A framework for relating, implementing and verifying argumentation models and their translations

Bas van Gijzel

Supervisor: Henrik Nilsson

Thesis submitted to The University of Nottingham
for the degree of Doctor of Philosophy

October 2015

Voor oma, die altijd voor ons klaarstond

Abstract

Computational argumentation theory deals with the formalisation of argument structure, conflict between arguments and domain-specific constructs, such as proof standards, epistemic probabilities or argument schemes. However, despite these practical components, there is a lack of implementations and implementation methods available for most structured models of argumentation and translations between them.

This thesis addresses this problem, by constructing a general framework for relating, implementing and formally verifying argumentation models and translations between them, drawing from dependent type theory and the Curry-Howard correspondence. The framework provides mathematical tools and programming methodologies to implement argumentation models, allowing programmers and argumentation theorists to construct implementations that are closely related to the mathematical definitions. It furthermore provides tools that, without much effort on the programmer's side, can automatically construct counter-examples to desired properties, while finally providing methodologies that can prove formal correctness of the implementation in a theorem prover.

The thesis consists of various use cases that demonstrate the general approach of the framework. The Carneades argumentation model, Dung's abstract argumentation frameworks and a translation between them, are implemented in the functional programming language Haskell. Implementations of formal properties of the translation are provided together with a formalisation of AFs in the theorem prover, Agda. The result is a verified pipeline, from the structured model Carneades into existing efficient SAT-based implementations of Dung's AFs. Finally, the ASPIC$^+$ model for argumentation is generalised to incorporate content orderings, weight propagation and argument accrual. The framework is applied to provide a translation from this new model into Dung's AFs, together with a complete implementation.

Acknowledgements

I would like to thank my supervisor Henrik Nilsson, for giving me the freedom to decide what research I could do during the PhD. I enjoyed the relaxed atmosphere at the Functional Programming Lab and I will miss the drinks at the Johnson Arms with Graham, Henrik, Thorsten, Venanzio, Laurence, and the other PhD students from the FP lab. Also, I would like to thank all the students from Nottingham that I was lucky enough to teach in the past few years. (I won't miss marking your course work.)

I would like to thank Tom Gordon for his general interest in my work, the fruitful discussions about type theory and Carneades, and the extensive comments on my papers. I am grateful to the people from both the argumentation and functional programming research communities for always making me feel welcome. Also, I would like to thank Jan Kuper, Henry Prakken and Andres Löh in particular, for sparking my interest in both research fields. I would like to thank Henry Prakken, Sjoerd Timmer, Bas Testerink and colleagues for the productive and "gezellige" time in Utrecht.

I am very happy with all my friends, both old and new, for not complaining too much, but complaining enough, that I was so busy in the past four years. It will be good to see you all again!

I would like to thank my family for supporting me throughout the years, whether it was with the PhD or life in general. Thank you for visiting me in the UK many times, I know I can always count on you.

Finally, all the love for Hardeep, for supporting me and always being able to make me relax.

Contents

List of Figures

Chapter 1

Introduction

Argumentation theory is an interdisciplinary field studying how conclusions can be reached through logical reasoning. Argumentation should here be understood in a general sense, including for example political debates along with more rigorous settings such as a legal or a scientific argument. A central aspect is that there is usually not going to be any clear-cut truth, but rather arguments and counter-arguments, possibly carrying different weights, and possibly relative to the understanding of a specific audience and what is known at a specific point in time. The question, then, is what it means to systematically evaluate such a set of arguments to reach a rational conclusion. Fields that intersect with argumentation theory thus include philosophy (notably epistemology and the philosophy of science and mathematics), logic, rhetoric, and psychology.

Computational argumentation theory is studied in the context of artificial intelligence. Computational argumentation models deal with the formalisation of argument structure as well as conflict and domain-specific constructs such as proof standards. Most argumentation models can apply one or more types of inferencing mechanisms on a problem has been already been formalised, e.g. a specific court case, to derive a set of conclusions and/or applicable arguments.

There are two main approaches to defining computational argumentation models: the abstract approach and the structured approach. The abstract approach makes no specific assumptions about the form of arguments. Thus this approach is generally applicable across domains. In contrast, the structured approach assumes an argument structure, more or less specific to domains such as legal or scientific argumentation[1]. Irrespective of the approach, computational argumentation theory is concerned with the mathematical

[1]The ASPIC$^+$ is slightly more general in that it assumes only a basic structure of arguments, but leaves the logical language abstract.

analysis and specification of translations between models of argumentation.

This thesis puts forward a new, principled approach to the design, implementation and formalisation of computational argumentation models and translations between them drawing from dependent type theory [115, 116, 167] and the Curry-Howard correspondence [43, 44, 93, 167, 188]. This approach has a number of advantages:

- It intuitively captures existing abstract and structured argumentation models, providing implementations that also define a mathematical specification (see Chapter 4 and 5).
- Correctness of implementations is shown by implementing desirable properties, such as correspondence properties (see Chapter 7).
- It allows the specification and the proofs of an argumentation model or translation to be unified with its implementation by formalising it in a dependently typed theorem prover such as Agda (see Chapter 7).

Some of these ideas have previously been explored by Krause et al. [103], but their work has so far not had any major impact on the field of argumentation with respect to how argumentation models are implemented and formalised. This thesis develops these ideas much further in a contemporary argumentation setting.

Implementations of argumentation models and their translations

Dung's argumentation frameworks [48] are one of the most influential abstract models of argumentation. They capture the essence of argumentation, defining arguments as an abstract set together with a notion of conflict represented by a relation on this set. Dung's model sets the standard for various other abstract approaches to argumentation, including abstract dialectical frameworks [23] (see Section 3.2 and in particular Section 3.2.5).

Dung's argumentation frameworks (AFs) have an established relationship [48] to logic programming [112]. It is therefore not surprising that AFs have seen significant developments in the area of efficient implementation and elegant implementation methods [35, 36], particularly through implementations written in logic programming and answer set programming [56, 55] or through implementations based on SAT-solvers [34, 56]. Several other abstract models, are either direct extensions of AFs or are closely related. These models can thus also be implemented with relative ease through encoding into answer set programming clauses [57, 37], translation into other mathematical formalisms [19], or by direct implementation into a logic programming language such as Prolog [68]. See Section 4.4 for further related work on implementations of abstract argumentation models.

There has been a resurgence of interest in structured argumentation models, with new developments in both general frameworks [145, 3, 17, 14, 130] and more domain-specific approaches [86, 83, 180]. However, the state of implementations for these structured models is markedly different from the abstract models:

- As abstract argumentation is closely related to logic programming it facilitates developing intuitive implementations, closely aligned with the logical specification, that are also efficient [35, 36]. In contrast, no mainstream, general-purpose language or paradigm provides an equally close fit for structured argumentation. Java has been used in a few implementation efforts [164]. However, because Java fundamentally is an imperative language, those implementations tend to be quite far removed from the logical specification. This makes it difficult to verify whether they actually are correct.

- Most implementations of structured argumentation models are not publicly available. Simari [164] gives an overview of some structured argumentation models, but most implementations of those are now unavailable or closed source, meaning that details of these specific implementation techniques effectively have been lost. New implementers thus have to start from scratch.

  Two notable exceptions are the Carneades implementation [80][2] and the Tweety project [174][3]. See Section 5.5 for further related work on implementations of structured argumentation models.

- There are existing translations from structured into abstract argumentation frameworks (via structured argumentation models) [145, 77, 76, 19, 122], which in principle should allow abstract argumentation implementations to be leveraged. Examples include the translation of Carneades into ASPIC$^+$ (which generates AFs) [77, 76] and the translation of abstract dialectical frameworks into Dung [19]. Both proofs are substantial and very technical, and thus hard to verify even for experts in the field. The complexity of these translations is a probable cause of the lack of implementations of translations. See Section 7.3

[2]See `https://carneades.github.com`.
[3]See `http://tweetyproject.org/`.

for further discussion of implementations of translations between argumentation models.

To address these problems, my research takes a principled approach to the relating, implementing and generalising of argumentation models based on functional programming and the Curry-Howard correspondence. Functional programming and the Curry-Howard correspondence are applied to obtain, respectively, intuitive implementations and mechanical formalisations that are both very close to the mathematical specifications.

Functional programming

My thesis attempts to address the lack of implementations of structural argumentation frameworks and translation between models by exploiting functional programming. We will see that verification of structured argumentation frameworks and their translations is facilitated by a declarative implementation where the code is close to the actual mathematics. Functional programming languages thus provide a good basic fit. An additional advantage is their proven track record as hosts for Embedded Domain-Specific Languages (EDSL) [95, 94], allowing tailoring to specific requirements which is exactly what is needed to support particular structured argumentation frameworks.

The choice of functional programming language is Haskell [100, 114, 99]. The use of Haskell in the thesis is motivated by previous work [72], in which the Carneades argumentation model was implemented in a way that is easily understandable to argumentation theorists with no prior Haskell knowledge. Further, two of the Cabal packages[4] discussed in previous papers [72, 73] (see also Section 4 and 5), the Dung and CarneadesDSL package, have been used in AI for programming (e.g. a closed source natural language processor) and teaching[5].

Haskell has several advantages for the implementation of structured and abstract argumentation models and their translations:

- Programs are *statically* and *strongly typed*, i.e. code can be checked on compile-time for its type and the compiler can catch a significant amount of programming errors before running a program.
- Haskell supports *purely functional* programming. A pure function does not have side-effects, making it closer to how functions are specified in mathematics.

[4]For a discussion of Cabal packages, see Appendix A.

[5]School of Informatics, University of Edinburgh, AILP 2012–2013, 2013–2014, 2014–2015: `http://www.inf.ed.ac.uk/teaching/courses/ailp/`

- Haskell has very flexible syntax and furthermore has facilities for defining (embedded) domain specific languages. Together with pure functional programming, it is possible to develop an implementation that serves as a mathematical specification in its own right.

- Haskell allows for automatic random testing of the state space by using the QuickCheck [40] library. Programmers can thus quickly check correctness of their programs by specifying intuitive properties and letting QuickCheck search for possible counter-examples[6].

- Although Haskell itself is not a theorem prover, code programmed in Haskell can be relatively easily lifted to a theorem prover such as Agda, allowing formal verification of an implementation.

See Section 2.1 for an in depth discussion of the technical programming language terminology. For a further discussion of the motivations for using functional programming, see [105, 8, 96].

Curry-Howard correspondence

While tools like QuickCheck [40] can help finding problems automatically, firm correctness guarantees can only be obtained through formal proofs. Given that we are working in a pure, functional, strongly typed setting, theorem provers based on the Curry-Howard correspondence [43, 44, 93, 167, 188] offer a particularly attractive approach. The Curry-Howard correspondence is a connection between logic and computation that directly relates types with propositions and programs with proofs. Thus, to write down a proof for a theorem is the same as implementing a function for the corresponding type.

The Logic of Argumentation by Krause et al. [103] is an argumentation model that directly exploits this connection by generalising the link between typed lambda calculus and intuitionistic type theory [115, 116], by defining a notion of argument as proof terms with weights corresponding to lambda terms with free variables, given a context (see Section 3.7.1 for further details). The approach in the thesis is partly based on the insight gained from this connection, i.e., arguments and inferencing are closely related to functional programming and evaluation.

Agda [128] is a theorem prover based on dependent type theory [115, 116, 167] with syntax that is very close to that of Haskell, making the step from

[6]QuickCheck generates a specified number of test cases before it gives up finding a counter-example.

from a Haskell implementation to an Agda formalisation relatively small. Agda checks that all functions are terminating. Thus, if we successfully implement an algorithm in Agda, we immediately know that our algorithm is terminating on all (finite) inputs.

Furthermore, translations between argumentation models can be notoriously complex (various proofs including [145, 77, 76, 19] have later been amended or said to be very hard to understand even by authors of the respective models). Given the complexity of proofs of correctness, and the difficulty even for experts of the field to check this work, I believe that mechanical formalisation of translations and their correctness proofs also have significant benefits.

The general framework

The thesis develops a principled approach to the implementation and formalisation of argumentation models and their translations. In particular, a methodology is provided for implementing argumentation models, translating between different argumentation models, testing correctness of implementations and finally proving correctness of implementations.

1. **Implementation of abstract and structured argumentation models:**

    - Various abstractions and implementation methods of existing argumentation concepts are provided, e.g., a general implementation of Dung's AFs allowing different instantations and providing various high level algorithm implementations is given in Chapter 4. The various algorithms and definitions implemented in Chapter 4, 5 and 7 are deliberately implemented as close to the mathematics as possible, using high-level programming, to make it easy to verify that the implementations are indeed correct. The implementations in the various chapters are therefore also meant as reference implementations.
    - Argumentation theorists that intend to implement a different abstract argumentation model, can apply the implementation methods of Chapter 4 to develop their own implementation, or alternatively, use the implementation of Chapter 4 as a translation target (see also point 2.).
    - Domain specific languages can be built from an argumentation model implementation using the methodology outlined in Chapter 5.

- All programming code and programming methodology is fully documented, open source and available making the work reproducible and suitable as teaching material (see Chapter 4 and 5)[7].

2. **Implementation of translations between argumentation models:**

   - Chapter 4 is a general and complete implementation of Dung's AFs that is intended to be used as a translation target from other argumentation models. Chapter 7 and Chapter 8 provide use cases for this approach, providing high-level implementation of a translation from Carneades/generalised ASPIC$^+$ into Dung's AFs. The implementation methods of Chapter 7 are intended to be complemented with testing or formal verification (see point 3. and 4.).
   - Argumentation theorists that intend to provide an efficient implementation of their argumentation model can also rely on the implementation of Chapter 4, by implementing a translation to this implementation. Although the implementation of Dung's AFs in Chapter 4 in itself is not optimised, the implementation does provide various output formats that are readily usable by various efficient SAT-based implementations in the field (see Chapter 7).

3. **Verification/testing of implementations of argumentation models and their translations:**

   - Argumentation models can be verified by implementing expected properties directly into Haskell and applying the testing methods used in Chapter 7 to do a limited search of the state space, possibly finding counter-examples when these properties are violated (see point 4. for formal verification).
   - Translations can be verified by implementing both the original model and the translation and again applying the testing methods used in Chapter 7 to verify that the original model implementation and the result in the translated Dung AF correspond.

4. **Formal verification of implementations of argumentation models and their translations through formalisation in a theorem prover:**

[7] `https://github.com/nebasuke/thesis`

- Chapter 7 provides a reference implementation of Dung's AFs in the theorem prover Agda, documenting how to formalise definitions up to grounded semantics.
- The implementation of Chapter 7 is intended as a translation target for mechanical formalisations of translations from argumentation models.
- Again, all proofs and programming methodologies are fully documented, open source and available making the work reproducible and suitable as teaching material[8].

The ultimate goal here is that the functional programming approach taken is as suitable for implementing and formalising structured argumentation frameworks and their translations as logic programming is for implementing abstract argumentation models. See also Section 1.1 and Figure 1.1, for details on how the thesis contributions apply to each of these points.

1.1 Contributions

The specific contributions of the thesis are the following:

- An implementation in Haskell of Dung's abstract argumentation frameworks [48], its standard semantics and the more recent semi-stable semantics [28]. The code is very close to the mathematical definitions, allowing the work to simultaneously serve as documentation and implementation and showing how functional programming is indeed suitable for implementing abstract argumentation models.

- An implementation and domain specific language for the Carneades argumentation model [86, 83], providing one of the first functional programming implementations of a structured argumentation model. The implementation has been shown to be intuitive to read and use by argumentation theorists, and is now used as the basis of a university module.

- A deeper treatment of the previous work translating Carneades into ASPIC$^+$ [69, 76]. Further theorems about the correspondence of the translation are proved and a derivative translation from Carneades into Dung is constructed. Further algorithms and properties are given for the derived translation.

[8]`https://github.com/nebasuke/thesis`

- An implementation of the translation from Carneades into Dung is given, providing one of the first *implementations* of a translation between argumentation models. The implementation techniques are fully documented and all work is made publicly available and reusable. The desired properties of such a translation are discussed, together with their implementation in Haskell. In addition, a sketch is provided how such properties might be formalised in a theorem prover.

- A formalisation of the implementation of Dung's AFs in a theorem, Agda, is given, thereby providing the first fully machine-checkable formalisation of an argumentation model, and showcasing the benefits of using a functional programming language as an initial implementation.

- The above implementations and formalisation are combined, to provide a verified pipeline, starting from an input file reading a Carneades argument structure, resulting in a file containing a Dung AF, readable by one of the fastest current implementations [56].

- A generalisation of the ASPIC$^+$ model is developed, extending the model with content orderings, weight propagation and argument accrual. The framework constructed in this thesis is then applied to provide an implementation of extended ASPIC$^+$, by translating it directly into Dung's AFs.

- Finally, all work has been made open source, publicly available and are immediately installable, either as a Hackage package or as a literate programming file. All the implementations and formalisations are also extensively documented and contain various examples of their usage.

These contributions and the relationship between them are summarised in a schematic overview in Figure 1.1. The top half of the figure (using cloudy outlines) denote mathematical specifications and implementations of argumentation models and their translations, while the bottom half denotes completely mechanised formal specifications (in a theorem prover). Dashed lines indicate that part of the implementation is left to future work, while solid lines indicate the project has been completely finished.

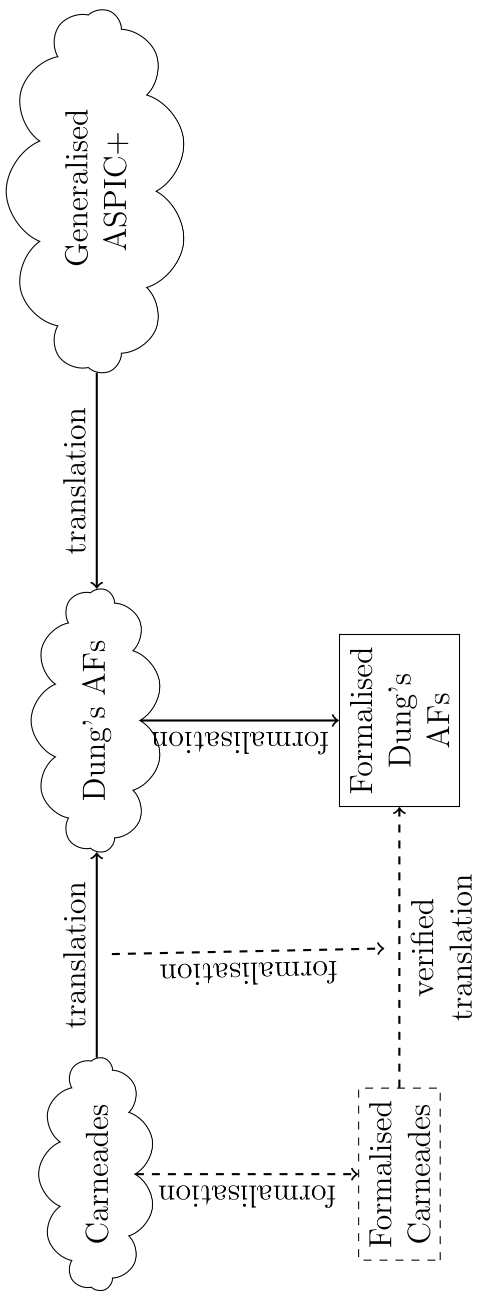


Figure 1.1: Overview of completed and future research

There are also various small technical contributions to existing argumentation models and translations in the thesis, see Appendix B.

1.2 Structure of the thesis

This section gives an overview of the research I have done, by discussing how each section contributes to the overall thesis. The thesis is organised into two parts:

1. The first part discusses the relevant background needed to read the main part of the thesis.

2. The second part presents my framework for relating, implementing and verifying argumentation models and their translations. Various use cases of this framework are provided.

Part I is organised as follows.

Chapter 2 discusses the necessary functional programming preliminaries. Section 2.1 discusses functional programming in general and furthermore provides an overview of the Haskell language, including various examples. Section 2.2 gives an introduction to the Curry-Howard correspondence, dependent type theory and the Agda programming language.

In **Chapter 3** the necessary argumentation background is discussed. Section 3.1 starts with an introduction to process of argumentation, covering the relationship between abstract and structured argumentation. Section 3.2 introduces the abstract argumentation frameworks by Dung. Section 3.3 discusses the 2010 version of the ASPIC$^+$ model. Section 3.4 gives a different treatment of the 2009 version of the Carneades model, explicitly splitting the model in a stage-specific and dialogical part. Section 3.5 discusses ASPIC$^+$ with proof standards/burdens. Section 3.6 gives a treatment of rationality postulates, constraints that can be imposed on the evaluation of argumentation models with structure. Finally, Section 3.7 provides a literature review of other related computational argumentation models.

Part II is organised as follows.

Chapter 4 presents an implementation in Haskell of Dung's argumentation frameworks, its standard semantics and the more recent semi-stable semantics. Furthermore, the labelling algorithm for calculating preferred, stable and semi-stable semantics and all the definitions discussed in Caminada [28] are presented and faithfully implemented. Finally, a library and command-line interface for these implementations are included and discussed.

Chapter 5 presents a case study for implementing a structured argumentation model, Carneades, into Haskell, while simultaneously defining an embedded domain specific language. This implementation is then lifted to a Haskell library.

Chapter 6 relates the Carneades argumentation model with the ASPIC$^+$ framework, by presenting a translation that allows arbitrary Carneades argument evaluation structures to be faithfully represented in ASPIC$^+$. In particular, it is proved that important correspondence results and rationality postulates hold for the translation. Finally, the translation is applied to generalise the Carneades model to allow for evaluation on cycle-containing structures.

Chapter 7 builds on the use cases presented in the previous chapters, providing a general methodology for developing abstract and structured argumentation models, and translations between them. The chapter discusses an implementation in Haskell of a translation from Carneades into Dung, derived from the translation of Carneades into ASPIC$^+$. Properties of the implementation are implemented and tested using the QuickCheck library. Finally, a mechanical formalisation of Dung's AFs up to grounded semantics is presented in Agda.

Chapter 8 presents a use case of the framework constructed in my thesis. A new structured argumentation model is defined by extending ASPIC$^+$ with content orderings, propagation of weights and argument accrual. This model is then, with relative ease, implemented in Haskell by building on the implementations of the previous chapters.

Chapter 9 concludes this thesis, providing an overview of the work done and giving suggestions for future work.

The appendices are structured as follows.

Appendix A gives an overview of the techniques and technologies used in the thesis.

Appendix B elaborates the minor technical contributions to existing argumentation models and translations.

1.3 Relation to previous work

Parts of this thesis were previously published in my MSc. thesis or in scientific articles. All the publications mentioned in this section were written as a lead or only author.

Part of the background material in **Chapter 3**, in particular the basic definitions and four standard semantics of Dung and the majority of the Carneades and ASPIC$^+$ sections, are based on previous work as produced in my MSc thesis [69], the article with Henry Prakken [77] and the articles with Henrik Nilsson [72, 73, 75, 74].

Chapter 4 contains earlier work from [73, 75, 74]. The implementation has been significantly extended to include implementations for the preferred, stable and semi-stable labelling algorithms [28], along with all the accompanying definitions. The Dungell application is based on the work published at ICCMA [71].

Chapter 5 is based on the article published at Trends in Functional Programming [72]. The work has been updated to incorporate changes required to support the translation from Carneades into Dung in [75, 74] and has furthermore been extended to incorporate a parser, an output module and a significantly extended related work section.

Chapter 6 is largely based on the article published in Argument & Computation [77], building on the work of [76, 69].

Chapter 7 is the final product of multiple iterations [72, 73, 75, 74] of developing a framework for the implementation and formalisation of abstract models, structured models and translations between them. The chapter has been extended to include the complete Agda formalisation of Dung's AFs, it describes in more detail how the translation between Carneades and Dung is implemented and it also covers an in depth discussion of related work.

Chapter 8 has not been published before. Part of the research was done with guidance of Henry Prakken, when visiting Utrecht University.

Appendix A is an extended and rewritten version of a previously published article [70].

1.4 Roadmap

This thesis is interdisciplinary, containing technical contributions in both argumentation theory and functional programming. I have attempted to accommodate argumentation theorists, functional programmers and general computer scientists by making the thesis largely self-contained. This section provides three suggested paths for reading the thesis, matching to the three intended audiences.

There is deliberate overlap between chapters, in particular between the background and the functional programming implementations of argumentation models. It is intended that the reader can use the background section as a central reference point for all the mathematical definitions. The functional programming chapters give a self-contained introduction to argumentation models, repeating or rephrasing mathematical definitions to be more suited for a functional programming audience, while providing corresponding implementations.

The following order of progressing through the thesis is advised:

- If the reader is a functional programmer, an introduction to argumentation theory through functional programming in Chapter 4 and Chapter 5 is advised.
  Section order: 3.1 $\Rightarrow$ A $\Rightarrow$ 4 $\Rightarrow$ 5 $\Rightarrow$ 7 $\Rightarrow$ 8 $\Rightarrow$ 9, referring back to the models in Chapter 3 and the translation in Chapter 6 when needed. A book length introduction to argumentation theory and some of the here discussed models is given in Besnard and Hunter [14].

- An argumentation theorist can linearly progress through the thesis, skipping the argumentation background Chapter 3, only referring to it when needed as a refresher or as an introduction to a specific model. Argumentation theorists with no background in (functional) programming, might want to consult a standard introduction to functional programming [99, 111] in addition to the functional programming background Chapter 2, before tackling Chapter 4.

- A reader with neither a background in argumentation theory nor in functional programming could either progress linearly through the report while consulting literature on argumentation [14, 157] or alternatively follow the route of the functional programmer while reading an introduction to functional programming [99, 111].

Part I

Background

Chapter 2

Functional programming background

This chapter gives an introduction to functional programming, discussing the necessary concepts and providing concrete examples to help the reader understand the approach taken in the thesis. For a more in depth introduction the reader should consult Section 2.1.7 and 2.2.4, which discuss introductory texts on (dependently typed) functional programming and other relevant literature[9].

Section 2.1 introduces the functional programming paradigm and in specific, the purely functional programming language Haskell. In particular, it will be made clear what it means for a programming language to be purely functional and what it entails to have strong static typing, lazy evaluation and referential transparency. The reader will also be introduced to concrete Haskell syntax w.r.t. its types and functions, algebraic data types and typeclasses. Finally, the Haskell library QuickCheck will be discussed and references to further reading material will be given.

Section 2.2 introduces the dependently typed functional programming language Agda, a programming language and an interactive system for writing and checking proofs.

2.1 Functional programming: Haskell

This section discusses the paradigm of functional programming and in specific, the programming language Haskell [100, 114, 99]. For a further motivation of the use of functional programming in this thesis, see Chapter 1 and

[9]Further technical and practical concepts used in this thesis, such as literate programming and Haskell packages, are discussed in Appendix A.

further references in Section 2.1.7 and 2.2.4.

2.1.1 Purely functional programming

Purely functional programming takes a quite different approach to programming than that of imperative and object oriented programming languages such as Java [87], C++ [169] and Pascal [190]. Java, C++ and Pascal all rely on the assignment statement, a command that changes the state of a variable to a new value, thereby overwriting the previous value. Assignment statements, and other commands that induce side effects, make it harder to perform reasoning about programs, including proving correctness and termination of programs. This is particularly true in the context of parallelism and shared data structures [170]. To prove correctness of programs that have shared state requires advanced reasoning tools such as separation logic [162].

In contrast, *purely functional programming* deliberately does not allow side effects and shared state, and instead depends on *pure functions*, i.e. functions that do not change the global state and are therefore side effect free. Haskell is a purely functional programming language [100, 114][10], allowing us to more easily prove properties of our Haskell programs by using equational reasoning (see Section 2.1.6 and Hutton [99]) and it also shortens the gap between our programs and a fully mechanically formalised specification in a theorem prover, such as Agda (see Section 2.2). Even Haskell's unpure code, including mutable state and concurrency, can be reasoned about purely using equational reasoning [172].

Consider the following Java code fragment:

```
public static int fibonacci(int n) {
        ...
}
```

We can see that the above function takes an int as argument and returns an int as result. However, despite that intuitively the Fibonacci function should only work on calculating the correct corresponding Fibonacci number, we are not barred from writing to the file system, printing out a document or firing a nuclear missile. A *pure* function does not allow for such side effects. In Haskell we have a type declaration that is very similar to the Java type of *fibonacci*:[11]

```
fibonacci :: Int → Int
fibonacci n = ...
```

[10]Strictly speaking, Haskell does allow for one side effect: non-termination.

[11]We use slightly idealised syntax for Haskell, as generated by the tool *lhs2tex*. See Appendix A for further details.

The same type declaration in Haskell provides more guarantees about the behaviour of the function than the Java specification. By disallowing side effects, we are guaranteed that a function call applied to the same argument will always give the same result, making Haskell functions closer to functions as used in mathematics[12]. Results of computations are then computed deterministically, allowing us to substitute a function application in Haskell by its result, an important property of Haskell called *referential transparency* [99].

2.1.2 Functions and types

A program in a functional programming language consists of a number of defined functions. Functions in Haskell are defined by giving it a name, giving a name to its possible arguments (juxtaposed) and defining a body (after the equals sign) that specifies how the calculation depends on the argument(s).

$$\mathit{double}\ x = x + x$$

Here *double* is a function taking numbers and doubling them. Computation then occurs by applying functions to appropriate arguments (where arguments can again be functions). Function application in Haskell is denoted by juxtaposition, e.g. $f\ 3\ \mathit{True}$ will apply f to arguments 3 and *True*. So *double* 4 will give the expected 8.

Defining *double* as a function can also be seen as defining an equation, relating left and right hand sides of the equals sign. Haskell's referential transparency allows us to substitute equals for equals, combining this with the idea of functions defining equations, we can do *equational reasoning.* For example, if we encounter the expression $\mathit{double}\ x$ in another function, we can without problem replace this by its right hand side $x + x$. For further examples of equational reasoning, see Chapter 13 of Hutton [99].

Haskell is a *statically, strongly typed* language [100, 114]. Static typing implies that the Haskell compiler knows on compile time for each piece of code, what type it has. Haskell is also *strongly typed*, i.e. supplying incorrectly typed arguments or making incorrect type combinations will throw a compiler error. This combination of strong and static typing means that many of the programming errors can be caught at compile time. For an in depth discussion of static type checking and its use in programming languages, see Pierce's book on type systems [135].

[12]By specifying a function as pure, we disallow the implementer of this function to do any type of I/O. In Section 2.1.5 we will see how we can do I/O in Haskell.

Compilers for the Haskell language also provide *type inference*. For instance, the previously defined function *double* will get a type that is automatically inferred by the compiler (or the compiler would throw an error in case it was impossible to infer a valid one). A programmer can always opt to supply the type signature, together with its definition, explicitly.

```
double :: Int → Int
double x = x + x
```

A type expression $e :: T$ can be read as: expression e has type T.

Type

Haskell provides a mechanism for defining *type synonyms*, either as a shorthand for a complex type or to clarify the usage of a type. For instance, in Section 4.1 the simplest instance of an abstract argument is defined to be a *String*, but we might want the users of the code to still think of these arguments as abstract arguments, in contrast to how they are actually implemented.

```
type AbsArg = String
```

Note that the above definition only defines a synonym and the compiler therefore does not handle *AbsArg* and *String* differently. If a programmer does want to make a distinction on names, for instance for types that should be handled differently but are not structurally different, she can use *newtypes* defined in Section 2.1.3.

Polymorphic types

Int, *Bool* and type synonyms like *AbsArg* are all concrete types. Haskell also provides *polymorphic types*, that is a type variable that can be instantiated by any concrete type allowing users to write functions that work for every type, i.e. *parametric polymorphism* [78, 160, 117, 135][13]. Haskell has tuples, meaning that the programmer can define tuples of *Int*s, i.e. (*Int*, *Int*), or any other combination such as (*String*, (*Int*, *Bool*)). A generic tuple is polymorphic and has type (a, b), with a and b being type variables.

A function that takes the first element of a tuple should not need to depend on the concrete instantiation of a and b and this indeed the case in Haskell:

[13]In Section 2.1.5 we will see how we can restrict a type variable to a certain class of types by using typeclasses, where each typeclass can give a a possibly different behaviour, giving rise to *ad-hoc polymorphism.*

$$\begin{aligned}&\mathit{fst} :: (a, b) \to a\\&\mathit{fst}\ (x, y) = x\end{aligned}$$

The type of *fst* can be read as a universally quantified statement: for all types a and b, given a tuple (a, b) return an a. If we ignore infinitely looping and partial definitions, then by *parametricity* [161, 187], the correct definition is also the only possible implementation of this function. Parametricity will be discussed in more detail in Section 2.2.3.

2.1.3 Data types and pattern matching

We have seen that all Haskell expressions (values) have types. For example:

$$\begin{aligned}&\mathit{False} :: \mathit{Bool}\\&\mathit{True} :: \mathit{Bool}\\&\neg \quad\ :: \mathit{Bool} \to \mathit{Bool}\end{aligned}$$

Programmers who want to define new types, such as the *Bool* type, can do so by using **data** declarations. For example, *Bool* is defined in the Haskell standard library as following:

$$\textbf{data}\ \mathit{Bool} = \mathit{True} \mid \mathit{False}$$

On the left hand side is the **data** keyword followed by the name of the type. After the equal signs we get one or more *value constructors*, separated by the bar symbol which can be read as "or".

Type parameters

Value constructors in a **data** definition can have one or more arguments, which can either be concrete types, or variables thereby resulting in a polymorphic type. A good example of the power of Haskell's algebraic data types is the *Maybe* type.

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$$

The *Maybe* data type is a data type respresenting exception. *Maybe* has two constructors, *Just a* in the case of a successful computation, and *Nothing* in case an exception occurs. For example, division on integers has an exceptional case when dividing by zero:

$$\begin{aligned}&\mathit{safeDiv} :: \mathit{Double} \to \mathit{Double} \to \mathit{Maybe}\ \mathit{Double}\\&\mathit{safeDiv}\ m\ 0 = \mathit{Nothing}\\&\mathit{safeDiv}\ m\ n = \mathit{Just}\ \$\ m\ /\ n\end{aligned}$$

Double is a data type for floating point numbers. The *Double* data type can intuitively be seen as a very big data type definition having a case for every floating number it is able to represent. The function *safeDiv* above is defined by pattern matching on the *Double* data type, splitting the function into two lines. The first line matches on the 0 constructor, returning *Nothing* as the result. In the case a number other than 0 is encountered, the result of the division is wrapped in the *Just* constructor. Here $ is shorthand for lowest priority application, meaning *Just* $ m / n is equal to *Just* (m / n). If we would then want to use the result of such a division we can pattern match on the *Maybe* data type, we can use the result of a *Just Double* and for *Nothing* we can either propagate the Nothing type or possibly give an error message.

Lists

Haskell has very strong syntactical support for lists, allowing for easier *pattern matching* and *list comprehensions*. For example, the following function takes the first element of a list:

$$\begin{aligned}&\mathit{head} :: [\,a\,] \to a\\&\mathit{head}\ (x : \mathit{xs}) = x\end{aligned}$$

Here $[\,a\,]$ is a polymorphic list, $(x : \mathit{xs})$ pattern matches on the list, splitting it into its first element and the *tail* of the list, returning the *head*. Note that we do not have a case for the empty list, which will therefore throw an error when taking the *head* of the empty list, $[\,]$. We can however define a safe version of this using the previously discussed *Maybe* datatype.

$$\begin{aligned}&\mathit{safeHead} :: [\,a\,] \to \mathit{Maybe}\ a\\&\mathit{safeHead}\ [\,] \qquad\ = \mathit{Nothing}\\&\mathit{safeHead}\ (x : \_) = \mathit{Just}\ x\end{aligned}$$

The underscore represents a wild card, matching any remaining patterns.

Finally, *list comprehensions* are a Haskell syntactical construct that allows us to specify lists close to mathematical set builder notation. Given that we defined *isEven*, the following function takes the even numbers of a list of numbers:

$$\begin{aligned}&\mathit{evenList} :: [\mathit{Int}] \to [\mathit{Int}]\\&\mathit{evenList}\ \mathit{ns} = [\,n \mid n \leftarrow \mathit{ns}, \mathit{isEven}\ n\,]\end{aligned}$$

This is similar to the mathematical $\{n \mid n \leftarrow ns, n\ is\ even\}$, and indeed the Haskell list comprehension is interpreted in a similar manner. The bar can be read as “such that” and the $n \leftarrow \mathit{ns}$ is called a *generator* and can be read as n is drawn from ns. The other allowed statement beside a generator is a guard statement, i.e. a logical expression resulting in a *Bool*. Only those elements for which the result of the Boolean expression is *True* will be added to the list. Finally, note that all this syntax can be desugared into standard Haskell syntax.

Newtype

While type synonyms are useful for giving types a different name and intuition, we sometimes would like to make two equivalent types explicitly different by using a **newtype** declaration[14]. We can do this to prevent users from mixing up a file name and with another type of *String* or to define functions with different behaviours depending on which **newtype** is used.

```
newtype Feet = Feet Double
newtype CM   = CM  Double
```

Here *Feet* and *CM* are two **newtype** declarations that both use a *Double* as its only argument. Despite that the two are equivalent, a user cannot call a function using *Feet* with a *CM* type and vice versa, thereby preventing some programming errors on the type level.

Finally, newtypes can be used to define different type class (see Section 2.1.5) for what is essentially the same type.

2.1.4 Recursive functions

Haskell's basic mechanism for looping is *recursion* [46], a function is recursive if it is defined "in terms of itself". Standard recursive function have at least one base case (to allow for termination) and one or more recursive cases. For example, the Fibonacci function from Section 2.1.1 can be defined as following:

```
fibonacci :: Int → Int
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n − 1) + fibonacci (n − 2)
```

The multi line definition can be seen as defining three equations, when the argument is 0, 1 or something else. A calculation of *fibonacci* 3 would go as following:

[14]**newtype** declarations are also used to declare cyclic type definitions, which are disallowed in type synonyms.

$$\begin{aligned}
\mathit{fibonacci}\ 3 &= \mathit{fibonacci}\ 2 + \mathit{fibonacci}\ 1 && \text{(applying } \mathit{fibonacci}\ n\text{)}\\
&= (\mathit{fibonacci}\ 1 + \mathit{fibonacci}\ 0) + \mathit{fibonacci}\ 1 \\
& && \text{(applying } \mathit{fibonacci}\ n\text{)}\\
&= (1 + \mathit{fibonacci}\ 0) + \mathit{fibonacci}\ 1 && \text{(applying } \mathit{fibonacci}\ 1\text{)}\\
&= (1 + 0) + \mathit{fibonacci}\ 1 && \text{(applying } \mathit{fibonacci}\ 0\text{)}\\
&= 1 + \mathit{fibonacci}\ 1 && (1 + 0 = 1)\\
&= 1 + 1 && \text{(applying } \mathit{fibonacci}\ 1\text{)}\\
&= 2 && (1 + 1 = 2)
\end{aligned}$$

An important thing to note is that having a base case for a recursive function in Haskell (or most other programming languages), does not guarantee termination. The Agda programming language discussed in Section 2.2 does guarantee this property.

Partial application

In the previous subsection we saw the function *safeDiv*:

$$\mathit{safeDiv} :: \mathit{Double} \to \mathit{Double} \to \mathit{Maybe}\ \mathit{Double}$$

The type of *safeDiv* gives the impression that the function takes two arguments. However, the arrow notation hides the fact that the arrows in types of Haskell functions associate to the right.

$$\mathit{safeDiv} :: \mathit{Double} \to (\mathit{Double} \to \mathit{Maybe}\ \mathit{Double})$$

safeDiv is thus a function that takes one argument, a *Double*, and returns a function *Double* → *Maybe Double*. Using this idea, we can do a *partial application* as following:

```
plus :: Int → Int → Int
plus x y = x + y

plusThree :: Int → Int
plusThree = plus 3
```

Then, we have that:

```
> plusThree 5
8
```

Anonymous functions

In the previous sections, functions were defined by defining an equation, e.g. $double\ x = x + x$, where *double* defines the name. Functions can also be defined anonymously when understood as a lambda expression:

$$\lambda x \rightarrow x + x$$

2.1.5 Typeclasses

Section 2.1.2 discussed polymorphic functions, i.e. functions that work generically for any instantiation of types. Haskell has its own version of *ad-hoc polymorphism* (function overloading) [31] by the use of *typeclasses* [187, 92]. A typeclass declaration is similar to a Java interface, defining a general type of behaviour for all instances of that particular typeclass.

A Haskell programmer can syntactically define the type of functions that work for type variables in a specific typeclass by using a *class constraint* $C\ a$, where C is the name of the typeclass and a is the type variable. For example, the definition of + in the Prelude (the Haskell standard library) uses a class constraint of *Num*, the class of numeric types:

$$(+) :: Num\ a \Rightarrow a \rightarrow a \rightarrow a$$

The type of + can be read as, given that a is an instance of the typeclass *Num*, + has type $a \rightarrow a \rightarrow a$. Any **data** or **newtype** declaration can be turned into an instance of a typeclass, provided that we give an implementation for the methods that typeclass specifies.

