

Tools for the implementation of argumentation models

Bas van Gijzel

Functional Programming Laboratory,
School of Computer Science,
University of Nottingham,
United Kingdom
bmv@cs.nott.ac.uk

Abstract

The structured approach to argumentation has seen a surge of models, introducing a multitude of ways to deal with the formalisation of arguments. However, while the development of the mathematical models have flourished, the actual implementations and development of methods for implementation of these models have been lagging behind. This paper attempts to alleviate this problem by providing methods that simplify implementation, i.e. we demonstrate how the functional programming language Haskell can naturally express mathematical definitions and sketch how a theorem prover can verify this implementation. Furthermore, we provide methods to streamline the documenting of code, showing how literate programming allows the implementer to write formal definition, implementation and documentation in one file. All code has been made publicly available and reusable.

1998 ACM Subject Classification I.2.3 Nonmonotonic reasoning and belief revision

Keywords and phrases argumentation, implementation, functional programming, Haskell, Carneades

Digital Object Identifier 10.4230/OASISs.ICCSW.2013.43

1 Introduction

Argumentation theory is an interdisciplinary field studying how conclusions can be reached through logical reasoning in settings where the soundness of arguments might be subjective and arguments can be contradictory. There are two main approaches: the structured approach giving a predetermined structure to arguments, including for example legal and scientific arguments, while the abstract approach makes no specific assumptions about the form of arguments and is thus generally applicable. Structured argumentation models have seen a recent surge, with new developments in both general frameworks [17, 1, 3] and more domain specific approaches [12, 11]. For the abstract approach, a significant effort has been directed towards the construction of usable tools and efficient implementations; see [4] for a survey. In addition there has been a recent development of translations between structured and abstract argumentation models, allowing an implementer to sidestep the implementation of the structured model by implementing the translation instead and relying on an existing efficient implementation of the translation target. However, despite these tools and existing translations of structured argumentation models into abstract argumentation frameworks in the literature [17, 10, 9, 2], there is a lack of implementations of the structured models.


We give a number of potential reasons:

- Most implementations of structured argumentation models are not publicly available. Simari [18] gives an overview of some of the structured argumentation models, but most



© Bas van Gijzel;
licensed under Creative Commons License CC-BY
2013 Imperial College Computing Student Workshop (ICCSW'13).
Editors: Andrew V. Jones, Nicholas Ng; pp. 43–48

OpenAccess Series in Informatics

 OASIS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

implementations are now unavailable or closed source. In the case the code has never been published it means that the information regarding the techniques of implementation are effectively lost, forcing new implementers to develop methods from scratch.

- Translations can be notoriously complex, both in implementation and in verification. As a good example, consider the translation of Carneades to ASPIC⁺ [10, 9], or the translation of abstract dialectical frameworks to Dung [2]. Both proofs are at least a page long, and are hard to verify even by experts in the field.

To tackle this problem, we introduce a set of methods and tools in Section 2. Then in Section 3 we provide an introduction to the definitions of Carneades each time providing corresponding implementations using previously discussed methods and tools. We conclude in Section 4 with a discussion of our study and how to go from this implementation to an automatic verification in a theorem prover.

2 Tools and methods

2.1 Functional programming

When looking at recent developments in abstract argumentation [4], we can see that answer set programming (ASP) and Prolog have taken a significant role in the efficient implementation and development of general tools. Part of this success can be explained by the paradigm of ASP and logic programming which can express computational problems for Dung’s argumentation frameworks [6] in a very natural way, making it possible to make the code partly self-documenting. See also [6], which relates abstract argumentation to logic programming with negation as failure.

For structured argumentation, there has not been such a convincing implementation language yet. There are various implementations done in Java [18], but they are quite far removed from the logical specification making it significantly harder to verify whether the implementation is actually correct. In this paper we instead apply functional programming, using the programming language Haskell [16]. The declarative nature of functional programming, similar to logic programming, is a natural candidate to express structured argumentation frameworks in such a way that the code is close to the actual mathematical definitions [13], while additionally simplifying future verification of such implementations.

2.2 Literate programming

Although implementations of structured argumentation models often have appropriate user instructions, it is less common that such an implementation also documents its methods of implementation. To make this process more attractive, we employ literate programming [14], a technique that allows the user to write both the implementation and documentation, including the formal definitions, in one file. Literate Haskell is Haskell’s native version of literate programming, which allows programmers to intermix the writing of L^AT_EX and Haskell code, while still being readable by a standard Haskell compiler. Additionally, the tool called lhs2tex [15] provides the user with the automatic typesetting of Haskell code within a Literate Haskell file, generating appropriate L^AT_EX code. This ensures that the documentation is kept up to date along with the programming code.

2.3 Open source and public repositories

As we discussed in Section 1, most implementations of structured argumentation models are not publicly available (anymore) or are closed source. We believe that to progress the know-

ledge of implementing techniques for (structured) argumentation models, implementations should be made publicly available. The implementation in this paper has also been made available through the standard Haskell package repository called Hackage¹, providing source files and automatic generation of html documentation of the API. The public availability and documentation of the implementation has attracted other people to contribute as well, e.g. see Stefan Sabev’s github² where he extended our implementation for a university module.

