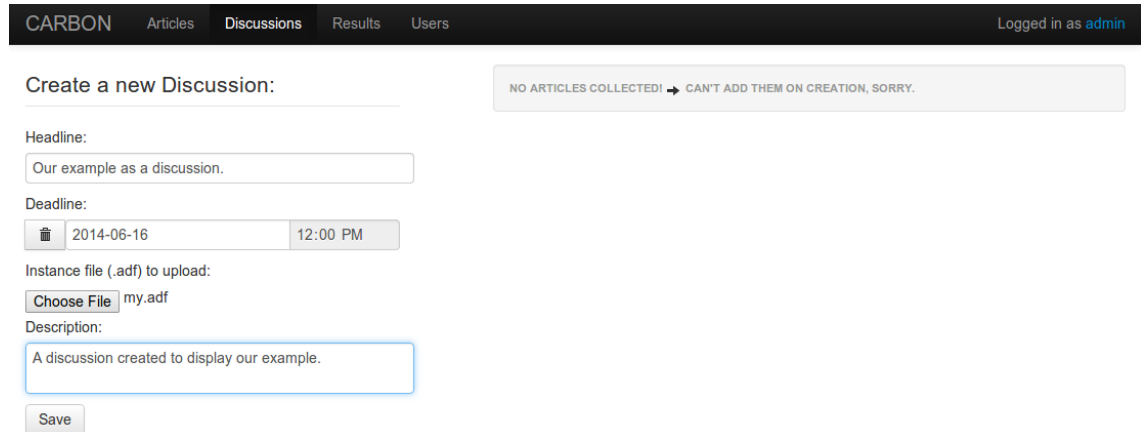
Figure D.3: This view pages the recently created articles and also makes it possible for users to create new articles.

```

1 % Translation of our example ADF:
2
3 s(a).
4 s(e).
5 s(b).
6 s(f).
7 s(m).
8 s(s).
9
10 ac(a,or(e,neg(b))).
11 ac(b,or(f,neg(a))).
12 ac(e,c(v)).
13 ac(f,c(v)).
14 ac(m,neg(s)).
15 ac(s,c(v)).

```

Listing D.1: The file `my.adf`, which is a direct translation of \mathcal{F}_{ADF} from 2.3(a). Since DIAMOND cannot process \bar{a} or $\neg a$, we choose to use the next character from the alphabet instead.



CARBON Articles Discussions Results Users Logged in as admin

Create a new Discussion: NO ARTICLES COLLECTED! → CAN'T ADD THEM ON CREATION, SORRY.

Headline:

Deadline:

Instance file (.adf) to upload:
 my.adf

Description:

Figure D.4: A new discussion is created, based on the code listed in D.1.

CARBON Articles Discussions Results Users Logged in as admin

Our example as a discussion.

Creation: 2014-06-12 02:22:44
A discussion created to display our example.

Evaluate Discussion

Current articles Collected articles Graph view Participants .adf files

#	Headline	Description	Actions
373	a		Remove from Discussion
377	b		Remove from Discussion
383	e		Remove from Discussion
386	f		Remove from Discussion
389	m		Remove from Discussion
391	s		Remove from Discussion

Figure D.5: The discussion created by D.4 is displayed and the list of articles contained in that discussion is displayed.

CARBON Articles Discussions Results Users Logged in as admin

Our example as a discussion.

Creation: 2014-06-12 02:22:44
A discussion created to display our example.

Evaluate Discussion

Current articles Collected articles Graph view Participants .adf files

↑ 📄 * →

Figure D.6: The discussion created by D.4 is displayed and the statements are displayed as a graph.

CARBON Articles Discussions Results Users Logged in as [admin](#)

Our example as a discussion.

Creation: 2014-06-12 02:22:44
A discussion created to display our example.

[Evaluate Discussion](#)

[Current articles](#) [Collected articles](#) [Graph view](#) [Participants](#) [.adf files](#)

[Leave the discussion.](#) You're currently a participant in this discussion.

Participants:

[admin](#)

Figure D.7: A list of participants belonging to a discussion is being displayed.

CARBON Articles Discussions Results Users Logged in as [admin](#)

Our example as a discussion.

Creation: 2014-06-12 02:22:44
A discussion created to display our example.

[Evaluate Discussion](#)

[Current articles](#) [Collected articles](#) [Graph view](#) [Participants](#) [.adf files](#)

Instance file (.adf) to upload:

[Choose File](#) No file chosen [Upload](#)

Current .adf for this discussion:

```
s (a) .
s (b) .
s (e) .
s (f) .
s (m) .
s (s) .
ac (e, c (v)) .
ac (f, c (v)) .
ac (s, c (v)) .
ac (a, or (e, neg (b))) .
ac (b, or (f, neg (a))) .
ac (m, neg (s)) .
```

Figure D.8: The ADF instance for every discussion can be viewed, downloaded or extended by additional uploads.

#	Headline	Description	Creation
395	Our example as a discussion.	A discussion created to display our example.	2014-06-12 02:24:48.395831
346	A second guess.	Kragen!	2014-05-28 14:10:31.52153
289	A new hope	We start again after some wild debugging.	2014-05-28 12:41:03.827114
232	hewgpin	knewgoiwergipn	2014-05-27 15:08:15.175746
177	Test1	For Cthulhu!	2014-05-21 17:36:14.039779
173	Test2	One more test...	2014-05-19 14:26:24.245239
113	Test2	inwegiwneqionwegoinweg	2014-05-14 13:13:41.640678

Figure D.9: After evaluation, a discussion is displayed in the results tab. We can see a list of results for several evaluated discussions.

Set	Extensions	Score	Vote:
{¬a,¬b,¬e,¬f,¬m,¬s}	TwoValued	Votes: <input type="text"/>	<input type="checkbox"/>
∅	Admissible	Votes: <input type="text"/>	<input type="checkbox"/>
{a,b,e,f}	Admissible	Votes: <input type="text"/>	<input type="checkbox"/>
{a,b,e,f,s,¬m}	Stable, Grounded, Complete, Admissible, Preferred	Votes: 7	<input type="checkbox"/>
{a,b,e,f,s}	Admissible	Votes: <input type="text"/>	<input type="checkbox"/>
{a,e}	Admissible	Votes: <input type="text"/>	<input type="checkbox"/>
{a,e,f}	Admissible	Votes: <input type="text"/>	<input type="checkbox"/>
{a,e,f,s,¬m}	Admissible	Votes: <input type="text"/>	<input type="checkbox"/>
{a,e,f,s}	Admissible	Votes: <input type="text"/>	<input type="checkbox"/>
{a,e,s,¬m}	Admissible	Votes: <input type="text"/>	<input type="checkbox"/>

Figure D.10: Some of the models for our discussion created by D.4 are displayed. Users can vote on these results to express their preferences.

Eigenständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.

Leipzig, den

Jakob Runge