Eq

Eq is the class of types that are able to be compared for equality and inequality.

$$\begin{aligned}&\textbf{class}\ Eq\ a\ \textbf{where}\\ &\quad (\equiv) :: a \rightarrow a \rightarrow Bool\\ &\quad (\not\equiv) :: a \rightarrow a \rightarrow Bool\end{aligned}$$

The *Eq* typeclass specifies two methods, $\equiv$ to compare two elements for equality and $\not\equiv$ to compare two elements for inequality. The default definition in the Haskell Prelude implements both methods using the other definition, which means that to successfully define an instance for $Eq\ T$ for type T we only need to define $\equiv$ or $\not\equiv$.

```
instance Eq CM where
  (CM d1) ≡ (CM d2) = d1 ≡ d2
```

Here we define an instance of *Eq* for the *CM* **newtype**, using the already defined $\equiv$ on *Double*s.

Ord

The *Ord* typeclass contains those types that first of all are an instance of the *Eq* typeclass and furthermore can be totally ordered.

```
class Eq a ⇒ Ord a where
  compare :: a → a → Ordering
  (<)     :: a → a → Bool
  (⩾)     :: a → a → Bool
  (>)     :: a → a → Bool
  (⩽)     :: a → a → Bool
  max     :: a → a → a
  min     :: a → a → a
```

The functions $<$, $\geqslant$, $>$, $\leqslant$ implement an ordering in the expected way; *max* and *min* respectively return the maximum and minimum of two elements of *a*; *compare* takes two elements of *a* and determines whether the first element is smaller than (*LT*), equal to (*EQ*) or greater than (*GT*) the second element, with **data** *Ordering* = *LT* | *EQ* | *GT*.

Show

Finally, members of the *Show* typeclass are types that can be presented as *String*s.

```
class Show a where
  showsPrec :: Int → a → ShowS
  show :: a → String
  showList :: [a] → ShowS
```

Show defines three methods: *show* takes any element of *a* and returns a *String* that represents that *a*; *showList* and *showsPrec* are defined for efficiency reasons and will not be discussed.

2.1.6 QuickCheck

QuickCheck [40][15] is a lightweight tool that aids programmers in formulating and automatically testing properties of their Haskell code. Properties are defined as Haskell functions. QuickCheck can, dependent on the type of data used as input for the property, automatically generate data that will test satisfaction of the property.

Properties

Instead of writing unit tests, QuickCheck allows users to state general properties that are required of a function. For example, sensible properties for the *reverse* function from the Haskell standard library would be the following (with $+\!\!+$ defining list concatenation):

$$\begin{aligned}
&\mathit{reverse}\ [x] &&= [x]\\
&\mathit{reverse}\ (\mathit{xs} +\!\!+ \mathit{ys}) &&= \mathit{reverse}\ \mathit{ys} +\!\!+ \mathit{reverse}\ \mathit{xs}\\
&\mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}) &&= \mathit{xs}
\end{aligned}$$

QuickCheck provides a DSL that provides tools to specify such properties:

$$\begin{aligned}
&\mathit{propRevSingleton} :: \mathit{Int} \to \mathit{Bool}\\
&\mathit{propRevSingleton}\ x = \mathit{reverse}\ [x] \equiv [x]\\
&\mathit{propRevApp} :: [\mathit{Int}] \to [\mathit{Int}] \to \mathit{Bool}\\
&\mathit{propRevApp}\ \mathit{xs}\ \mathit{ys} = \mathit{reverse}\ (\mathit{xs} +\!\!+ \mathit{ys}) \equiv \mathit{reverse}\ \mathit{ys} +\!\!+ \mathit{reverse}\ \mathit{xs}\\
&\mathit{propRevRev} :: [\mathit{Int}] \to \mathit{Bool}\\
&\mathit{propRevRev}\ \mathit{xs} = \mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}) \equiv \mathit{xs}
\end{aligned}$$

Note that the type of the properties has to be defined specific to a type (monomorphically), otherwise QuickCheck will not be able to generate arbitrary data for testing the properties.

Using QuickCheck to check properties

QuickCheck provides a basic function *quickCheck* and further customisable variants that can automatically test a property, given that QuickCheck knows how to generate arbitrary test cases. For example, running *quickCheck* on one of the properties defined before, would look as following:

$$\begin{aligned}
&> \mathit{quickCheck}\ \mathit{propRevRev}\\
&\mathit{OK} : \mathit{passed}\ 100\ \mathit{tests}.
\end{aligned}$$

[15]See: `http://hackage.haskell.org/package/QuickCheck`.

Alternatively, in the case of a buggy implementation or an ill-defined property, we could have the following:

```
propRevWrong :: [Int] → Bool
propRevWrong xs = xs ≡ rev xs

> quickCheck propRevWrong
  Falsifiable, after 2 tests :
  [2, 3]
```

Here QuickCheck correctly extracts a counter-example for the parameter xs, which violates the property.

QuickCheck has various pre-defined instances for generating *Int*s, lists and various other standard Haskell data types. If an instance is not available or suited for the specific use case, then the user can define their own *Arbitrary* type class instance.

For example, the instance for your own type of *Int*s that only returns small positive values, might look as following:

```
newtype MyInt = M Int
instance Arbitrary MyInt where
  arbitrary = M (choose (1, 100))
```

Here *choose* is a predefined combinator generating a value between two given ranges.

2.1.7 References

This section gave a quick introduction to various functional programming features and Haskell in specific. There are various resources that give a more in depth introduction to functional programming in Haskell, including the books by Hutton [99], Lipovača [111], O'Sullivan et al. [131] and others [177, 16]. The Haskell language reports [100, 114] are also a good source of information for the Haskell language and libraries.

For a further discussion of the motivations for using functional programming in general, see [105, 8, 96].

2.2 Dependently typed functional programming: Agda

This section discusses the dependently typed (functional) programming language, Agda [128, 127, 129]. Agda, like Haskell, is a pure functional programming language, requiring functions to not have side-effects or shared state.

The type system of Agda forms a consistent logical system, called a *type theory*. The type theory is an extension of Martin-Löf type theory [116, 115], with features such as *dependent types*, *indexed families*, *totality* and the ability to write *proofs* for theorems that are simultaneously (well-typed) terms implementing a type.

2.2.1 Functions and types

Similar to Haskell, Agda programmers can define new data types by using **data** declarations.

```
data Bool : Set where
  true  : Set
  false : Set
data ℕ : Set
  zero : ℕ
  suc  : ℕ → ℕ
```

Since Agda has *kinds* [135], i.e. the type of a type constructor, we need to make it explicit that *Bool* and ℕ are base types by declaring their types to be *Set*. Data type definitions can also take type parameters.

Below we define Agda's equivalents of *Maybe*, *Either* and lists.

```
data Maybe (A : Set) : Set where
  just    : A → Maybe A
  nothing : Maybe A
data _+_ (A B : Set) : Set where
  inl : A → A + B
  inr : B → A + B
data List (A : Set) : Set where
  []    : List A
  _::_ : A → List A → List A
```

Agda does not have all of Haskell's syntactical conveniences, making the declarations of lists more explicit. Agda does allow numerals to be used as a syntactic short hand for naturals.

```
natList : List ℕ
natList = zero :: (suc zero) :: (suc (suc zero)) :: []
  -- or equivalently
natList2 : List ℕ
natList2 = 0 :: 1 :: 2 :: []
```

Pattern matching

Given our inductive type for natural numbers we can define the + function on naturals by pattern matching on the two types of the constructor.

$$
\begin{array}{l}
\_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
\mathit{Zero} \quad\ + \ m = m \\
\mathit{Succ}\ n \ + \ m = \mathit{Succ}\ (n \ + \ m)
\end{array}
$$

To ensure consistency of Agda as a theorem prover, functions need to be total and terminating. To enforce this, Agda has a coverage checker and a termination checker. The coverage checker ensures that functions, such as _+_, are defined on all cases of a data type. The termination checker will determine whether a function is obviously terminating, by checking whether it is structurally recursive (see Norell's thesis [127] and the latest Agda release [129] for details on the exact algorithm used). Termination and coverage checking are undecidable in general. Agda, thus instead places the burden on the programmer to define their functions such that the type checker can automatically detect whether it is terminating and fully covered.

For example, the *safeHead* function, as defined in Section 2, can be defined in Agda as follows:

$$
\begin{array}{l}
\mathit{safeHead} : \mathit{List}\ A \to \mathit{Maybe}\ A \\
\mathit{safeHead}\ [\,] \qquad\quad = \mathit{nothing} \\
\mathit{safeHead}\ (x :: \mathit{xs}) = \mathit{just}\ x
\end{array}
$$

Both cases of the list data type are covered, and there are no recursive calls. Agda will therefore accept this definition. However, Agda equivalent to Haskell's *head* function is not fully covered ([ ]) and will therefore be rejected.

$$
\begin{array}{l}
\quad \text{-- Not accepted by the type checker:} \\
\mathit{head} : \mathit{List}\ A \to A \\
\mathit{head}\ (x :: \mathit{xs}) = x
\end{array}
$$

Similarly, we can try to cheat the type checker by writing a case for [ ], but Agda will mark the *head′* function (and the specific offending case) as possibly non-terminating.

$$
\begin{array}{l}
\quad \text{-- Also not accepted:} \\
\mathit{head}' \ : \mathit{List}\ A \to A \\
\mathit{head}'\ (x :: \mathit{xs}) = x \\
\mathit{head}'\ [\,] \qquad\quad = \mathit{head}'\ [\,]
\end{array}
$$

The append function *app*, appending two lists together, is a polymorphic function taking a type argument A. It is defined by structural recursion on the list's structure and accepted as total by the Agda type checker.

$$\begin{array}{l} \mathit{app} :: (A : \mathit{Set}) \to \mathit{List}\ A \to \mathit{List}\ A \to \mathit{List}\ A \\ \mathit{app}\ A\ [\,]\qquad\quad \mathit{ys} = \mathit{ys} \\ \mathit{app}\ A\ (x :: \mathit{xs})\ \mathit{ys} = x :: (\mathit{app}\ A\ \mathit{xs}\ \mathit{ys}) \end{array}$$

The append function can then be specialised to natural numbers, by explicitly supplying a type argument.

$$\begin{array}{l} \mathit{appNat} :: \mathit{List}\ \mathbb{N} \to \mathit{List}\ \mathbb{N} \to \mathit{List}\ \mathbb{N} \\ \mathit{appNat} = \mathit{app}\ \mathbb{N} \end{array}$$

Agda is a dependently typed programming language, allowing functions to not only depend on type (as is the case for *app*) but also on values. In the case the function does not depend on a value, Agda provides the convenient option to define the parameter as implicit, making it not required to supply the parameter as long as it can be inferred by the type checker (otherwise it will still have to explicitly passed).

Implicit arguments

The append function can be redefined to instead have an implicit argument for the type A by writing it between curly braces. The Agda compiler will then normally be able to be infer the argument through its use.

$$\begin{array}{l} \_{+\!\!+}\_ :: \{A : \mathit{Set}\} \to \mathit{List}\ A \to \mathit{List}\ A \to \mathit{List}\ A \\ [\,]\qquad\quad {+\!\!+}\ \mathit{ys} = \mathit{ys} \\ (x :: \mathit{xs})\ {+\!\!+}\ \mathit{ys} = x :: (\mathit{xs}\ {+\!\!+}\ \mathit{ys}) \end{array}$$

The underscores denote the position(s) in which arguments are supplied around the operator, making it possible to define our own infix but also mixfix operators. If needed, arguments can always be provided explicitly by providing them within curly braces:

$$\begin{array}{l} \mathit{natList3} : \mathbb{N} \\ \mathit{natList3} = \mathit{natList1}\ \ {+\!\!+}\ \ \mathit{natList2} \\ \mathit{natList4} : \mathbb{N} \\ \mathit{natList4}\ = \_{+\!\!+}\_\ \{\mathbb{N}\}\ \mathit{natList1}\ \mathit{natList2} \\ \quad \text{-- a mixfix operator} \\ \mathit{if\_then\_else\_} : \{A : \mathit{Set}\} \to \mathit{Bool} \to A \to A \to A \\ \mathit{if\ true\ then\ lb\ else\ \_} = \mathit{lb} \\ \mathit{if\ false\ then\ \_\ else\ rb} = \mathit{rb} \end{array}$$

2.2.2 Dependent types and functions

Agda generalises data types and the function space to types and functions *dependent on values*. Lists of a defined length, called *vectors*, can be defined similarly to lists, providing an additional natural number index. The *Vec* data type has two constructors. The empty vector [] (for a type A) is a vector with length *zero*. The :: constructor takes an implicit natural number, an element of type A, a vector of type A with length n and constructs a vector of type A with total length $n + 1$.

```
data Vec (A : Set) : ℕ → Set where
  []    : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

The data type declaration for vector has a type parameter A left of the colon. Right of the colon is an index, giving rise to an indexed family of data types, i.e., for each natural number n, *Vec A n* defines a type. Note that the _::_ also defines a dependent function, taking a natural number and returning a vector type depending on the *value*. Finally, Agda does not require that functions or data type constructors use different names, making it possible to use the _::_ constructor for both lists and vectors. The compiler will try to infer the appropriate type given the context.

Given the *Vec* data type, we can now define a *head* function that is total:

```
head : {A : Set} {n : ℕ} → Vec A (suc n) → A
head (x :: xs) = x
```

The coverage checker will accept the above definition, despite the lack of a case for the [] constructor. The Agda compiler (correctly) detects that it is impossible to supply the empty vector to the function while satisfying the type signature of the function (requiring an $n + 1$ element vector).

Absurd patterns

The Agda compiler is not always able to tell directly whether a term or type is satisfiable. If this is the case, then the programmer can provide more information about the data type and explicitly tell the Agda compiler that the pattern is absurd, by supplying the impossible pattern (). It is then not required to provide a definition for that pattern. For example, we can make the absurd pattern explicit in the definition of *head* for a vector by instead matching on the implicit natural:

$$
\begin{aligned}
&\mathit{head}' : \{A : \mathit{Set}\}\ \{n : \mathbb{N}\} \to \mathit{Vec}\ A\ (\mathit{suc}\ n) \to A\\
&\mathit{head}'\ \{\mathit{zero}\}\quad ()\\
&\mathit{head}'\ \{\mathit{suc}\ \_\}\ (x :: \mathit{xs}) = x
\end{aligned}
$$

Dependent pairs

Dependent pairs are a dependent data type that will be useful later in this thesis. Non-dependent pairs are defined similar to Haskell pairs:

$$
\begin{aligned}
&\mathbf{data}\ (A : \mathit{Set}) \times (B : \mathit{Set}) : \mathit{Set}\ \mathbf{where}\\
&\quad <\_,\_> : A \to B \to A \times B
\end{aligned}
$$

A dependent pair is more general, letting the type of the second component of the pair possibly depend on the value of the first component:

$$
\begin{aligned}
&\mathbf{data}\ \Sigma\ (A : \mathit{Set})\ (B : A \to \mathit{Set}) : \mathit{Set}\ \mathbf{where}\\
&\quad <\_,\_> : A \to B \to \Sigma\ A\ B
\end{aligned}
$$

Dependent pairs are useful for pairing a value, such as a natural number, together with a proof about that specific number, e.g. $\Sigma\ (n{:}\mathbb{N})\ (n{+}1 \equiv 1{+}n)$.

The *with* construct

Agda defines a construct for pattern matching on expressions in a function definition by means of the *with* construct. When defining a function, we can add *with p*, where *p* is a valid expression, after the ordinary pattern matches of the function. For example, the function *headOrElse* which takes the *head* of a function, or in the case it is empty, returns a default value, can be defined as following:

$$
\begin{aligned}
&\mathit{headOrElse} : \{A : \mathit{Set}\} \to \mathit{List}\ A \to A \to A\\
&\mathit{headOrElse}\ \mathit{xs}\ y\ \mathit{with}\ \mathit{head}\ \mathit{xs}\\
&\mathit{headOrElse}\ \mathit{xs}\ y \mid \mathit{just}\ x \quad = x\\
&\mathit{headOrElse}\ \mathit{xs}\ y \mid \mathit{nothing} = y
\end{aligned}
$$

The repeating part of the function definition, *headOrElse xs y*, can also be omitted using the ... notation:

$$
\begin{aligned}
&\mathit{headOrElse}' : \{A : \mathit{Set}\} \to \mathit{List}\ A \to A \to A\\
&\mathit{headOrElse}'\ \mathit{xs}\ y\ \mathit{with}\ \mathit{head}\ \mathit{xs}\\
&\ldots \mid \mathit{just}\ x \quad = x\\
&\ldots \mid \mathit{nothing} = y
\end{aligned}
$$

2.2.3 Proving in Agda

The Curry-Howard correspondence [43, 44, 93, 104, 167] observes a correspondence between proofs for theorems in intuitionsitic logic and functions implementing types in typed lambda calculus. This correspondence was extended by Scott and Martin-Löf [116, 115] to develop a foundation for constructive mathematics, called *Martin-Löf's intuitionistic type theory.* A proposition within constructive mathematics is *true* iff it the set of its proofs is inhabited. Conversely, a proposition is *false* iff the set of its proof is empty.

Agda is directly based on Martin-Löf's type theory, including the previously discussed features such as dependent types and totality of functions. This section will demonstrate how we can apply the C-H correspondence between proofs and functions by implementing some of the constructive connectives and constructive proofs.

Intuitionistic logic in Agda

We will start with giving definitions for intuitionistic logic within Agda, based on the Brouwer-Heyting-Kolmogorov (BHK) [167] interpretation of logic (and refined by Martin-Löf [116, 115]). $\bot$ can be represented as the empty type (a type that has no inhabitants).

$\textbf{data}\ \bot : \mathit{Set}\ \textbf{where}$

There are no constructors for *false*, making it impossible to construct a value of it. Conversely, *true* can be represented by a data type with only one constructor and no arguments. Instead, we use a record, to allow Agda to automatically infer the only allowed value (the value can be constructed explicitly by calling *record* { }).

$\mathit{record}\ \top : \mathit{Set}\ \textbf{where}$

Agda can take a Boolean and convert this to the type level, making it possible to test conditions on compile time[16]. The *isTrue* and *isFalse* below respectively convert a *Bool* to the type level.

$\mathit{isTrue} : \mathit{Bool} \to \mathit{Set}$
$\mathit{isTrue}\ \mathit{true}\ = \top$
$\mathit{isTrue}\ \mathit{false} = \bot$

$\mathit{isFalse} : \mathit{Bool} \to \mathit{Set}$

[16]Note that having arbitrary type level computation is justified, because Agda only allows computations that are total.

$$\begin{aligned}&\mathit{isFalse}\ \mathit{true}\ = \bot\\&\mathit{isFalse}\ \mathit{false} = \top\end{aligned}$$

With $\bot$ defined, we can construct the first elimination (or proof rule) in Agda. Given that we have derived $\bot$, it is possible to derive any proposition A.

$$\begin{aligned}&\bot\text{–}\mathit{Elim} : \{A : \mathit{Set}\} \to \bot \to A\\&\bot\text{–}\mathit{Elim}\ ()\end{aligned}$$

Note that the () is a use of an absurd pattern, as defined in Section 2.2.1.

Negation of a proposition in intuitionistic logic is defined to be equivalent to proving that the proposition leads to absurdity.

$$\begin{aligned}&\mathit{Not} : \mathit{Set} \to \mathit{Set}\\&\mathit{Not}\ A = A \to \bot\end{aligned}$$

Constructive conjunction is defined by the non-dependent pair data type. A proof of propositions A and B is thus a proof of A paired up with a proof of B.

$$\begin{aligned}&\_\&\_ : \mathit{Set} \to \mathit{Set} \to \mathit{Set}\\&A\ \&\ B = A \times B\end{aligned}$$

The elimination rules for conjunction are then the projections into the pair.

$$\begin{aligned}&\mathit{fst} : \{A\ B : \mathit{Set}\} \to A\ \&\ B \to A\\&\mathit{fst} < a, b> = a\\&\mathit{snd} : \{A\ B : \mathit{Set}\} \to A\ \&\ B \to B\\&\mathit{snd} < a, b> = b\end{aligned}$$

The $\_\vee\_$ connective can be defined by means of the previously defined disjount union type $(\_+\_)$. In contrast to classical logic, a proof of a disjunction $A + B$ is constructed by providing a proof for either A or B.

$$\begin{aligned}&\_\vee\_ : \mathit{Set} \to \mathit{Set} \to \mathit{Set}\\&A\ \vee\ B = A + B\end{aligned}$$

A disjunction can be eliminated by means of a case statement on its two constructors. Again, notice that this is in strong contrast with classical logic, where having a disjunction does not imply we have access to a proof for one of the two disjuncts.

$$\begin{aligned}
&\textbf{case} : \{A\ B\ C : \mathit{Set}\} \to A\ \vee\ B \to (A \to C) \to (B \to C) \to C\\
&\textbf{case}\ (\mathit{inl}\ a)\ d\ e = d\ a\\
&\textbf{case}\ (\mathit{inr}\ b)\ d\ e = e\ b
\end{aligned}$$

Implication is in BHK is interpreted as a computable function that takes a proof for A and returns a proof for B. The implication constructor is thus as synonym for the Agda function space:

$$\begin{aligned}
&\_\Rightarrow\_ : (A\ B : \mathit{Set}) \to \mathit{Set}\\
&A \Rightarrow B = A \to B
\end{aligned}$$

Bi-implication is a conjunction of the two implications:

$$\begin{aligned}
&\_\Leftrightarrow\_ : \mathit{Set} \to \mathit{Set} \to \mathit{Set}\\
&A \Leftrightarrow B = (A \Rightarrow B)\ \&\ (B \Rightarrow A)
\end{aligned}$$

Given the connectives and their proof rules, it is now possible to construct a small constructive proof in Agda for $A\&B \Leftrightarrow (B\&(A\&A))$ either by applying the elimination rules, or by directly pattern matching:

$$\begin{aligned}
&\mathit{conjProof} : \{A\ B : \mathit{Set}\} \to A\ \&\ B \Rightarrow B\ \&\ (A\ \&\ A)\\
&\mathit{conjProof}\ p = <\mathit{snd}\ p, <\mathit{fst}\ p, \mathit{fst}\ p>>\\
&\mathit{conjProof'} : \{A\ B : \mathit{Set}\} \to A\ \&\ B \Rightarrow B\ \&\ (A\ \&\ A)\\
&\mathit{conjProof'} < a, b> = <b, <a, a>>\\
&\mathit{conjProof2} : \{A\ B : \mathit{Set}\} \to B\ \&\ (A\ \&\ A) \Rightarrow A\ \&\ B\\
&\mathit{conjProof2}\ p = <\mathit{snd}\ (\mathit{snd}\ p), \mathit{fst}\ p>\\
&\mathit{conjProof2'}\ : \{A\ B : \mathit{Set}\} \to B\ \&\ (A\ \&\ A) \Rightarrow A\ \&\ B\\
&\mathit{conjProof2'} < b, <a, a'>> = <a, b> \quad \text{-- or } <a', b>\\
&\mathit{biImplProof} : \{A\ B : \mathit{Set}\} \to A\ \&\ B \Leftrightarrow B\ \&\ (A\ \&\ A)\\
&\mathit{biImplProof} = <\mathit{conjProof}, \mathit{conjProof2}>
\end{aligned}$$

Proving with lists

This section builds up the necessary definitions to perform some simple constructive proofs on list, based on the tutorial of Norell [128]. In particular, definitions and data types will be given to be able to find a given element in a list, constructing a proof whether it was found or not found.

In the previous section, we saw that a *Bool* could be lifted to the type level, using the functions *isTrue* and *isFalse*. Similarly, a decidable predicate on a type A can be lifted to the type level by applying the *satisfies* function:

```
satisfies : { A : Set } → (A → Bool) → A → Set
satisfies p x = isTrue (p x)
```

In Haskell we can find an element an in a list, by calling the *find* function: $find :: Eq\ a \Rightarrow (a \to Bool) \to [a] \to Maybe\ a$. It returns *Just a*, when an element satisfying the predicate is found, or *Nothing* when it is not present. While this might match the intuition of what a *find* function should return, it does not keep any information on why the list contained or did not contain the element. For example, we would not be able to extract from the *find* function, where the element was present or whether it correctly returned *Nothing* (*find* _ _ = *Nothing* would happily type check).

Instead, in Agda we can make it explicit what it means to find to an element in a list by constructing a data type that demonstrates this. The *Find* data type takes two parameters: a type A, a predicate on A and an list that indexes the type. The constructor *found* takes the (possibly empty) front part of the list (*xs*), an element (y), proof that the element satisfies the predicate, the back part of the list (*ys*), and returns a witness that we found the element for the complete list (*xs* ++ (y :: *ys*)). The other constructor *notFound* takes a list (*xs*) and a proof that all the elements of *xs* do *not* satisfy the predicate, returning a witness that we did not find the element.

```
data Find { A : Set } (p : A → Bool) : List A → Set where
  found     : (xs : List A) (y : A) → satisfies p y → (ys : List A) →
              Find p (xs ++ (y :: ys))
  notFound : forall { xs }             → All (satisfies (¬ ∘ p)) xs        →
              Find p xs
```

The *All* data type encapsulates the proof that a type-level predicate P holds on all elements of a given list.

```
infixr 30 _:all:_
data All { A : Set } (P : A → Set) : List A → Set where
all[ ]    : All P [ ]
_:all:_ : forall { x xs } → P x → All P xs → All P (x :: xs)
```

Before the *find* function can be defined in Agda, we need a few more constructs. In particular, equality in Agda can be defined by means of the _ ≡ _ data type, which has a single constructor *refl* that takes an element a, and returns a witness for the proof that $a \equiv a$.

```
data _≡_ { A : Set } : A → A → Set where
  refl : { a : A } → a ≡ a
```

Similar to *Bool* and predicates, we can lift equality statements to the type level. Although it might be obvious to the reader we can never construct a value of *refl* for which we have an a that is not equal to the other element, this information might not be immediately clear to the Agda compiler. The following two functions are therefore particularly useful when dealing with equalities:

```
trueIsTrue : {x : Bool} → x ≡ true → isTrue x
trueIsTrue refl = _
falseIsFalse : {x : Bool} → x ≡ false → isFalse x
falseIsFalse refl = _
```

Given the *refl*, and the corresponding implicit argument a, Agda can infer the correct implementation for the function, allowing us to define the function using the wildcard _.

Finally, we need a *lemma* that can convert a proof that a proposition is false, to the negation of that proposition being true.

```
lemma : {x : Bool} → isFalse x → isTrue (¬ x)
lemma {true} ()
lemma {false} prf = prf
```

With the above tools we can now define a *find* function in Agda, that given a predicate and a list, returns a proof whether an element satisfying that predicate has been found.

```
find : {A : Set} (p : A → Bool) (xs : List A) → Find p xs
find p [] = notFound all []
find p (x :: xs) with p x | inspect p x
... | true  | [prf] = found [] x (trueIsTrue prf) xs
... | false | _ with find p xs
find p (x :: ._) | false | [prf] | found xs y py ys
    = found (x :: xs) y py ys
find p (x :: xs) | false | [prf] | notFound npxs
    = notFound (lemma (falseIsFalse prf) : all : npxs)
```

with takes an expression, and allows us to pattern match on the expression as an additional argument to the function. However, although it is obvious to us that in the first pattern match for a non-empty list that $p\ x \equiv \mathit{true}$ holds, Agda does not keep a reference to the original $p\ x$ expression. We therefore need an additional *with* argument, *inspect p x* that keeps this information around as a proof.

In the case the list is empty, the proof is trivially *notFound all* [ ]. Otherwise, there are two cases: the current element in front of the list satisfies the predicate, which implies we can construct a *found* constructor, or alternatively, we have not found the element yet so we pattern match on a recursive call of *find p xs* and construct the *Find* datatype by adding x correctly to the pattern matched *find p xs*.

For example:

```
exampleList : List AbsArg
exampleList = "A" :: "B" :: "C" :: []
exampleFind : Find (String._≡_ "A") exampleList
exampleFind = find (String._≡_ "A") exampleList
```

Here, *String._ ≡ _* is primitive equality on Strings. Then, executing/normalising *exampleFind*:

```
> exampleFind
found [] "A" (record { }) ("B" :: "C" :: [])
```

2.2.4 References

There are various resources for the Agda programming languages, including tutorials [128, 18], Norell's thesis [127] and the Agda wiki [129]. Further references for intuitionistic logic and Martin-Löf type theory are the work by Martin-Löf [116, 115], the book by Nordstrom, Petersson and Smith [126], and the book by Sorensen and Urzyczyn [167]. For further motivation using dependently typed programming, see also Oury and Swierstra [132].

Chapter 3

Argumentation background

In this chapter an introduction to various computational models of argumentation theory will be presented. The abstract and structured approach to defining argumentation models are introduced, covering motivations, technical definitions and theoretical results. Finally, further relevant literature is reviewed and references to in depth introductions to both argumentation models and argumentation theory in general are given.

Section 3.1 introduces the reader to the argumentation process and the instantiation of arguments.

Section 3.2 introduces Dung's (abstract) argumentation frameworks [48] and covers Dung's standard semantics, semi-stable semantics and argument labellings.

Section 3.3 introduces the ASPIC$^+$ structured argumentation model and general argumentation framework is introduced.

Section 3.4 gives an introduction to both the static, stage-specific and the dialogical versions of the Carneades argumentation model.

Section 3.5 introduces a specialisation of the ASPIC$^+$ model from Section 3.3, enhanced with proof standards and proof burdens.

Section 3.6 discusses rationality postulates, constraints that can be imposed on the evaluation of argumentation models with structure.

Finally, Section 3.7 provides a literature review of other related computational argumentation models.

3.1 Argumentation process

The argumentation models discussed in the thesis are generally static, in the sense that they assume a given set of arguments or knowledge base. However, argumentation can still be seen as a reasoning process. The steps below are based on Caminada and Amgoud [29]:

1. Construct *arguments* given a knowledge base and optional preferences, audiences, proof standards or other constructs.
2. Determine the different *conflicts* between arguments.
3. Evaluate the *acceptability* of the different arguments.
4. Draw conclusions depending on which arguments are *justified*.

Abstract argumentation considers an already given set of arguments and notion of conflict. Users can apply different argumentation semantics to make sense of the arguments and their relationships (see also Section 3.2, for Dung's abstract argumentation frameworks). Thus, abstract argumentation models generally start their reasoning process from step 3.

Structured argumentation instead starts from a given knowledge base containing facts, assumptions or axioms. Arguments are then constructed given a given or user-defined set of rules (defeasible or strict). Structured argumentation models thus generally start from step 1, or possibly even without a pre-constructed knowledge base.

Instantiation

Although the structured and abstract approach might seem fundamentally different, we can apply a structured approach to argumentation using an abstract model by instantiating its arguments and conflict relation. For example, ASPIC$^+$ (see Section 3.3) builds arguments based on a knowledge bases and two types of rules (strict and defeasible). The resulting arguments and defeat relation are then used as input by abstract argumentation frameworks and evaluated using Dung's standard semantics. The conclusions can then be drawn by linking back the acceptable arguments to the original ASPIC$^+$ knowledge base. For a further treatment of instantiation and another instance of it, see Besnard and Hunter [14]. Caminada and Amgoud [29] further discuss instantiation and some reasonable properties a structured argumentation system should adhere to (see also Section 3.6).

3.2 Dung's abstract argumentation frameworks

In 1993, in an attempt to capture the fundamental basics of human argumentation, Dung defined his model of (abstract) argumentation frameworks

(AFs) [47, 48]. An abstract argumentation framework consists of a set of abstract arguments and a binary relation representing attack, the notion that one argument may refute another. The arguments' internal structure is kept open for different instantiations. Arguments can be internally structured as graphs or trees and allowing different types of inferences in the arguments. Later work, including ASPIC$^+$, interprets Dung's attack relation as abstracting from the use of preferences. Instantiating the attack relation then leads to a notion of defeat between arguments, e.g. an argument A defeats another argument B iff A attacks B and A is preferred to B[17]. Although historically the binary relation is called a relation of attack in Dung's formulation, we will, to unify notation with other sections, refer it as a defeat relation, without any further changes of Dung's original definitions.

Dung's work has had a large role in the development of computational argumentation theory. First of all, due to abstract nature of AFs, Dung was able to model logic programming and several of the contemporary approaches to non-monotonic reasoning. Showing that these forms of reasoning can be represented as a form of argumentation allowed concepts to be unified and clarified the relationship between these approaches. Secondly, AFs have become a starting point for developing and relating computational models of abstract argumentation. For instance, preference based AFs [4, 5], value based AFs [12] and various others are inspired by or formally an instance of (translatable to) Dung's AFs (see also Subsection 3.2.5). In Chapter 4 we will see our implementation of Dung's AFs and in Section 6 and 3.3 we will see how Dung's AFs can also be related to structured argumentation models or used to define semantics to structured argumentation models.

3.2.1 Standard definitions

This subsection discusses a large part of the standard definitions of Dung's AFs and is meant to be used as a reference for the further content of the thesis.

Definition 3.1 (Abstract argumentation framework (Adapted Def. 2 of [48])). An *abstract argumentation framework* is a tuple $\langle \mathit{Args}, \mathit{Def} \rangle$, where Args is a set of arguments and $\mathit{Def} \subseteq \mathit{Args} \times \mathit{Args}$ is an arbitrary relation on Args representing defeat.

Note that no restrictions are imposed on the defeat relation as such: what meanings can be ascribed to an argumentation framework is instead

[17]For an overview of the use of preferences in determining the defeat relation cf. Section 3.3 of Prakken [145].

wholly captured by the notion of *extensions* defined later (Definition 3.8). In particular, a defeat relation is not assumed to be symmetric as an defeated argument does not necessarily constitute a counter-defeat of the defeating argument. The relation is not even assumed to be anti-reflexive; i.e., self-contradicting arguments are not ruled out.

Example 3.2. Consider an abstract argumentation framework with three arguments: A, B and C. For instance, the arguments might pertain to whether a murder has been committed or not: C = "The accused is guilty of murder since there was a killing and it was done with intent"; B = "Witness X testified the accused did not have the intent to murder the victim"; and A = "Witness X is known to be unreliable". Thus B defeats C and A defeats B. Consequently A *reinstates* C as A defeats the defeater of C. This is formally captured by $AF_1 = \langle\{A, B, C\}, \{(A, B), (B, C)\}\rangle$; see Figure 3.1.

$$\begin{tikzcd} A \arrow[r] & B \arrow[r] & C \end{tikzcd}$$

Figure 3.1: An (abstract) argumentation framework

The following are standard definitions for AFs such as the acceptability of arguments and admissibility of sets. We use an arbitrary but fixed argumentation framework $AF = \langle Args, Def\rangle$.

Definition 3.3 (Set-defeats (Remark 4 of [48])). A set $S \subseteq Args$ of arguments defeats an argument $A \in Args$ iff there exists a $B \in S$ such that $(B, A) \in Def$.

For example, in Figure 3.1, $\{A, B\}$ set-defeats C, because B defeats C and $B \in \{A, B\}$.

Definition 3.4 (Conflict-free (Def. 5 of [48])). A set $S \subseteq Args$ of arguments is called *conflict-free* iff there are no X, Y in S such that $(X, Y) \in Def$.

Considering a set of arguments as a position an agent can take with regards to its knowledge, conflict-freeness is often taken as the minimal requirement for a reasonable position. For example, in Figure 3.1, $\{A, C\}$ is a conflict-free set.

Definition 3.5 (Acceptability (Def. 6.1 of [48])). An argument $X \in Args$ is acceptable with respect to a set S of arguments, or alternatively S *defends* X, iff for all arguments $Y \in Args$: if $(Y, X) \in Def$ then there exists a $Z \in S$ for which $(Z, Y) \in Def$.

An argument is acceptable (w.r.t. to some set S) if all its defeaters are defeated in turn. (Note that although the acceptability is w.r.t. to a set S, *all* defeaters are taken in account.) For example, in Figure 3.1, C is acceptable w.r.t. $\{A, B, C\}$, because A defeats the only defeater of C, i.e. B.

Dung defined the semantics of argumentation frameworks by using the concepts of *characteristic function* of an AF and *extensions*.

Definition 3.6 (Characteristic function (Def. 16 of [48])). The *characteristic function* of AF, $F_{AF} : 2^{Args} \to 2^{Args}$, is a function, such that, given a set of arguments S, $F_{AF}(S) = \{X \mid X \text{ is acceptable w.r.t. to } S\}$.

For example, in Figure 3.1, $F_{AF}(\emptyset) = \{A\}$, $F_{AF}(\{A\}) = \{A, C\}$ and $F_{AF}(\{A, B, C\}) = \{A, C\}$.

The characteristic function for a given AF is monotonic (with respect to set inclusion). We can see this by noticing that if an argument A is acceptable w.r.t. to S, then it is also acceptable w.r.t. any superset of S.

A conflict-free set of arguments is said to be *admissible* if it is a defendable position, that is, it can defend itself from incoming defeats.

Definition 3.7 (Admissibility (Def. 6.2 and Lemma 18 of [48])). A conflict-free set of arguments S is admissible iff every argument X in S is acceptable with respect to S, i.e. $S \subseteq F_{AF}(S)$.

Note that not every conflict-free set is necessarily admissible. For example, in Figure 3.1, $\{C\}$ is conflict-free but is not an admissible set, since $(B, C) \in \mathit{Def}$ and there is no argument in $\{C\}$ that defends it from this defeat. Note that by definition, every empty set is also admissible.

Extensions can be seen as a refinement of admissible sets that do not reject arguments without reason. An extension is a subset of *Args* that are acceptable when taken together. An extension thus constitute a possible meaning, a *semantics*, for an argumentation framework. Below we define the four standard extensions as given by Dung [48]. The definition of extensions through F_{AF} relies on the monotonicity of F_{AF} given a subset ordering on AF.

Definition 3.8 (Extensions (Def. 7, Def. 13, Def. 20, Def. 23, Lemma 24 and Theorem 25 of [48])). Given an argumentation framework AF and a conflict-free set S of arguments, $S \subseteq \mathit{Args}$, then, with the ordering determined by set inclusion, S is a:

- *complete extension* iff $S = F_{AF}(S)$; i.e., S is a fixed point of F_{AF}.
- *grounded extension* iff S is the least fixed point of F_{AF}.
- *preferred extension* iff S is a maximal fixed point of F_{AF}.

- *stable extension* iff it is a preferred extension defeating all arguments in $Args \backslash S$.

Alternatively, a set of arguments S is a complete extension, if it is admissible and for each A defended by S, $A \in S$ holds. The grounded and preferred extensions can, respectively, be characterised as the smallest and a maximal complete extension.

Intuitively, a *complete extension* is a set of arguments that is able to defend itself, including all arguments it defends. It is mainly used to define the other extensions. Further, the (unique) *grounded extension* is a minimal standpoint, including only those arguments without defeaters and those that are "completely defended". A *stable extension* is an extension that is able to defeat all arguments not included in it. Finally, a *preferred extension* is a relaxation from that requirement, weakening it to an as large as possible set still able to defend itself from defeats.

Given our definition of extension, we can determine the *justification status* of an argument according to a *sceptical* or *credulous* viewpoint. Sceptical justification under a certain semantics, implies that an argument is in all extensions for that semantics, while credulous justification only implies existence of an extension including that argument.

Definition 3.9 (Justification status (Based on Def. 3.23 of [145] and [26]))**.** For $s \in \{complete, grounded, preferred, stable\}$, A is sceptically or credulously justified under the s semantics if A belongs to all, respectively at least one, s extension.

We wrap up this subsection by demonstrating the definitions on an extensive example.

Example 3.10 (Calculating extensions)**.** Given the following argumentation framework, $AF = \langle Args, Def \rangle$ with $Args = \{A, B, C, D, E, F, G\}$ and $Def = \{(A, B), (C, B), (C, D), (D, C), (D, E), (E, G), (F, E), (G, F)\}$, as depicted in Figure 3.2.

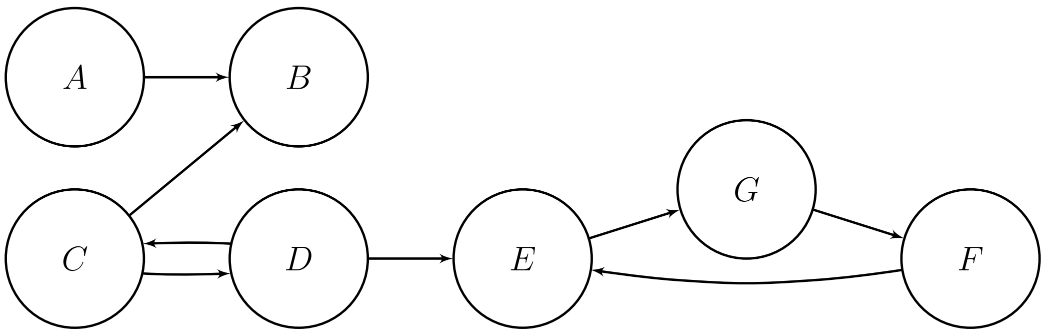


Figure 3.2: A more complex argumentation framework

Since there are no self-defeating loops, we have that in addition to the empty set, $\{\}$, all single argument sets, $\{A\}$, $\{B\}$, $\{C\}$, $\{D\}$, $\{E\}$, $\{F\}$, $\{G\}$, are conflict-free. All pairs of arguments without defeating edges are conflict-free: $\{A, C\}$, $\{A, D\}$, $\{A, E\}$, $\{A, F\}$, $\{A, G\}$, $\{B, D\}$, $\{B, E\}$, $\{B, F\}$, $\{B, G\}$, $\{C, E\}$, $\{C, F\}$, $\{C, G\}$, $\{D, F\}$, $\{D, G\}$. The remaining conflict-free sets are the following: $\{A, C, F\}$, $\{A, C, G\}$, $\{A, D, F\}$, $\{A, D, G\}$, $\{B, D, F\}$, $\{B, D, G\}$.

A conflict-free set is admissible if every argument in that set can defend itself from incoming defeats. The empty set is therefore admissible by definition, the single argument set, $\{A\}$ is admissible since it does not have incoming defeats, while $\{C\}$ and $\{D\}$ defeat their only defeater (D and C respectively). $\{C\}$ and $\{D\}$ can be extended with A into $\{A, C\}, \{A, D\}$ since A is undefeated, while $\{D, G\}$ is admissible because D defeats, E, the only defeater of G. Finally, we can extend $\{D, G\}$ into $\{A, D, G\}$ with the same reasoning as before.

The complete extensions exclude those admissible sets for which the characteristic function F returns a strict superset of the set. For example, the admissible sets $\{C\}$, $\{D, G\}$ also trivially defend A (A is undefeated), resulting in $\{A, C\}$ and $\{A, D, G\}$ after applying the characteristic function, thereby excluding them as complete extensions. Similarly by defeating all defeaters, $\{D\}$ and $\{A, D\}$, will both give $\{A, D, G\}$ and are therefore not a complete extension. This leaves $\{A\}$, $\{A, D, G\}$ and $\{A, C\}$ as complete extensions.

From the complete extension we can compute the other extensions. The grounded extension can be calculated by taking the smallest complete extension, which is $\{A\}$. Preferred extensions (maximal complete extensions) then are $\{A, C\}$ and $\{A, D, G\}$. The set $\{A, C\}$ does not attack arguments E, F and G and is therefore not a stable extension, while $\{A, D, G\}$ is.

Finally, we can see A is sceptically and credulously accepted under complete, grounded, preferred and stable semantics. C is only credulously accepted under preferred and complete semantics and not accepted under stable semantics. D and G are credulously accepted under preferred semantics, but credulously and sceptically accepted under stable semantics. All other arguments are not acceptable under either justification.

3.2.2 Properties and further definitions

This subsection briefly discusses the most important theorems of Dung's paper [48] and introduces a few remaining definitions that will be used later in this thesis.

Dung's fundamental lemma relates admissibility and acceptability, demonstrating that admissible sets are compatible with additional acceptable arguments.

Lemma 3.11 (Fundamental lemma (Lemma 10 of [48])). *Let S be an admissible set of arguments, and A and A' be arguments which are acceptable w.r.t. S. Then:*
(1) $S' = S \cup \{A\}$ is admissible, and
(2) S' defends A'.

The following theorem then follows directly from the fundamental lemma above.

Theorem 3.12 (Theorem 11 of [48]). *Let AF be an argumentation framework. Then:*
(1) The set of all admissible sets of AF form a complete partial order with respect to set inclusion.
(2) For each admissible set S of AF, there exists a preferred extension E of AF such that $S \subseteq E$.

Given Theorem 3.12 and the fact that the empty set is always admissible, it follows that the following corollary holds:

Corollary 3.13 (Corollary 12 of [48]). *Every argumentation framework has at least one preferred extension.*

We now continue to define a few properties with respect to the characteristic function of an argumentation framework.

Lemma 3.14 (Lemma 19 of [48]). *The characteristic function for a given AF, F_{AF}, is monotonic with respect to set inclusion.*

Given that F_{AF} is monotonic, we now know that the grounded extension, defined as the least fixed point of F_{AF}, is guaranteed to exist.

The grounded extension can thus be calculated by iterating F_{AF} over the empty set, the least element in the domain, until a fixed point is reached. The calculation of the least fixed point is guaranteed to succeed (given that we apply a finite set of arguments), due to the previously stated properties of F_{AF}.

Finally, we use Dung's definition of a *well-founded argumentation framework*. A well-founded AF is an AF without cycles or an infinite defeating chain of arguments. This definition will be useful later when we prove the translation of Carneades to be well-founded.

Definition 3.15 (Well-founded argumentation framework (Def. 29 of [48])). An argumentation framework is *well-founded* iff there does not exist an infinite sequence of arguments: $A_0, A_1, \ldots, A_n, \ldots$ such that for each i, $(A_{i+1}, A_i) \in \mathit{Def}$.

The differences between the semantics collapse in an argumentation framework in which there are no cycles.

Theorem 3.16 (Theorem 30 of [48]). *Every well-founded argumentation framework has exactly one complete extension which is grounded, preferred and stable.*

3.2.3 Semi-stable extensions

In this subsection we define another semantics for Dung's AFs, the semi-stable semantics by Caminada et al.[27, 28]

Before defining the semi-stable semantics, we define sets of defeating and defeated arguments, on both arguments and argument sets. A^+ and A^- respectively are the set of arguments defeated by A and the set of arguments that defeat A. Similarly, for a set of arguments S, S^+ and S^- denote the set of arguments defeated by at least one argument in S and the set of arguments that defeat at least one argument in S.

Definition 3.17 (Sets of defeating and defeated arguments (Def. 2 of [28])). Let $AF = \langle \mathit{Args}, \mathit{Def} \rangle$ be an argumentation framework, $A \in \mathit{Args}$ and $S \subseteq \mathit{Args}$. Then:

- A^+ and S^+ are respectively defined as $\{B \mid (A, B) \in \mathit{Def}\}$ and $\{B \mid (A, B) \in \mathit{Def}, A \in S\}$,
- A^- and S^- are respectively defined as $\{B \mid (A, B) \in \mathit{Def}\}$ and $\{B \mid (A, B) \in \mathit{Def}, A \in S\}$.

In Definition 3.8 we introduced both preferred and stable extensions. Stable extension capture the idea of having a viewpoint (set of arguments) that can defeat all arguments not included in it; however, it is not guaranteed to exist.

The preferred semantics can be seen as a relaxation of the stable semantics. However, the preferred semantics is not the closest possible semantics to stable semantics. The *semi-stable* semantics occupies the space betweens stable and preferred semantics. The semi-stable semantics always exist and furthermore, given the existence of at least one stable extension, it coincides with the stable semantics.

Definition 3.18 (Semi-stable extension (Def. 3 of [28])). Given an argumentation framework AF and a conflict-free set S of arguments, $S \subseteq Args$, then, with the ordering determined by set inclusion, S is a *semi-stable extension* iff it is a complete extension where $Args \cup Args^+$ is maximal.

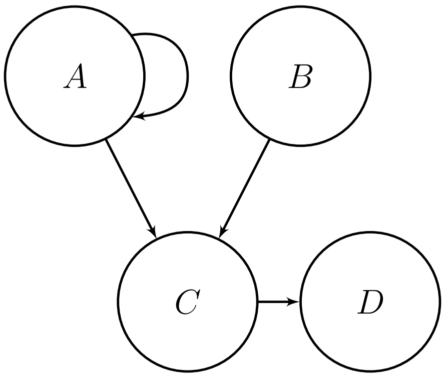


Figure 3.3: An argumentation framework with no stable extensions

Example 3.19 (Stable and semi-stable extensions). In Figure 3.3 we have an argumentation framework with no stable extensions, however the AF does have a preferred and semi-stable extension: $\{B, D\}$.

3.2.4 Argument labellings

Given an argumentation framework, we can determine which arguments are justified by applying an argumentation semantics from Definition 3.8 and Definition 3.18. This subsection we will instead take the *labelling-based* approach to calculating semantics of an argumentation framework. The labelling approach was originally put forward by Pollock [139] and later extended by Caminada [26, 28].

Formally, the labelling approach is a generalisation of the extension-based approach [28], permitting further possibilities than arguments being either *in* or *out* of an extension; e.g. the state of an argument might be *undecided.* In the thesis the standard labels, in, out and undec will be adopted.

Definition 3.20 (Labelling (Definition 6.1 of [119]))**.** Let $\langle Args, Def \rangle$, be an argumentation framework.

- A labelling is a total function from arguments into labels, $\mathcal{L} : Args \to \{In, Out, Undec\}$.
- $\text{in}(\mathcal{L}) = \{x \mid \mathcal{L}(x) = In\}$; $\text{out}(\mathcal{L}) = \{x \mid \mathcal{L}(x) = Out\}$; $\text{undec}(\mathcal{L}) = \{x \mid \mathcal{L}(x) = Undec\}$.

We will often refer to a labelling $\mathcal{L}$ as a triple $\langle i, o, u \rangle$ where i, o, u are the sets of arguments that are *in*, *out*, or of *undecided* status respectively. We further write $\text{in}(\mathcal{L}), \text{out}(\mathcal{L}), \text{undec}(\mathcal{L})$ to refer to the first, second, and third field of a labelling $\mathcal{L}$.

Caminada defines the concept of *illegally labelled* arguments to determine whether an argument should or should not be part of a labelling corresponding to some semantics.

Definition 3.21 (Illegal arguments (Def. 5 of [28]))**.** Given an argumentation framework $\langle Args, Def \rangle$, an argument $A \in Args$ and a labelling $\mathcal{L}$ over $Args$, we have that:

1. A is *illegally* in iff A is labelled in
   but not all its defeaters are labelled out;
2. A is *illegally* out iff A is labelled out
   but does not have a defeater labelled in;
3. A is *illegally* undec iff A is labelled undec
   but either all of its defeaters are labelled out or it has a defeater that is labelled in.

A labelling has *no* illegal arguments iff there is no argument that is illegally in, illegally out or illegally undec.

Similarly, we can define *legally labelled arguments.*

Definition 3.22 (Legally labelled arguments (Def. 5 of [28]))**.** Given an argumentation framework $\langle Args, Def \rangle$, an argument $A \in Args$ and a labelling $\mathcal{L}$ over $Args$, we have that:

1. A is *legally* in iff A is labelled in and it is not illegally in;

2. A is *legally* out iff A is labelled out and it is not illegally out;
3. A is *legally* undec iff A is labelled undec and it is not illegally undec.

Given the definition of illegal arguments we can define admissible labellings and the labellings corresponding to the previously defined semantics.

Definition 3.23 (Admissible labellings (Def. 6 of [28]))**.** An *admissible labelling* is a labelling with no arguments that are illegally in or illegally out.

The complete labelling is a strengthening of the admissible labelling, additionally requiring that no arguments are illegally undec.

Definition 3.24 (Complete labellings (Def. 7 of [28]))**.** A *complete labelling* is a labelling with no arguments that are illegally in, illegally out or illegally undec.

We can then define different semantics based on the complete labelling.

Definition 3.25 (Further labellings (Def. 8 of [28]))**.** Given an argumentation framework $\langle Args, Def \rangle$ and a labelling $\mathcal{L}$ over $Args$, with the ordering determined by set inclusion, we define $\mathcal{L}$ to be a

- *grounded labelling* iff in($\mathcal{L}$) is minimal;
- *preferred labelling* iff in($\mathcal{L}$) is maximal;
- *stable labelling* iff undec($\mathcal{L}$) $= \emptyset$;
- *semi-stable labelling* iff undec($\mathcal{L}$) is minimal.

Example 3.26 (Labellings)**.** In Figure 3.4 we have an argumentation framework with two cycles and an attack from B to C. The argumentation framework has six admissible labellings ($\langle \text{in}, \text{out}, \text{undec} \rangle$):

$$\begin{array}{lll}
\mathcal{L}_1 = \langle \{A\}, & \{B\}, & \{C, D, E\} \rangle \\
\mathcal{L}_2 = \langle \{B, D\}, & \{A, C, E\}, & \emptyset \rangle \\
\mathcal{L}_3 = \langle \{B, D\}, & \{A, C\}, & \{E\} \rangle \\
\mathcal{L}_4 = \langle \{B\}, & \{A, C\}, & \{D, E\} \rangle \\
\mathcal{L}_5 = \langle \{B\}, & \{A\}, & \{C, D, E\} \rangle \\
\mathcal{L}_6 = \langle \emptyset, & \emptyset, & \{A, B, C, D, E\} \rangle
\end{array} \quad .$$

A few of these labellings are very similar due to arguments being allowed to be illegally undec. The set of complete labelling is a refinement of the admissible labellings, disallowing labellings $\mathcal{L}_3, \mathcal{L}_4$ and $\mathcal{L}_5$. For example, labelling $\mathcal{L}_3$ is

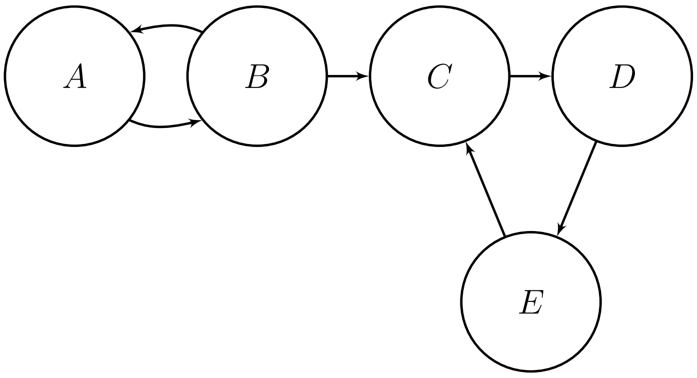


Figure 3.4: An argumentation framework with a stable extension

not a complete labelling, since argument D is labelled in making argument E illegally undec. The complete labellings are thus:

$$\begin{array}{llll}
\mathcal{L}_1 = \langle\{A\}, & \{B\}, & \{C, D, E\}\rangle & \\
\mathcal{L}_2 = \langle\{B, D\}, & \{A, C, E\}, & \emptyset\rangle & \\
\mathcal{L}_6 = \langle\emptyset, & \emptyset, & \{A, B, C, D, E\}\rangle & .
\end{array}$$

From the complete labellings, we can see that $\mathcal{L}_6$ has the minimum number of arguments labelled in and is therefore the grounded labelling. Similarly, we have that $\mathcal{L}_1$ and $\mathcal{L}_2$ are both preferred labellings. From the preferred labellings, $\mathcal{L}_2$ is the only semi-stable and stable labelling.

Although the above definitions give us the means to determine whether a labelling is e.g. admissible or preferred, we have not yet defined an algorithm to compute the labellings. In Chapter 4 we will discuss an algorithm for the grounded labelling [119] and an algorithm for computing preferred, semi-stable and stable labellings [28] and provide implementations of both the algorithms and of all definitions discussed in this and previous subsections.

Finally, based on the results of Caminada, we can relate a labelling to its respective extension by taking the arguments that are labelled in.

Theorem 3.27 (Theorem 1, Theorem 2 and Theorem 3 of [28])**.** *Let $\langle Args, Def\rangle$ be an argumentation framework and $S \subseteq Args$, then S is:*

- *admissible iff there exists an admissible labelling $\mathcal{L}$ with* $\text{in}(\mathcal{L}) = Args$;
- *a $s \in \{$complete, grounded, preferred, stable, semi-stable$\}$ extension iff there exists an s-labelling $\mathcal{L}$ with* $\text{in}(\mathcal{L}) = Args$.

3.2.5 References

This section introduced argumentation frameworks as proposed by Dung [48]. Our introduction to AFs handled most of Dung's standard definitions, while also introducing semi-stable semantics [27] and the labelling approach [139, 26, 28] to calculating semantics. See Baroni and Giacomin [11] for an introduction to argumentation frameworks based on possible principles of extension-based semantics, including further semantics such as stage semantics [182] and ideal semantics [51].

Recently, there has been a strong effort in implementing abstract argumentation frameworks and other abstract systems. Charwat et al. [36] give a good overview of the various implementations, including fast implementations that in contrast to the labelling approach or other standard algorithms for computing extensions, use answer set programming or constraint satisfaction programming. See Section 4.4 for an in depth discussion of various implementations and the recent ICCMA competition [175].

3.3 ASPIC$^+$ (2010)

The abstract argumentation frameworks by Dung [48], see Section 3.2, define an abstract argumentation model[18], keeping the structure and nature of arguments and the defeat relation unspecified. This allows for general reasoning about the acceptability status of arguments, but provides no guidance for the modelling of actual argumentation problems. Other research, including work before Dung (see Section 3.7), has taken a structured approach to argumentation [2, 83, 130, 14, 49].

Structured argumentation frameworks (ASPIC$^+$) as defined by Prakken [145], are a further development of the ASPIC framework as defined by Amgoud et al. [3]. Prakken's frameworks instantiate[19] the abstract argumentation model of Dung, defining the internal structure of arguments; defining multiple types of attack and adding preferences to the attack relation; resulting in a defeat relation.

This additional structure, such as the arguments built up as inference

[18]See Section 3.7 for a discussion of other abstract models of argumentation.

[19]Prakken [145] calls the general argumentation model, defined by Dung [48], argumentation frameworks. This is in contrast to the use of Dung, where an argumentation framework is a specific set of arguments and defeat relation. Instantiations of the abstract argumentation model by Dung (for example, arguments based on propositional logic) are in ASPIC$^+$ called *argumentation systems* and argumentation frameworks are called *argumentation theories*. Dung's attack relation is called defeat, and is defined by a combination of an ASPIC$^+$ attack relation and preferences.

trees, a distinction between defeasible and strict rules and preferences determining actual defeat, make it easier to implement other concrete argumentation systems such as Carneades [86]. Prakken [145] has already shown assumption-based argumentation [52, 17], a structured argumentation approach using assumptions from which conclusions are drawn using strict inference rules, to be a special case of ASPIC$^+$. A recent and more general version of ASPIC$^+$ is discussed in Modgil and Prakken [122]. In their paper they prove additional relationships to other approaches of argumentation such as Hunter and Besnard's [14] classical logic approach to argumentation.

Throughout the thesis, various versions of ASPIC$^+$ are used. The work in Chapter 6 translates Carneades into the ASPIC$^+$ (2010) framework [145], since it was developed when ASPIC$^+$ (2010) was current[20]. In Chapter 8 we will discuss and extend the 2013 version of ASPIC$^+$, while taking ideas from the proof burdens and standards version of ASPIC$^+$ [154] (see Section 3.5).

3.3.1 Basic definitions

The basic building block of a structured argumentation framework is the concept of an *argumentation system*, extending the standard notion of a proof system. In argumentation systems the logical language is left unspecified except for the existence of a contrariness relation (generalisation of logical negation to asymmetric conflict). Inference rules are divided into *strict* and *defeasible* rules. The strict rules can contain standard deductive (domain independent) inference rules, but can also be used to model domain specific inference rules such as $Bachelor \rightarrow \neg Married$. Defeasible rules can contain general reasoning patterns, such as abduction, or domain-specific knowledge such as birds generally being able to fly. Defeasible rules are susceptible to exceptions and hence disputable. Finally, defeasible rules in an argumentation system are ordered on strength by means of a partial preorder.

Contrary and *contradictory* formulas in the language will now be defined. Beside preferences, contraries also play a role in determining the defeat relation between arguments.

Definition 3.28 (Logical language (Def. 3.2 of [145])). Let $\mathcal{L}$, a set, be a logical language and $^-$ a contrariness function from $\mathcal{L}$ to $2^{\mathcal{L}}$. Given that $\varphi \in \overline{\psi}$, then:

- if $\psi \notin \overline{\varphi}$ then φ is called a *contrary* of ψ,
- otherwise, $\psi \in \overline{\varphi}$ and φ and ψ are called *contradictory*, i.e. $\varphi \in \overline{\psi}$ and $\psi \in \overline{\varphi}$.

[20]There is no existing formal translation from ASPIC$^+$ 2010 to ASPIC$^+$ (2013), making it a significant contribution outside of the scope of the thesis to update the translation.

Definition 3.29 (Strict and defeasible rules (Def. 3.4 of [145])). Let $\varphi_1, \ldots, \varphi_n, \varphi$ be elements of $\mathcal{L}$.

- A *strict rule* is of the form $\varphi_1, \ldots, \varphi_n \rightarrow \varphi$.
- A *defeasible rule* is of the form $\varphi_1, \ldots, \varphi_n \Rightarrow \varphi$.

$\varphi_1, \ldots, \varphi_n$ are called the *antecedents* of the rule and φ its *consequent*.

Definition 3.30 (Argumentation system (Def. 3.1 of [145])). An *argumentation system* is a tuple $AS = \langle \mathcal{L}, ^-, \mathcal{R}, \leqslant \rangle$ where

- $\mathcal{L}$ is a logical language,
- $^-$ is a contrariness function from $\mathcal{L}$ to $2^{\mathcal{L}}$,
- $\mathcal{R} = \mathcal{R}_s \cup \mathcal{R}_d$ is a set of strict ($\mathcal{R}_s$) and defeasible inference rules ($\mathcal{R}_d$) such that $\mathcal{R}_s \cap \mathcal{R}_d = \emptyset$,
- $\leqslant$ is a partial preorder on $\mathcal{R}_d$.

Since this definition leaves the nature of the logical language and the inference rules largely unspecified, it is possible to reformulate specific argumentation systems as instances of ASPIC$^+$. For example, Prakken [145] has shown that assumption-based argumentation [52, 17], a structured argumentation approach using assumptions from which conclusions are drawn using strict inference rules, is a special case of ASPIC$^+$, and Modgil and Prakken [121] have proven the same for variants of argumentation using classical logic (cf. Besnard and Hunter [14]).

With the argumentation system defined, we can now look at the construction of arguments by means of a knowledge base in an argumentation system. The set of rules contains both a strict and defeasible kind and the knowledge base can be inconsistent. Vreeswijk (Chapter 8 of [184]) distinguished two ways of reasoning on with uncertainty: plausible reasoning, which is sound reasoning on an uncertain basis, and defeasible reasoning, which is unsound reasoning on a solid basis. For instance, the deductive account of argumentation given by Besnard and Hunter [14] is a case of plausible reasoning; arguments are built by using only strict rules on consistent subsets of a possibly inconsistent knowledge base. An example of defeasible reasoning is Defeasible Logic [130]. In Defeasible Logic one can apply defeasible and strict rules on a set of indisputable statements giving the ability to infer definite or defeasible conclusions depending on the type of rule used. ASPIC$^+$ [145] combines plausible and defeasible reasoning.

In addition to the possible inconsistency, the knowledge base also contains four different types of facts, inspired by a similar distinction of [83]. Similar to the axioms in deductive logic, there are (unattackable) premises called *necessary axioms* ($\mathcal{K}_n$), (attackable) *ordinary premises* ($\mathcal{K}_p$), *assumptions*

($\mathcal{K}_a$) — which are a weak type of premise always defeated by an attack — and *issues* ($\mathcal{K}_i$) — which are premises that are not acceptable unless backed by further argument.

Definition 3.31 (Knowledge base (Def. 3.5 of [145]))**.** A *knowledge base* in an argumentation system $\langle \mathcal{L}, ^-, \mathcal{R}, \leqslant \rangle$ is a pair $\langle \mathcal{K}, \leqslant' \rangle$ where $\mathcal{K} \subseteq \mathcal{L}$ and $\leqslant'$ is a partial preorder on $\mathcal{K} \backslash \mathcal{K}_n$. Here $\mathcal{K} = \mathcal{K}_n \cup \mathcal{K}_p \cup \mathcal{K}_a \cup \mathcal{K}_i$ where these subsets of $\mathcal{K}$ are disjoint.

3.3.2 Arguments

With the knowledge base and inference rules defined as above, the construction of arguments can be defined by adopting Vreeswijk's [184, 185] definition of an argument. The smallest argument is simply a fact from the knowledge base. More complex arguments can be constructed by chaining inference rules on previous arguments, resulting in an argument in tree form (containing subarguments).

Definition 3.32 (Arguments (Def. 3.6 of [145]))**.** An *argument* A on the basis of a knowledge base $\langle \mathcal{K}, \leqslant' \rangle$ in an argumentation system $\langle \mathcal{L}, ^-, \mathcal{R}, \leqslant \rangle$ is:

1. φ if $\varphi \in \mathcal{K}$ with:
   $\mathit{Prem}(A) = \{\varphi\}$,
   $\mathit{Conc}(A) = \varphi$,
   $\mathit{Sub}(A) = \{\varphi\}$,
   $\mathit{DefRules}(A) = \emptyset$,
   $\mathit{TopRule}(A) =$ undefined.

2. $A_1, \ldots, A_n \rightarrow \psi$ if $A_1, \ldots, A_n$ are arguments such that there exists a strict rule $\mathit{Conc}(A_1), \ldots, \mathit{Conc}(A_n) \rightarrow \psi$ in $\mathcal{R}_s$,
   $\mathit{Prem}(A) = \mathit{Prem}(A_1) \cup \ldots \cup \mathit{Prem}(A_n)$,
   $\mathit{Conc}(A) = \psi$,
   $\mathit{Sub}(A) = \mathit{Sub}(A_1) \cup \ldots \cup \mathit{Sub}(A_n) \cup \{A\}$,
   $\mathit{DefRules}(A) = \mathit{DefRules}(A_1) \cup \ldots \cup \mathit{DefRules}(A_n)$,
   $\mathit{TopRule}(A) = \mathit{Conc}(A_1), \ldots, \mathit{Conc}(A_n) \rightarrow \psi$.

3. $A_1, \ldots, A_n \Rightarrow \psi$ if $A_1, \ldots, A_n$ are arguments such that there exists a defeasible rule $\mathit{Conc}(A_1), \ldots, \mathit{Conc}(A_n) \Rightarrow \psi$ in $\mathcal{R}_d$,
   $\mathit{Prem}(A) = \mathit{Prem}(A_1) \cup \ldots \cup \mathit{Prem}(A_n)$,
   $\mathit{Conc}(A) = \psi$,
   $\mathit{Sub}(A) = \mathit{Sub}(A_1) \cup \ldots \cup \mathit{Sub}(A_n) \cup \{A\}$,
   $\mathit{DefRules}(A) = \mathit{DefRules}(A_1) \cup \ldots \cup \mathit{DefRules}(A_n) \cup$

$\{Conc(A_1), \ldots, Conc(A_n) \Rightarrow \psi\}$,
$TopRule(A) = Conc(A_1), \ldots, Conc(A_n) \Rightarrow \psi$.

Example 3.33. Given an argumentation system and a knowledge base in that argumentation system with the following rules and facts (where $q, r \rightarrow s$ means with q and r derived, derive s):

$$\begin{aligned}
\mathcal{R}_s &= \{z, s \rightarrow t;\ q, r \rightarrow s\} \\
\mathcal{R}_d &= \{p, u \Rightarrow q\} \\
\mathcal{K}_n &= \{p;\ z\} \\
\mathcal{K}_p &= \{u\} \\
\mathcal{K}_a &= \{r\} \\
\mathcal{K}_i &= \{s\}
\end{aligned}$$

An argument for t can be constructed by using an issue premise from $\mathcal{K}_i$, as seen in Figure 3.5. (The type of premise is used as superscript.)

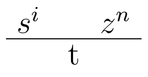


Figure 3.5: An argument for t using an issue premise

Arguments using issue premises will result in not being acceptable. Therefore to produce a possibly acceptable argument for t after evaluation, we will want to derive an argument for s instead of using the issue premise. Such an argument for t can be seen in Figure 3.6. Here double lines indicate a defeasible inference.

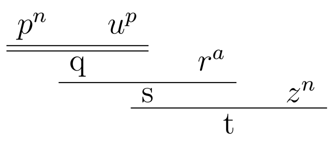


Figure 3.6: Another argument for t

This argument contains several subarguments which can formally be written as follows:

$$\begin{array}{ll}
A_1 : p & A_5 : A_1, A_2 \Rightarrow q \\
A_2 : u & A_6 : A_5, A_3 \rightarrow s \\
A_3 : r & A_7 : A_6, A_4 \rightarrow t \\
A_4 : z &
\end{array}$$

Here A_7 is the argument from Figure 3.6.

Definition 3.34 (Argument properties (Def. 3.8 of [145])). An argument A is

- *strict* if $\mathit{DefRules}(A) = \emptyset$;
- *defeasible* if $\mathit{DefRules}(A) \neq \emptyset$;
- *firm* if $\mathit{Prem}(A) \subseteq \mathcal{K}_n$;
- *plausible* if $\mathit{Prem}(A) \nsubseteq \mathcal{K}_n$.

Given the construction of arguments and its properties we can now define *argument orderings*, $\preceq$, where $A \preceq B$ can be read as: argument B is at least as good as argument A.

Definition 3.35 (Admissible argument orderings (Def. 3.10 of [145])). Let A be a set of arguments. Then a partial preorder $\preceq$ on A is an *argument ordering* iff

1. if A is firm and strict and B is defeasible or plausible, then $B \prec A$;
2. if $A = A_1, \ldots, A_n \rightarrow \psi$ then for all $1 \leqslant i \leqslant n$, $A \preceq A_i$ and for some $1 \leqslant i \leqslant n$, $A_i \preceq A$.

Below we give two possible admissible argument orderings that are commonly used in the literature [3]: the weakest-link and last-link principle. The *weakest-link principle* considers the strength of the premises between arguments and defeasible rules used. So the weakest-link principle considers all uncertain elements in an argument to determine the ordering.

Definition 3.36 (Weakest-link principle (Def. 6.17 of [145])). Let A and B be two arguments. Then $A \preceq B$ iff either condition (1) of Definition 3.35 holds; or

1. $\mathit{Prem}(A) \prec_S \mathit{Prem}(B)$; and
2. If $\mathit{DefRules}(B) \neq \emptyset$ then $\mathit{DefRules}(A) \prec_S \mathit{DefRules}(B)$.

The *last-link principle* considers the strength of the last used defeasible rule between two arguments. Only when both arguments are strict, we consider the strength of the premises.

Definition 3.37 (Last defeasible rules (Def. 6.12 of [145])). Let A be an argument, then:

- $\mathit{LastDefRules}(A) = \emptyset$ iff $\mathit{DefRules}(A) = \emptyset$.
- If $A = A_1, \ldots, A_n \Rightarrow \varphi$, then $\mathit{LastDefRules}(A) = \{\mathit{Conc}(A_1), \ldots, \mathit{Conc}(A_n) \Rightarrow \varphi\}$, otherwise $\mathit{LastDefRules}(A) = \mathit{LastDefRules}(A_1) \cup \ldots \cup \mathit{LastDefRules}(A_n)$.

Definition 3.38 (Last-link principle (Def. 6.14 of [145])). Let A and B be two arguments. Then $A \preceq B$ iff either condition (1) of Definition 3.35 holds; or

1. $\mathit{LastDefRules}(A) \prec_S \mathit{LastDefRules}(B)$; or
2. $\mathit{LastDefRules}(A)$ and $\mathit{LastDefRules}(B)$ are empty and $\mathit{Prem}(A) \prec_S \mathit{Prem}(B)$.

Now we can define the notion of an argumentation theory.

Definition 3.39 (Argument theories (Def. 3.11 of [145])). An *argumentation theory* is a triple $AT = \langle AS, KB, \preceq \rangle$ where AS is an argumentation system, KB is a knowledge base in AS and $\preceq$ is an argument ordering on the set of all arguments that can be constructed from KB in AS.

3.3.3 Attack and defeat

With the internal structure of arguments defined it is now possible to distinguish between types of attack.

An *undermining attack* is an attack on the premises on an argument and is the only attack possible in the context of strict rules. An *undercutting attack* is an attack on the (defeasible) inference step and is a way to provide “exceptions to the rule”. Finally, *rebutting attack* is done by constructing a contrary or contradictory conclusion for the attacked argument’s (sub)conclusion.

The definition of undercutting attack assumes that inference rules can be named in the object language, $\mathcal{L}$. Prakken [145] leaves the precise nature of this naming convention implicit (see Modgil and Prakken [122] for an example of an explicit naming convention).

Definition 3.40 (Types of attack (Def. 3.16, 3.11, 3.14 of [145])).

- Argument A *undermines* argument B (on φ) iff $\mathit{Conc}(A) \in \overline{\varphi}$ for some $\varphi \in \mathit{Prem}(B) \backslash \mathcal{K}_n$. In such a case A *contrary-undermines* B iff $\mathit{Conc}(A)$ is a contrary of φ or if $\varphi \in \mathcal{K}_a$.

- Argument A *undercuts* argument B (on B') iff $\mathit{Conc}(A) \in \overline{B'}$ for some $B' \in \mathit{Sub}(B)$ of the form $B_1'', \ldots, B_n'' \Rightarrow \psi$.

- Argument A *rebuts* argument B (on B') iff $\mathit{Conc}(A) \in \overline{\varphi}$ for some $B' \in \mathit{Sub}(B)$ of the form $B_1'', \ldots, B_n'' \Rightarrow \psi$. In such a case A *contrary-rebuts* B iff $\mathit{Conc}(A)$ is a contrary of φ.

The types of attack can be combined with an argument ordering to define the notion of defeat. Similar to contraries, an undercutting attack does not take the ordering into account and hence always results in defeat. Intuitively the undercutter contains an argument for an exception to the rule of the attacked argument, otherwise an undercutter and the attacked argument using that rule could be in the same extension.

Definition 3.41 (Types of defeat (Def. 3.19, 3.20 of [145]))**.**

- Argument A *successfully rebuts* argument B if A rebuts B on B' and either A contrary-rebuts B' or $A \nprec B'$.
- Argument A *successfully undermines* argument B if A undermines B on φ and either A contrary-undermines B or $A \nprec \varphi$.

The previous notions can be combined in an overall definition of defeat:

Definition 3.42 (Defeat (Def. 3.21 of [145]))**.** Argument A *defeats* argument B iff no premise of A is an issue and A undercuts or successfully rebuts or successfully undermines B. Argument A *strictly defeats* argument B iff $(A, B) \in \mathit{Def}$ and B does not defeat A.

To deal with issue premises, an argument is acceptable only if it contains no issue premises, therefore changing Definition 3.5 to:
An argument $A \in \mathit{Args}$ is acceptable with respect to a set S of arguments, or alternatively S *defends* A, iff A *contains no issue premises and* for all arguments $B \in S$: if $\mathit{Def}(B, A)$ holds then there is a $C \in S$ for which $\mathit{Def}(C, B)$ holds [21].

With arguments and the defeat relation fully defined, it is possible to link the argumentation theories of the structured approach to Dung's abstract argumentation frameworks, thereby formally making the correspondence between the structured and abstract approach.

Definition 3.43 (Argumentation framework (Def. 3.22 of [145]))**.** An *abstract argumentation framework (AF) corresponding to an argumentation theory AT* is a pair $\langle \mathit{Args}, \mathit{Def} \rangle$ such that:

- Args is the set of arguments on the basis of AT as defined by Definition 3.32,
- Def is the relation on Args given by Definition 3.42.

[21]This slightly changes the definition of Prakken [145], disallowing arguments with issue premises to be acceptable at all, instead of only excluding them from extensions.

Finally, the acceptability of conclusions (of a mathematical language $\mathcal{L}$) is defined in the corresponding argumentation framework.

Definition 3.44 (Acceptability of conclusions (Def. 3.23 of [145]))**.** For any semantics S and for any argumentation framework AF and formula $\varphi \in \mathcal{L}_{AF}$:

1. φ is *sceptically S-acceptable* in AF if and only if all S-extensions of AF contain an argument with conclusion φ;

2. φ is *credulously S-acceptable* in AF if and only if there exists an S-extensions of AF that contain an argument with conclusion φ.

Example 3.45. Given an argumentation system and a knowledge base in that argumentation system with the following rules and facts (where $q, r \to \neg r_1$ means that given q and r, rule r_1 does not apply, ie. undercut):

$$\begin{aligned}
\mathcal{R}_d &= \{bird \Rightarrow_r fly;\ penguin \Rightarrow \neg r\} \\
\mathcal{K}_p &= \{bird;\ penguin\} \\
\mathcal{K}_n &= \mathcal{K}_a = \mathcal{K}_i = \mathcal{R}_s = \emptyset \\
\leqslant &= \leqslant' = \emptyset \\
{}^{-} &= \{(r, \neg r)\}
\end{aligned}$$

The arguments on the basis of this knowledge base are the following:

$$\begin{array}{ll}
A_1 : bird & B_1 : penguin \\
A_2 : A_1 \Rightarrow_r fly & B_2 : B_1 \Rightarrow \neg r
\end{array}$$

The defeat relation on basis of the argumentation system and knowledge base (independent of the ordering $\preceq$), $Def = \{(B_2, A_2)\}$, can be visualised together with the arguments constructed:

$$\begin{tikzcd}
A_1 & B_1 \\
A_2 & B_2 \arrow[l]
\end{tikzcd}$$

Figure 3.7: Corresponding argumentation framework

For this argumentation framework we have one complete (and thus one grounded, preferred and stable) extension, namely $E = \{A_1, B_1, B_2\}$. We can see that *bird* and *penguin* are sceptically and credulously justified in E.

3.3.4 Properties of argumentation theories

Argumentation theories, see Definition 3.39, satisfy various desirable properties [121]. In particular, the rules of an argumentation theory are *closed under contraposition* and *transposition*, the strict rules and axioms should be logically consistent (*axiom-consistent*), and finally a rule should not have a conclusion that is contrary of another strict rule's conclusion (*well-formed*).

Definition 3.46 (Properties of an argumentation theory [121]). Given an argumentation theory AT with language $\mathcal{L}$, then:

- AT is *closed under contraposition* iff for all $S \subseteq \mathcal{L}$, $s \in S$ and $\varphi \in \mathcal{L}$, if $S \vdash \varphi$ then $S \backslash \{s\} \cup \{-\varphi\} \vdash \neg s$.
- AT is *closed under transposition* iff for all $S \subseteq \mathcal{L}$, $s \in S$ and $\varphi_1 \ldots \varphi_n, \psi \in \mathcal{L}$, if $\varphi_1, \ldots, \varphi_n \rightarrow \psi \in \mathcal{R}_s$, then for $i = 1 \ldots n$, $\varphi_1, \varphi_{i-1}, -\psi, \varphi_{i+1}, \ldots, \varphi_n \rightarrow \neg\varphi_i \in \mathcal{R}_s$.
- AT is *axiom-consistent* iff $Cl_{\mathcal{R}_s}(\mathcal{K}_n)$ is consistent (where $Cl_{\mathcal{R}_s}(P)$ is the smallest set containing P and the consequent of any strict rule in $\mathcal{R}_s$, whose antecedents are in $Cl_{\mathcal{R}_s}(P)$).
- AT is *well-formed* iff if φ is a contrary of ψ then [$\psi \notin \mathcal{K}_n$ and ψ is not the consequent of a strict rule].

If these properties hold for an argumentation theory in ASPIC$^+$, then the AT also satisfies the rationality postulates as defined by Caminada and Amgoud [29] (see Section 3.6). See Prakken [145] for proofs.

Alternatively, if an argument ordering is reasonable, then the rationality postulates hold as well. First we define a *maximum fallible subargument*:

Definition 3.47 (Maximum fallible subarguments (Def. 6.5 of [145])). For any argument A, an argument $A' \in Sub(A)$ is a *maximum fallible subargument* of A if

1. A''s final inference is defeasible or A' is a non-axiom premise; and
2. there is no $A'' \in Sub(A)$ such that $A'' \neq A$ and $A' \in Sub(A'')$ and A'' satisfies condition 1.

The set of maximum fallible subarguments of an argument A will be denoted by $M(A)$.

A *reasonable argument ordering* is then defined as follows:

Definition 3.48 (Reasonable argument orderings (Def. 6.7 of [145])). Argument ordering $\preceq$ is *reasonable* if it satisfies the following condition. Let A and B be arguments with contradictory conclusions such that $B \prec A$. Then there exists a $B_i \in M(B)$ and an A^+ with $A \in \mathit{Sub}(A^+)$ such that $\mathit{Conc}(A^+) = -\mathit{Conc}(B_i)$ and $A^+ \not\prec B_i$.

Both the weakest-link and last-link principle, Definition 3.36 and 3.38, have been proved by Prakken [145] to be reasonable.

Proposition 3.49 (Proposition 6.15 and 6.18 of [145]). *The weakest-link and last-link argument orderings are reasonable.*

3.3.5 References

The European ASPIC project [3] developed a structured argumentation model unifying various existing approaches to argumentation, including the work of Pollock [137, 138] and Vreeswijk [184] on the structure of arguments, and Pollock's different types of defeat [136, 137].

Prakken's ASPIC$^+$ [145] is a significant extension of the ASPIC model integrating additional concepts of the argumentation community into a single framework, including the premise types of the Carneades argumentation model [83], undermining attack as defined in Vreeswijk [184], the contrariness relation from Verheij [183] and Bondarenko et al. [17], and an extension of preference orderings as defined in the original ASPIC [3][22].

Later versions of ASPIC$^+$ [122, 123], including the work on EAFs$^+$ [120], added preferences over arguments, and reformulated and repaired various of the original definitions. The latest version of ASPIC$^+$ also captures Besnard and Hunter's approach to argumentation based on propositional logic [14].

Finally, Grooters and Prakken's recent work [91] explores para-consistent logic within the ASPIC$^+$ system.

3.4 Carneades (2009)

Carneades is a formal model of argumentation incorporating both static and dynamic aspects of argumentation. In Carneades argumentation is seen as a dialogical process, determining the acceptability of arguments by applying proof standards, where the assignment of proof standards to arguments is determined by the various proof burdens. The version of Carneades that we will discuss is most recent version by Gordon and Walton [86], putting

[22]See the article of Prakken [145] for further discussion and references.