2.4 Formalisation in a theorem prover

Given the complexity of some of the structured models and translation we might want to be able to verify the correctness of our implementation. One way to achieve this beyond the proofs done on paper, is to formalise the implementation through an interactive theorem prover of our choice. Haskell, allows for code very close to the mathematics and additionally is of the same functional nature as most theorem provers, making the step from a Haskell program to a theorem prover very natural. We do not have the space to demonstrate this approach here, however an implementation of Dung’s argumentation frameworks has been made available online³. See [8] for an expanded exposition.

3 A documented implementation of Carneades

In this section we will give definitions of Carneades [12, 11], an argumentation model designed to capture standards and burdens of proof. We discuss the version as given in Gordon and Walton [12], after each definition showing our corresponding implementations in Haskell⁴. Due to space constraints, we do not give the complete implementation, however the source code of this section, is available as a literate programming source file, containing all the left out definitions, corresponding implementations and explanations⁵. This literate programming file can immediately be loaded into the Haskell compiler and is also available as an open source library on Hackage⁶. Although Carneades is very suitable to demonstrate the implementation techniques explained in this paper; it does already have a quite mature implementation available⁷.

3.1 Arguments

We strive for a realisation in Haskell that mirrors the mathematical model of Carneades argumentation framework as closely as possible. Ideally, there would be little more to a realisation than a transliteration. In Carneades all logical formulae are literals in propositional logic; i.e., all propositions are either positive or negative atoms. Taking atoms to be strings suffice in the following, and propositional literals can then be formed by pairing this atom with a Boolean to denote whether it is negated or not:

```
type PropLiteral = (Bool, String)
```

We write \bar{p} for the negation of a literal p . The realisation is immediate:

¹ <http://hackage.haskell.org/>

² https://github.com/SSabev/Haskell_Carneades

³ <http://www.cs.nott.ac.uk/~bmv/Code/AF2.agda>

⁴ This section is largely based on previous work in [7].

⁵ See <http://www.cs.nott.ac.uk/~bmv/CarneadesDSL/> for the source code and instructions.

⁶ <http://hackage.haskell.org/package/CarneadesDSL>

⁷ <http://carneades.github.com/>

```
negate :: PropLiteral → PropLiteral
negate (b, x) = (¬ b, x)
```

An argument is a tuple of two lists of propositions, its *premises* and its *exceptions*, and a proposition that denotes the *conclusion*:

```
newtype Argument = Arg ([PropLiteral], [PropLiteral], PropLiteral)
```

Due to lack of space, we will not discuss the details and the implementation of the set of arguments; see [7] for details.

3.2 Carneades Argument Evaluation Structure

The main structure of the argumentation model is called a Carneades Argument Evaluation Structure (CAES):

► **Definition 1** (Carneades Argument Evaluation Structure (CAES)). A *Carneades Argument Evaluation Structure* (CAES) is a triple $\langle arguments, audience, standard \rangle$ where *arguments* is an acyclic set of arguments, *audience* is an audience as defined below (Def. 2), and *standard* is a total function mapping each proposition to its specific proof standard.

The transliteration into Haskell is almost immediate

```
newtype CAES = CAES (ArgSet, Audience, PropStandard)
```

► **Definition 2** (Audience). Let \mathcal{L} be a propositional language. An *audience* is a tuple $\langle assumptions, weight \rangle$, where *assumptions* $\subset \mathcal{L}$ is a propositionally consistent set of literals (i.e., not containing both a literal and its negation) assumed to be acceptable by the audience and *weight* is a function mapping arguments to a real-valued weight in the range $[0, 1]$.

This definition is captured by the following Haskell definitions:

```
type Audience = (Assumptions, ArgWeight)
type Assumptions = [PropLiteral]
type ArgWeight = Argument → Weight
type Weight = Double
```

Further, as each proposition is associated with a specific proof standard, we need a mapping from propositions to proof standards. A proof standard is a function that given a proposition p , aggregates arguments pro and con p and decides whether it is acceptable or not:

```
type PropStandard = PropLiteral → ProofStandard
type ProofStandard = PropLiteral → CAES → Bool
```

The above definition of proof standard demonstrates that implementation in a typed language such as Haskell is a useful way of verifying definitions from argumentation theoretic models. Our implementation effort revealed that the original definition of proof standard as given in [12] could not be realised as stated, because proof standards in general not only depend on a set of arguments and the audience, but may need the whole CAES. However, due to space constraints we will omit the definition of specific proof standards.

3.3 Evaluation

Two concepts central to the evaluation (semantics) of a CAES are *applicability of arguments*, which arguments should be taken into account, and *acceptability of propositions*, which conclusions can be reached under the relevant proof standards, given the beliefs of a specific audience.