emphasis on the stage-specific part of the model. It is primarily to our interest because of the existing reduction to abstract dialectical frameworks [22, 23]. Contrary to how Carneades has been introduced in the literature [83, 86], but similar to Brewka and Gordon [22], we will introduce proof standards to be part of the static, evaluative part of Carneades.

Now follows an introduction to Carneades in two parts: a static, stage specific part and a dynamic part including the dialogical elements. This is in line with the results of the reduction of static, stage-specific Carneades to Dung's argumentation frameworks in Chapter 6.

3.4.1 Stage-specific Carneades

We will start with an introduction to Carneades' concept of arguments. Similar to structured abstract argumentation frameworks introduced in the previous section, arguments in Carneades are not left abstract but given certain structure. Arguments are constructed by linking premises and exceptions to a conclusion. Arguments *pro* and *con* a conclusion are later aggregated and evaluated through proof standards.

Definition 3.50 (Arguments (Def. 1 of [22])). Let $\mathcal{L}$ be a propositional language. An *argument* is a tuple $\langle P, E, c\rangle$ where $P \subset \mathcal{L}$ are its *premises*, $E \subset \mathcal{L}$ with $P \cap E = \emptyset$[23] are its *exceptions* and $c \in \mathcal{L}$ is its *conclusion*. For simplicity, c and all members of P and E must be literals, i.e. either an atomic proposition or a negated atomic proposition. Let p be a literal. If p is c, then the argument is an argument *pro* p. If p is the complement of c, then the argument is an argument *con* p.

In the Carneades model a dialogue consists of three phases, i.e. an *opening*, *argumentation* and *closing* phase. These phases each consist of multiple stages, where the sequence of stages intuitively corresponds to the process of building up an argumentation. For evaluating the current set of arguments in a specific stage we do not need the information of the other stages or phases and we can thus, analogous to Brewka and Gordon [22], introduce a *stage specific Carneades argument evaluation structure*.

Before introducing the evaluation structure we will need to define the concept of an *audience* and an *acyclic set* of arguments. An audience evaluates a set of arguments with respect to their assumptions and the weights they assign to these arguments.

Definition 3.51 (Audience (Adapted Def. 3 of [86])). Let $\mathcal{L}$ be a propositional language. An *audience* is a tuple $\langle$*assumptions*, *weight*$\rangle$, where

[23]The constraint $P \cap E$ was not explicit in the original definition of [86].

assumptions $\subset \mathcal{L}$ is a consistent set of literals assumed to be acceptable by the audience and *weight* is a function[24] mapping arguments to real numbers in the range $0.0 \ldots 1.0$, representing the relative weights assigned by the audience to the arguments.

Carneades is defined with an acyclic set of arguments to simplify the definitions and time needed for the evaluation of arguments[25]. In the original definition by Gordon and Walton [86], sets of arguments were restricted to be acyclic in the sense that the chains of arguments $a_1, \ldots, a_n$ constructable from the set should be acyclic. A chain is constructed by sequencing arguments,arguments, intuitively giving the possibility to link the conclusion of an argument to a premise of a later argument. Acyclicity of a chain implied that a premise of an argument a_i could not be used in a conclusion of an argument a_j later in the chain. This however still results in a non-well-founded definition of acceptability in Carneades due to possible cycles in exceptions in arguments, e.g. consider $arguments = \{a, b\}$ with $a = \langle \emptyset, \{p\}, q\rangle$ and $b = \langle \emptyset, \{q\}, p\rangle$. This is probably a small oversight, contrasting the correct definitions in Gordon et al. [83].

Instead, we will use the more general concept of a dependency graph to determine acyclicity of a set of arguments, based on the definition of Brewka and Gordon [22].

Definition 3.52 (Acyclic set of arguments (Adapted Def. 2.1 of [22]))**.** A set of *arguments* is *acyclic* iff its corresponding dependency graph is acyclic. The corresponding dependency graph has nodes for every literal appearing in the set of arguments. A node p has a link to node q whenever p depends on q in the sense that there is an argument pro or con p that has q or $\overline{q}$ in its set of premises or exceptions.

The previous definitions can now be combined to define Carneades' concept of an evaluation structure:

Definition 3.53 (Stage specific Carneades argument evaluation structure (Adapted Def. 4 of [86]))**.** A *(stage specific) Carneades argument evaluation structure* (CAES) is a tuple $\langle arguments, audience, standard\rangle$, where

[24]In contrast to how it was presented by Gordon and Walton [86], the *weight* function will not be partial. Evaluation of arguments without weights was, in exception of the rare case that the standard was *scintilla* for both c and $\overline{c}$, undefined, and therefore the partiality of the weight function will be assumed to be incorrect.

[25]A possible generalisation to cyclic structures was left as future work [86]. This restriction was lifted in the translation done to abstract dialectical frameworks by Brewka and Gordon [22] and we will see that similar results hold for the translation done in Section 6.2.3.

arguments is an acyclic set of arguments, *audience* is an audience and *standard* is a total function mapping literals in $\mathcal{L}$ to their applicable proof standards.

In the (legal) dialogues that Carneades models, proof burdens, such as the burden of production or the burden of persuasion, can be assigned to a propositional literal. Intuitively, an assigned proof burden obliges a participant of the dialogue to provide proof for that proposition, under the condition that the proof satisfies a certain standard of proof. The burden of production, called an "evidential burden", is a burden that requires the party to whom the burden was assigned to produce some extent of evidence. The burden of persuasion is an often stronger burden, requiring the obligated party to convince that a statement holds "beyond reasonable doubt" or according to some other proof standard [153]. We will discuss proof burdens in more detail in the next subsection.

In the model, the assignment of the proof standard is determined by the function *standard* provided in the previous definition. This proof standard can be evaluated in a specific stage and thus can be taken as a static part of Carneades. Proof standards included in the Carneades model, originating from the work of Freeman and Farley [63, 62], are *scintilla of evidence*, *preponderance of the evidence*, *clear and convincing evidence*, *beyond reasonable doubt* and finally *dialectical validity.* A proof standard is a function that given a literal p, aggregates arguments pro and con p and evaluates to *true* or *false* depending on a specific audience[26].

Definition 3.54 (Proof standard (Def. 4 of [86])). A *proof standard* is a function mapping tuples $\langle \mathit{issue}, \mathit{arguments}, \mathit{audience}\rangle$ to $\{\mathit{true}, \mathit{false}\}$, where *issue* is a literal in $\mathcal{L}$, *arguments* is an acyclic set of arguments and *audience* is an audience.

Given a CAES and the concept of a proof standard we can define the acceptability of a literal. The acceptable set of literals can be seen as the collection of literals for which the proof standard is satisfied.

Definition 3.55 (Acceptability of literals (Adapted Def. 5 of [86])[27]). Given $C = \langle \mathit{arguments}, \mathit{audience}, \ \mathit{standard}\rangle$ a CAES, p a literal in $\mathcal{L}$ and $s = \mathit{standard}(p)$ the proof standard corresponding to P. Then the literal p is *acceptable* in C iff $s(p, \mathit{arguments}, \ \mathit{audience})$ is *true*.

[26]This slightly generalises Carenades in that we allow complementary literals to be given different proof standards, in contrast to [83] and the implementations of Carneades.

[27]This (stage-specific) definition fixes a small technical error in the original definition by Gordon and Walton [86]. The function *standard* returning a proof standard was instead directly called with the arguments for a proof standard.

All proof standards defined depend on the concept of argument applicability and thus this needs to be defined first.

Definition 3.56 (Applicability of arguments (Adapted Def. 6 of [22])). Let $C = \langle \textit{arguments}, \textit{audience}, \textit{standard} \rangle$ be a CAES. An argument $\langle P, E, c \rangle \in$ *arguments* is *applicable* in C iff

- $p \in P$ implies p is an assumption of the *audience* or [$\overline{p}$ is not an assumption and p is acceptable in C] and
- $e \in E$ implies e is not an assumption of the *audience* and [$\overline{e}$ is an assumption or e is not acceptable in C].

Now we can turn to the definition of Carneades' proof standards. There is one subtle matter concerning the first proof standard, *scintilla of evidence.* A literal that is assigned scintilla of evidence as its proof standard obliges the party who puts forward an argument for this literal to produce an applicable argument. So to satisfy the scintilla of evidence standard only an applicable argument needs to be constructed; contradictory arguments do not influence the acceptability, thus allowing both p and its contradiction $\overline{p}$ to be acceptable in a given CAES.

Definition 3.57 (Proof standards (Adapted Def. 7, 8, 9, 10 and 11 of [86])). Given a CAES $C = \langle \textit{arguments}, \textit{audience}, \textit{standard} \rangle$ and a literal p in $\mathcal{L}$.

- *scintilla*$(p, \textit{arguments}, \textit{audience}) = \textit{true}$ iff there exists at least one applicable argument pro p in *arguments*.
- *preponderance*$(p, \textit{arguments}, \textit{audience}) = \textit{true}$ iff there exists at least one applicable argument pro p in *arguments* for which the weight assigned by the *audience* is greater than the weight of the applicable arguments con p.
- *clear-and-convincing*$(p, \textit{arguments}, \textit{audience}) = \textit{true}$ iff there is an applicable argument A, pro p for which:
    - *preponderance*$(p, \textit{arguments}, \textit{audience})$ holds and
    - the weight for A exceeds the threshold α, and
    - the difference between the weight of A and the maximum weight of the applicable con arguments exceeds the threshold β.
- *beyond-reasonable-doubt*$(p, \textit{arguments}, \textit{audience}) = \textit{true}$ iff *clear-and-convincing*$(p, \textit{arguments}, \textit{audience})$ holds and the maximum weight of the applicable con arguments is less than the threshold γ.

- *dialectical-validity*$(p, \mathit{arguments}, \mathit{audience}) = \mathit{true}$ iff there exists at least one applicable argument pro p in *arguments* and no argument con p in *arguments* is applicable.

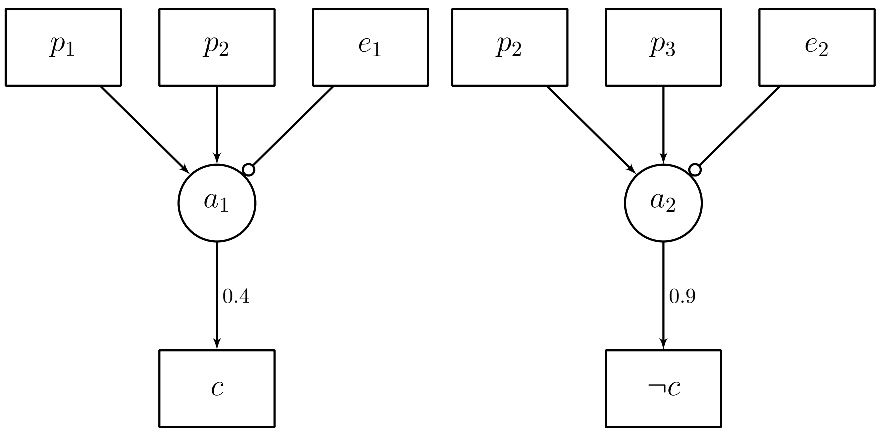
The theory of a (stage-specific) Carneades argument evaluation structure is constructed by combining the acceptable literals in that CAES with the assumptions of the audience. In Carneades propositional logic is assumed as the logical language, so the theory is taken to be closed under propositional inferences.

Definition 3.58 (Theory of a CAES (Page 12 of [86])). Let $C = \langle \mathit{arguments}, \mathit{audience}, \mathit{standard} \rangle$ be a CAES. The *theory* of C is the deductive closure, in propositional logic, of the union of *assumptions* and the set of literals acceptable in C.

Example 3.59. Let $\alpha = 0.3$, $\beta = 0.3$ and $\gamma = 0.6$. Consider a CAES $C = \langle \mathit{arguments}, \mathit{audience}, \mathit{standard} \rangle$ and $\mathit{audience} = \langle \mathit{assumptions}, \mathit{weight} \rangle$ with

$$\begin{aligned}
\mathit{arguments} &= \{a_1, a_2, a_3, a_4\},\\
a_1 &= \langle \{p_1, p_2\}, \{e_1\}, c\rangle,\ a_2 = \langle \{p_2, p_3\}, \{e_2\}, \neg c\rangle,\\
a_3 &= \langle \{p_2\}, \{e_3\}, \neg c\rangle,\ a_4 = \langle \emptyset, \{e_4\}, \neg c\rangle,\\
\mathit{assumptions} &= \{p_1, p_2, e_4\},\\
\mathit{weight}(a_1) &= 0.4;\ \mathit{weight}(a_2) = 0.9;\ \mathit{weight}(a_3) = 0.5;\ \mathit{weight}(a_4) = 0.6,\\
\mathit{standard}(c) &= \mathit{preponderance},\ \mathit{standard}(\neg c) = \mathit{clear\text{-}and\text{-}convincing}.
\end{aligned}$$

We can visualise these arguments (arrows denote premises/inferences and open circles denote exceptions):



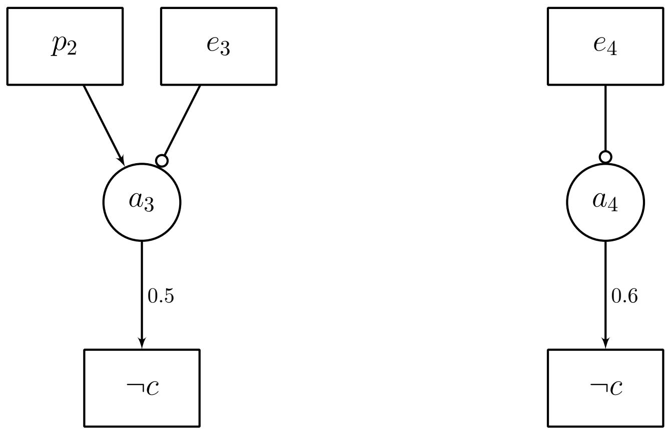


Figure 3.8: Arguments in Carneades

Then we have that argument a_2 is not applicable because $p_3 \notin \mathit{assumptions}$ and p_3 is not acceptable because there is no argument with p_3 as conclusion. Argument a_4 is not applicable because $e_4 \in \mathit{assumptions}$. Argument a_1 and a_3 are applicable.

The conclusion c (of argument a_1) is not acceptable because $\mathit{standard}(c) = \mathit{preponderance}$ and $\mathit{weight}(a_1) \not> \mathit{weight}(a_3)$ while a_3 is an applicable con argument for c. The conclusion $\neg c$ is also not acceptable because $\mathit{standard}(\neg c) = \mathit{clear\text{-}and\text{-}convincing}$ and when considering the argument a_4 it holds for the applicable con argument a_1 that: $\mathit{weight}(a_3) \not> \mathit{weight}(a_1) + \beta$ (although $\mathit{weight}(a_3) > \alpha$).

3.4.2 Dialogical notions

Although the dialogical elements of Carneades are not directly of our interest, to complete the introduction to Carneades, we will also briefly treat the dialogical notions (adapted to our stage-specific definitions). The main component of Carneades is the concept of a dialogue, needed to model the different kinds of burden of proof. For this purpose three different phases are introduced, an *opening*, *argumentation* and *closing* phase. Each phase contains a sequence of stages, intuitively modelling the progression of the dialogue. With each specific stage, Carneades tracks the *arguments* made and the *dialectical status* of the conclusions of these *arguments*.

Parties participating in the dialogue are supposed to give arguments to satisfy the various proof burdens. A dialogue protocol for Carneades can then change the assumptions of an audience in such a way, that neglecting a burden causes the proposition at issue to be assumed at the opposite or at the default value in the rest of the dialogue, thus "failing to convince the audience".

Definition 3.60 (Dialogue (Adapted Def. 2 of [86]))**.** A *dialogue* is a tuple $\langle O, A, C\rangle$, O, A, and C, the *opening*, *argumentation*, and *closing* phases of the dialogue, respectively, are each sequences of *stages*. A stage is a tuple $\langle \mathit{arguments}, \mathit{status}\rangle$, where *arguments* is an acyclic set of arguments and status is a function mapping the conclusions of the arguments in *arguments* to their *dialectical status* in the stage, where the status is a member of $\{\mathit{claimed}, \mathit{questioned}\}$.

Carneades defines various burdens of proof that can be assigned to propositional literals. Each burden is relevant in certain phases of the dialogue. First, in the opening phase of the dialogue, we have the *burden of claiming* and the *burden of questioning*. A proposition that has been claimed in the opening stage, by means of the dialectical status, will be deemed conceded unless it is questioned. Although the *dialectical status* information is not explicitly used in the evaluation of arguments and propositional literals it can be used for the reconstruction of the *assumptions* of an audience. For instance, if a proposition has been questioned in the opening phase of the dialogue and there was no argument produced for it in the argumentation phase, it can be deemed conceded by the audience, defaulting the proposition to its opposite value as an assumption of the audience for the rest of the dialogue.

Definition 3.61 (Burden of claiming and questioning (Def. 12 of [86]))**.** Let $s_1, \ldots, s_n$ be the stages of the opening phase of a dialogue. Let $\langle \mathit{arguments}_n$, $\mathit{status}_n\rangle$ be the last stage, s_n, of the opening phase. A party has met the *burden of claiming* for a proposition p if and only if $\mathit{status}_n(p) \in \{\mathit{claimed}$, $\mathit{questioned}\}$, that is, if and only if $\mathit{status}_n(p)$ is defined. The *burden of questioning* for a proposition p has been met if and only if $\mathit{status}_n(p) = \mathit{questioned}$.

In the argumentation phase of a dialogue, a proposition is acceptable with respect to the burden of production if it satisfies the weakest proof standard, scintilla of evidence. The burden of production is then satisfied for a certain proposition p, if there is at least one applicable pro argument for p.

Definition 3.62 (Burden of production (Def. 13 of [86]))**.** Let $s_1, \ldots, s_n$ be the stages of the argumentation phase of a dialogue. Let $\langle \mathit{arguments}_n, \mathit{status}_n\rangle$ be the last stage, s_n, of the argumentation phase. Let *audience* be the relevant audience for assessing the burden of production, depending on the protocol of the dialogue. Let $C = \langle \mathit{arguments}_n, \mathit{audience}, \mathit{standard}\rangle$ be a

CAES, where *standard* is a function mapping every proposition to the scintilla of proof standard. The *burden of production* for a proposition p has been met if and only if p is acceptable in C.

The final part determining the acceptability of an issue is the *burden of persuasion*; here the burden on the participants of the dialogue is to convince the audience that their arguments in favour of an issue are stronger in a certain sense, i.e. the assigned proof standard, than the arguments against the issue by the other parties.

Definition 3.63 (Burden of persuasion (Def. 14 of [86]))**.** Let $s_1, \ldots, s_n$ be the stages of the closing phase of a dialogue. Let $\langle \mathit{arguments}_n, \mathit{status}_n \rangle$ be the last stage, s_n, of the closing phase. Let *audience* be the relevant audience for assessing the burden of persuasion, depending on the dialogue type and its protocol. Let $C = \langle \mathit{arguments}_n, \mathit{audience}, \mathit{standard} \rangle$ be a CAES, where *standard* is a function mapping every proposition to its applicable proof standard for this type of dialogue. The *burden of persuasion* for a proposition p has been met if and only if p is acceptable in C.

The burden of persuasion is assigned to parties in the final stage of the closing phase. However, to make it possible for parties participating in the dialogue to evaluate the risk of losing an issue while still in the middle of the argumentation phase, they can assess their situation using the *tactical burden of proof*. The tactical burden of proof is assessed by evaluating the burden of persuasion, as if it was the final stage of the closing phase. Because evidence in favour and against an issue accumulates during the dialogue, the tactical burden of proof is the only burden that can shift between parties during the dialogue.

Definition 3.64 (Tactical burden of proof (Def. 15 of [86]))**.** Let $s_1, \ldots, s_n$ be the stages of the argumentation phase of a dialogue. Assume *audience* is the audience which will assess the burden of persuasion in the closing phase. Assume *standard* is the function which will be used in the closing phase to assign a proof standard to each proposition. For each stage s_i in $s_1, \ldots, s_n$, let C_i be the CAES $\langle \mathit{arguments}_n, \mathit{audience}, \mathit{standard} \rangle$. The *tactical burden of proof* for a proposition p is met at stage s_i if and only if p is acceptable in C_i.

The burdens of proof of Carneades are based on the original distinction made between the burden of production, the burden of persuasion and the tactical burden of proof in Prakken and Sartor [150]. Carneades also borrows ideas from their early attempts at modelling these burdens of proof [151, 152]. Finally, for a recent overview and logical analysis of the burdens of proof, that also covers the burdens of claiming and questioning, the reader can confer the work of Prakken [153].

3.4.3 References

The Carneades version described in the previous section is the version in Gordon and Walton [86]. However, the Carneades argumentation model, and its accompanying implementations and tools are continuously evolving. There is an ecosystem of tools: the Carneades argumentation support system [80], consisting of diagramming tools, web-based interfaces [82] and various different implementations [83, 86][28], each providing different functionality.

3.5 ASPIC$^+$ with proof standards/burdens

In Section 3.4 we saw that Carneades is an argumentation model defining burdens and standards of proof. Carneades already has been translated to ASPIC$^+$ [77, 76, 69], so it might look like the translation of these definitions of burdens and standards of proof to ASPIC$^+$ would be sufficient. However, Prakken and Sartor [154] argue that the treatment of the burdens and standard of proof by the Carneades argumentation model is not satisfactory, because the shifts of the burden of persuasion are not handled correctly. Therefore Prakken and Sartor define a more specific version of ASPIC$^+$ by instantiating the ASPIC$^+$ model as described in Section 3.3, together with an adapted definition of defeat to formalise the burden of persuasion more accurately.

3.5.1 Basic definitions

We will first treat definitions which are special cases of the standard ASPIC$^+$ argumentation model.

An argumentation system in ASPIC+ with proof standards/burdens is a special case of Definition 3.30, with instead a language closed under classical negation and a symmetric contrariness relation (classical negation).

Definition 3.65 (Argumentation system (Def 2.1 of [154])). An *argumentation system* is a tuple $AS = \langle \mathcal{L}, ^-, \mathcal{R}, \leqslant \rangle$ where

- $\mathcal{L}$ is a logical language closed under classical negation,
- $^-$ is a symmetric contrariness relation on $\mathcal{L}$ (p and $-p$ are said to be each other's *contradictories*),

[28]See `http://carneades.github.io/` for the most recent version of Carneades

- $\mathcal{R} = \mathcal{R}_s \cup \mathcal{R}_d$ is a set of strict ($\mathcal{R}_s$) and defeasible inference rules ($\mathcal{R}_d$) such that $\mathcal{R}_s \cap \mathcal{R}_d = \emptyset$,
- $\leqslant$ is a partial preorder on $\mathcal{R}_d$.

Knowledge bases are restricted to contain only necessary axioms and ordinary premises.

Definition 3.66 (Knowledge base (Def 2.2 of [154]))**.** A *knowledge base* in an argumentation system $\langle \mathcal{L}, ^-, \mathcal{R}, \leqslant \rangle$ is a pair $\langle \mathcal{K}, \leqslant' \rangle$ where $\mathcal{K} \subseteq \mathcal{L}$ and $\leqslant'$ is a partial preorder on $\mathcal{K}_p$. Here $\mathcal{K}$ is partitioned into two subsets $\mathcal{K}_p$ (ordinary premises) and $\mathcal{K}_a$ (the assumptions).

Arguments and argumentation theories are constructed in a similar way to before, but now instead depend on the above definitions of knowledge base and argumentation system. For ease of reading we restate the definitions.

Definition 3.67 (Arguments (Def 2.3 of [154]))**.** An *argument* A on the basis of a knowledge base $\langle \mathcal{K}, \leqslant' \rangle$ in an argumentation system $\langle \mathcal{L}, ^-, \mathcal{R}, \leqslant \rangle$ is:

1. φ if $\varphi \in \mathcal{K}$ with:
   $Prem(A) = \{\varphi\}$,
   $Conc(A) = \varphi$,
   $Sub(A) = \{\varphi\}$.

2. $A_1, \ldots, A_n \rightarrow \psi$ if $A_1, \ldots, A_n$ are arguments such that there exists a strict rule $Conc(A_1), \ldots, Conc(A_n) \rightarrow \psi$ in $\mathcal{R}_s$,
   $Prem(A) = Prem(A_1) \cup \ldots \cup Prem(A_n)$,
   $Conc(A) = \varphi$. $Sub(A) = Sub(A_1) \cup \ldots \cup Sub(A_n) \cup \{A\}$,

3. $A_1, \ldots, A_n \Rightarrow \psi$ if $A_1, \ldots, A_n$ are arguments such that there exists a defeasible rule $Conc(A_1), \ldots, Conc(A_n) \Rightarrow \psi$ in $\mathcal{R}_d$,
   $Prem(A) = Prem(A_1) \cup \ldots \cup Prem(A_n)$,
   $Conc(A) = \varphi$,
   $Sub(A) = Sub(A_1) \cup \ldots \cup Sub(A_n) \cup \{A\}$.

Definition 3.68 (Argumentation theories (Def. 2.4 of [154]))**.** An *argumentation theory* is a triple $AT = \langle AS, KB, \preceq \rangle$ where AS is an argumentation system, KB is a knowledge base in AS and $\preceq$ is an argument ordering on the set of all arguments that can be constructed from KB in AS.

Since a symmetric contrariness relation is assumed and furthermore issue premises nor necessary axioms are present in the knowledge base, the definitions for attacks and defeats between arguments can be simplified. Furthermore, attacks on ordinary premises are disallowed.

Definition 3.69 (Attacks (Def 2.5 of [154])). Let A and B be two arguments.

- *A undercuts B* (on B') iff $Conc(A) = -B'$ for some $B' \in Sub(B)$ of the form $B''_1, \ldots, B''_n \Rightarrow \psi$.
- *A rebuts B* (on B') iff $Conc(A) = -\varphi$ for some $B' \in Sub(B)$ of the form $B''_1, \ldots, B''_n \Rightarrow \psi$.
- *A undermines B* (on φ) iff $Conc(A) = -\varphi$ for some $\varphi \in Prem(B) \cap \mathcal{K}_a$.

Definition 3.70 (Successful rebuttal and defeat (Def 2.6 of [154])).

- *A successfully rebuts B* if A rebuts B on B' and $A \not\prec B'$.
- *A defeats B* iff A undermines, undercut or successfully rebuts B.

Again an argumentation framework can be built up by using the corresponding definitions of the argument set and defeat relation.

Definition 3.71 (Argumentation framework (Def. 2.7 of [154])). An *abstract argumentation framework (AF) corresponding to an argumentation theory AT* is a pair $\langle Args, defeats \rangle$ such that:

- *$Args$* is the set $\mathcal{A}_{AT}$ as defined by Definition 3.67,
- *$defeats$* is the relation on $Args$ given by Definition 3.70.

3.5.2 Burden of persuasion and standards of proof

We will now focus on the formalisation of the burden of persuasion and the standards of proof. The standards of proof used are those given in the Carneades model (see Section 3.4, Definition 3.57). The standards of proof are formalised in ASPIC$^+$ by adapting the definition of defeat. This is achieved by changing the definition of successful rebuttal to a definition that relies on the weights of the arguments and an assigned proof standard. Handling of (inverted) proof burdens is done by extending the definition of an argumentation theory to contain a set of propositions (from the logical language $\mathcal{L}$) which have an explicitly assigned proof burden. From this set inverted (implicit) proof burdens are to be derived.

Definition 3.72 (General bop-argumentation theories (Adapted Def. 5.1 of [154])). A *general bop argumentation theory* is a tuple $AT = \langle AS, KB, t, B, w, \alpha, \beta, \gamma \rangle$ where AS is an argumentation system, KB is a knowledge base in AS as before, and

- $t \in \mathcal{L}$ (the main topic of the AT)
- $B \subseteq \mathcal{L}$ such that for no φ and $-\varphi$ are in B (we write *dbop*(φ) iff $\varphi \in B$),

- $I \subseteq B$ determines a set of inverted proof burdens (we write *ibop*(φ) iff $\varphi \in I$),
- $w : \mathcal{A}_{AT} \to \mathcal{R}^+ \cup \{0\}$:
- $\alpha, \beta, \gamma \in \mathcal{R}^+ \cup \{0\}$.

Definition 3.72 has been adapted to explicitly contain inverted proof burdens. It is assumed that B was meant to define the default burden of proof *dbop*, instead of *ebop*.

For any $A \in \mathcal{A}_{AT}$ such that $Conc(A)$ has the proof standard beyond reasonable doubt $\mathcal{R}_s$ is assumed to contain rules $\to \neg A$ if $w(A) < \alpha$, and rules of the forum $B_1, \ldots, B_n \to \neg A$ for any $B = B_1, \ldots, B_n \to \neg Conc(A)$ such that $w(B) \geqslant \gamma$. Note that the weight function, w, is assumed to only depend on the content of A, avoiding circularity.

The conditions on which an argument A successfully rebuts an argument B (on B') depends on the assignment of an explicit or implicit proof burden. The original definition in Prakken and Sartor is not strictly correct, e.g. allowing arguments with an inverted proof burden to still satisfy the second or last rule.

Definition 3.73 (Original definition of successful rebuttal under burden of persuasion (Def. 5.2 of [154])). Argument A *successfully rebuts* argument B if A rebuts B on B' and

1. *ibop*$(Conc(A))$ and $w(A) > w(B') + \beta$; or else
2. *dbop*$(Conc(A))$ and $w(A) \nless w(B')$; or else
3. $w(A) + \beta \nless w(B')$.

My corrected definition is the following:

Definition 3.74 (Successful rebuttal under burden of persuasion (Adapted Def. 5.2 of [154])). Argument A *successfully rebuts* argument B if A rebuts B on B' and

1. *ibop*$(Conc(A))$ and $w(A) > w(B') + \beta$; or else
2. $\neg$*ibop*$(Conc(A))$ and *dbop*$(Conc(A))$ and $w(A) \nless w(B')$; or else
3. $\neg$*ibop*$(Conc(A))$ and $\neg$*dbop*$(Conc(A))$ and $w(A) + \beta \nless w(B')$.

3.6 Rationality postulates

The abstract argumentation model by Dung provides a basis for argumentation systems. The structure imposed, a set of arguments and a defeat (attack) relation, gives rise to multiple semantics to evaluate argumentation frameworks. These semantics, extensions of acceptable arguments, can be seen as rationality constraints on how to evaluate argumentation frameworks. Therefore, when additional structure is imposed on arguments, as done for the structured models in Section 3.3, 3.4 and 3.5, more rationality constraints can be imposed on the evaluation. This approach was taken by Caminada and Amgoud [29], who introduced a set of *rationality postulates* for argumentation systems with more structure. Caminada and Amgoud argued that extensions of these systems should satisfy postulates regarding consistency and closure. We repeat here four postulates[29] in the formulation of Prakken [145]:

- **Closure under subarguments:** for every argument in an extension all its subarguments are in the extension.
- **Closure under strict rules:** the set of conclusions of all arguments in an extension is closed under strict-rule application.
- **Direct consistency:** the set of conclusions of all arguments in an extension is consistent.
- **Indirect consistency:** the closure of the set of conclusions of all arguments in an extension under strict-rule application is consistent.

3.7 Literature review of further models of argumentation

This section discusses various abstract and structured argumentation approaches that are relevant to the rest of the thesis. In particular, argumentation approaches that assign weights or preferences to arguments/attacks are reviewed, together with various structured models of argumentation (both are relevant to Chapter 6, 7 and 8).

3.7.1 Logic of Argumentation

The Logic of Argumentation [103] (LA) is an argumentation model based on the established connections between intuitionistic logic, typed lambda calculus and Cartesian closed categories through the Curry-Howard-Lambek correspondence [104, 42, 167]. The Lambek part of the Curry-Howard-Lambek correspondence is the correspondence between Cartesian closed categories (CCCs) from category theory and intuitionistic logic [104].

[29]The results of the other postulates follow directly from these four.

The LA model starts out by defining a rule set for a subset of intuitionistic logic, i.e. minimal logic, a logic containing implication ($\rightarrow$), conjunction ($\wedge$) and falsum ($\perp$). The proof rules are defined as the standard typed lambda calculus inference rules, where the most important difference to standard logical systems is a labelling of propositions (so $x : A$ means x is a proof for the proposition A). Under the standard interpretation, labels would be considered as proofs of propositions, but in LA a label is instead taken to be an argument for a proposition. The defeasibility, in contrast deductivity in standard proofs, is gained by adding the possibility to have open variables in a proof (now to be called an argument). These open variables can then later be given values corresponding to a certainty through a context (like a variable assignment). An argument with no free variables would be equivalent to a standard deductive proof.

Semantics of this system is then given by building on the standard correspondence of Cartesian closed categories (CCCs) to intuitionistic logic [104]. However, to deal with multiple arguments for one proposition, orderings of arguments and aggregations of arguments need to be considered. This is done by enriching the CCC with a join-semilattice, describing how arguments can be taken together. This ordering is then composed with a confidence measures, which maps the ordering to a "dictionary of weights", i.e. a bounded set of values with an operator (such as multiplication) that respects the ordering.

3.7.2 Preference based argumentation frameworks

Preference based argumentation frameworks [4, 5] (PAFs) are an extension of AFs [48] that allow some attacks to fail by adding a preference relation over arguments that determine whether an attack is successful or not.

Definition 3.75 (Preference-based argumentation framework (Def. 1 of [4])). A *preference-based argumentation framework* is a triplet $\langle \mathcal{A}, \mathcal{R}, \mathit{Pref} \rangle$, where $\mathcal{A}$ is a set of arguments, $\mathcal{R}$ is a binary relation representing a defeat relation between arguments, $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{A}$, and *Pref* is a (partial or complete) preordering on $\mathcal{A} \times \mathcal{A}$.

Then, with $>>^{\mathit{Pref}}$ denoting strict preference of arguments, attack can be defined, while keeping the preference ordering in mind.

Definition 3.76 (Attack in PAFs (Def. 2 of [4])). Let A, B be two arguments of $\mathcal{A}$. *B attacks A* iff $B \ \mathcal{R} \ A$ and not ($A >>^{\mathit{Pref}} B$).

Arguments are defined using a classical logic notion of entailment, given a consistent set of premises. The work in Amgoud [5] adds a notion of acceptability defined by two points of view, *individual* and *joint acceptability*, representing whether an argument can defend itself from defeat/an argument set can defend itself from defeat.

Note that, in contrast to the approach taken in ASPIC$^+$ [145], Carneades [83], the argumentation model developed in Chapter 8, and other structured models of argumentation, PAFs do not keep the internal structure of arguments into account when determining the preference ordering. Later work, including [146], has shown that this can lead to counter-intuitive results and violating rationality postulates [29].

PAFs have been generalised by the structured approach in Prakken [145] and the abstract approach in Modgil and Prakken [120]. Both models address the counter-intuitive results of PAFs, respectively, by taking the internal structure in account, and by appropriate generalisation of the preference relation.

3.7.3 Value-based argumentation frameworks

Value-based argumentation frameworks [12] (VAFs) are an extension of AFs [48] that add values to arguments, denoting their strength. Bench-Capon [12] argues that the strength of an argument is based on the social values that it advances. The comparative strengths between arguments, then determine whether an attack succeeds or fails. In VAFs, values are assigned to arguments by an audience (each holding its own social values).

Definition 3.77 (Value-based argumentation framework (Def 5.1 of [12]))**.** A *value-based argumentation framework* is a 5-tuple:

$$\langle AR, \mathit{attacks}, V, \mathit{val}, P\rangle$$

where AR is a finite set of arguments, *attacks* is an irreflexive binary relation on AR, V is a non-empty set of values, *val* is a function $AR \to V$ and P is the set of possible audiences.

A VAF can also be defined w.r.t. a specific audience, called an *audience-specific value-based argumentation framework* (AVAF).

Definition 3.78 (Audience-specific value-based argumentation framework (Def 5.2 of [12]))**.** An *audience-specific value-based argumentation framework* is a 5-tuple:

$$\langle AR, \mathit{attacks}, V, \mathit{val}, \mathit{Valpref}_a\rangle$$

where AR, *attacks*, V, and *val* are as for a VAF, and $\mathit{Valpref}_a$ is an irreflexive, transitive and asymmetric relation on AR, reflecting the value preferences of an audience a.

Defeat between arguments is then determined by the existence of an attack and the values given to the arguments in question.

Definition 3.79 (Defeat for an audience (Def 5.3 of [12]))**.** An argument $A \in AR$ $\mathit{defeats}_a$ an argument $B \in AR$ for audience a iff $(A, B) \in \mathit{attacks}$ and $(\mathit{val}(B), \mathit{val}(A)) \notin \mathit{Valpref}_a$.

Conflict-freeness, acceptability, admissibility and preferred extensions are then defined similarly to Dung [48], using the VAFs notion of defeat.

VAFs suffer from the same modelling problems as PAFs [146], giving unintuitive results and not adhering to rationality postulates.

3.7.4 Defeasible logic programming

Defeasible logic programming [67] (DeLP) combines strict and defeasible rules, based on the family of logics developed by Nute in his Defeasible Logic [130].

The language of defeasible logic programs is built up using facts (ground atoms) and strict/defeasible rules consisting of a head (literal) and body (non-empty set of literals).

Definition 3.80 (Defeasible logic program (Def. 2.4 of [67]))**.** A defeasible logic program P is a possibly infinite set of facts, strict rules and defeasible rules. In a program P, we will distinguish the subset Π of facts and strict rules, and the subset Δ of defeasible rules. When required, we will denote P as (Π, Δ).

Arguments are constructed from defeasible logic programs, by building up defeasible derivations (sequence of facts/rules inferring the conclusion) that are minimal and non-contradictory:

Definition 3.81 (Argument structure (Def. 3.1 of [67]))**.** Let h be a literal, and $P = (\Pi, \Delta)$ a defeasible logic program. We say that $\langle \mathcal{A}, h \rangle$ is an argument structure for h, if $\mathcal{A}$ is a set of defeasible rules of Δ, such that:

1. there exists a defeasible derivation for h from $\Pi \cup \mathcal{A}$,
2. the set $\Pi \cup \mathcal{A}$ is non-contradictory, and
3. $\mathcal{A}$ is minimal: there is no proper subset $\mathcal{A}'$ of $\mathcal{A}$ such that $\mathcal{A}'$ satisfies condition 1. and 2.

A query for a conclusion q is then warranted if there is an argument A that supports q, while the dialectical procedure dealing with defeaters demonstrates that the argument A is undefeated. Conflicting arguments can be compared either with priorities, or using a generalised notion of Poole's specificity [143].

3.7.5 Bipolar argumentation frameworks

Bipolar argumentation frameworks [32, 33] (BAFs) extend AFs [48] to have both an attack and a support relation on arguments.

Definition 3.82 (Bipolar argumentation framework)**.** A *bipolar argumentation framework* is a triple $\langle \mathcal{A}, \mathcal{R}_{att}, \mathcal{R}sup \rangle$[30], where $\mathcal{A}$ is a set of arguments, $\mathcal{R}_{att}$ a binary relation on $\mathcal{A}$ called the attack relation, and $\mathcal{R}_{sup}$ is another binary relation on $\mathcal{A}$ called the support relation.

Support and supported attack are defined by following paths in the graph.

Definition 3.83 (Support and supported attack)**.** Given arguments $A, B \in \mathcal{A}$, A *supports* B if there is a *support sequence* $(A, B_1) \in \mathcal{R}_{sup}, \ldots, (B_n, B) \in \mathcal{R}_{sup}$. There is a *supported attack* for argument B by argument A if there is a support sequence to an argument X and $(X, B) \in \mathcal{R}_{att}$.

The definitions of acceptability, admissibility and conflict-freeness are then generalised to deal with supported attacks. There are multiple semantics for BAFs [32, 33], including a gradual labelling of the argument graph.

BAFs defines support and various constraints on the attack and support relation on the abstract level. This causes problems similar to those of PAFs and VAFs. See [147] and [118] for a discussion of the various issues.

3.7.6 Abstract dialectical frameworks

Abstract dialectical frameworks [23, 21] (ADFs) generalise the attack relation of AFs [48] to a boolean function, capturing support, attack and more complex relationships, including set attacks and mutual exclusion.

Definition 3.84 (Abstract dialectical framework (Def. 1 of [21]))**.** An *abstract dialectical framework* is a tuple $D = \langle S, L, C \rangle$ where

- S is a set of statements (positions, nodes),
- $L \subseteq S \times S$ is a set of links,
- $C = \{C_s\}_{s \in S}$ is a set of total functions $C_s : 2^{par(s)} \to \{\mathbf{t}, \mathbf{f}\}$, one for each statement s. C_s is called an acceptance condition of s.

The initial version of ADFs [23] generalised the grounded, preferred and stable semantics of AFs to ADFs, with the preferred and stable semantics

[30]In [32] the defeat and support relation are assumed to be disjoint. This is not the case in the later version of the framework in [33].

being defined for a restricted class of ADFs, called bipolar ADFs (ADFs restricted to attack/support relations). However, the formalisation had modelling problems [58, 168] for certain examples in these semantics. The later iteration of ADFs [21] defines the grounded, preferred and stable semantics using two and three-valued interpretations similar to Caminada's labellings [26], simultaneously generalising the semantics to arbitrary ADFs and fixing the modelling issues.

ADFs are capable of modelling preferences, in the sense of VAFs [12] and PAFs [4, 5] by encoding them as acceptance conditions [23, 21]. Dynamic preferences are modelled similar to the work by Modgil [120]. It is still an open question whether this form of dynamic preferences gives any modelling problems.

AFs can be directly represented as ADFs[23, 21], with the generalised semantics of ADFs giving the same result for the encoded AFs. ADFs are therefore also a good translation target for structured models that have a notion of support (or more complex interactions).

3.7.7 Weighted argument systems

Weighted argumentation systems [54] add weights to attacks indicating the relative strength.

Definition 3.85 (Weighted argument system (Def. 4 of [54]))**.** A *weighted argument system* is a triple $\langle \mathcal{A}, \mathcal{R}, w \rangle$, where $\langle \mathcal{A}, \mathcal{R} \rangle$ is a Dung AF and $w : \mathcal{A} \to \mathcal{R}_>$ is a function assigning positive, non-zero, real valued weights to attacks.

According to [54] various of the weight and preference based approaches were developed, because of the problems with Dung's standard semantics. In particular, although it is guaranteed that a grounded, preferred and complete extension exist, they can be empty. By recognising that not all arguments have equal strength, weights can be assigned to arguments and used to more finely determine which arguments belong to an extension.

Although, the authors of the weighted argument systems approach [54] argue that arguments weight are a derived (internal) notion, while weights on attacks are a primitive (external) notion, it can also be argued conversely. For example, weights on attacks can be derived from the amount of conflict between arguments based on their internal structure. Indeed, it is argued by Prakken [146] that weighted argument systems suffer from the same problems as PAFs and VAFs.

3.7.8 Pollock's approach to degrees of justification

Pollock's approach to degrees of justification [142, 140] is an argumentation system built on top of his previous work using a recursive inference graph with support and defeat-links [141]. Degrees of justification are measured by real numbers, possibly with top/bottom values ($\infty, -\infty$). One of Pollock's first conclusions when analysing possibilities for assigning and calculating degrees of justification to propositions, is that a probabilistic view is largely untenable, since this would justify belief in a necessary truth before the evidence has even been presented, e.g., Fermat's conjecture before there was a proof.

Instead, Pollock considers the possibility of *diminishers*, counter-arguments that can weaken the degree of justification in the defeasible reasoning process. Conjunction also differs from standard probability theory, taking instead what he calls the *weakest link principle for conjunctions.*

Definition 3.86 (Weakest link principle for conjunctions (Def. 14 of [142]))**.** Given two propositions P and Q:

$$\begin{aligned} \mathit{degree\text{-}of\text{-}justification}(P\&Q) = \mathit{min}\{&\mathit{degree\text{-}of\text{-}justification}(P), \\ &\mathit{degree\text{-}of\text{-}justification}(Q)\} \end{aligned}$$

Pollock's approach to degrees of justification [142] is extended by Liang and Wei [109] to Dung's semantics and beyond, using the ASPIC$^+$ framework. The approach taken in Chapter 8 is in contrast to Pollock's work on degrees of justification. My approach, based on the Logic of Argumentation [103], does not allow for diminishers and does allow for accrual to make arguments stronger. The justification of my approach is that it is not based on standard probability theory, but relies on Dempster-Shafer theory [45, 163, 103, 173] instead.

3.7.9 Besnard and Hunter's classical logic approach to argumentation

Besnard and Hunter [14] define an argumentation system, based on deductive arguments. Their starting position is that the claim of a deductive argument, as well the collection of statements used to support that claim, are denoted by formulae of classical logic. Entailment of the argumentative system as a whole, is then identified as deduction in classical logic.

Assuming a fixed set of classical logic formulae, Δ, and argument can be defined as following.

Definition 3.87 (Argument (Def 3.2.1 of [14]))**.** An *argument* is a pair $\langle \Phi, \alpha \rangle$ such that:

1. $\Phi \nvdash \bot$.
2. $\Phi \vdash \alpha$.
3. Φ is a minimal subset of Δ satisfying 2.

Conflict between deductive arguments can occur through undercutting (undermining in Prakken [145]) and rebutting of other arguments, respectively corresponding to an attack on the premises and conclusion of an other argument.

Besnard and Hunters approach has been shown to be representable in ASPIC$^+$ through the work of Modgil and Prakken [122].

3.7.10 Hunter's probabilistic approach

Hunter [97] develops various approaches and foundations to assigning probabilities to arguments and the attack relation, phrased in the classical logic approach to argumentation [14].

Probabilities over arguments are defined as for probabilistic argumentation frameworks [108] (PAFs), that is, they represent the probability that an argument is believed to hold. The probability of arguments is then used to generate a probability distribution over the sub-graphs that can be generated from a PAF. From these sub-graphs it can be determined which sets of arguments hold in a given extension [48].

Hunter [97] extends the approach of PAFs to probability distributions over models of classical logical languages, extending the work of Paris [133].

Definition 3.88 (Belief function (Def. 7 of [97]))**.** Let $\mathcal{L}$ be a propositional language, and let $\Phi \subseteq \mathcal{L}$. A *belief function* on Φ is a function $P : \mathcal{L} \to [0..1]$. If P is a belief function on $\mathcal{L}$ (i.e. Φ is $\mathcal{L}$), then P is *complete*, otherwise P is *incomplete.*

Definition 3.89 (Probability function (Def. 8 of [97]))**.** Let $\mathcal{L}$ be a propositional language, and let $\mathcal{M}^{\mathcal{L}}$ be the models of the language $\mathcal{L}$. A belief function P on $\mathcal{L}$ is a *probability function* on $\mathcal{L}$ iff for each $\varphi \in \mathcal{L}$:

$$P(\varphi) = \sum_{m \in Models(\varphi)} P(m)$$

Various reasonable properties of probability functions and probabilistic argument subgraphs are discussed. The properties apply to abstract arguments, probability distributions over arguments and models of classical logic, making the approach orthogonal to my work in Chapter 8.

The work by Hunter and Thimm [98] builds on the work by Hunter [97] by adding epistemic extensions and various constraints on probability assignments.

3.7.11 Further work

There are various other approaches to abstract and structured argumentation. Below, I summarise a selection of further models in computational argumentation theory:

- Pollock [136, 137, 141] developed a theory on argument inference schemes and recursive evaluation of argument graphs;
- Argument systems [110] provide a unifying framework for non-monotonic logics, reformulating any logic as a system for constructing arguments;
- Simari and Loui [165] present a defeasible system combines Pollock's style of arguments [137] with Poole's notion of specificity [143];
- Abstract argumentation systems [184, 185] captures arguments combining defeasible and strict rules, ordering the arguments in strength;
- Prakken and Sartor [148, 149] developed an argumentation system that extends the default logic language and defines priorities that are derived as defeasible conclusions of arguments;
- Assumption-based argumentation [17] defines arguments as backward deductions, supported by sets of assumptions;
- Constrained argumentation systems [6] are systems for practical reasoning that combine deliberation and means-ends reasoning in one step;
- Argumentation context systems [20] is a framework allowing group argumentation;
- Extended argumentation frameworks [120] extend AFs and ASPIC$^+$ with attacks on attacks;
- Argumentation frameworks with recursive attacks [9] adds recursive attacks to AFs;
- Fibring argumentation frames and (probabilistic) argumentation networks [64, 65, 66] extend AFs to disjunctive attacks, joint attacks, equations on AFs and various other concepts;

- Temporal argumentation frameworks [25] considers temporal availability of arguments in an AF;
- Probabilistic argumentation frameworks [108] associate probabilities both to the attack and arguments relation, inducing derived AFs with a certain likelihood;

For an in depth review of some of the argumentation models, see Prakken and Vreeswijk [155], Chesñevar et al.[38] and the Argumentation in Artificial Intelligence book [156].

Part II

General framework and use cases

Chapter 4

A reference implementation of Dung's argumentation frameworks in Haskell

Dung's abstract argumentation frameworks and most other abstract approaches to argumentation are closely aligned with logic and/or answer set programming. Their mathematical specifications can thus be captured almost directly in program code [35].

In the following, I argue that functional programming, with Haskell in particular, is a suitable paradigm for capturing abstract argumentation models by implementing Dung's abstract argumentation frameworks and various of its semantics and algorithms. The chapter serves both as a use case, demonstrating that Haskell and functional programming languages are indeed a good candidate for implementing abstract argumentation frameworks, and secondly providing us with a complete and modular implementation that can immediately be used and re-used as a translation target. Chapter 7 does exactly that by using the implementation in this chapter for the translation of Carneades into Dung (see Chapter 6).

This chapter is organised as follows. Section 4.1 illustrates the functional programming approach by giving some of the standard definitions of Dung's AFs [48] and demonstrating how to implement the four standard semantics and semi-stable semantics [27] into Haskell. Section 4.2 discusses the labelling approach as put forward by Pollock [139] and extended by Caminada [26], including a labelling algorithm for the *grounded semantics* [119] and semi-stable semantics [28], along with accompanying definitions. Section 4.3 presents an application that provides a parser, an output module compatible with other mainstream implementations, a command-line interface, an application programming interface (API) and documentation to the previously discussed implementation. Section 4.4 discusses related implementations, particularly those from the ICCMA competition [175]. Section 4.5 concludes.

4.1 Basic definitions

An abstract argumentation framework consists of a set of arguments and a binary relation on this set representing *defeat*: the notion of one argument conflicting with another. To keep the framework completely general, the notion of argument is abstract; i.e., no assumptions are made as to their nature.

Definition 4.1 (Abstract argumentation framework (Adapted Def. 2 of [48])). An *abstract argumentation framework* is a tuple $\langle \mathit{Args}, \mathit{Def} \rangle$, where Args is a set of arguments and $\mathit{Def} \subseteq \mathit{Args} \times \mathit{Args}$ is a relation on Args representing defeat. An argument A is said to *defeat* an argument B iff $(A, B) \in \mathit{Def}$.

Example 4.2. Consider $AF_1 = \langle \{A, B, C\}, \{(A, B), (B, C)\} \rangle$.

$$\begin{tikzcd} A \arrow[r] & B \arrow[r] & C \end{tikzcd}$$

Figure 4.1: An (abstract) argumentation framework

The Haskell counterpart of this definition takes the form of an *algebraic data type*:

```
data DungAF arg = AF [arg] [(arg, arg)]
```

Note how this essentially is a transliteration of the mathematical definition, even if lists are used in place of sets. Additionally, the definition is parametrised on the type of argument, *arg*. Initially, arguments will be represented by strings for simplicity. However, in Section 7, the utility of this parametrisation will become apparent when arguments are represented by propositions or complete proof trees from a different (structured) model such as Carneades (see Section 3.4).

```
type AbsArg = String
a, b, c :: AbsArg
a = "A"; b = "B"; c = "C"
AF1 :: DungAF AbsArg
AF1 = AF [a, b, c] [(a, b), (b, c)]
```

Below the definitions needed before defining a semantics for an argumentation framework, are given. For an in-depth treatment of these definitions and further discussion of Dung's work, including examples, see Section 3.2.

Definition 4.3. Let $AF = \langle Args, Def \rangle$ and $S \subseteq Args$.

1. S (set-)defeats an argument $A \in Args$ iff there exists a $B \in S$ such that $(B, A) \in Def$.
2. S is called *conflict-free* iff $\neg\exists A, B \in S$ such that $(A, B) \in Def$.
3. An argument $A \in Args$ is *acceptable* w.r.t. S iff $\forall B \in Args$, if $(B, A) \in Def$ then $\exists C \in S$ such that $(C, B) \in Def$.
4. The *characteristic function* of an AF, F_{AF} is a function such that:
   - $F_{AF} : 2^{Args} \to 2^{Args}$,
   - $F_{AF}(S) = \{A \mid A$ is acceptable w.r.t. to $S\}$.
5. A conflict-free set of arguments S is *admissible* iff every argument $A \in S$ is acceptable w.r.t. S, i.e. $S \subseteq F_{AF}(S)$.

The Haskell implementation is straightforward and closely follows the mathematical definitions.

```
setDefeats :: Eq arg ⇒ DungAF arg → [arg] →
              arg → Bool
setDefeats (AF _ def) args arg
  = or [y ≡ arg | (x, y) ← def, x ∈ args]
conflictFree :: Eq arg ⇒ DungAF arg → [arg] → Bool
conflictFree (AF _ def) args
  = null [(x, y) | (x, y) ← def, x ∈ args, y ∈ args]
acceptable :: Eq arg ⇒ DungAF arg → arg →
              [arg] → Bool
acceptable af@(AF _ def) x args
  = and [setDefeats af args y | (y, x') ← def, x ≡ x']
f :: Eq arg ⇒ DungAF arg → [arg] → [arg]
f af@(AF args' _) args
  = [x | x ← args', acceptable af x args]
f_AF1 :: [AbsArg] → [AbsArg]
f_AF1 = f AF1
admissible :: Ord arg ⇒ DungAF arg → [arg] → Bool
```

$$\begin{aligned} \mathit{admissible}\ \mathit{af}\ \mathit{args} = {} & \mathit{conflictFree}\ \mathit{af}\ \mathit{args} \wedge \\ & \mathit{args} \subseteq f\ \mathit{af}\ \mathit{args} \end{aligned}$$

Then, for the argumentation framework of Figure 4.1 (see also Section 3.2.1), the expected output holds for the Haskell implementation:

$\mathit{setAttacks}\ AF_1\ [a, b]\ c$
$> \mathit{True}$
$\mathit{conflictFree}\ AF_1\ [a, c]$
$> \mathit{True}$
$\mathit{acceptable}\ AF_1\ c\ [a, b, c]$
$> \mathit{True}$
$f_{AF_1}\ [a]$
$>$ ["A", "C"]
$\mathit{admissible}\ AF_1\ [a]$
$> \mathit{False}$

4.1.1 Standard semantics

In this subsection an implementation of the four standard semantics for Dung's argumentation frameworks (grounded, complete, preferred and stable semantics) will be discussed. The implementation is based on the definitions using the characteristic function and therefore closely follows the original mathematical definitions of Dung. Section 4.2 discusses implementations of the labelling approach to the four standard and semi-stable semantics.

Definition 4.4 (Extensions (Def. 7, Def. 13, Def. 20, Def. 23, Lemma 24 and Theorem 25 of [48])). Given an argumentation framework AF and a conflict-free set S of arguments, $S \subseteq \mathit{Args}$, then, with the ordering determined by set inclusion, S is a:

- *grounded extension* iff S is the least fixed point of F_{AF}.
- *complete extension* iff $S = F_{AF}(S)$; i.e., S is a fixed point of F_{AF}.
- *preferred extension* iff S is a maximal complete extension.
- *stable extension* iff it is a preferred extension defeating all arguments in $\mathit{Args} \backslash S$.

The grounded extension can be calculated by iterating F_{AF} over the empty set, the least element in the domain, until a fixed point is reached. The calculation of the least fixed point is guaranteed to succeed (given that we apply a finite set of arguments), due to the previously stated properties of F_{AF}.

```
groundedF :: Eq arg ⇒ ([arg] → [arg]) → [arg]
groundedF f = groundedF′ f []
  where groundedF′ f args
          | f args ≡ args = args
          | otherwise     = groundedF′ f (f args)
```

Then as expected:

```
groundedF f_AF1
 > ["A","C"]
```

The further semantics of Dung's argumentation frameworks can be computed naively by constructing the powerset of the arguments in a given argumentation framework and then defining an appropriate filter.

The powerset can be constructed recursively in Haskell:

```
powerset :: [a] → [[a]]
powerset []       = [[]]
powerset (x : xs) = powerset xs ++ map (x:) (powerset xs)
```

Given an argumentation framework, we can compute all complete extension, by taking all sets of arguments of the powerset of arguments of that AF, given that they are conflict-free and $f\ af \equiv f$, for all arguments A in that set.

```
completeF :: Ord arg ⇒ DungAF arg → [[arg]]
completeF af@(AF args _) =
  let fAF = f af
  in filter (λa → conflictFree af a ∧ a ≡ fAF a)
            (powerset args)
```

A complete extension in a given argumentation framework is also a preferred extension if it is not a subset of one of the other complete extensions.

```
isPreferredExt :: Ord arg ⇒ DungAF arg → [[arg]] → [arg] → Bool
isPreferredExt af exts ext = all (¬ ∘ (ext ⊆))
                                 (delete ext exts)
```

A preferred extension can then be computed by taking the set of complete extension and applying the above defined filter.

```
preferredF :: Ord arg => DungAF arg -> [[arg]]
preferredF af@(AF args _) =
  let cs = completeF af
  in filter (isPreferredExt af cs) cs
```

A set of arguments S is a stable extension iff it is equal to the set of arguments not defeated by S. Given a list of arguments that are not in an extension, an argument *arg* is *undefeated* if the list of its defeaters, ignoring the arguments outside the extension, is empty.

```
isStableExt :: Ord arg => DungAF arg -> [arg] -> Bool
isStableExt af@(AF args _) ext = filter (undefeated (args \\ ext) af)
                                        args == ext
undefeated :: Ord arg => [arg] ->
               DungAF arg -> arg -> Bool
undefeated outs (AF _ def) arg =
  let defeaters = [a | (a, b) <- def, arg == b]
  in null (defeaters \\ outs)
```

Similarly, we can then define stable extension by giving the appropriate filter over preferred extensions.

```
stableF :: Ord arg => DungAF arg -> [[arg]]
stableF af@(AF args _) =
  let ps = preferredF af
  in filter (isStableExt af) ps
```

Semi-stable extensions are defined by taking a complete extension and not just maximizing the set of arguments, as is done for preferred extensions, but also maximizing the defeated arguments, $Args^+$.

Definition 4.5 (Semi-stable extension (Def. 3 of [28]))**.** Given an argumentation framework AF and a conflict-free set S of arguments, $S \subseteq Args$, then, with the ordering determined by set inclusion, S is a *semi-stable extension* iff it is a complete extension where $Args \cup Args^+$ is maximal.

The Haskell code corresponding to the semi-stable extension is again a filter over complete extensions, with the implementation of *isSemiStableExt* being based on *isPreferredExt*. Instead, we now check whether the possible extension together with its defeated arguments is itself a subset of $Args \cup$

$Args^+$ of any complete extensions. The implementations of *argplus* is given at the start of Section 4.2.1.

```
isSemiStableExt :: Ord arg ⇒ DungAF arg → [[arg]] → [arg] → Bool
isSemiStableExt af exts ext =
  let extPlus  = argplus af ext
      extsPlus = map (argplus af) exts
  in  all (¬ ∘ (extPlus ⊆))
      (delete extPlus extsPlus)
semiStableF :: Ord arg ⇒ DungAF arg → [[arg]]
semiStableF af@(AF args _) =
  let cs = completeF af
  in filter (isSemiStableExt af cs) cs
```

Example 4.6 (Calculating extensions)**.** Given the following argumentation framework, $AF = \langle Args, Def \rangle$ with $Args = \{A, B, C, D, E, F, G\}$ and $Def = \{(A, B), (C, B), (C, D), (D, C), (D, E), (E, G), (F, E), (G, F)\}$, depicted in Figure 3.2.

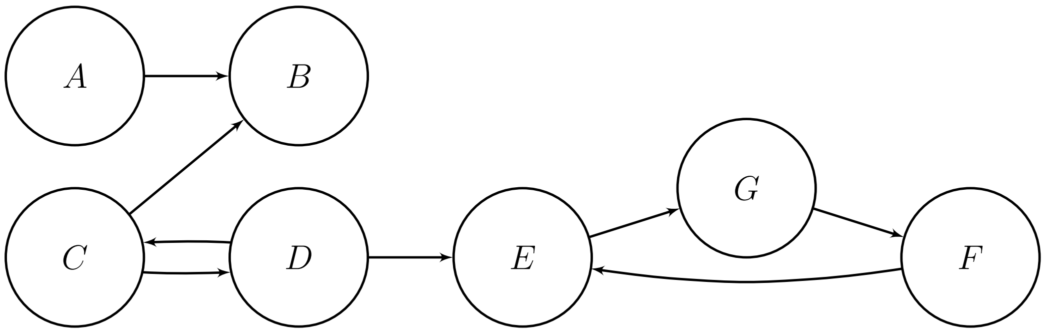


Figure 4.2: A more complex argumentation framework

```
AF₂ = AF [a, b, c, d, e, f2, g]
         [(a, b), (c, b), (c, d), (d, c), (d, e),
          (e, g), (f2, e), (g, f2)]
```

Then:

```
completeF AF₂
 > [["A"], ["A", "D", "G"], ["A", "C"]]
preferredF AF₂
 > [["A", "D", "G"], ["A", "C"]]
```

```
stableF AF₂
> [["A","D","G"]]
semiStableF AF₂
> [["A","D","G"],["A","C"]]
```

4.2 Labelling

This section implements the labelling algorithms and definitions as given in Caminada [28]. The purpose of this section is to demonstrate that more complex algorithms and definitions can be translated into Haskell. The labelling based approach is also more commonly used in actual implementations (see Section 4.4). Opting for this thus facilitates comparisons. Furthermore, by using the grounded labelling we obviate the need to formalise fixed points, significantly reducing the amount of work needed when implementing everything in a theorem prover (see Chapter 7). It will also be shown that the direct implementation of the various definitions can immediately be exploited in the definition of the actual algorithms, providing a faithful representation.

4.2.1 Basic labelling definitions

A^+ and A^- respectively are the set of arguments defeated by A and the set of arguments that defeat A. Similarly, for a set of arguments S, S^+ and S^- denote the set of arguments defeated by at least one argument in S and the set of arguments that defeat at least one argument in S.

Definition 4.7 (Sets of defeating and defeated arguments (Def. 2 of [28]))**.** Let $AF = \langle Args, Def \rangle$ be an argumentation framework, $A \in Args$ and $S \subseteq Args$. Then:

- A^+ and S^+ are respectively defined as $\{B \mid (A, B) \in Def\}$ and $\{B \mid (A, B) \in Def, A \in S\}$,
- A^- and S^- are respectively defined as $\{B \mid (A, B) \in Def\}$ and $\{B \mid (A, B) \in Def, A \in S\}$.

```
aplus :: Eq arg ⇒ DungAF arg → arg → [arg]
aplus (AF args def) a = [b | (a′, b) ← def, a ≡ a′]
amin :: Eq arg ⇒ DungAF arg → arg → [arg]
amin (AF args def) a = [b | (b, a′) ← def, a ≡ a′]
```

```
argplus :: Ord arg ⇒ DungAF arg → [arg] → [arg]
argplus af = nub ∘ concatMap (aplus af)

argmin :: Ord arg ⇒ DungAF arg → [arg] → [arg]
argmin af = nub ∘ concatMap (amin af)
```

A labelling of an argumentation framework is an assignment of exactly one label from $\{In, Out, Undec\}$ to each argument, where *In* means the argument is justified, *Out* means overruled, and *Undec* means status as yet undecided.

Definition 4.8 (Labelling (Def. 6.1 of [119]))**.** Let $\langle Args, Def\rangle$, be an argumentation framework.

- A labelling is a total function from arguments into labels, $\mathcal{L} : Args \rightarrow \{In, Out, Undec\}$.
- $\text{in}(\mathcal{L}) = \{x \mid \mathcal{L}(x) = In\}$; $\text{out}(\mathcal{L}) = \{x \mid \mathcal{L}(x) = Out\}$; $\text{undec}(\mathcal{L}) = \{x \mid \mathcal{L}(x) = Undec\}$.

A labelling $\mathcal{L}$ may then be presented as a triple of the form $\langle \text{in}(\mathcal{L}), \text{out}(\mathcal{L}), \text{undec}(\mathcal{L})\rangle$.

Labellings are represented in the implementation by a list of pairs, i.e. arguments paired with their status. This representation enables us to compare labellings for equality.

```
data Status = In | Out | Undecided
  deriving (Eq, Show, Ord)

type Labelling arg = [(arg, Status)]

inLab :: Labelling arg → [arg]
inLab labs = [a | (a, In) ← labs]
```

The definitions for $\text{out}(\mathcal{L})$ and $\text{undec}(\mathcal{L})$ (*outLab* and *undecLab*) are defined similarly to *inLab*.

Definition 4.9 (Basic labellings (Def. 4 of [28]))**.** Let $\langle Args, Def\rangle$, be an argumentation framework.

- The *all-in labelling* is a labelling that labels every argument $A \in Args$ as In.
- The *all-out labelling* is a labelling that labels every argument $A \in Args$ as Out.
- The *all-undec labelling* is a labelling that labels every argument $A \in Args$ as Undec.

allIn is given below. *allOut* and *allUndec* are defined similarly.

```
allIn :: [arg] → Labelling arg
allIn = map (λa → (a, In))
```

4.2.2 Grounded labelling

In this subsection an algorithm for computing the grounded labelling will be discussed. The algorithm in Algorithm 4.10 is commonly used for computing the grounded labelling [119]. It takes a set of arguments, initially making every argument unlabelled. The algorithm iteratively assigns out or in to any unlabelled arguments until the labellings do not change any longer. Any unlabelled arguments are then assigned undec.

However, our version is clarified slightly: the set of unlabelled arguments explicit as the set of undecided arguments, undec. The variable i refers the ith step of calculating the labelling; its incrementation is handled implicitly.

Algorithm 4.10. Algorithm for grounded labelling (Algorithm 6.1 of [119])

```
1. $\mathcal{L}_0 := \langle \emptyset, \emptyset, Args \rangle$
2. **repeat**
3. $\text{in}(\mathcal{L}_{i+1}) := \text{in}(\mathcal{L}_i) \cup \{x \mid x \in \text{undec}(\mathcal{L}_i),$
   $\forall y : \text{if } (y,x) \in Def \text{ then } y \in \text{out}(\mathcal{L}_i)\}$
4. $\text{out}(\mathcal{L}_{i+1}) := \text{out}(\mathcal{L}_i) \cup \{x \mid x \in \text{undec}(\mathcal{L}_i),$
   $\exists y : (y,x) \in Def \text{ and } y \in \text{in}(\mathcal{L}_{i+1})\}$
5. $\text{undec}(\mathcal{L}_{i+1}) := \text{undec}(\mathcal{L}_i) - \{x \mid x \text{ is newly labelled in or out}\}$
6. **until** $\mathcal{L}_{i+1} = \mathcal{L}_i$
7. $\mathcal{L}_{\mathcal{G}} := \langle \text{in}(\mathcal{L}_i), \text{out}(\mathcal{L}_i), \text{undec}(\mathcal{L}_i) \rangle$
```

Implementing this algorithm in Haskell is straightforward. This is illustrated by showing the translation of the two conditions for x containing quantifiers in line 3 and 4.

```
  -- if all defeaters are Out
undefeated :: Eq arg ⇒ [arg] →
                DungAF arg → arg → Bool
undefeated outs (AF _ def) arg =
  let defeaters = [x | (x, y) ← def, arg ≡ y]
  in null (defeaters \\ outs)
  -- if there exists a defeater that is In
```

```
defeated :: Eq arg ⇒ [arg] →
            DungAF arg → arg → Bool
defeated ins (AF _ def) arg =
  let defeaters = [x | (x, y) ← def, arg ≡ y]
  in ¬ (null (defeaters `intersect` ins))
```

The implementation as such consists of two parts: a main function and a helper function implementing the actual algorithm having three additional accumulation arguments for keeping track of the labelling. All arguments are initially labelled *Undecided*.

```
grounded :: Eq arg ⇒ DungAF arg → [(arg, Status)]
grounded af@(AF args _) = grounded' [] [] args af

grounded' :: Eq a ⇒ [a] → [a] →
             [a] → DungAF a → [(a, Status)]
grounded' ins outs [] _
  =     map (λx → (x, In)) ins
    ++  map (λx → (x, Out)) outs
grounded' ins outs undec af =
  let newIns  = filter (undefeated outs af) undec
      newOuts = filter (defeated ins af)    undec
  in if null (newIns ++ newOuts)
     then map (λx → (x, In)) ins
       ++ map (λx → (x, Out)) outs
       ++ map (λx → (x, Undecided)) undec
     else grounded' (ins ++ newIns)
                    (outs ++ newOuts)
                    (undec \\ (newIns ++ newOuts))
                    af
```

Note how closely the structure of this implementation is aligned with the specification.

4.2.3 Complete, preferred and stable labellings

This section discusses the definitions of complete, preferred and stable labellings, computing the labellings naively by using a powerset construction and subsequently applying a filter. Section 4.2.4 presents a more sophisticated approach to computing labels, covering an implementation of an adapted version of Caminada's algorithm [28] for computing semi-stable labellings.

The *powerset* of labellings can be constructed by mapping all three possible labellings of an argument in front of the lists of the recursive call.

```
powerLabel :: [arg] → [Labelling arg]
powerLabel []       = [[]]
powerLabel (x : xs) =  map ((x, In):)        (powerLabel xs)
                    ++ map ((x, Out):)       (powerLabel xs)
                    ++ map ((x, Undecided):) (powerLabel xs)
```

Caminada defines the concept of *illegally* and *legally labelled* arguments to determine whether an argument should or should not be part of a labelling.

Definition 4.11 (Illegally labelled arguments (Def. 5 of [28]))**.** Given an argumentation framework $\langle Args, Def \rangle$, an argument $A \in Args$ and a labelling $\mathcal{L}$ over $Args$, we have that:

1. A is *illegally* in iff A is labelled in
   but not all its defeaters are labelled out;
2. A is *illegally* out iff A is labelled out
   but does not have a defeater labelled in;
3. A is *illegally* undec iff A is labelled undec
   but either all of its defeaters are labelled out or it has a defeater that is labelled in.

A labelling has *no* illegal arguments iff there is no argument that is illegally in, illegally out or illegally undec.

Definition 4.12 (Legally labelled arguments (Def. 5 of [28]))**.** Given an argumentation framework $\langle Args, Def \rangle$, an argument $A \in Args$ and a labelling $\mathcal{L}$ over $Args$, we have that:

1. A is *legally* in iff A is labelled in and it is not illegally in;
2. A is *legally* out iff A is labelled out and it is not illegally out;
3. A is *legally* undec iff A is labelled undec and it is not illegally undec.

The Haskell implementation has two pattern matching cases for each of the illegal labelling concepts, one corresponding to the condition defined following the "but" and a second part that returns *False* if it is not the correct label (an Out labelled argument can not be illegally In). The functions rely on a helper function, *labAttackers*. *labAttackers* takes an argumentation framework, an argument from that AF and a labelling, and returns the labelled defeaters of those arguments.

```
labAttackers :: Eq arg ⇒ DungAF arg → arg →
                Labelling arg → Labelling arg
```

```
labAttackers (AF args def) a labs =
  [lab | lab@(b, _) ← labs, (b, a) ∈ def]
```

```
illegallyIn :: Eq arg ⇒ DungAF arg → Labelling arg →
               (arg, Status) → Bool
illegallyIn af labs (a, In) =
  ¬ ∘ null $ [lab | lab@(_, l) ← labAttackers af a labs, l ≢ Out]
illegallyIn _ _ _          = False
```

```
illegallyOut :: Eq arg ⇒ DungAF arg → Labelling arg →
                (arg, Status) → Bool
illegallyOut af labs (a, Out) =
  null [lab | lab@(_, In) ← labAttackers af a labs]
illegallyOut _  _    _        = False
```

```
illegallyUndec :: Eq arg ⇒ DungAF arg → Labelling arg →
                  (arg, Status) → Bool
illegallyUndec af labs (a, Undecided) =
  and [l ≡ Out | (_, l) ← labAttackers af a labs]
       ∨ (¬ ∘ null) [lab | lab@(_, In) ← labAttackers af a labs]
illegallyUndec _  _    _              = False
```

Legal labellings are defined in terms of pattern matching and the negation of the corresponding illegal labelling functions.

```
legallyIn :: Eq arg ⇒ DungAF arg → Labelling arg →
             (arg, Status) → Bool
legallyIn af labs arg@(_, In) = ¬ $ illegallyIn af labs arg
legallyIn _  _    _           = False
```

legallyOut and *legallyUndec* are defined similarly.

Given the definition of illegal arguments, we can define admissible labellings and the labellings corresponding to the previously defined semantics.

Definition 4.13 (Admissible labellings (Def. 6 of [28]))**.** An *admissible labelling* is a labelling with no arguments that are illegally in or illegally out.

The implementation of an admissible labelling is a check whether the concatenation of the arguments that are *illegallyIn* and *illegallyOut* is empty.

```
isAdmissible :: Eq arg ⇒ DungAF arg → Labelling arg → Bool
isAdmissible af labs = null $
```

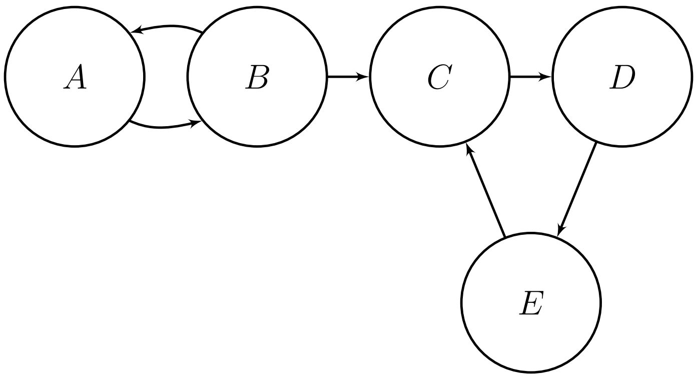


Figure 4.3: An argumentation framework with a stable extension

$$\begin{array}{l} [\mathit{lab} \mid \mathit{lab}@(a, \mathit{In}) \leftarrow \mathit{labs}, \mathit{illegallyIn}\ \mathit{af}\ \mathit{labs}\ \mathit{lab}] \\ +\!\!+ [\mathit{lab} \mid \mathit{lab}@(a, \mathit{Out}) \leftarrow \mathit{labs}, \mathit{illegallyOut}\ \mathit{af}\ \mathit{labs}\ \mathit{lab}] \end{array}$$

Figure 4.3 is the argumentation framework from the example in Section 3.2.4. It will be shown that the implementations of the labelling definitions, discussed below, produce the expected results w.r.t. this example.

The implementation of *isAdmissible* captures the definition of what it means to be an admissible labelling, but also computes whether a labelling is admissible[31]. The *isAdmissible* function can thus immediately be applied to calculate the admissible labellings of an argumentation framework, by first constructing the powerset of labellings, using *powerLabel*, and subsequently filtering out the labellings that are admissible.

```
> filter (isAdmissible AF₃) (powerLabel [a, b, c, d, e])
[[("A", In), ("B", Out), ("C", Undec), ("D", Undec), ("E", Undec)],
 [("A", Out), ("B", In), ("C", Out), ("D", In), ("E", Out)],
 [("A", Out), ("B", In), ("C", Out), ("D", In), ("E", Undec)],
 [("A", Out), ("B", In), ("C", Out), ("D", Undec), ("E", Undec)],
 [("A", Out), ("B", In), ("C", Undec), ("D", Undec), ("E", Undec)],
 [("A", Undec), ("B", Undec), ("C", Undec), ("D", Undec), ("E", Undec)]]
```

The complete labelling is a strengthening of the admissible labelling, additionally requiring that no arguments are illegally undec.

Definition 4.14 (Complete labellings (Def. 7 of [28])). A *complete labelling* is a labelling with no illegal arguments.

[31]This is an application of the Curry-Howard correspondence, see also Chapter 7.

isComplete follows the definition of no illegal arguments, that is, the concatenation of the list of arguments that are illegally in, illegally out and illegally undec should be empty.

```
isComplete :: Eq arg ⇒ DungAF arg → Labelling arg → Bool
isComplete af labs = null $
        [lab | lab@(a, In) ← labs, illegallyIn af labs lab]
    ++ [lab | lab@(a, Out) ← labs, illegallyOut af labs lab]
    ++ [lab | lab@(a, Undecided) ← labs, illegallyUndec af labs lab]
```

Similar to filtering admissible labelling, we can define the complete labellings by filtering the powerset of labelling using the above defined *isComplete* function.

```
completes :: Ord arg ⇒ DungAF arg → [Labelling arg]
completes af@(AF args _) = filter (isComplete af) (powerLabel args)
```

Then, as expected:

```
> completes AF3
[[("A", In), ("B", Out), ("C", Undec), ("D", Undec), ("E", Undec)],
 [("A", Out), ("B", In), ("C", Out), ("D", In), ("E", Out)],
 [("A", Undec), ("B", Undec), ("C", Undec), ("D", Undec), ("E", Undec)]]
```

The other semantics are defined in terms of the complete labelling:

Definition 4.15 (Further labellings (Def. 8 of [28])). Given an argumentation framework $\langle Args, Def\rangle$ and a complete labelling $\mathcal{L}$ over *Args*, with the ordering determined by set inclusion, we define $\mathcal{L}$ to be a

- *grounded labelling* iff in($\mathcal{L}$) is minimal;
- *preferred labelling* iff in($\mathcal{L}$) is maximal;
- *stable labelling* iff undec($\mathcal{L}$) $= \emptyset$;
- *semi-stable labelling* iff undec($\mathcal{L}$) is minimal.

We can restructure the definition of the grounded labelling to make it clear how it could be implemented.

Definition 4.16 (Grounded labelling (2)). Let $AF = \langle Args, Def\rangle$ be an argumentation framework and $\mathcal{L}$ a labelling over *Args*. A labelling $\mathcal{L}$ is grounded, if it is complete, and for all complete labellings $\mathcal{L}_2$ of AF, it is the case that in($\mathcal{L}$) is a subset in($\mathcal{L}_2$).

```
isGrounded :: Ord arg ⇒ DungAF arg → [Labelling arg] →
                Labelling arg → Bool
isGrounded af labss labs = isComplete af labs ∧
                                all (inLab labs ⊆)
                                   (map inLab labss)
groundedL :: Ord arg ⇒ DungAF arg → Labelling arg
groundedL af@(AF args _) = head $
  filter (isGrounded af (completes af))
        (powerLabel args)
```

The grounded labelling, *groundedL*, gives back the smallest complete labelling:

```
> groundedL AF₃
[("A", Undec), ("B", Undec), ("C", Undec), ("D", Undec), ("E", Undec)]
```

The other semantics are defined in a similar manner (preferreds, stables, and semiStables are omitted):

```
isPreferred :: Ord arg ⇒ DungAF arg → [Labelling arg] →
                Labelling arg → Bool
isPreferred af labss labs = isComplete af labs ∧
                                all (¬ ∘ (inLab labs ⊆))
                                   (map inLab (delete labs labss))
isStable :: Eq arg ⇒ DungAF arg → [Labelling arg] →
            Labelling arg → Bool
isStable af labss labs = isComplete af labs ∧
                            null (undecLab labs)
isSemiStable :: Ord arg ⇒ DungAF arg → [Labelling arg] →
                  Labelling arg → Bool
isSemiStable af labss labs = isComplete af labs ∧
                                all (undecLab labs ⊆)
                                   (map undecLab labss)
```

```
> preferreds AF₃
[[("A", In), ("B", Out), ("C", Undec), ("D", Undec), ("E", Undec)],
 [("A", Undec), ("B", Undec), ("C", Undec), ("D", Undec),
  ("E", Undec)]]
> stables AF₃
[[("A", In), ("B", Out), ("C", Undec), ("D", Undec), ("E", Undec)]]
```

```
> semiStables AF_3
[[("A", In), ("B", Out), ("C", Undec), ("D", Undec), ("E", Undec)]]
```

Finally, we relate the labellings back to the extensions. The complete extension of an argumentation framework is just the complete labelling, keeping only those arguments that were labelled 'In'. Analogously for *preferredExt*, *stableExt* and *semiStableExt*.

```
completeExt :: Ord arg ⇒ DungAF arg → [[arg]]
completeExt af = [[arg | (arg, In) ← c] | c ← complete af]
```

4.2.4 Caminada's labelling algorithm

This section discusses an implementation of Caminada's algorithm [28] for calculating labellings. The original algorithm has three variants that can calculate preferred, semi-stable and stable labellings. The implementation treated in this section is simplified, performing no caching of potential semi-stable labellings, and furthermore removing checks inside the algorithm. The resulting algorithm generates a subset of the admissible labellings that include at least all preferred labellings (and therefore the semi-stable and stable labellings). The preferred, semi-stable and stable labellings are then obtained by applying the appropriate filter.

The initial state of the algorithm is an all-in labelling of the given argumentation framework, trivially satisfying that no arguments are illegally labelled Out (or Undec). However, to satisfy both conditions of an admissible labelling, the labelling additionally needs to have no arguments that are illegally labelled In. This is achieved by taking a sequence of *transition steps* that each take an illegally In labelled argument and relabel it to Out (or Undec, if it is illegally Out). Then, for each of A its defeated arguments, if it is now illegally Out, it will be relabelled Undec.

Definition 4.17 (Transition step (Def. 9 of [28]))**.** Given an argumentation framework $\langle Args, Def\rangle$, a labelling $\mathcal{L}$ over $Args$ and $A \in Args$ with A illegally In in $\mathcal{L}$, then a transition step on A in $\mathcal{L}$ consists of the following:

1. the label of A is changed from 'In' to 'Out'
2. for every $B \in \{A\} \cup A^+$, if B is illegally out, then change the label for B from 'Out' to 'Undecided'

The Haskell equivalent changes the status of A to Out, binds *bs* to the defeated arguments of A and A itself, and then partitions the current labelling (with A changed) into two lists, depending on whether they are illegally Out or not. The result is then a mapping of all the illegally out arguments to *Undecided* together with the remaining partition list (which are not illegally Out).

```
transitionStep :: Eq arg ⇒ DungAF arg → Labelling arg →
                  arg → Labelling arg
transitionStep af labs a =
let labs'              = (a, Out) : delete (a, In) labs
    bs                 = a : aplus af a
    (newUndecs, rem)   = partition (λlab@(b, l) →
                                        b ∈ bs
                                      ∧ illegallyOut af labs' lab)
                           labs'
in map (λ(b, _) → (b, Undecided)) newUndecs
 ++ rem
```

A labelling is *terminated* in a transition sequence if the labelling does not contain any argument that is illegally in[32].

```
terminatedLabelling :: Eq arg ⇒ DungAF arg → Labelling arg → Bool
terminatedLabelling af labs = ¬ ∘ or $ map (illegallyIn af labs) labs
```

As an optimisation step in devising the algorithm, Caminada [28] defines the concept of a superillegally In argument.

Definition 4.18 (Superillegally in (Def. 11 of [28]))**.** Given an argumentation framework $\langle Args, Def\rangle$, a labelling $\mathcal{L}$ over $Args$ and $A \in Args$. A is superillegally In in $\mathcal{L}$ iff A is labelled In by $\mathcal{L}$ and it is defeated by an argument that is legally In in $\mathcal{L}$ or Undec in $\mathcal{L}$.

A transition step on an illegally In argument might result in an admissible, but not complete labelling. Choosing a superillegally In argument makes this at least less likely.

```
superIllegallyIn :: Eq arg ⇒ DungAF arg → Labelling arg →
                    (arg, Status) → Bool
superIllegallyIn af labs (a, In) =
  ¬ ∘ null $
    [lab | lab ← labAttackers af a labs,
           legallyIn af labs lab ∨ legallyUndec af labs lab]
superIllegallyIn _ _ _          = False
```

[32]Caminada [28] instead defines a whole transition sequence.

Given the previously defined concepts of a transition step and a superillegally In argument, we can now define the algorithm for calculating at least all maximal admissible labellings.

Algorithm 4.19 (Simplified algorithm computing at least all maximal admissible labellings (Based on Algorithm on page 10 of [28]))**.** Given an argumentation framework $\langle Args, Def\rangle$:

```
find_admissible($\mathcal{L}$: Labelling) → Set(Labelling):
  1. if $\mathcal{L}$ does not have an argument that is superillegally In:
     (a) if $\mathcal{L}$ does not have an argument that is illegally In:
           i. return $\mathcal{L}$ as a set;
     (b) else
           i. for each argument $A$ that is illegally In, in $\mathcal{L}$:
              A. find_admissible(transition_step($A$, $\mathcal{L}$));
  2. else
     (a) $A$ := An argument that is super-illegally In, in $\mathcal{L}$;
     (b) find_admissible(transition_step($A$, $\mathcal{L}$));
```

The set of at least all maximal admissible labellings can then be calculated by calling *find_admissible*(*all-in*), where *all-in* is the all-in labelling.

The corresponding Haskell implementation closely follows the structure of the above defined algorithm, using **case** instead of **if** statements. One thing that is more explicit in the Haskell version, is that if we have a list illegally In labelled arguments, we can indeed do a transition step for each argument, but we also need to combine the results into one list (or set) by using *concatMap*.

```
findAdmissibles :: Ord arg ⇒ DungAF arg → [Labelling arg]
findAdmissibles af@(AF args def) =
let allInArgs = allIn args
    fAdm :: Eq arg ⇒ DungAF arg →
              Labelling arg → [Labelling arg]
    fAdm af labs =
      case filter (superIllegallyIn af labs) labs of
        []          → case filter (illegallyIn af labs) labs of
                          []   → [labs]
                          ills → concatMap (fAdm af) $
                                   map (transitionStep af labs ∘ fst)
```

```
                                    ills
                  ((a, _) : _) → fAdm af (transitionStep af labs a)
    in nub ∘ map sort $ fAdm af allInArgs
```

Computes all preferred labellings for a Dung argumentation framework, by taking the maximally in admissible labellings and applying *isPreferred* as a filter. The *stable* and *semiStable* functions are defined similarly.

```
preferred :: Ord arg ⇒ DungAF arg → [Labelling arg]
preferred af@(AF args def) =
let adms = findAdmissibles af
in filter (isPreferred af adms) adms
```

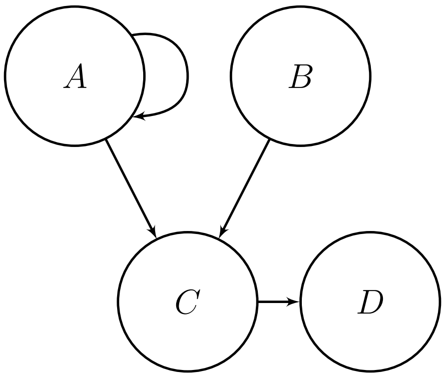


Figure 4.4: An argumentation framework with no stable extension

In Figure 4.4 the example from Section 3.2.3 is repeated. The Haskell equivalent is the following:

```
AF₃ = AF [a, b, c, d]
         [(a, a), (a, c), (b, c), (c, d)]
```

The implementation gives the expected results:

```
findAdmissibles AF₃
> [[("A", Undec), ("B", In), ("C", Out), ("D", In)]]
preferred AF₃
> [[("A", Undec), ("B", In), ("C", Out), ("D", In)]]
semiStable AF₃
> [[("A", Undec), ("B", In), ("C", Out), ("D", In)]]
stable AF₃
> []
```

4.3 Dungell: a command-line interface and API for the AFs in Haskell implementation

This section describes Dungell [71] an application accompanying the previously discussed implementation Dung's AFs in Haskell. Dungell provides a parser, an output module compatible with other mainstream implementations, a command-line interface, an application programming interface (API) and extended documentation.

There are various other efficient solvers that exist for AFs, which are faster than Dungell (see also Section 4.4). Instead, the approach taken in the implementation of Dungell and its accompanying library, is to provide an implementation that is as clear and close to the mathematical definitions as possible. The main reasons this approach is taken, is that:

- the library becomes intuitive, reproducible and easier to verify;
- the implemented definitions can more easily be converted to and proven correct in a theorem prover;
- the library can be used as a translation target.

The library also provides output formats readable by the current fastest implementations.

The combination of these features is intended to allow implementers of structured argumentation models to use Dungell to implement a translation from a structured model into Dungell, possibly performing formal verification, while still providing an efficient evaluation method.

4.3.1 ICCMA competition

Recently, there has been significant interest in the implementation of Dung's argumentation frameworks, fueled by the organisation of the first International Competition of Computational Models of Argumentation (ICCMA)[33] [175, 176]. The Dungell solver discussed in this chapter was also submitted for ICCMA [71]. Although the competition results did not make Dungell qualify as a fast solver, it was one of the only solvers that took a direct implementation approach that was close to the mathematics, while still providing the option to perform fast evaluation by calling other solvers.

[33]See `http://argumentationcompetition.org/2015/`.

4.3.2 Input and output

Dungell provides parsing facilities based on the argument format used by CEGARTIX [56] and the ICCMA competition[34]. A text file specifying an argumentation framework consists of one or more lines declaring an argument, `arg(a1).`, followed by zero or more lines defining an attack (defeat) between two already defined arguments, `att(a1,a2).` or `atk(a1,a2).`. The parser, implemented using the Parsec parsing library [106][35], is omitted for brevity, but can be found in the literate Haskell source code of this section.

Arguments are parsed as a *DungAF* data type (see Section 4.1), with *String*s as its arguments. Arguments can then be printed using the output module, outputting arguments and attacks (defeats) in the same format that arguments are parsed. Although the parser produces a *DungAF String*, the output module can output any type of argument, as long as it has or is provided with a *Show* instance. This makes it immediately possible to output an implemented translation (see Section 7) and output it as standard CEGARTIX format, allowing efficient evaluation by CEGARTIX or any other AF solver supporting this basic format. The output functions are not particularly interesting and are omitted, but are included in the source code of this section.

4.3.3 Command-line interface

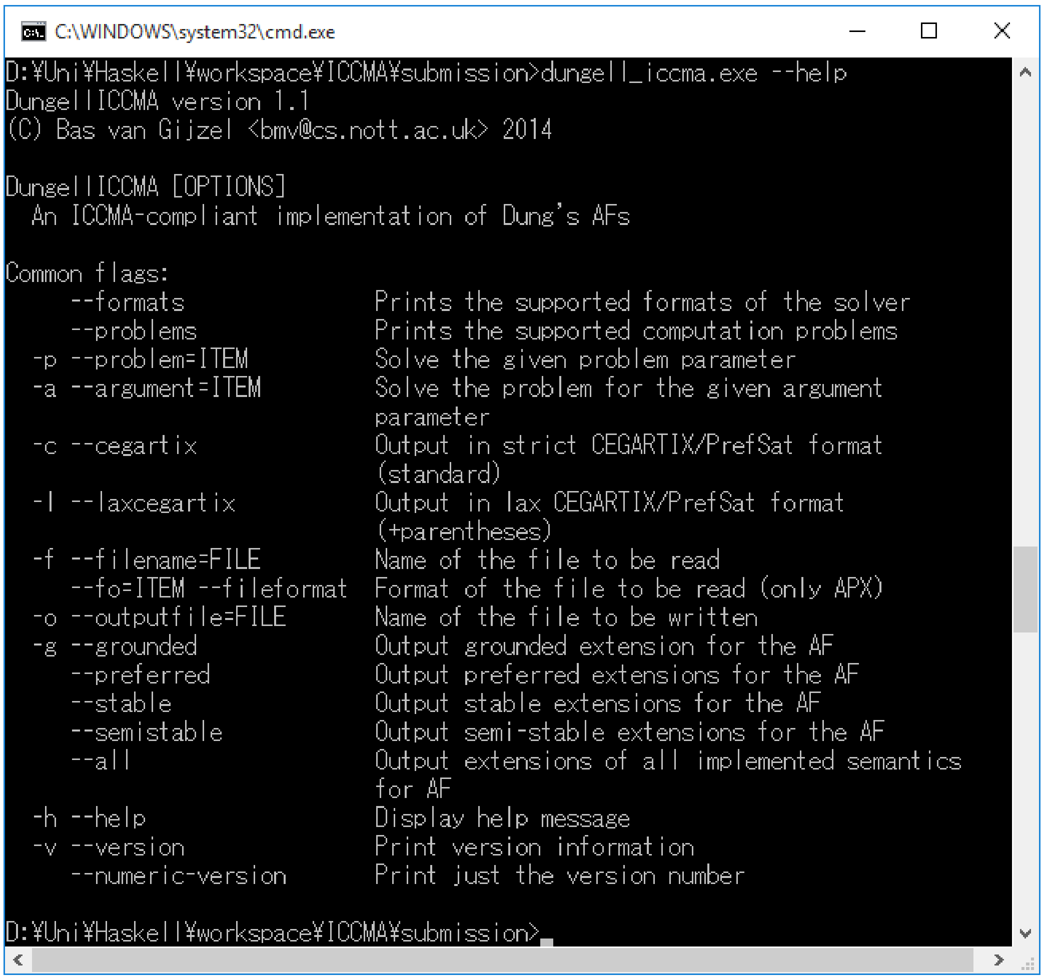
Dungell provides two command-line interfaces (CLI): a basic CLI used for quick testing of examples and a more elaborate CLI that adheres to the Probo interface used in the ICCMA competition, see Figure 4.5.

The ICCMA interface is provided as a separate Cabal package[36], and provides a CLI that takes as input the complete range of problems used in the ICCMA competition. The functionality of the interface, parsing command-line arguments and reading files, is separate from the calls to the actual solver. This allows programmers interested in programming their own CLI in Haskell for an upcoming ICCMA to easily adapt the existing implementation to their own requirements. See Figure 4.6 for output of running the grounded, preferred and stable extension an an example AF.

[34]The specification for the format used in the ICCMA competition can be found here: `http://argumentationcompetition.org/2015/`.

[35]Refer to Section 5.4.1 for a parser for Carneades argument evaluation structures.

[36]See: `https://github.com/nebasuke/DungICCMA`.



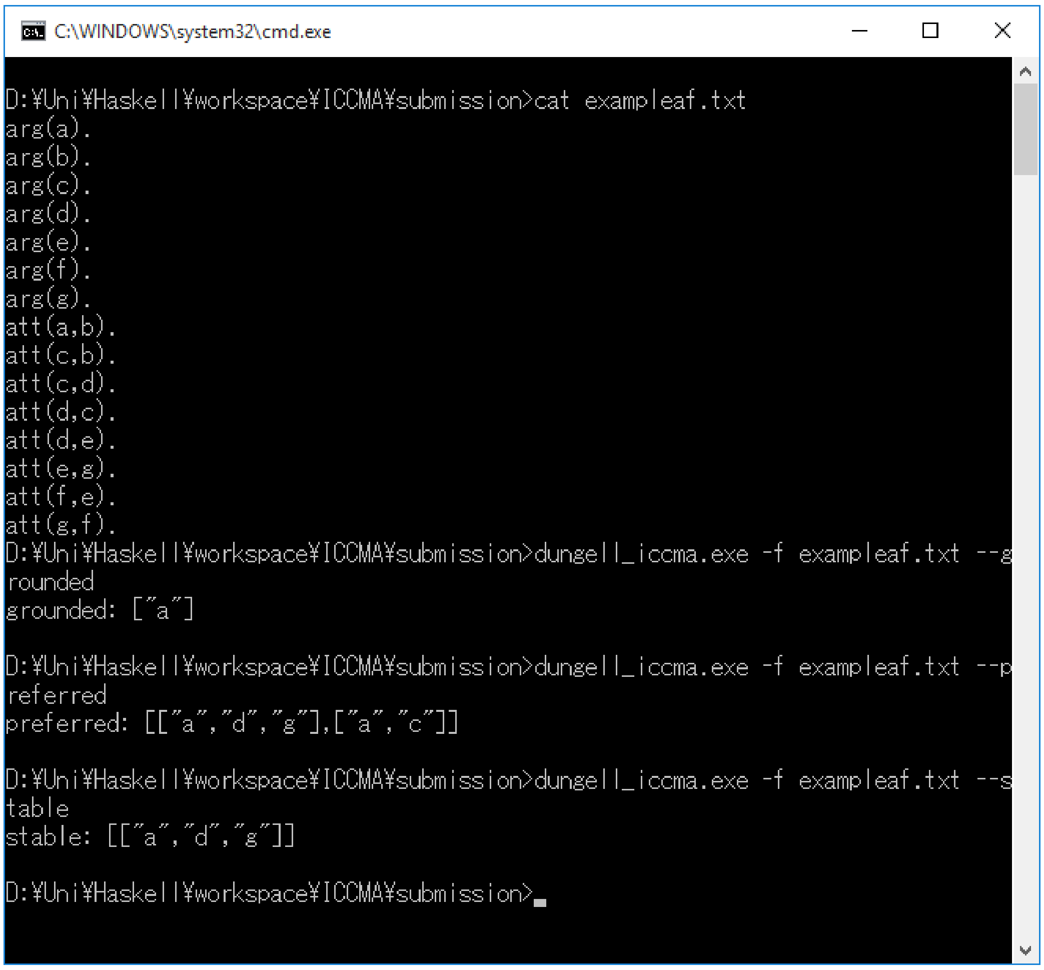
```
D:¥Uni¥Haskell¥workspace¥ICCMA¥submission>dungell_iccma.exe --help
DungellICCMA version 1.1
(C) Bas van Gijzel <bmv@cs.nott.ac.uk> 2014

DungellICCMA [OPTIONS]
  An ICCMA-compliant implementation of Dung's AFs

Common flags:
     --formats              Prints the supported formats of the solver
     --problems             Prints the supported computation problems
  -p --problem=ITEM         Solve the given problem parameter
  -a --argument=ITEM        Solve the problem for the given argument
                            parameter
  -c --cegartix             Output in strict CEGARTIX/PrefSat format
                            (standard)
  -l --laxcegartix          Output in lax CEGARTIX/PrefSat format
                            (+parentheses)
  -f --filename=FILE        Name of the file to be read
     --fo=ITEM --fileformat Format of the file to be read (only APX)
  -o --outputfile=FILE      Name of the file to be written
  -g --grounded             Output grounded extension for the AF
     --preferred            Output preferred extensions for the AF
     --stable               Output stable extensions for the AF
     --semistable           Output semi-stable extensions for the AF
     --all                  Output extensions of all implemented semantics
                            for AF
  -h --help                 Display help message
  -v --version              Print version information
     --numeric-version      Print just the version number

D:¥Uni¥Haskell¥workspace¥ICCMA¥submission>_
```

Figure 4.5: The Dungell ICCMA command line interface



```
D:¥Uni¥Haskell¥workspace¥ICCMA¥submission>cat exampleaf.txt
arg(a).
arg(b).
arg(c).
arg(d).
arg(e).
arg(f).
arg(g).
att(a,b).
att(c,b).
att(c,d).
att(d,c).
att(d,e).
att(e,g).
att(f,e).
att(g,f).
D:¥Uni¥Haskell¥workspace¥ICCMA¥submission>dungell_iccma.exe -f exampleaf.txt --g
rounded
grounded: ["a"]

D:¥Uni¥Haskell¥workspace¥ICCMA¥submission>dungell_iccma.exe -f exampleaf.txt --p
referred
preferred: [["a","d","g"],["a","c"]]

D:¥Uni¥Haskell¥workspace¥ICCMA¥submission>dungell_iccma.exe -f exampleaf.txt --s
table
stable: [["a","d","g"]]

D:¥Uni¥Haskell¥workspace¥ICCMA¥submission>
```

Figure 4.6: The Dungell ICCMA evaluated on an example AF

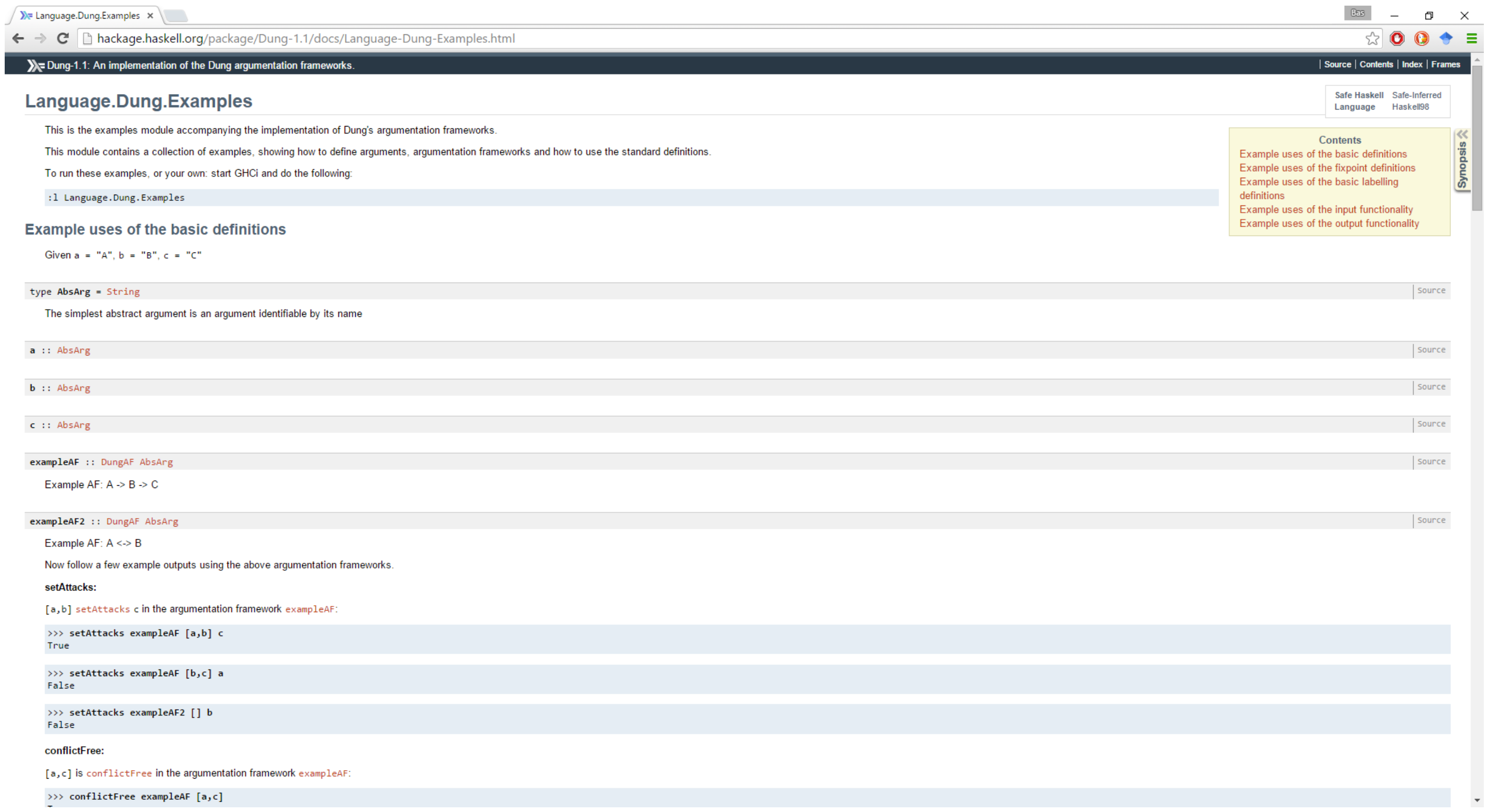
4.3.4 API and documentation on Hackage

The library accompanying the Dungell application is documented by means of a literate programming implementation see Section 4.1 and Section 4.2). The library has also been made available as a Hackage package[37]. The packages includes extensive documentation with various examples of the usage, see Figure 4.7) and the webpage[38], and a well-documented API, see Figure 4.8 and the webpage[39].

[37]`http://hackage.haskell.org/package/Dung`

[38]`http://hackage.haskell.org/package/Dung-1.1/docs/`
`Language-Dung-Examples.html`

[39]`http://hackage.haskell.org/package/Dung-1.1/docs/Language-Dung-AF.html`


Language.Dung.Examples

hackage.haskell.org/package/Dung-1.1/docs/Language-Dung-Examples.html

Dung-1.1: An implementation of the Dung argumentation frameworks.

Source | Contents | Index | Frames

Safe Haskell: Safe-Inferred
Language: Haskell98

Contents
Example uses of the basic definitions
Example uses of the fixpoint definitions
Example uses of the basic labelling definitions
Example uses of the input functionality
Example uses of the output functionality

Language.Dung.Examples

This is the examples module accompanying the implementation of Dung's argumentation frameworks.

This module contains a collection of examples, showing how to define arguments, argumentation frameworks and how to use the standard definitions.

To run these examples, or your own: start GHCi and do the following:

```
:l Language.Dung.Examples
```

Example uses of the basic definitions

Given `a = "A", b = "B", c = "C"`

```
type AbsArg = String
```

The simplest abstract argument is an argument identifiable by its name

```
a :: AbsArg
```

```
b :: AbsArg
```

```
c :: AbsArg
```

```
exampleAF :: DungAF AbsArg
```

Example AF: A -> B -> C

```
exampleAF2 :: DungAF AbsArg
```

Example AF: A <-> B

Now follow a few example outputs using the above argumentation frameworks.

setAttacks:

`[a,b]` `setAttacks` c in the argumentation framework `exampleAF`:

```
>>> setAttacks exampleAF [a,b] c
True
```

```
>>> setAttacks exampleAF [b,c] a
False
```

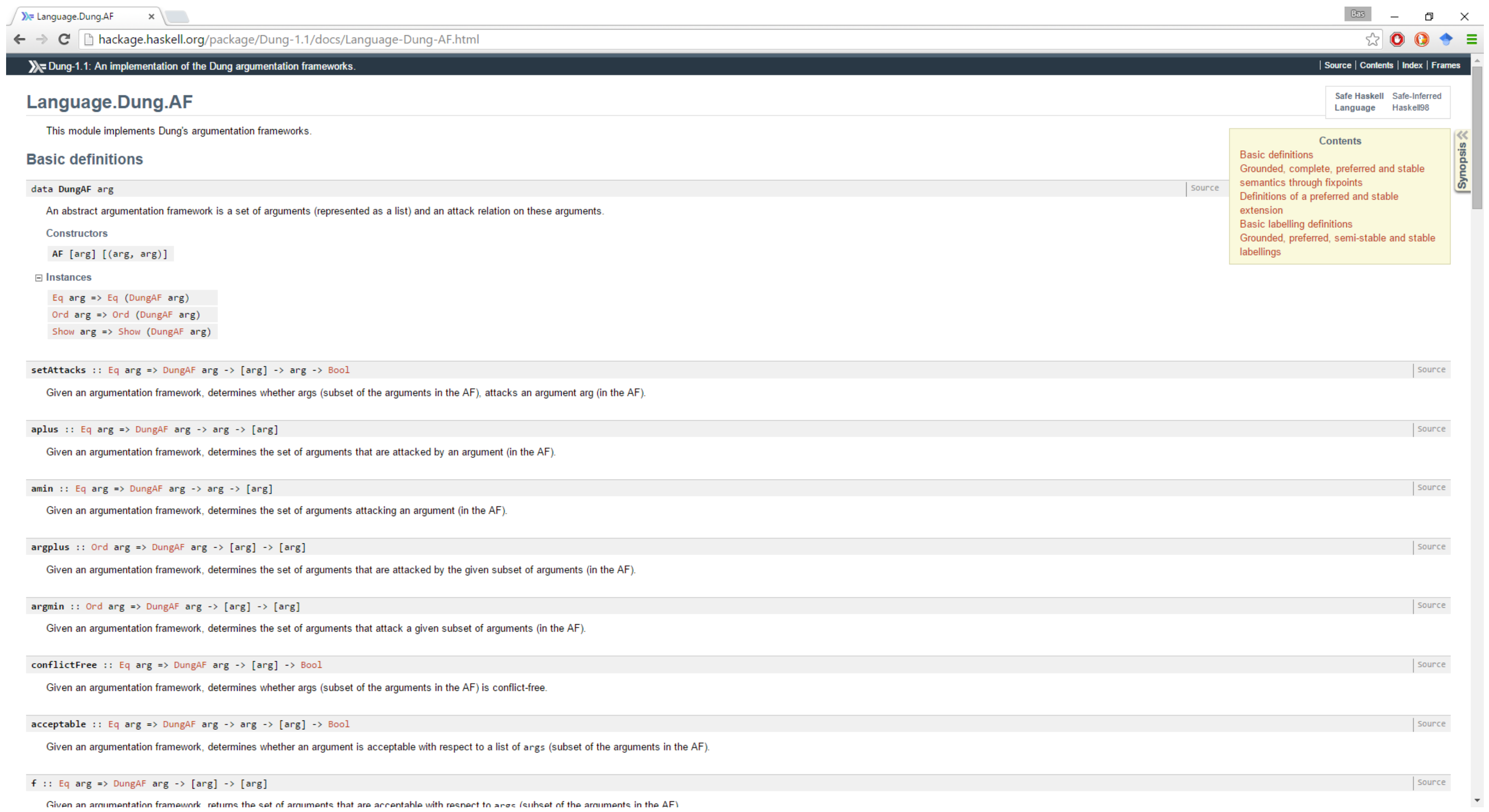
```
>>> setAttacks exampleAF2 [] b
False
```

conflictFree:

`[a,c]` is `conflictFree` in the argumentation framework `exampleAF`:

```
>>> conflictFree exampleAF [a,c]
```

Figure 4.7: A screenshot of part of the documentation of the Dung package on Hackage


Language.Dung.AF
hackage.haskell.org/package/Dung-1.1/docs/Language-Dung-AF.html
Dung-1.1: An implementation of the Dung argumentation frameworks.
Source | Contents | Index | Frames
Safe Haskell Safe-Inferred
Language Haskell98
Synopsis

Contents
Basic definitions
Grounded, complete, preferred and stable semantics through fixpoints
Definitions of a preferred and stable extension
Basic labelling definitions
Grounded, preferred, semi-stable and stable labellings

Language.Dung.AF

This module implements Dung's argumentation frameworks.

Basic definitions

```
data DungAF arg
```

An abstract argumentation framework is a set of arguments (represented as a list) and an attack relation on these arguments.

Constructors

```
AF [arg] [(arg, arg)]
```

Instances

```
Eq arg => Eq (DungAF arg)
Ord arg => Ord (DungAF arg)
Show arg => Show (DungAF arg)
```

```
setAttacks :: Eq arg => DungAF arg -> [arg] -> arg -> Bool
```

Given an argumentation framework, determines whether args (subset of the arguments in the AF), attacks an argument arg (in the AF).

```
aplus :: Eq arg => DungAF arg -> arg -> [arg]
```

Given an argumentation framework, determines the set of arguments that are attacked by an argument (in the AF).

```
amin :: Eq arg => DungAF arg -> arg -> [arg]
```

Given an argumentation framework, determines the set of arguments attacking an argument (in the AF).

```
argplus :: Ord arg => DungAF arg -> [arg] -> [arg]
```

Given an argumentation framework, determines the set of arguments that are attacked by the given subset of arguments (in the AF).

```
argmin :: Ord arg => DungAF arg -> [arg] -> [arg]
```

Given an argumentation framework, determines the set of arguments that attack a given subset of arguments (in the AF).

```
conflictFree :: Eq arg => DungAF arg -> [arg] -> Bool
```

Given an argumentation framework, determines whether args (subset of the arguments in the AF) is conflict-free.

```
acceptable :: Eq arg => DungAF arg -> arg -> [arg] -> Bool
```

Given an argumentation framework, determines whether an argument is acceptable with respect to a list of `args` (subset of the arguments in the AF).

```
f :: Eq arg => DungAF arg -> [arg] -> [arg]
```

Given an argumentation framework, returns the set of arguments that are acceptable with respect to `args` (subset of the arguments in the AF).

Figure 4.8: The Dung package API

4.4 Related work

The survey by Charwat et al. [36] provides a comprehensive overview of the existing software that implement Dung's abstract argumentation frameworks. They classify two approaches to implementing an argumentation system: the *direct approach* and the *reduction approach*.

The direct approach implements AFs by directly employing algorithms for calculating extensions, labellings or argument games. The reduction approach instead reduces the problem of calculating the semantics to a problem phrased in propositional logic solvable by SAT a constraint satisfaction problem or other domains including answer set programming. See Charwat et al. [36] for further references to implementations and implementation methods.

4.5 Conclusions

In this chapter we have seen that Haskell can be used to implement Dung's abstract argumentation frameworks, including more sophisticated algorithms, such as an algorithm for calculating semi-stable labellings [28]. The implementation in this chapter should be seen as a demonstration that Haskell and functional programming in general are as suitable a method for implementing Dung's AFs as the logic programming approach, Prolog and ASP are.

The implementation methodology of Dungell is a mixture between the direct and reduction approach as classified by Charwat et al. [36]. The approach taken in this thesis is mainly a direct approach, translating the algorithms for calculating extensions and labellings directly into Haskell. However, my approach combines the intuitiveness of the direct approach with the efficiency of the existing implementations used in the reduction approach, by providing the option to calculate extensions using an output format generated by Dungell. Indeed Charwat et al. single out my approach for its reuse of computational engines and as a general framework suitable for translations.

Chapter 5

Haskell gets argumentative

Computational argumentation theory is studied in the context of artificial intelligence, and a number of computational argumentation frameworks have been put forward to date. However, as discussed in Chapter 1, there is a lack of concrete, high level realisations of these frameworks. This hampers research and applications at a number of levels. The lack of suitable domain-specific languages in which to formalise argumentation frameworks could be a contributing factor. In this chapter, a formalisation of a particular computational argumentation framework, Carneades is presented. It serves as a case study to determine the extent to which functional languages are useful as a means to realising computational argumentation frameworks and reason about them. This chapter repeats relevant definitions both for convenience and to make the literate Haskell implementation in this chapter completely self-contained. For a further, in depth treatment of the Carneades model, see Section 3.4.

This chapter is structured as follows. Section 5.1 gives the reader an introduction to the problem stated and the contributions made towards solving it. Section 5.2, gives an intuitive introduction to Carneades, providing a concrete and easy to grasp example to provide a grounding for the technical account of Carneades and the implementation of it that follows. Section 5.3 repeats relevant formal definitions of the central parts of Carneades juxtapositioned with a realisation in Haskell. The section covers central notions such as the argumentation graph that captures the relationships between arguments and counter arguments, the exact characterisation of proof standards (including "beyond reasonable doubt"), and the notion of an audience with respect to which arguments are assigned weights. Section 5.4 shows how the implementation can be lifted to a Haskell library, discussing a parser and library documentation. Related work is discussed in Section 5.5, and Section 5.6 concludes with a discussion of what was learnt from this case study, its relevance to argumentation theorists, and various avenues for future work.

5.1 Introduction

Since Dung's seminal work, a number of other computational argumentation frameworks have been proposed, and the study of their relative merits and exact, mathematical relationships is now an active sub-field in its own right [22, 19, 77, 88, 145]. However, a problem here is the lack of concrete realisations of many of these frameworks, in particular realisations that are sufficiently close to the mathematical definitions to serve as specifications in their own right. This hampers communication between argumentation theorists, impedes formal verification of frameworks (see Chapter 7) and their relationships as well as investigation of their computational complexity, and raises the barrier of entry for people interested in developing practical applications of computational argumentation.

A possible contributing factor to this state of affairs is the lack of a language for expressing such frameworks that on the one hand is sufficiently high-level to be attractive to argumentation theorists, and on the other is rigorous and facilitates formal (preferably machine-checked) reasoning. A further hypothesis is that a functional, domain-specific language (DSL) would be a good way to address this problem, in particular if realised in close connection with a proof assistant.

The work presented in this chapter is a first step towards such a language. In order to learn how to best capture argumentation theory in a functional setting, this chapter undertakes a case study of casting a particular computational argumentation framework, Carneades [86, 83], into Haskell. Ultimately, the goal is to generalise this into an embedded DSL for argumentation theory, possibly within the dependently typed language Agda with a view to facilitate machine checking of proofs about arguments and the relationships between argumentation frameworks. In Chapter 8 an initial step to formalisation is taken, by casting the implementation of Dung's AF into Agda and proving certain properties.

The initial experience from the case study has been positive: the formalisation in Haskell was deemed to be intuitive and readable as a specification on its own by Tom Gordon, an argumentation theorist and one of the authors of the Carneades argumentation framework [81]. It furthermore has been used as the basis of a course in . Finally, the case study is a contribution in its own right in that it:

- already constitutes a helpful tool for argumentation theorists;

- demonstrates the usefulness of a language like Haskell itself as a tool for argumentation theorists, albeit assuming a certain proficiency in functional programming;
- is a novel application of Haskell that should be of interest for example to researchers interested in using Haskell for AI research and applications.

This is not to say that there are no implementations of *specific* argumentation theory frameworks around; see Section 5.5 for an overview. However, the goals and structure of those systems are rather different from what we are concerned with in this case study. In particular, a close and manifest connection between argumentation theory and its realisation in software appears not to be a main objective of existing work. For the work in this thesis, on the other hand, maintaining such a connection is central, as this is the key to the ultimate goal of a successful *generic* DSL suitable for realising *any* argumentation framework.

5.2 Background: the Carneades argumentation model

The main of purpose of the Carneades argumentation model is to formalise argumentation problems in a legal context. Carneades contains mathematical structures to represent arguments placed in favour of or against atomic propositions; i.e., an argument in Carneades is a *single* inference step from a set of *premises* and *exceptions* to a *conclusion*, where all propositions in the premises, exceptions and conclusion are literals in the language of propositional logic. For example, Figure 5.1 gives an argument in favour of the proposition *murder* mimicking an argument that might be put forward in a court case.

For ease of reference, the argument is named (*arg1*). However, arguments are not formally named in Carneades, but instead identified by their logical content. An argument is only to be taken into account if it is *applicable* in a technical sense defined in Carneades. In this case, *arg1* is applicable given that its two premises *kill* and *intent* are *acceptable*, also in a technical sense defined in Carneades. (We will come back to exceptions below.) In other words, we are able to derive that there was a murder, given that we know (with sufficient certainty) that someone was killed and that this was done with intent.

In Carneades, a set of arguments is evaluated relative to a specific *audience* (jury). The audience determines two things: a set of *assumptions*, and

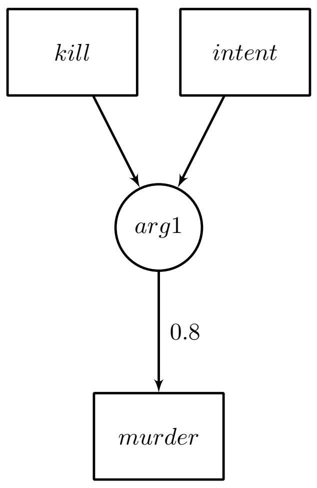


Figure 5.1: Carneades argument for murder

the *weight* of each argument, ranging from 0 to 1. The assumptions are the premises and exceptions that are taken for granted by the audience, while the weights reflect the subjective merit of the arguments. In our example, the weight of *arg1* is 0.8, and it is applicable if *kill* and *intent* are either assumptions of the audience, or have been derived by some other arguments, relative to the *same* audience.

Things get more interesting when there are arguments both for and against the same proposition. The conclusion of an argument against an atomic proposition is the propositional negation of that proposition, while an argument against a negated atomic proposition is just the (positive) proposition itself. Depending on the type of proposition, and even the type of case (criminal or civil), there are certain requirements the arguments should fulfil to tip the balance in either direction. These requirements are called *proof standards*. Carneades specifies a range of proof standards, and to model opposing arguments we need to assign a specific proof standard, such as *clear and convincing evidence*, to a proposition.

Consider the two arguments in Figure 5.2, where the arrows with circular heads indicate exceptions. The first argument represents an argument in favour of *intent*. It is applicable given that the premise *witness* is acceptable and the exception *unreliable* does not hold. The second argument represents an argument against *intent*. It involves a second witness, *witness2*, who claims the opposite of the first witness. Let us assume that the required proof standard for *intent* indeed is *clear and convincing evidence*, which Carneades

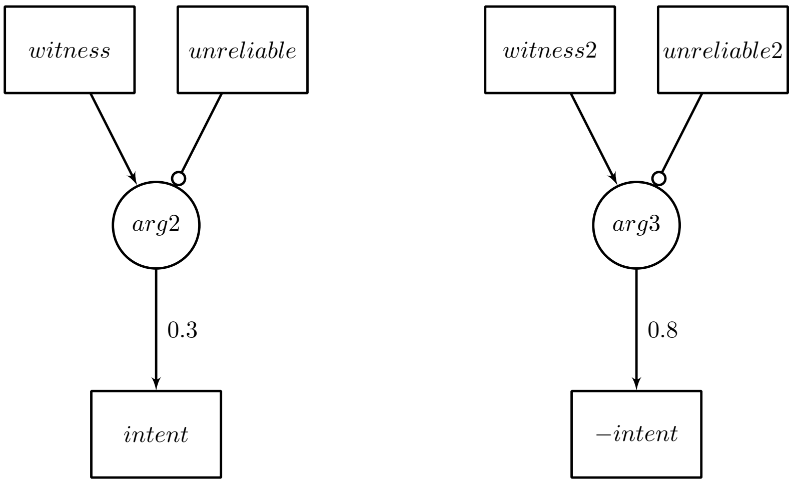


Figure 5.2: Arguments pro and con *intent*

formally defines as follows:[40]

Definition 5.1 (Clear and convincing evidence (Def. 9 of [86]))**.** Given two globally predefined positive constants α and β; clear and convincing evidence holds for a specific proposition p iff

- There is at least one applicable argument for proposition p that has at least a weight of α.

- The maximal weight of the applicable arguments in favour of p are at least β stronger than the maximal weight of the applicable arguments against p.

Taking $\alpha = 0.2$, $\beta = 0.3$, and given an audience that determines the argument weights to be as per the figure and that assumes $\{\mathit{witness}, \mathit{witness2}\}$, we have that $-\mathit{intent}$ is acceptable, because *arg3* and *arg2* are applicable, $\text{weight}(\mathit{arg3}) > \alpha$, and $\text{weight}(\mathit{arg3}) > \text{weight}(\mathit{arg2}) + \beta$.

For another example, had *unreliable2* been assumed as well, or found to be acceptable through other (applicable) arguments, that would have made *arg3* inapplicable. That in turn would make *intent* acceptable, as the weight 0.3 of *arg2* satisfies the conditions for clear and convincing evidence given that there now are no applicable counter arguments, and we could then proceed to establish *murder* by *arg1* had it been established that someone indeed was killed.

[40]Similar to Chapter 3.4, all definitions in this chapter are adapted to be stage-specific.

5.3 Towards a DSL for Carneades in Haskell

5.3.1 Arguments

As one of the goals of this thesis is a DSL for argumentation theory, the realisation in Haskell will strive to mirror the mathematical model of Carneades argumentation framework as closely as possible. Ideally, there would be little more to a realisation than a transliteration. This section proceeds with the central definitions of Carneades along with a realisation of them in Haskell.

Definition 5.2 (Arguments (Def. 1 of [86]))**.** Let $\mathcal{L}$ be a propositional language. An *argument* is a tuple $\langle P, E, c\rangle$ where $P \subset \mathcal{L}$ are its *premises*, $E \subset \mathcal{L}$ with $P \cap E = \emptyset$ are its *exceptions* and $c \in \mathcal{L}$ is its *conclusion*. For simplicity, all members of $\mathcal{L}$ must be literals, i.e. either an atomic proposition or a negated one. An argument is said to be *pro* its conclusion c (which may be a negative atomic proposition) and *con* the negation of c.

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 $\overline{p}$ for the negation of a literal p. The realisation is immediate:

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

An *argument* has been realised as a newtype to allow a user-defined overloading of the equality operator, by providing an Eq instance (see also Section 2.1.3). It contains 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)
```

Arguments are considered equal if their premises, exceptions and conclusion are equal; thus arguments are identified by their logical content. The equality instance for *Argument* takes this into account by comparing the lists as sets.

```
instance Eq Argument where
(Arg (prems, excs, c)) ≡ (Arg (prems′, excs′, c′))
```

```
= Set.fromList prems ≡ Set.fromList prems' ∧
  Set.fromList excs ≡ Set.fromList excs'   ∧
  c ≡ c'
```

A set of arguments determines how propositions depend on each other. Carneades requires that there are no cycles among these dependencies. Following Brewka and Gordon [22], a dependency graph is used to determine acyclicity of a set of arguments.