► **Definition 3** (Applicability of arguments). Given a set of arguments and a set of assumptions (in an audience) in a CAES C , then an argument $a = \langle P, E, c \rangle$ is *applicable* iff

- $p \in P$ implies p is an assumption or $[\bar{p}$ is not an assumption and p is acceptable in C] and
- $e \in E$ implies e is not an assumption and $[\bar{e}$ is an assumption or e is not acceptable in C].

► **Definition 4** (Acceptability of propositions). Given a CAES C , a proposition p is *acceptable* in C iff $(s \ p \ C)$ is *true*, where s is the proof standard for p .

The realisation of applicability and acceptability in Haskell is straightforward:

```

applicable :: Argument → CAES → Bool
applicable (Arg (prems, excns, -)) caes@(CAES (_, (assumptions, -), -))
  = and $ [(p ∈ assumptions) ∨ (p 'acceptable' caes) | p ← prems]
    ++
    [(e ∈ assumptions) ↓ (e 'acceptable' caes) | e ← excns ]
  where
    x ↓ y = ¬ (x ∨ y)

acceptable :: PropLiteral → CAES → Bool
acceptable c caes@(CAES (_, -, standard))
  = c 's' caes
  where s = standard c

```

4 Conclusions and future work

As we have seen, functional programming allows to realise 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. Furthermore, by making the code publicly available and open source we improve the chances that implementation methods will progress more efficiently. For related work, researchers working in answer set programming have been working on extensions of ASP to make it more of a general purpose programming language. This has also allowed Charwat et al. to implement argument generation and visualisation for structured based approaches [5].

For future work, we are putting the formalisation methods discussed in Section 2.4 into practice, see also [8]. Thus, in addition to the discussed implementation of argumentation models using the described methods and tools, we want to verify the correctness of implementations and furthermore employ the same techniques for translations between argumentation models. Our ultimate goal is to have verified translations from structured argumentation models into efficiently implemented abstract argumentation models, resulting in a verified and efficient implementation method for structured argumentation models.

References

- 1 Andrei Bondarenko, Phan Minh Dung, Robert A. Kowalski, and Francesca Toni. An abstract, argumentation-theoretic framework for default reasoning. *Artificial Intelligence*, 93:63–101, 1997.
- 2 Gerhard Brewka, Paul E. Dunne, and Stefan Woltran. Relating the semantics of abstract dialectical frameworks and standard AFs. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-11)*, pages 780–785, 2011.
- 3 Gerhard Brewka and Stefan Woltran. Abstract dialectical frameworks. In *Proceedings of the Twelfth International Conference on the Principles of Knowledge Representation and Reasoning*, pages 102–111. AAAI Press, 2010.
- 4 Günther Charwat, Wolfgang Dvorák, Sarah Alice Gaggl, Johannes Peter Wallner, and Stefan Woltran. Implementing abstract argumentation - a survey. Technical Report DBAI-TR-2013-82, Vienna University of Technology, 2013.
- 5 Günther Charwat, Johannes Peter Wallner, and Stefan Woltran. Utilizing ASP for generating and visualizing argumentation frameworks. *CoRR*, abs/1301.1388, 2013.
- 6 Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321–357, 1995.
- 7 Bas van Gijzel and Henrik Nilsson. Haskell gets argumentative. In *Proceedings of the Symposium on Trends in Functional Programming (TFP 2012)*, LNCS 7829, pages 215–230, St Andrews, UK, 2013. LNCS.
- 8 Bas van Gijzel and Henrik Nilsson. Towards a framework for the implementation and verification of translations between argumentation models. Draft Proceedings of The 25th symposium on Implementation and Application of Functional Languages, August 2013.
- 9 Bas van Gijzel and Henry Prakken. Relating Carneades with abstract argumentation. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-11)*, pages 1113–1119, 2011.
- 10 Bas van Gijzel and Henry Prakken. Relating Carneades with abstract argumentation via the ASPIC⁺ framework for structured argumentation. *Argument & Computation*, 3(1):21–47, 2012.
- 11 Thomas F. Gordon, Henry Prakken, and Douglas Walton. The Carneades model of argument and burden of proof. *Artificial Intelligence*, 171(10-15):875–896, 2007.
- 12 Thomas F. Gordon and Douglas Walton. Proof burdens and standards. In Guillermo Simari and Iyad Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 239–258. Springer US, 2009.
- 13 John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, April 1989.
- 14 D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.
- 15 Andres Löh. lhs2tex. <http://www.andres-loeh.de/lhs2tex/>. Accessed July 10, 2013.
- 16 Simon Marlow et al. Haskell 2010 language report. <http://www.haskell.org/onlinereport/haskell2010>, 2010.
- 17 Henry Prakken. An abstract framework for argumentation with structured arguments. *Argument & Computation*, 1:93–124, 2010.
- 18 Guillermo R. Simari. A brief overview of research in argumentation systems. In *Proceedings of the 5th international conference on Scalable uncertainty management, SUM’11*, pages 81–95, Berlin, Heidelberg, 2011. Springer-Verlag.