Definition 5.3 (Acyclic set of arguments (Adapted Def. 2.1 of [22])). A set of *arguments* is *acyclic* iff its corresponding dependency graph is acyclic. The corresponding dependency graph has a node for every literal appearing in the set of arguments. A node p has a link to node q whenever p depends on q in the sense that there is an argument pro or con p that has q or $\overline{q}$ in its set of premises or exceptions.

Our realisation of a set of arguments is considered abstract for DSL purposes, only providing a check for acyclicity and a function to retrieve arguments pro a proposition. FGL [59] is used to implement the dependency graph, forming nodes for propositions and edges for the dependencies. For simplicity, we opt to keep the graph also as the representation of a set of arguments.

```
type ArgSet = Gr (PropLiteral, [Argument]) ()
```

An argument set is thus an inductive graph, with at its nodes a *PropLiteral* together will all its pro arguments and at its edges no label (represented by the unit element). Note that for practical purposes we do not need to know the following implementation but can use the abstraction further below.

```
type ArgSet = ...
getArgs    :: PropLiteral → ArgSet → [Argument]
checkCycle :: ArgSet → Bool
```

The Carneades model requires the argument set to be acyclic, which can be checked using the abstract checkCycle function.

```
cyclic :: (DynGraph g) ⇒ g a b → Bool
cyclic g | ¬ (null leafs) = cyclic (delNodes leafs g)
  | otherwise = ¬ (isEmpty g)
  where leafs = filter (isLeaf g) (nodes g)
isLeaf :: (DynGraph g) ⇒ g a b → Node → Bool
isLeaf g n = n ∉ map fst (edges g)
```

The *cyclic* function checks whether there are any leafs present and if so deletes all the leafs from the graph, calling the check recursively. A leaf is defined to be a node in the graph that does not have an edge to another node in the graph (including itself). Defining *checkCycle* = *cyclic* then gives us the previously given abstraction.

The code of the cycle check is inspired by the work of a student of the University of Edinburgh, Stefan Sabev[41]. Stefan and other students taking the Artificial Intelligence Large Practical module[42] were asked to extend a previous version of the implementation discussed in this chapter of which some students republished their source code on GitHub [79]. I believe that this result is a strong indication of the difference it can make if an argumentation theorist writing an implementation does make their code well documented, open source and publicly available.

5.3.2 Carneades Argument Evaluation Structure

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

Definition 5.4 (Carneades Argument Evaluation Structure (CAES) (Adapted Def. 4 of [86]))**.** 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 (Definition 5.5), and *standard* is a total function mapping each proposition to to its specific proof standard.

Note that propositions may be associated with *different* proof standards. This is considered a particular strength of the Carneades framework. The transliteration into Haskell is almost immediate[43]:

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

Definition 5.5 (Audience (Adapted Def. 3 of [86]))**.** 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

[41]See `https://github.com/SSabev/Haskell_Carneades` for his GitHub repository.

[42]School of Informatics, University of Edinburgh, AILP 2012–2013, 2013–2014, 2014–2015: `http://www.inf.ed.ac.uk/teaching/courses/ailp/`

[43]Note that we use a newtype to prevent a cycle in the type definitions.

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. It is possible to give a faithful representation by directly translating the mapping into a function type:

```
type OldPropStandard = PropLiteral → ProofStandard
```

Exactly this was done for the initial implementation [72]. However, to support the translation of proof standards in Chapter 7, we need to have access to the proof standard used and therefore opt to have a representation that can be checked for equality (*String*s representing the name of the proof standard), together with a mapping to the corresponding proof standard, *psMap*.

```
type PropStandard = PropLiteral → PSName
data PSName = Scintilla
  | Preponderance | ClearAndConvincing
  | BeyondReasonableDoubt | DialecticalValidity
deriving (Show, Eq)
```

Proof standard names can then be mapped to their according proof standards (to be defined later).

```
psMap :: PSName → ProofStandard
psMap Scintilla             = scintilla
psMap Preponderance         = preponderance
psMap ClearAndConvincing    = clear_and_convincing
psMap BeyondReasonableDoubt = beyond_reasonable_doubt
psMap DialecticalValidity   = dialectical_validity
```

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 ProofStandard = PropLiteral → CAES → Bool
```

This aggregation process will be defined in detail in the next section, but note that it is done relative to a specific CAES, and note the cyclic dependence at the type level between *CAES* and *ProofStandard*.

The above definition of proof standard also demonstrates that implementation in a typed language such as Haskell is a useful way of verifying definitions from argumentation theoretic models. The implementation effort in this chapter revealed that the original definition as given in [86] 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.

5.3.3 Evaluation

Two concepts central to the evaluation 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 5.6 (Applicability of arguments (Adapted Def. 6 of [86]))**.** 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 [$\overline{p}$ is not an assumption and p is acceptable in C] and
- $e \in E$ implies e is not an assumption and [$\overline{e}$ is an assumption or e is not acceptable in C].

Definition 5.7 (Acceptability of propositions (Adapted Def. 5 of [86]))**.** 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.

Note that these two definitions in general are mutually dependent because acceptability depends on proof standards, and most sensible proof standards depend on the applicability of arguments. This is the reason that Carneades restricts the set of arguments to be acyclic. (Specific proof standards are considered in the next section.) The realisation of applicability and acceptability in Haskell is straightforward, adding only a call to psMap to translate the proof standard name:

```
applicable :: Argument → CAES → Bool
applicable  (Arg (prems, excns, _))
                caes@(CAES (_, (assumptions, _), _))
    = and $ [p ∈ assumptions ∨
```

```
                  (negate p ∉ assumptions ∧
                   p `acceptable` caes) | p ← prems]
                ++
                [(e ∉ assumptions) ∧
                  (negate e ∈ assumptions ∨
                   ¬ (e `acceptable` caes)) | e ← excns]
acceptable :: PropLiteral → CAES → Bool
acceptable c caes@(CAES (_, _, standard))
   = c `s` caes
  where s = psMap $ standard c
```

5.3.4 Proof standards

Carneades predefines five proof standards, originating from the work of Freeman and Farley [63, 62]: *scintilla of evidence*, *preponderance of the evidence*, *clear and convincing evidence*, *beyond reasonable doubt* and *dialectical validity*. Some proof standards depend on constants such as α, β, γ; these are assumed to be defined once and globally. This time, definitions will directly be given in Haskell, as they really only are translitarations of the original definitions.

For a proposition p to satisfy the weakest proof standard, scintilla of evidence, there should be at least one applicable argument pro p in the CAES:

```
scintilla :: ProofStandard
scintilla p caes@(CAES (g, _, _))
  = any (`applicable`caes) (getArgs p g)
```

Preponderance of the evidence additionally requires the maximum weight of the applicable arguments pro p to be greater than the maximum weight of the applicable arguments con p. The weight of zero arguments is taken to be 0. As the maximal weight of applicable arguments pro and con is a recurring theme in the definitions of several of the proof standards, those notions should be defined first:

```
maxWeightApplicable :: [Argument] → CAES → Weight
maxWeightApplicable as caes@(CAES (_, (_, argWeight), _))
  = foldl max 0 [argWeight a | a ← as, a `applicable` caes]
maxWeightPro :: PropLiteral → CAES → Weight
maxWeightPro p caes@(CAES (g, _, _))
```

```
 = maxWeightApplicable (getArgs p g) caes
maxWeightCon :: PropLiteral → CAES → Weight
maxWeightCon p caes@(CAES (g, _, _))
 = maxWeightApplicable (getArgs (negate p) g) caes
```

The preponderance proof standard can then be defined:

```
preponderance :: ProofStandard
preponderance p caes = maxWeightPro p caes > maxWeightCon p caes
```

Clear and convincing evidence strengthen the preponderance constraints by insisting that the difference between the maximal weights of the pro and con arguments must be greater than a given positive constant β, and there should furthermore be at least one applicable argument pro p that is stronger than a given positive constant α:

```
clear_and_convincing :: ProofStandard
clear_and_convincing p caes
 =  (mwp > α) ∧ (mwp − mwc > β)
   where
     mwp = maxWeightPro p caes
     mwc = maxWeightCon p caes
```

Beyond reasonable doubt has one further requirement: the maximal strength of an argument con p must be less than a given positive constant γ; i.e., there must be no reasonable doubt:

```
beyond_reasonable_doubt :: ProofStandard
beyond_reasonable_doubt p caes
 = clear_and_convincing p caes ∧ (maxWeightCon p caes < γ)
```

Finally dialectical validity requires at least one applicable argument pro p and no applicable arguments con p:

```
dialectical_validity :: ProofStandard
dialectical_validity p caes
   = scintilla p caes ∧ ¬ (scintilla (negate p) caes)
```

5.3.5 Convenience functions

A set of functions to facilitate construction of propositions, arguments, argument sets and sets of assumptions is provided. Together with the definitions covered so far, this constitute the DSL for constructing Carneades argumentation models.

```
mkProp         :: String → PropLiteral
mkArg          :: [String] → [String] → String → Argument
mkArgSet       :: [Argument] → ArgSet
mkAssumptions  :: [String] → [PropLiteral]
```

A string starting with a `'-'` is taken to denote a negative atomic proposition.

To construct an audience, native Haskell tupling is used to combine a set of assumptions and a weight function, exactly as it would be done in the Carneades model:

```
audience :: Audience
audience = (assumptions, weight)
```

Carneades Argument Evaluation Structures and weight functions are defined in a similar way, as will be shown in the next subsection.

Finally, we provide a function for retrieving the arguments for a specific proposition from an argument set, a couple of functions to retrieve all arguments and propositions respectively from an argument set, and functions to retrieve the (not) applicable arguments or (not) acceptable propositions from a CAES:

```
getArgs            :: PropLiteral → ArgSet → [Argument]
getAllArgs         :: ArgSet               → [Argument]
getProps           :: ArgSet               → [PropLiteral]
applicableArgs     :: CAES                 → [Argument]
nonApplicableArgs  :: CAES                 → [Argument]
acceptableProps    :: CAES                 → [PropLiteral]
nonAcceptableProps :: CAES                 → [PropLiteral]
```

5.3.6 Implementing a CAES

This subsection shows how an argumentation theorist given the Carneades DSL developed in this section, can quickly and at a high level of abstraction implement a Carneades argument evaluation structure and evaluate it as well.

We assume the three arguments given in Figure 5.3.

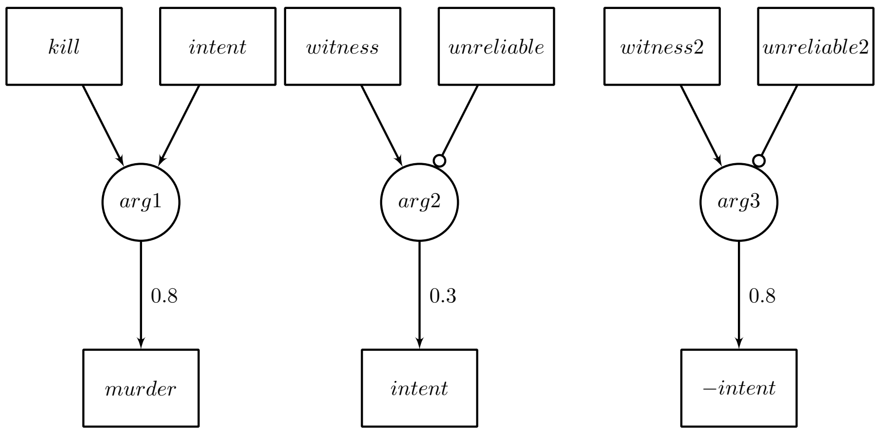


Figure 5.3: Carneades argument for murder, and arguments pro and con *intent*

We furthermore assume the following:

$$\begin{aligned}
\mathit{arguments} &= \{\mathit{arg1}, \mathit{arg2}, \mathit{arg3}\},\\
\mathit{assumptions} &= \{\mathit{kill}, \mathit{witness}, \mathit{witness2}, \mathit{unreliable2}\},\\
\mathit{standard}(\mathit{intent}) &= \mathit{beyond\text{-}reasonable\text{-}doubt},\\
\mathit{standard}(x) &= \mathit{scintilla},\ \text{for any other proposition x},\\
\alpha = 0.4,\ \beta &= 0.3,\ \gamma = 0.2.
\end{aligned}$$

Arguments and the argument graph are constructed by calling *mkArg* and *mkArgSet* respectively:

```
arg1, arg2, arg3 :: Argument
arg1 = mkArg ["kill", "intent"] [] "murder"
arg2 = mkArg ["witness"] ["unreliable"] "intent"
arg3 = mkArg ["witness2"] ["unreliable2"] "-intent"

argSet :: ArgSet
argSet = mkArgSet [arg1, arg2, arg3]
```

The audience is implemented by defining the *weight* function and calling *mkAssumptions* on the propositions which are to be assumed. The audience is just a pair of these:

```
weight :: ArgWeight
weight arg | arg ≡ arg1 = 0.8
weight arg | arg ≡ arg2 = 0.3
weight arg | arg ≡ arg3 = 0.8
weight _                = error "no weight assigned"
assumptions :: [PropLiteral]
assumptions = mkAssumptions ["kill", "witness",
                              "witness2", "unreliable2"]
audience :: Audience
audience = (assumptions, weight)
```

Finally, after assigning proof standards in the *standard* function, we form the CAES from the argument graph, audience and function *standard*:

```
standard :: PropStandard
standard (_, "intent") = BeyondReasonableDoubt
standard _             = Scintilla
caes :: CAES
caes = CAES (argSet, audience, standard)
```

We can now try out the argumentation structure. Arguments are pretty printed in the format *premises* $\sim$ *exceptions* $\Rightarrow$ *conclusion*:

```
getAllArgs argSet
 > [["witness2"]        ~["unreliable2"] ⇒ "-intent",
    ["witness"]         ~["unreliable"]  ⇒ "intent",
    ["kill", "intent"]~[]                 ⇒ "murder"]
```

As expected, there are no applicable arguments for $-$*intent*, since *unreliable2* is an exception, but there is an applicable argument for *intent*, namely *arg2*:

```
filter (`applicable`caes) $ getArgs (mkProp "-intent") argSet
 > []
filter (`applicable`caes) $ getArgs (mkProp "intent") argSet
 > [["witness"] ⇒ "intent"]
```

However, despite the applicable argument *arg2* for *intent*, *murder* should not be acceptable, because the weight of *arg2* $< \alpha$. Interestingly, note that we can't reach the opposite conclusion either:

```
acceptable (mkProp "murder") caes
 > False
acceptable (mkProp "-murder") caes
 > False
```

As a further extension, one could for example imagine giving an argumentation theorist the means to see a trace of the derivation of acceptability. It would be straightforward to add further primitives to the DSL and keeping track of intermediate results for acceptability and applicability to achieve this.

5.4 The DSL as a Haskell library

This section describes how the implementation of the Carneades model is lifted to a programming library by adding parsing functionality and documentation. The resulting library has been made open source and can be found on Hackage[44] and GitHub[45].

5.4.1 Parsing

This section describes the input module accompanying the implementation of Carneades. It defines a simple parser for a Carneades Argument Evaluation Structure (CAES) implemented using the Parsec parsing library [106]. The EBNF for the CAES taken as input has been defined in Figure 5.5. The example from Section 5.3.6 is given as a file in Figure 5.4.

Parsec is a monadic parser combinator library, providing parsers that are efficient and composable. Parsec provides both primitive parsers and higher order functions that work over parsers; e.g., a *many* function that can takes a parser as an argument and applies it zero or more times. The construction of parsers in Parsec often closely follow the structure of the language grammar.

Lexer

Parsec defines its lexing functions parametrised on a user definable *TokenParser*. This means that we can define our lexing functions by building on the existing Haskell style *TokenParser* and then parametrise the lexing functions with the constructed *TokenParser*.

```
lexer :: P.TokenParser ()
lexer = P.makeTokenParser
(haskellStyle
  {P.reservedNames = ["Scintilla", "Preponderance",
    "ClearAndConvincing", "BeyondReasonableDoubt",
    "DialecticalValidity", "scintilla",
    "preponderance", "clear_and_convincing",
```

[44]Cabal package on Hackage: `http://hackage.haskell.org/package/CarneadesDSL`.
[45]CarneadesDSL on GitHub: `https://github.com/nebasuke/CarneadesDSL/`.

```
argument arg1 ["kill", "intent"] [ ] "murder"
argument arg2 ["witness" ] ["unreliable"] "intent"
argument arg3 ["witness2"] ["unreliable2"] "-intent"

weight arg1 0.8
weight arg2 0.3
weight arg3 0.8

assumptions ["kill", "witness", "witness2", "unreliable2"]

-- Haskell style commments are allowed.

{- also valid for standards:
   standard "intent" BeyondReasonableDoubt -}

standard "kill" scintilla
standard "intent" beyond_reasonable_doubt
```

Figure 5.4: A file containing the definition of the CAES from Section 5.3.6

```
          "beyond_reasonable_doubt", "dialectical_validity"]
        }
      )
```

The above definition makes a *TokenParser* that enables Haskell style comments, Haskell style identifiers and reserves the keywords needed to parse the proof standards. The P refers to a shorthand name defined for the appropriate Parsec module.

Below the definition of *float* and *symbol* are given. The other lexing functions (*identifier*, *stringLiteral* and *whiteSpace*) can be defined analogously.

$$
\begin{array}{l}
\mathit{float} :: \mathit{Parser\ Double} \\
\mathit{float} = P.\mathit{float\ lexer} \\
\mathit{symbol} :: \mathit{String} \to \mathit{Parser\ String} \\
\mathit{symbol} = P.\mathit{symbol\ lexer}
\end{array}
$$

Parser

A named version of a Carneades argument is provided to enable corresponding weights to be assigned to be names.

```
⟨caes⟩          ::= ⟨argument⟩* ⟨weight⟩* ⟨assumptions⟩ ⟨standards⟩* eof

⟨argument⟩      ::= ‘arg’ ⟨argName⟩ ⟨propList⟩ ⟨propList⟩ ⟨prop⟩
                  | ‘argument’ ⟨argName⟩ ⟨propList⟩ ⟨propList⟩ ⟨prop⟩

⟨weight⟩        ::= ‘weight’ ⟨argName⟩ [0-9] ‘.’ [0-9]

⟨standard⟩      ::= ‘standard’ ⟨prop⟩ ⟨psName⟩

⟨assumptions⟩   ::= ‘assumptions’ ⟨propList⟩

⟨propList⟩      ::= ‘[’ ⟨props⟩ ‘]’

⟨props⟩         ::= ϵ
                  | ⟨prop⟩ ⟨propElems⟩

⟨propElems⟩     ::= ϵ
                  | ‘,’ ⟨prop⟩ ⟨propElems⟩

⟨prop⟩          ::= ⟨stringLiteral⟩

⟨psName⟩        ::= ‘Scintilla’
                  | ‘Preponderance’
                  | ‘ClearAndConvincing’
                  | ‘BeyondReasonableDoubt’
                  | ‘DialecticalValidity’
                  | ‘scintilla’
                  | ‘preponderance’
                  | ‘clear_and_convincing’
                  | ‘beyond_reasonable_doubt’
                  | ‘dialectical_validity’

⟨argName⟩       ::= ‘-’? ⟨identifier⟩
                  | ⟨stringLiteral⟩

⟨stringLiteral⟩ ::= ‘"’ ‘-’? ⟨identifier⟩ ‘"’

⟨identifier⟩    ::= [‘a’-‘Z’]+ [‘a’-‘Z’,‘0’-‘9’,‘_’,‘’’]*
```

Figure 5.5: An EBNF grammar for the CAES input language

```
data Argument′ = Arg′ String
                        ([PropLiteral], [PropLiteral], PropLiteral)
```

An argument parser tries to parse the string `"argument"` or `"arg"`, allows any whitespace including newlines, parses an *argName* (defined further), two proposition lists, and a proposition corresponding to the conclusion. Given that the parser succeeds, it constructs an *Argument′* as the parsing result. The *try* indicates that Parsec should backtrack if needed.

```
pArgument :: Parser Argument′
pArgument = do
  try (string "argument") <|> string "arg"
  whiteSpace
  name ← argName
  prems ← pPropositions
  excs ← pPropositions
  c ← pProposition
  return (Arg′ name (prems, excs, c))
```

An argument name is a Haskell style identifier or a string literal. A proposition is then read by parsing an *argname* and calling *mkProp* on the resulting String (a function from the Carneades DSL, transforming the String into an argument and handling negation).

```
argName :: Parser String
argName = try identifier <|> stringLiteral
pProposition :: Parser PropLiteral
pProposition = do
  p ← argName
  whiteSpace
  return (mkProp p)
```

A list of propositions can be parsed by employing the higher order function *sepBy*. The *sepBy* function will parse zero or more of an already defined parsing function (*pProposition*) separated by another parser (a parser symbol parsing a comma and any whitespace) and combines the result of the *pProposition*s into a list.

```
pPropositions :: Parser [PropLiteral]
pPropositions = do
  char '[' ≫ whiteSpace
  ps ← pProposition `sepBy` (symbol "," ≫ whiteSpace)
```

```
    char ']' ≫ whiteSpace
    return ps
```

Finally, the definition of a complete Carneades Argument Evaluation Structure (CAES). The parsing of a CAES closely follows the language grammar as defined in Figure 5.5. After parsing, all the parser results are combined by transforming named arguments/weights/standards into unnamed ones and taking those into a CAES.

```
pCAES :: Parser CAES
pCAES = do
    whiteSpace
    args ← many pArgument
    weights ← many pWeight
    assumps ← pAssumptions
    standards ← many pStandard
    eof
    let weight = weightToWeight args weights
    let audience = (assumps, weight)
    let standard = standardToStandard standards
    let argSet = mkArgSet (map argToArg args)
    return (CAES (argSet, audience, standard))
```

The other parsing functions are defined analogously and will be omitted for brevity.

5.4.2 Examples

Below we define a function parse that takes a file, `examplecaes.txt`, and parses it, printing any possible error. If the parse is successful, we print the argument set, the assumptions and the standard corresponding to the *intent* proposition. The contents of `examplecaes.txt` are given in Figure 5.4.

```
parse :: IO ()
parse = do
    input ← readFile "examplecaes.txt"
    (CAES (argSet, (assumps, weight), standard))
      ← case parseCAES input of
      Left err → do putStrLn "Parsing error: "
                    print err
                    exitWith (ExitFailure 1)
      Right caes → return caes
```

```
  print $ getAllArgs argSet
  print assumps
  print $ standard (mkProp "intent")

print $ getAllArgs argSet
> [["witness2"]        ~["unreliable2"] ⇒ "-intent",
   ["witness"]         ~["unreliable"]  ⇒ "intent",
   ["kill", "intent"]~[]                ⇒ "murder"]
print assumps
> [(True, "kill"), (True, "witness"),
   (True, "witness2"), (True, "unreliable2")]
print $ standard (mkProp "intent")
> BeyondReasonableDoubt
```

5.5 Related work

In this section related work of direct relevance to DSLs for argumentation theory will be considered. Specifically, both implementations of structured argumentation models, and DSLs in closely related areas with similar design goals will be discussed.

For a general overview of implementations and a discussion of limitations regarding experimental testing, see Bryant and Krause [24]. Most closely related to the work presented here is likely the well-developed implementation [80] of Carneades in Clojure (see also Section 3.4.3). However, the main aim of that implementation is to provide efficient tools, GUIs, and so on for non-specialists, not to express the implementation in a way that directly relates it to the formal model. Consequently, the connection between the implementation and the model is not immediate. This means that the implementation, while great for argumentation theorists only interested in modelling argumentation problems, is not directly useful to a computational argumentation theorist interested in relating models and implementations, or in verifying definitions. The Clojure implementation is thus in sharp contrast to our work, and reinforces our belief in the value of a high-level, principled approach to formalising argumentation theory frameworks.

Recent, still unpublished work by Gordon and van Gijzel in Agda [84] and ongoing work in Isabelle attempt to address that exact problem by specifying a newer version of Carneades directly in a theorem prover.

There are other implementations of structured argumentation models, see Simari [164] and the overview given in Table 5.1. The majority of these imple-

Name	**Available online**	**Open source**	**Library**
Carneades [86, 83, 80, 82]	Yes	Yes	Yes
Gorgias [125]	Yes	Yes	No
Pollock's work [141]	Yes	Yes	No
Araucaria [158, 181]	Yes	No	No
ASPIC [3]	No	No[46]	No
ArguGrid [180]	No	No[47]	No
CaSAPI [68]	Yes	Yes[48]	No
TOAST [166]	Yes	No	No
VISPARTIX [37]	Yes	Yes	No
Tweety [174]	Yes	Yes	Yes

Table 5.1: An overview of structured argumentation systems

mentations (except for Tweety [174] and Carneades [86, 83, 80, 82]) are not open source and available as a library. It is furthermore rare that the applied implementation techniques, the choice of algorithms, and design decisions for an implementation of a structured argumentation model are published, making it hard to reuse implementation efforts. The implementations in this thesis attempt to break away from this trend.

One of the main attempts to unify work in argumentation theory, encompassing arguments from the computational, philosophical and the linguistic domains, is the Argument Interchange Format (AIF) [39, 156]. The AIF aims to capture arguments stated in the above mentioned domains, while providing a common core ontology for expressing argumentative information and relations. Recent work has given a logical specification of AIF [15], providing foundations for interrelating argumentation semantics of computational models of argumentation, thereby remedying a previous weaknesses of AIF. The implementation discussed in this chapter tackles the problem from another direction, starting with a formal and computationally oriented language instead.

Walkingshaw and Erwig [189, 60] have developed an EDSL for neuron diagrams [107], a formalism in philosophy that can model complex causal relationships between events, similar to how premises and exceptions determine a conclusion in an argument. Walkingshaw and Erwig extend this model to work on non-Boolean values, while at the same time providing an implementation, thereby unifying formal description and actual implementation. This

[46]It is supposedly open source [164], but it is not available online.
[47]It is supposedly open source [164], but it is not available online.
[48]Although the source code is available online, it has restrictions on its usage.

particular goal is very similar to ours. Furthermore, the actual formalisms of neuron diagrams and the Carneades argumentation model are technically related: while an argument on its own is a simple graph, the dependency graph corresponding to the whole Carneades argument evaluation structure is much more complex and has a structure similar to a full neuron diagram. Arguments in Carneades could thus be seen as an easy notation for a specific kind of complex neuron diagrams for which manual encoding would be unfeasible in practice. However, due to the complexity of the resulting encoding, this also means that for an argumentation theorist, neuron diagrams do not offer directly relevant abstractions. That said, Walkingshaw's and Erwig's EDSL itself could offer valuable input on the design for a DSL for argumentation.

Similarly, causal diagrams are a special case of Bayesian networks [134] with additional constraints on the semantics, restricting the relations between nodes to a causal relation (causal diagrams are a graphic and restricted version of Bayesian networks). Building on the already existing relation between Carneades and Bayesian networks [90], we can view the neuron diagrams generalised to non-Boolean values in Carneades by generalising the negation relation and proof standards to non-Boolean values in the obvious way, and picking scintilla of evidence as the proof standard for all propositions. So, in a way, neuron diagrams are a specific case of arguments, using scintilla of evidence as the proof standard. Finally, to compute an output for every combination of inputs, as is done for neuron diagrams, we can vary the set of assumptions accordingly.

However, formal connections between Bayesian networks and (dialectical) argumentation are still in its infancy; most of the work such as Grabmair [90], Keppens [101] and Vreeswijk [186] are high level relations or comparisons, containing no formal proofs. Sjoerd Timmer and Henry Prakken's recent research [178, 179] establishes an initial connection between support graphs in argumentation and Bayesian networks.

5.6 Conclusions and future work

This chapter discussed the Carneades argumentation model and an implementation of it in Haskell. This implementation effort should be seen as a case study and a step towards a generic DSL for argumentation theory, providing a unifying framework in which various argumentation models can be implemented and their relationships studied. We have seen that the original mathematical definitions can be captured at a similar level of abstraction by Haskell code, thereby allowing for greater understanding of the implementation. At the same time we obtained a domain specific language for the Carneades argumentation framework, allowing argumentation theorists to realise arguments essentially only using a vocabulary with which they are already familiar.

The experience from this work has been largely positive. Comments from Tom Gordon [81], one of the authors and implementers of the Carneades argumentation model, suggests that the implementation is intuitive and would even work as an executable specification, which is an innovative approach in argumentation theory as a field. Furthermore, the literate programming paper on which this chapter is based [72], has been used as an educational tool to teach Bachelor students the Carneades model, implementation of argumentation models and argumentation theory in general[49].

The implementation can also be envisioned as being used as a testing framework for computational argumentation theorists and as an intermediate language between implementations, providing a much more formal alternative to the existing Argument Interchange Format [156].

One avenue of future work is the generalisation of the DSL to other related argumentation models. It is relatively common in argumentation theory to define an entirely new model to realise a small extension. However, this hurts the meta-theory as lots of results will have to be re-established from scratch. By reducing such an extension to an existing implementation/DSL such as the previously discussed, for instance by providing an implementation of an existing formal translation such as [77, 145], we effectively formalise a translation between both models, while gaining an implementation of this generalisation at the same time.

This could be taken even further by transferring the functional definitions of an argumentation model into an interactive theorem prover, such as Agda. First of all, the formalisation of the model itself would be more precise. While the Haskell model might seem exact, note that properties such as the acyclicity of arguments, or that premises and exceptions must not overlap, are not inherently part of this model. Second, this would enable formal, *machine-checked*, reasoning *about* the model, such as establishing desirable properties like consistency of the set of derivable conclusions. We will see exactly this in Chapter 7.

Then, if *multiple* argumentation models were to be realised in a theorem prover, relations *between* those models, such as translations, could be formalised. As mentioned in the introduction, there has recently been much work on formalisation of translations between conceptually very different ar-

[49]School of Informatics, University of Edinburgh, AILP 2012–2013, 2013–2014, 2014–2015: `http://www.inf.ed.ac.uk/teaching/courses/ailp/`

gumentation models [19, 77, 145, 88]. But such a translation can be very difficult to verify if done by hand. Using a theorem prover, the complex proofs could be machine-checked, guaranteeing that the translations preserve key properties of the models. An argumentation theorist might also make use of this connection by inputting an argumentation case into one model and, through the formal translation, retrieve a specification in another argumentation model, allowing the use of established properties (such as rationality postulates [29]) of the latter model.

Finally, we are interested in the possibility of mechanised argumentation as such; e.g., as a component of autonomous agents. We thus intend to look into realising various argumentation models efficiently by considering suitable ways to implement the underlying graph structure and exploiting sharing to avoid unnecessarily duplicated work. Ultimately we hope this would allow us to establish results regarding the asymptotic time and space complexity inherent in various argumentation models, while providing a framework for empirical evaluations and testing problems sets at the same time. Especially the latter is an area that has only recently received attention [24, 19], due to the lack of implementations and automated conversion of problem sets.

Chapter 6

Relating Carneades with abstract argumentation via the ASPIC$^+$ framework for structured argumentation

Carneades is a recently proposed formalism for structured argumentation with varying proof standards, inspired by legal reasoning but more generally applicable. Its distinctive feature is that each statement can be given its own proof standard, which is claimed to allow a more natural account of reasoning under burden of proof than existing formalisms for structured argumentation, in which proof standards are defined globally. In this chapter Carneades and the ASPIC$^+$ framework for structured argumentation are formally related by translating the former into the latter. Since ASPIC$^+$ is defined to generate Dung-style abstract argumentation frameworks, this in effect translates Carneades graphs into abstract argumentation frameworks. For this translation, a formal correspondence is proven and it certain rationality postulates are shown to hold. It is furthermore proven that Carneades always induces a unique Dung extension, which is the same in all of Dung's semantics, allowing us to generalise Carneades to cycle-containing structures.

6.1 Introduction

Argumentation involves the construction of arguments in favour of and against statements, selecting the acceptable arguments, and in the end determining which statements hold. How arguments support their conclusion depends on the knowledge they use and the inference rules they apply, so any full theory of argument evaluation should take the structure and content of arguments into account. One way to do so is to define a defeat relation between arguments that takes into account the structure and content of arguments and (if available) information on their relative strength. This approach thus results in an abstract argumentation framework in the sense of Dung [48] (see Section 3.2), so that the full theory of abstract argumentation can be applied. Two frameworks for structured argumentation that are designed following this approach are assumption-based argumentation [17, 49] and ASPIC$^+$ [145] (see Section 3.3). In fact, Prakken [145] shows that assumption-based argumentation can be translated into ASPIC$^+$ as a special case.

However, there have also been advances in structured argumentation that diverge from this approach. A recent application in legal reasoning is the Carneades argumentation system, both a logical model [86, 83] and a software toolbox for structured argument evaluation, construction and visualisation [80] (see Section 3.4). Carneades innovates models of structured argumentation by allowing varying proof standards to be assigned to individual propositions. It is claimed that this allows for a more natural account of reasoning under burden of proof than existing formalisms for structured argumentation, in which proof standards are defined globally [7, 13]. This makes the Carneades formalism potentially very attractive, as signified by the large number of citations due to its proof standards.

Recently, Brewka and Gordon [22] translated Carneades into Brewka and Woltran's [23] abstract dialectical frameworks. Moreover, Brewka and Gordon [19] have proved a formal correspondence between abstract dialectical frameworks and Dung's abstract argumentation frameworks. By combining these results, a formal relation between Carneades and Dung's semantics can be obtained. However, this relation is rather indirect[50]. In this chapter we therefore take a different approach, by translating Carneades into the ASPIC$^+$ framework. Since ASPIC$^+$ is defined to generate abstract argumentation frameworks, which are the input of Dung's approach, a translation of Carneades into ASPIC$^+$ provides a more direct way to translate Carneades' graphs into Dung's frameworks. It will furthermore be proved that Carneades can be modelled cycle-free, thus always inducing a unique Dung extension, which is the same in all of Dung's semantics. This allows us to generalise Carneades' argument evaluation structures to cycle-containing structures, addressing an important issue left for future research by Gordon and Wal-

[50]The translation from Brewka and Gordon [22] is comparable to the translation in this chapter. However, the polynomially sized translation from ADFs to AFs for stable models [19] is achieved through an intermediary representation (boolean networks) and creates various administrative nodes in the AF whose meaning is entire technical.

ton [86]. An additional advantage of translating Carneades to ASPIC$^+$ is that the results of Prakken [145] on the rationality postulates of Caminada and Amgoud [29] can be shown to hold for the translation.

This chapter is structured as follows. In Section 6.2 a formal relation between Carneades and Dung's frameworks by developing a translation and proving formal results[51]. In Section 6.3 related work is treated. Finally, Section 6.4 concludes and discusses future work.

6.2 Relation between Carneades and Dung's argumenation frameworks

In the next subsections Carneades will be related to Dung's model. First we will translate the static, stage specific part of Carneades to structured argumentation frameworks. We will study properties of our translation, for instance proving correspondence results and showing that the translation does not violate rationality postulates. Then we will generalise our translation of Carneades, allowing to lift the restriction of acyclicity on a CAES. Finally, we will relate our translation to the existing translations of Carneades to ADF's and Defeasible Logic [22, 88].

6.2.1 Translation of stage-specific Carneades

First the premises and exceptions of the arguments in Carneades will be related to a knowledge base in an argumentation system. The *assumptions* of the audience in a CAES are propositional literals which are unattackable and furthermore, as can be seen in Definition 3.58, part of the logical theory. Combining these characteristics, assumptions in Carneades are closely related to the concept of axioms in a in a knowledge base and thus will be modelled as necessary axioms, $\mathcal{K}_n$, in our knowledge base. Next, the use of conclusions as a premise in a later argument is similar to the chaining of subarguments to construct more complex arguments and can therefore be handled by the argument generation part of ASPIC$^+$. Finally, a premise with no backing, also called an *issue premise* in Gordon et al. [83], maps exactly to the issue premises in our knowledge base.

Combining these insights, the knowledge base corresponding to a CAES can now be defined.

Definition 6.1 (Knowledge base corresponding to a CAES)**.** Given a CAES $C = \langle arguments, audience, standard \rangle$ with $audience = \langle assumptions, weight \rangle$

[51]Part of this work has appeared before in [76] and [69].

and propositional language $\mathcal{L}_{CAES}$. Then the knowledge base in an argumentation system corresponding to C is a pair $\langle \mathcal{K}, \leqslant' \rangle$ where:

- $\mathcal{K}_n = assumptions$,
- $\mathcal{K}_p = \mathcal{K}_a = \emptyset$,
- $\mathcal{K}_i = \mathcal{L}_{CAES} \backslash (assumptions \cup \{c \mid \langle P, E, c \rangle \in \ arguments\})$,
- $\leqslant' = \{(k, k) \mid k \in (\mathcal{K} \backslash \mathcal{K}_n)\}$.

There is no need to differentiate in the strength of premises, making our preference relation on premises just the reflexive closure on non-axiom premises.

As shown in our visualisation of Carneades' arguments in Example 3.59, the link between the premises, the exceptions and the conclusion is a two part inference. The first part — applicability of the argument — is solely determined by the acceptability of the premises and exceptions. The second step — acceptability of the conclusion — requires the argument to be applicable and furthermore to satisfy the demands of the proof standard that is assigned to the conclusion.

So for every argument $a = \langle P, E, c \rangle$ in a CAES, a defeasible rule going from the premises to the applicability of the argument is added, $P \Rightarrow_{app_a} arg_a$, saying that if P then a is applicable[52]. The other inference is represented by a defeasible rule $arg_a \Rightarrow_{acc_a} c$, saying that if a is applicable, its conclusion is acceptable. As before, app_a and acc_a are rule names, which will need to added to the language, $\mathcal{L}$, of the CAES (rule names are assumed to be disjoint with $\mathcal{L}$).

Exceptions in Carneades' arguments express exceptions to inferring the conclusion. If we have an argument containing an exception that is acceptable or assumed by the audience, then that argument is made inapplicable, so the argument cannot make the conclusion acceptable. Given an acceptable argument containing exception $\overline{p}$, it is not implied that p can be assumed to be true; so two arguments with conflicting exceptions can both be acceptable. This use of exceptions, similar to the concept of justifications in default logic [159], implies that negations of exceptions cannot be modelled as an assumption, but instead need to modelled as an undercutter to the inference rule. So in our translation of argument a, for each exception $e \in E$, an undercutter $e \Rightarrow \neg app_a$ is added to $\mathcal{R}_d$.

[52]The idea to make the applicability step explicit by means of an argument node, is adapted from Brewka and Gordon [22].

Although it might seem natural to include the negation relation of Carneades into the contrariness relation of the corresponding argumentation system, this does not actually work. With scintilla of evidence as a proof standard that can determine acceptable literals of a CAES, both p and $\overline{p}$ are allowed to be acceptable, e.g. $arguments = \{\langle\emptyset, \emptyset, p\rangle, \langle\emptyset, \emptyset, \neg p\rangle\}$. It is furthermore possible to construct an acceptable argument for $\neg c$ while $c \in assumptions$[53]. To retain the properties of the original Carneades system, the negation relation of Carneades will therefore not be imported into the contrariness relation. Instead the contrariness relation in our argumentation system is used to let applicability conclusions for one argument defeat the acceptability of conflicting arguments, depending on the proof standards of their conclusions. This is essentially where the proof standards are encoded.

Definition 6.2 (Argumentation system corresponding to a CAES)**.** Given a CAES $C = \langle arguments, audience, standard\rangle$ with $audience = \langle assumptions, weight\rangle$ and propositional language $\mathcal{L}_{CAES}$, the corresponding argumentation system, AS, is a tuple $\langle\mathcal{L}, ^{-}, \mathcal{R}, \leqslant\rangle$ where:

- $\mathcal{L} = \mathcal{L}_{CAES} \cup$ argument nodes $\cup$ rule names,
- $^{-}$ consists of all tuples specified below,
- $\mathcal{R}_d = \bigcup_{a \in arguments} \mathcal{R}_{d_a}$,
- $\mathcal{R}_s = \bigcup_{a \in arguments} \mathcal{R}_{s_a}$,
- $\leqslant = \{(r, r) \mid r \in \mathcal{R}_d\}$.

For every argument $a = \langle P, E, c\rangle$ in $arguments$:

$$\begin{aligned} \mathcal{R}_{d_a} &= \{P \Rightarrow_{app_a} arg_a;\ arg_a \Rightarrow_{acc_a} c\} \cup \\ &\quad \{e_i \Rightarrow \neg app_a \mid e_i \in E\} \\ ^{-}(app_a) &= \{\neg app_a\} \end{aligned}$$

For every argument $a = \langle P, E, c\rangle$ in $arguments$ with $standard(c) = scintilla$:

$$\mathcal{R}_{s_a} = \emptyset$$

[53]This is probably a technical mistake of Gordon and Walton [86]. It can be fixed by slightly changing the definition of applicability of arguments, including the additional demand "there is not an assumption, $\overline{c}$ in the audience that is contradictory to the conclusion, c, of the argument".

For every argument $a = \langle P, E, c \rangle$ in *arguments* with *standard*(c) = *preponderance*:

$$\begin{aligned} \mathcal{R}_{s_a} &= \emptyset \\ {}^{-}(acc_a) &= \{arg_b \mid b = \langle P', E', \overline{c} \rangle \in arguments, \\ &\qquad weight(a) \leqslant weight(b)\} \end{aligned}$$

For every argument $a = \langle P, E, c \rangle$ in *arguments* with *standard*(c) = *clear-and-convincing*:

$$\begin{aligned} \mathcal{R}_{s_a} &= \{\rightarrow \neg acc_a \mid weight(a) \leqslant \alpha\} \\ {}^{-}(acc_a) &= \{arg_b \mid b = \langle P', E', \overline{c} \rangle \in arguments, \\ &\qquad weight(a) \leqslant weight(b) + \beta\} \\ &\cup \{\neg acc_a\} \end{aligned}$$

For every argument $a = \langle P, E, c \rangle$ in *arguments* with *standard*(c) = *beyond-reasonable-doubt*:

$$\begin{aligned} \mathcal{R}_{s_a} &= \{\rightarrow \neg acc_a \mid weight(a) \leqslant \alpha\} \\ {}^{-}(acc_a) &= \{arg_b \mid b = \langle P', E', \overline{c} \rangle \in arguments, \\ &\qquad weight(a) \leqslant weight(b) + \beta, \\ &\qquad \vee\, weight(b) \geqslant \gamma\} \\ &\cup \{\neg acc_a\} \end{aligned}$$

For every argument $a = \langle P, E, c \rangle$ in *arguments* with *standard*(c) = *dialectical-validity*:

$$\begin{aligned} \mathcal{R}_{s_a} &= \emptyset \\ {}^{-}(acc_a) &= \{arg_b \mid b = \langle P', E', \overline{c} \rangle \in arguments\} \end{aligned}$$

To illustrate the translation of one proof standard, notice that in a CAES, an argument a with standard *clear-and-convincing-evidence*, is unacceptable if either $weight(a) \leqslant \alpha$, or there is a contradictory applicable argument b for which $weight(a) \leqslant weight(b) + \beta$ holds. This is then translated by extending the set of contraries for the acceptability, acc_a, with an arg_b for **every** contradictory argument b for which $weight(a) \leqslant weight(b) + \beta$ holds. If the weight of a is less than α, there is also an inference $\rightarrow \neg acc_a$ added to the strict rules, R_s, that together with the contrary, $\neg acc_a$, will undercut the acceptability.

Having built up the corresponding argumentation system, we can now relate an argumentation theory and consequently an argumentation framework to a CAES.

Definition 6.3 (Argumentation theory corresponding to a CAES)**.** Given a CAES $C = \langle arguments, audience, standard\rangle$ with $audience = \langle assumptions, weight\rangle$ and propositional language $\mathcal{L}_{CAES}$ the argumentation theory AT corresponding to C is a tuple $\langle AS, KB, \preceq\rangle$ where:

- AS is the argumentation system corresponding to C according to Definition 6.2,
- KB is the knowledge base in the argumentation system AS corresponding to C according to Definition 6.1,
- $\preceq = \emptyset$.

Definition 6.4 (Argumentation framework corresponding to a CAES)**.** Given a CAES $C = \langle arguments, audience, standard\rangle$ with $audience = \langle assumptions, weight\rangle$, propositional language $\mathcal{L}_{CAES}$ and argumentation theory AT corresponding to C as given by Definition 6.3, the AF corresponding to C is the argumentation framework corresponding to AT as given by Definition 3.39.

To demonstrate our translation, we will show in detail how the CAES in Example 3.59 can be translated into its corresponding argumentation system generating the corresponding argumentation framework.

Example 6.5. First consider how the knowledge base in our argumentation system would correspond to the CAES given in Example 3.59. We have $K_n = assumptions = \{p_1, p_2, e_4\}$ while the other premises that are not a conclusion nor an assumption would be an issue premise, thus giving $K_i = \{p_3, e_1, e_2, e_3\}$.

Next we define the rules, $\mathcal{R}$, of the corresponding argumentation system. Every argument has a corresponding rule for applicability and for the acceptability of the conclusion, arguments containing an exception will have a corresponding inference rule generating an undercutter and finally rules with the proof standard *clear-and-convincing* or *beyond-reasonable-doubt* can have a strict rule undercutting the acceptability, if the weight of the argument is below α. For example, argument a_2 will generate:

$$\begin{aligned} R_{d_{a_2}} = \{\ & p_2, p_3 \Rightarrow_{app_{a_2}} arg_{a_2}; \\ & arg_{a_2} \Rightarrow_{acc_{a_2}} \neg c; \\ & e_2 \Rightarrow \neg app_{a_2}\} \\ R_{s_{a_2}} = \emptyset & \end{aligned}$$

Note that the set of strict rules is empty, because $weight(a_2) = 0.9 > 0.3 = \alpha$.

Given the previous rules, the structured arguments corresponding to the Carneades arguments can be visualised as follows:

$$\dfrac{\dfrac{p_1^n \qquad p_2^n}{arg_{a_1}}\ {\scriptstyle app_{a_1}}}{c}\ {\scriptstyle acc_{a_1}} \qquad\qquad \dfrac{\dfrac{p_2^n \qquad p_3^i}{arg_{a_2}}\ {\scriptstyle app_{a_2}}}{\neg c}\ {\scriptstyle acc_{a_2}}$$

$$\dfrac{\dfrac{p_2^n}{arg_{a_3}}\ {\scriptstyle app_{a_3}}}{\neg c}\ {\scriptstyle acc_{a_3}} \qquad\qquad \dfrac{\dfrac{}{arg_{a_4}}\ {\scriptstyle app_{a_4}}}{\neg c}\ {\scriptstyle acc_{a_4}}$$

$$\dfrac{e_1^i}{\neg app_{a_1}} \qquad\qquad \dfrac{e_2^i}{\neg app_{a_2}}$$

$$\dfrac{e_3^i}{\neg app_{a_3}} \qquad\qquad \dfrac{e_4^n}{\neg app_{a_4}}$$

Figure 6.1: Structured arguments corresponding to Example 3.59

These arguments contains several (sub)arguments. For example, the first argument can formally be written as follows:

$$A_1 : p_1 \qquad\qquad A_3 : A_1, A_2 \Rightarrow_{app_{a_1}} arg_{a_1}$$
$$A_2 : p_2 \qquad\qquad A_4 : A_3 \Rightarrow_{acc_{a_1}} c$$

What remains is the translation of the defeat relation, for which we will consider the argument A_4, related to a_1 in the CAES. The argument A_4 first of all is undercut on app_{a_1}, by means of the argument $e_1 \Rightarrow \neg app_{a_1}$. The other attackers are undercutters on the acceptability, acc_{a_1}. The proof standard of the conclusion c of a_1 is *preponderance*, while the weight of a_1 minus β is less than the other arguments in the CAES, so by the translation, the argument nodes arg_{a_2}, arg_{a_3} and arg_{a_4} will all be a contrary of acc_{a_1}. Then the (sub)arguments $p_2, p_3 \Rightarrow_{app_{a_2}} arg_{a_2}$, $p_2 \Rightarrow_{app_{a_3}} arg_{a_3}$ and $\Rightarrow_{app_{a_4}} arg_{a_4}$ will undercut A_4 on acc_{a_1}. Since the translation does not consider preferences, every attack that will be made, will result in a defeat.

Although there are some defeaters present that do not directly correspond to the original CAES, i.e. the exception $e_1 \Rightarrow \neg app_{a_1}$ and the undercutter

arg_{a_2}, this is not a problem since the arguments will not be deemed acceptable due to their issue premise. Thus in the end, the acceptable arguments will only be the non-issue premises, p_1, p_2 and e_4, the arguments for the applicability of a_1, a_3 and the argument for non-applicability of a_4 ($\neg app_{a_4}$), which is exactly what we want.

6.2.2 Translation properties

Now that that the argumentation framework corresponding to a CAES has been defined we can look at some interesting properties of the translation.

Well-foundedness

First of all it will be shown that an argumentation framework corresponding to a CAES contains no cycles and therefore is actually a well-founded argumentation framework. From this property and Theoorem 3.16, it can then immediately be deduced that every argumentation framework corresponding to a CAES induces a unique complete extension (which is grounded, preferred and stable).

Proposition 6.6. *Every argumentation framework corresponding to a (finite) CAES according to Definition 6.4 is well-founded.*

Proof. Given a CAES $C = \langle arguments, audience, standard \rangle$ with $audience = \langle assumptions, weight \rangle$ and corresponding argumentation framework AF. Assume AF is not well-founded and given that C is finite, there exists a sequence of arguments $A_1, \ldots, A_n$ in AF, such that $defeats(A_n, A_1)$ and for each $i < n$, $defeats(A_i, A_{i+1})$ hold.

Given that for our translation every Carneades argument a_i is assigned a unique argument node arg_i, we define ASPIC$^+$ arguments of the form $A_i : C_1, ...C_l \Rightarrow_{app_i} arg_i$, and the possible extension $A_{i'} : A_i \Rightarrow c$, to correspond with a_i. Note that by construction every $Conc(C_i)$ will be a premise of a_i and the set of contraries of arg_i will contain the exceptions of a_i. Then, denoting the direct and indirect parents of a node, c in a dependency graph as $ancestors(c)$, we will show that for every pair of arguments A_i, A_j for which $defeats(A_i, A_j)$ holds, $conc(a_i) \in ancestors(conc(a_j))$ holds for the corresponding Carneades arguments a_i and a_j. By proving this statement we can infer that for every pair of arguments $ancestors(conc(a_i)) \subseteq ancestors(conc(a_j))$ holds, entailing $ancestors(conc(a_i)) = ancestors(conc(a_j))$ by cyclity of the defeat sequence. Then by combining both statements we can infer that $conc(a_i) \in ancestors(conc(a_i))$, inducing a cycle in the dependency graph, contradicting our initial acylicity assumption of $arguments$, thereby proving what we want. (Notice that we are talking about defeat cycles in ASPIC$^+$ and dependency cycles in a CAES, which although related, are not of the same nature.)

We will prove the above property by considering the shapes of an arbitrary defeating argument A_i and its target A_j in our defeat sequence. By construction of our translation we can see that a defeating argument can only be of the following shapes:

Case 1: $A_i : \rightarrow \neg acc_j$. This argument cannot be defeated, and will therefore never be part of a defeat sequence, contradicting our assumption.

Case 2: $A_i : C_1, \ldots C_l \Rightarrow_{app_i} arg_i$ (with possible superargument: $A_i' \Rightarrow_{acc_i} c$). A_i can undercut an argument of the following form: $A_{subj} : D_1 \ldots D_m \Rightarrow_{app_{subj}} arg_{subj} \Rightarrow_{acc_{subj}} \overline{c}$, assuming that $arg_i \in \overline{acc_{subj}}$. However, a conclusion $\overline{c}$ cannot defeat the the next argument A_k in the defeat chain and therefore A_i must have defeated a proper subargument of A_j. So A_j either extended A_{subj} to an argument that is an exception to the next argument ($\neg app_j$) or it was extended to an argument that has an argument node (arg_j) as the conclusion (note that A_{subj} can be one of the many subarguments of A_j). In both cases we see that the argument a_j corresponding to A_j must have had $\overline{c}$ as a premise, inducing that a_j has c and $\overline{c}$ as dependencies. This establishes $conc(a_i) \in ancestors(conc(a_j))$.

Case 3: $A_i : e_i \Rightarrow \neg app_j$ with $e_i \in assumptions$. This argument cannot be defeated, and will therefore never be part of a defeat sequence, contradicting our assumption.

Case 4: $A_i : C_1, \ldots C_l \Rightarrow_{app_i} arg_i \Rightarrow_{acc_i} e_i \Rightarrow \neg app_j$. A_i can undercut A_j, on its subargument $D_1 \ldots D_m \Rightarrow_{app_j} arg_j$. This does not have to be a proper subargument of A_j. For $\neg app_j$ to be a contrary of app_j, e_i has to be an exception of the argument corresponding to A_j, a_j. This immediately establishes that $conc(a_i) \in ancestors(conc(a_j))$.

□

The next result follows directly from Proposition 6.6 and Theorem 3.16:

Corollary 6.7. *Every argumentation framework corresponding to a CAES according to Definition 6.4 has exactly one complete extension which is grounded, preferred and stable.*

Here we can see that contrary to the claim of Brewka and Gordon [22] – namely that modelling Carneades in Dung's approach could not be done without obtaining a cycle-free AF – we have proved that an argumentation framework corresponding to a CAES *is* well-founded and thus cycle-free. This means that the corresponding argumentation frameworks always induce a unique Dung extension which is the same in all Dung's semantics.

Carneades' semantics is therefore essentially a single status assignment approach.

We have seen that in Carneades the defeat relation that is generated through the translation depends on the audience and the proof standards. This use of audience is very similar to (and inspired by) the approach taken in value-based argumentation frameworks [13]. More interestingly, just as the uniqueness of preferred extensions in VAF's with respect to a single audience is guaranteed, in the translation of Carneades there is also a unique complete extension.

Computational complexity

The time to compute the extension of a well-founded argumentation framework can be determined to quadratic, by verifying that it is possible to topologically sort the acyclic dependency graph in $\Theta(|V|+|E|)$ (cf. page 549–552 of Cormen et al. [41]) and by checking that it is possible to compute the grounded extension in $\Theta(|V|+|E|)$ (by computing acceptability in order of dependency). We can therefore deduce that if the translation is polynomial, evaluating a CAES through our translation is also polynomial. Although this might seem immediate from our translation, there are some subtleties in the actual step that generates arguments. ASPIC$^+$ only declaratively states which arguments are to be generated from the argumentation system, but with a naive implementation/algorithm, the argumentation system corresponding to a CAES would actually generate an exponential number of arguments.

Example 6.8 (Exponential explosion)**.** Consider a class of CAES with $2n$ arguments, such that $arguments = \{a_{11}, a_{12}, a_{21}, a_{22}, \ldots, a_{n1}, a_{n2}\}$. Here the ith pair of arguments has the same conclusion c_i, with premises that depend on the previous, $i-1$th, pair of arguments. So we are building a large chain with pairs of arguments dependent on the previous pair. The start of the chain is not dependent on a previous conclusion, so $a_{11} = \langle\{p_{11}\}, \emptyset, c_1\rangle$, $a_{12} = \langle\{p_{11}\}, \emptyset, c_1\rangle$. The next arguments, for $i > 1$: $a_{i1} = \langle\{p_{i1}, c_{i-1}\}, \emptyset, c_i\rangle$, $a_{i2} = \langle\{p_{i2}, c_{i-1}\}, \emptyset, c_i\rangle$. And finally we have that $assumptions = \{p_{11}, p_{12}, \ldots, p_{n1}, p_{n2}\}$, $weight(a_i) = 0.5$ and $standard(a_i) = scintilla$.

Now consider the corresponding defeasible rules (leaving out rule names):

$$\begin{aligned} R_d = \{&p_{11} \Rightarrow arg_{a_{11}}; arg_{a_{11}} \Rightarrow c_1; \\ &p_{12} \Rightarrow arg_{a_{12}}; arg_{a_{12}} \Rightarrow c_1; \\ &\qquad\qquad\vdots \\ &p_{n1}, c_{n-1} \Rightarrow arg_{a_{n1}}; arg_{a_{n1}} \Rightarrow c_n; \\ &p_{n2}, c_{n-1} \Rightarrow arg_{a_{n2}}; arg_{a_{n2}} \Rightarrow c_n\} \end{aligned}$$

Although the ith argument only needs the conclusion of one of the $i-1$th arguments to be acceptable, when generating arguments we will generate every possible combination of subarguments, thereby generating 2^n arguments.

This exponential explosion is caused due to the implicit linking of arguments in Carneades that is made explicit when constructing arguments from the corresponding argumentation system. However, due to the acyclicity of the *arguments*, this explicit linking is not needed to compute the acceptable conclusions. The exponential explosion can be solved by the following (sketched) polynomial algorithm.

Definition 6.9 (CAES argument generation)**.**

1. *generatedArgs* $= \emptyset$.
2. *sortedArgs* $=$ Topological sort of *arguments* on its dependency graph.
3. **while** *sortedArgs* $\neq \emptyset$:
   (a) Pick the first argument in *sortedArgs*. Remove all arguments from *sortedArgs* that have the same conclusion, c, and put them in *argSet*.
   (b) Translate *argSet* and generate arguments, building on previously *generatedArgs* as subarguments, and put the generated arguments in *tempArgs*.
   (c) If present, pick one acceptable argument in *tempArgs* that has the conclusion c and add it to *generatedArgs*.
   (d) *argSet* $=$ *tempArgs* $= \emptyset$.

We leave it as future work to formally verify the time/space complexity of the translation.

This exponential explosion also sheds some light on the complexity of argument evaluation in Carneades. An important concept in the definition of Carneades is the "concept of a proof", where evaluating a proof (a CAES) should be possible in a tractable time. This tractability can now be proven by verifying the polynomial complexity of the translation.

Correspondence results

We can now prove the main theorem of this chapter, namely that every argumentation framework that corresponds to a CAES preserves the properties we would expect.

Theorem 6.10. *Let C be a CAES, $\langle arguments, audience, standard\rangle$, $\mathcal{L}_{CAES}$ the propositional language used and let the argumentation framework corresponding to C be AF. Then the following holds:*

1. *An argument $a \in arguments$ is applicable in C iff there is an argument contained in the complete extension of AF with the corresponding conclusion arg_a.*

2. *A propositional literal $c \in \mathcal{L}_{CAES}$ is acceptable in C or $c \in assumptions$ iff there is an argument contained in the complete extension of AF with the corresponding conclusion c.*

Proof. We prove 1. and 2. by induction on the number of arguments, n, in the CAES C.

For $n = 0$, there is neither an (applicable) argument nor an acceptable proposition in C. The knowledge base KB corresponding to C will only contain axioms in $\mathcal{K}_n$ for each assumption in C and issue premises in $\mathcal{K}_i$ for other propositional literals in $\mathcal{L}_{CAES}$. The defeasible and strict rules $\mathcal{R}_d$ and $\mathcal{R}_d$ will be empty. Therefore all arguments on the basis of KB will either be an argument using an issue premise and thus not in the complete extension of the argumentation framework (CE_{AF}), or an argument containing only an axiom and therefore in CE_{AF}. So CE_{AF} contains an argument with corresponding conclusion for every assumption in C and no argument with a conclusion of the form arg_a, therefore every conclusion of an argument in CE_{AF} is an assumption, making 1. and 2. hold.

Assuming 1. and 2. hold for n arguments we consider a CAES, C, with $n+1$ arguments. Due to acyclicity of *arguments* there is at least one argument $a = \langle P, E, c\rangle \in arguments$ for which the conclusion c is not contained in the premises or exceptions of another argument in *arguments*. Now consider the CAES C' constructed from C by taking $arguments' = arguments\backslash\{a\}$ and let AF' be the corresponding argumentation framework. We then obtain a CAES with n arguments for which the induction hypothesis holds.

(1. $\Leftrightarrow$) We must prove that for all (not) applicable arguments b in C there is (not) an argument in CE_{AF} with conclusion arg_b. For all arguments in C' this follows from the induction hypothesis. By our selection of a, the applicability of a does not influence applicability of the arguments that were in C'. In the translation of a to ASPIC$^+$, corresponding arguments

for arg_a will not defeat arguments in AF'. Then by the satisfaction of the directionality criterion of complete semantics [10] it follows that all arguments acceptable in $CE_{AF'}$ are also in CE_{AF}, thus leaving correspondence of the applicability of a in C to prove. Acceptability of the premises and exceptions of a is not influenced by the applicability of a, and thus by the induction hypothesis on C' and the directionality criterion, premises and exceptions of a are acceptable in C or part of the *assumptions* iff there is an argument contained in CE_{AF} with the corresponding conclusion. By our translation, we know that $P \Rightarrow_{app_a} arg_a$ and the set $\{e_i \Rightarrow \neg app_a \mid e_i \in E\}$ are in $\mathcal{R}_d$.

Now suppose first that a is applicable in C. Then by the induction hypothesis for all premises $p_i \in P$ there exists an argument A_i in CE_{AF}. We prove that if for $P = \{p_1, \ldots, p_n\}$ the argument $A_1, \ldots, A_n \Rightarrow_{app_a} arg_a$ also is in CE_{AF}. By conflict-freeness of CE_{AF}, no defeater of any A_i is in CE_{AF} so it suffices to prove that no argument for $\neg app_a$ is in CE_{AF}. By applicability of a and the induction hypothesis, for no $e \in E$ there exists an argument in CE_{AF} with conclusion e and thus this follows directly.

Suppose next that a is not applicable in C. Then by the induction hypothesis either not all A_i are in CE_{AF} or for some $e \in E$ an argument A_e with conclusion e is in CE_{AF}. In the first case $A = A_1, \ldots, A_n \Rightarrow_{app_a} arg_a \notin CE_{AF}$ by closure of CE_{AF} under subarguments (Proposition 6.1 of [145]). In the second case A for arg_a is defeated by A_e so $A \notin CE_{AF}$ by conflict-freeness of CE_{AF}.

(2. $\Rightarrow$) If d is an assumption, then by translation $d \in \mathcal{K}_n$ and thus there is an argument A with corresponding conclusion d in CE_{AF}.

Otherwise, we must prove that if a propositional literal $d \in \mathcal{L}_{CAES}$ is acceptable in C then there is an argument contained in CE_{AF} with the corresponding conclusion d. For the CAES C' defined before, the induction hypothesis holds and therefore acceptable literals (or literals in assumptions) of C' have an argument with corresponding conclusion in $CE_{AF'}$. By our selection of a and acyclicity of *arguments* we know that a only influences the acceptability of its conclusion and negation, c and $\overline{c}$. Then, again by the directionality criterion, we have (2. $\Rightarrow$) left to prove for c and $\overline{c}$ in C.

Suppose a is not applicable, then by (1.), no argument for arg_a will be in CE_{AF} and therefore neither will be an argument for its conclusion d in CE_{AF}. This prevents a from influencing acceptability of c (and d), letting (2. $\Rightarrow$) hold.

If a is applicable and $d = c$, then by (1.) there exists an argument A_1 with conclusion arg_a in CE_{AF}. By translation $arg_a \Rightarrow_{acc_a} c \in \mathcal{R}_d$, allowing A_1 to be extended to an argument A_2 for c. If c is acceptable in C, then its proof standard is satisfied. Then by translation there will be neither a contrary of acc_a in $^-$ nor a strict rule of the form $\rightarrow \neg acc_a \in \mathcal{R}_s$ and therefore there will be no undercutter of A_2 in CE_{AF} on the final inference. Furthermore, since A_1 is in CE_{AF}, by conflict-freeness no defeater of A_1 is in CE_{AF}. Thus $A_2 \in CE_{AF}$. Similarly, if a makes the proof standard for $\overline{c}$ unsatisfiable in C, by construction of AF, A_1 will defeat any argument b with conclusion $\overline{c}$ on its inference rule arg_b. So by conflict-freeness no such argument will be in CE_{AF}, correctly preserving acceptability of $\overline{c}$.

If a is applicable and $d = \overline{c}$, then we only need to consider the influence of the applicability of a on the acceptability of c, since the acceptability of $\overline{c}$ is irrelevant. First, by (1.) there exists an argument A_1 with conclusion arg_a in CE_{AF}. Consider $standard(c) =$ *clear-and-convincing*. Take an applicable argument b for c in C' with the highest weight. Since c is acceptable, b satisfies all (weight) conditions for *clear-and-convincing*. By the induction hypothesis there is an argument B_i, with $Conc(B_i) = c$, in $CE_{AF'}$. Then notice A_1 defeats B_i iff arg_a is a contrary of acc_b which holds iff $weight(b) \not> weight(a) + \beta$, correctly preserving acceptability of c.

(2. $\Leftarrow$) Proof by contraposition. First, $d \notin$ *assumptions* and therefore $d \notin \mathcal{K}_n$. Similar to the proof of (2. $\Rightarrow$), (2. $\Leftarrow$) holds if a is not applicable or d is neither c nor $\overline{c}$.

So assume a is applicable and $d = c$. Since c is not acceptable, the proof standard of c is not satisfied in C. Consider for example $standard(c) =$ *clear-and-convincing*. Then either $weight(a) \leqslant \alpha$ or $weight(a) \leqslant weight(b) + \beta$ for another applicable argument b with conclusion $\overline{c}$. Therefore the argumentation system either has $\rightarrow \neg acc_a \in \mathcal{R}_s$ or otherwise $arg_b \in {}^{-}(acc_a)$. Finally the AF on the basis of this argumentation system will either have an argument of the form $\rightarrow \neg acc_a$, or by applicability of b and the induction hypothesis, arg_b will be in CE_{AF} and defeats any argument using the defeasible inference acc_a. Concluding any argument constructed for the acceptability of c will be defeated and thus by conflict-freeness not in CE_{AF}.

If $d = \overline{c}$ and c is not acceptable, then applicability of a will not influence acceptability of c. □

From part 2. of Theorem 7.2 we can directly relate the theory of a CAES with the corresponding argumentation framework:

Corollary 6.11. *A proposition p is part of the theory of C iff p is contained in the closure under propositional logic of the complete extension of AF.*

We have now formally shown that the argumentation framework corresponding to a CAES keeps the properties we wanted to preserve. This proves that Carneades can be faithfully modelled in Dung's argumentation frameworks.

It can even be argued that it is a too faithful correspondence, keeping inconsistencies of the original model in the final translation. The inconsistencies caused by the use of the scintilla of evidence proof standard might suggest a change in the definition of proof standard in Carneades, for instance, by disallowing proof standards that make contradictory conclusions true at the same time.

Regardless, the intermediate translation of a CAES into an argumentation system allows for an easy reparation. We can import the original negation relation of the CAES, generating attacks for any conflicting conclusion. To be precise, given an argumentation system AS corresponding to a CAES, we can make the new contrariness relation: $^{-}_{AS'} = {}^{-}_{AS} \cup {}^{-}_{CAES}$. This will generate additional defeats in the final argumentation framework thereby breaking the correspondence results, however it will ensure that no inconsistencies can be generated in the final theory.

Ambiguity-blocking and ambiguity-propagating

The stage-specific part of Carneades can be called "ambiguity-blocking" in contrast to "ambiguity-propagating" (see Section 7.1 of Gordon et al. [83]). Here a non-monotonic logic is ambiguity-blocking if, when a conflict between two lines of reasoning with contradictory conclusions cannot be resolved, both lines of reasoning are cut-off and neither of the conclusions can be used for further reasoning. In such logics it may happen that other lines of reasoning remain undefeated even though one of the cut-off lines of reasoning interferes with it and is not weaker.

Consider the following example containing an ambiguity between q and $\neg q$, that does not interfere with the inference of $\neg s$ even though $\neg q$ is used as an argument for s.

Example 6.12. Consider the CAES $C = \langle arguments, audience, standard\rangle$ and $audience = \langle assumptions, weight\rangle$ with:

$$\begin{aligned}
arguments &= \{a_1, a_2, a_3, a_4\},\\
a_1 &= \langle\{p\}, \emptyset, q\rangle,\ a_2 = \langle\{r\}, \emptyset, \neg q\rangle, \rangle,\\
a_3 &= \langle\{\neg q\}, \emptyset, s\rangle,\ a_4 = \langle\{t\}, \emptyset, \neg s\rangle,\\
assumptions &= \{p, r, t\},\\
weight(a_1) &= weight(a_2) = weight(a_3) = weight(a_4) = 0.5,\\
standard(q) &= standard(\neg q) = standard(s) = standard(\neg s) = preponderance.
\end{aligned}$$

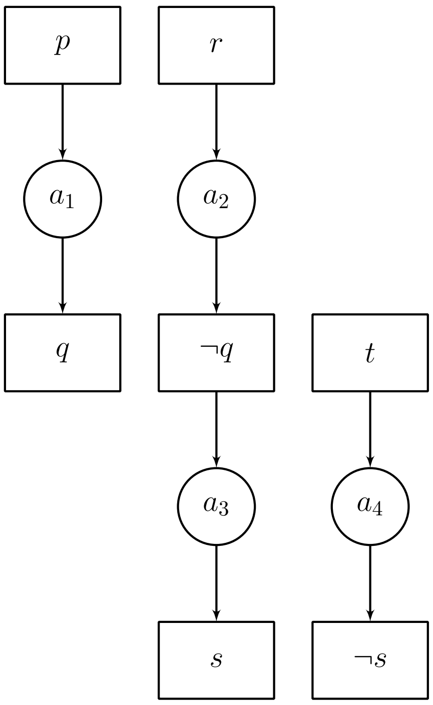


Figure 6.2: Ambiguity-blocking in Carneades

With the proof standard of q, $\neg q$, s and $\neg s$ being *preponderance* we can see that q, $\neg q$ and s will not be acceptable, but $\neg s$ will be acceptable. Now consider a naive, direct translation of the arguments into defeasible inference rules in ASPIC$^+$, i.e. $\mathcal{K}_n = \{p, r, t\}$ and $\mathcal{R}_d = \{p \Rightarrow q, r \Rightarrow \neg q, \neg q \Rightarrow s, t \Rightarrow \neg s\}$. This translation would instead make no corresponding arguments acceptable.

The translation according to Definition 6.4 solves this by using an explicit argument node, yielding undefeated undercutters for the acceptability of q and $\neg q$, thereby yielding an undefeated undercutter for the argument for s constructed by using the argument for q, so that $\neg s$ is acceptable in the corresponding AF.

The main difficulty in finding the translation of Carneades to ASPIC$^+$, was dealing with the ambiguity-blocking nature of Carneades, while ASPIC$^+$ is ambiguity-propagating. We have largely solved this problem by introducing additional argument nodes, allowing for an explicit representation of applicability and acceptability. We note that to our knowledge, we are the first to have achieved a translation of an ambiguity-blocking non-monotonic system to a standard Dung semantics.

Rationality postulates

We have shown in Section 6.2.1 that Carneades can be reduced to an ASPIC$^+$ argumentation theory. One advantage of going through this intermediate step is the possibility of applying existing results of ASPIC$^+$ regarding rationality postulates (see Section 3.6) to translated Carneades. To verify the rationality postulates we depend on the properties of argumentation theories as defined in Definition 3.46.

Proposition 6.13. *Given a CAES C with corresponding argumentation theory AT and corresponding argumentation framework AF, then the following rationality postulates hold for the unique extension of AF:*

1. *Closure under subarguments,*
2. *Closure under strict rules,*
3. *Direct consistency,*
4. *Indirect consistency.*

Proof. The first two rationality postulates follow directly from the translation of C to an ASPIC$^+$ argumentation theory and Propositions 6.1 and 6.2 of Prakken [145]. For the other two postulates, by Theorem 6.9 and 6.10 of Prakken [145], we will have to prove our argumentation theory to be closed under contraposition or transposition, axiom-consistent, well-formed and finally have a reasonable argument ordering (see Definition 3.48).

The corresponding AT of C has only one type of strict rule: $\rightarrow \neg acc_a$, for every argument a in $arguments$. Closure under contraposition (and transposition) is immediately satisfied by the lack of strict rules starting with a premise acc_a. Premises from $\mathcal{K}_n$ cannot be used as a premise for a strict rule, trivially satisfying closure under strict rules, given the consistency of $\mathcal{K}_n$. If φ is a contrary of ψ, then by construction of the translation, ψ is always of the form app_a or arg_a. Neither are in $\mathcal{K}_n$, nor a consequent of a strict rule, thus satisfying well-formedness. Finally, preferences are not used in the corresponding AT, allowing us to take any reasonable argument ordering. □

An important thing to note here is that although we have proven consistency for the extension of a corresponding argumentation framework, this consistency is relative to the contrariness relation of the argumentation theory. The achievement of gaining consistency in an argumentation framework corresponding to a CAES is mainly due to leaving out the negation relation of that CAES, in a sense circumventing the problem.

6.2.3 Generalisation of the translation

Important future work mentioned by Gordon and Walton [86], is to generalise Carneades to cycle-containing structures. Although it was claimed by Brewka and Gordon [22] that Carneades would need a cyclic representation in other frameworks, such as Dung's argumentation frameworks, our translation of Carneades translates to cycle-free, or well-founded argumentation frameworks. This same well-foundedness allows for an easy extension of Carneades's argument set to a possibly cycle-containing structure.

Since our translation of a CAES to an argumentation framework does not depend on possible cycles in the set of arguments, we can use the same translation for cycle-containing Carneades argument evaluation structures and deal with the resulting cycles by using the standard Dung semantics.

Definition 6.14. Given a CAES $C = \langle arguments, audience, standard\rangle$ without the acyclicity restriction, $\mathcal{L}_{CAES}$ the propositional language used and let the argumentation framework corresponding to C be AF. Then for $s \in \{complete,\ preferred, grounded, stable\}$:

- An argument $a \in arguments$ is applicable in C under sceptical (credulous) s semantics iff all (some) s extensions of AF contain an argument with conclusion arg_a.
- A propositional literal $c \in \mathcal{L}_{CAES}$ is acceptable in C or $c \in assumptions$ under sceptical (credulous) s semantics iff all (some) s extensions of AF contain an argument with conclusion c.

We will demonstrate our generalisation of Carneades by translating Example 2 of Brewka and Gordon [22] to an argumentation framework, showing intermediate steps.

Example 6.15 ((Cycle example))**.** Assume we have two possible destinations in mind for a summer vacation, Greece and Italy, but cannot afford to visit both destinations. We could formalise this as following. Let $arguments = \{a, b\}$ with:

$$a = \langle \emptyset, \{It\}, Gr\rangle, b = \langle \emptyset, \{Gr\}, It\rangle$$

These arguments contain an exception cycle and therefore cannot directly be handled by Carneades. We can give the example semantics by using the generalisation of our translation. The translation of this CAES would give the following argument trees:

$$\dfrac{\dfrac{\dfrac{}{arg_a}\,{\scriptstyle app_a}}{Gr}\,{\scriptstyle acc_a}}{\neg app_b} \qquad\qquad \dfrac{\dfrac{\dfrac{}{arg_b}\,{\scriptstyle app_b}}{It}\,{\scriptstyle acc_b}}{\neg app_a}$$

Figure 6.3: Greece versus Italy argument trees

Which can be written formally:

$$\begin{array}{ll}
A_1 : \Rightarrow_{app_a} arg_a & B_1 : \Rightarrow_{app_b} arg_b \\
A_2 : A_1 \Rightarrow_{acc_a} Gr & B_2 : B_1 \Rightarrow_{acc_b} It \\
E : A_2 \Rightarrow \neg app_b & F : B_2 \Rightarrow \neg app_a
\end{array}$$

From this formal description of arguments, together with the undercuts on applicability and acceptability we would get the following argumentation framework:

$$\begin{tikzcd}
A_1 & B_1 \\
A_2 & B_2 \\
E \arrow[uur] \arrow[ur] \arrow[r, leftrightarrow] & F \arrow[uul] \arrow[ul]
\end{tikzcd}$$

Figure 6.4: Greece versus Italy argumentation framework

The argumentation framework above can be evaluated through Dung's semantics. For instance, under credulous stable and preferred semantics, both *Gr* and *It* are acceptable. Under sceptical stable, sceptical preferred, or grounded semantics both would not be acceptable. These results are similar to the results in the generalisation by Brewka and Gordon [22].

6.3 Related work

Concurrent to the work done in the papers by van Gijzel and Prakken [77, 76], there have been translations of Carneades to other argumentation approaches. First of all, there is the translation of Carneades to abstract dialectical frameworks by Brewka and Gordon [22]. In this translation premises and exceptions, respectively, have a support and attack relation with the argument node, much in the same way that subarguments and undercuts are used in our translation. Carneades' proof standards are encoded as acceptance conditions from the argument node, supporting the conclusion and attacking the contradictory conclusion.

Although the translation of Brewka and Gordon clarified the relation between Carneades and abstract argumentation by relating it to ADFs, one of the main concerns about this translation was that it needed the full power of abstract dialectical frameworks, thus obscuring the direct relation with Dung's argumentation frameworks. This connection has now been made explicit by the paper of Brewka, Dunne and Woltran [19], developing a translation of ADFs to AFs using boolean networks [53]. The paper concerns itself mostly with the computational complexity of the translation, to keep a polynomial complexity in both size and time. However this translation introduces additional technical nodes in the final argumentation framework that have no intuitive meaning. So even though the translation gives a formal connection between the two argumentation models, the intuitive relation is mostly lost.

Recently, Carneades has been translated to Defeasible Logic [130] by Governatori [88]. Defeasible logic is a computational approach to non-monotonic reasoning with an argumentation-like flavour. Defeasible Logic has the possibility to handle both ambiguity-blocking and ambiguity-propagating behaviour, allowing for a rather direct representation of Carneades' proof standards. The translation by Governatori maps proof standards to a single inference mechanism, giving a natural representation of the proof standards.

While Governatori thus establishes an intuitive relation between Carneades and Defeasible Logic, he only partly relates Carneades to abstract argumentation, since only the ambiguity-propagating part of Defeasible Logic has an established direct formal relation with Dung's argumentation frameworks. Its ambiguity-blocking variant has instead been translated to a Dung-like semantics using a different notion of acceptability [89].

6.4 Conclusions and future work

This chapter has shown that Carneades can be reconstructed, through ASPIC$^+$, as Dung's abstract argumentation frameworks. We have seen that the idea of varying proof standards can be modelled within a Dungean approach, while retaining a correspondence of properties between both systems. These results show that Dung's approach to argumentation is able to model complex argumentation issues such as proof standards. Furthermore, by first translating Carneades through an ASPIC$^+$ argumentation theory, we were able to prove and instantly gain a number of useful results. First of all, we were able to use results about rationality postulates from Prakken [145] and directly apply these to the translated version of Carneades, proving consistency and strict closure of extensions. The translation also allows us to fully exploit the power of an ASPIC$^+$ argumentation theory, providing us for instance, with an explicit distinction between strict and defeasible inference rules. So in addition to providing a correspondence, the translation allows us to integrate Carneades with an extra set of tools provided by ASPIC$^+$.

An important property of our reconstruction of Carneades, is that our modelling gives a cycle-free argumentation framework, thus always inducing a unique Dung extension which is the same in all Dung's semantics. This shows that Carneades is essentially a single status assignment approach. This property allowed us to generalise Carneades to cycle-containing structures by using Dung's standard grounded, preferred and stable semantics, thereby addressing the issue put forward by Gordon and Walton [86]. This generalisation is done much in the same way as by Brewka and Gordon [22].

We note that our translation enables a standard Dung semantics for an 'ambiguity-blocking' non-monotonic logic (see Gordon et al. [83], Section 7.1); to our knowledge, we are the first to have achieved such a result.

Through this chapter and by the work of Prakken [145], several approaches to structured argumentation have been developed and subsequently related through the ASPIC$^+$ framework. Although the theoretical relations between these approaches have thus been clarified, we have seen that the actual step of generating arguments has not been given a concrete, efficient implementation. A useful path to take for future research would therefore be to develop a class of efficient argument generation algorithms for ASPIC$^+$. The possibility to efficiently generate arguments for a large class of argumentation approaches would give a better integration of these approaches.

The translation of Carneades to ASPIC$^+$ gives us access to the full power of an ASPIC$^+$ argumentation theory, thereby gaining the possibility to use strict and defeasible rules or use different types of knowledge. This additional power could also be used to show how the concept of argument generators [80] and the existing argument schemes of Carneades [85] relate to structured argumentation by translating them into schemes for defeasible inference rules (following the suggestion of Prakken [145] that most argument schemes can be seen as such). This would make the relation between ASPIC$^+$ and the Carneades argumentation system as a whole, more complete.

Our results raise the question whether it would now be better to use ASPIC$^+$ directly, instead of Carneades, to model argumentation when variable proof standards and the other features of Carneades are required. The answer depends on whether Carneades is sufficient as a model of reasoning with variable proof standards. Prakken and Sartor [154] claim that Carneades' ambiguity-blocking nature prevents an adequate modelling of the distinction between the burdens of production and persuasion. If they are right, then there is reason to change Carneades in the direction of ASPIC$^+$.

Chapter 7

Towards a framework for the implementation and verification of translations between argumentation models

This chapter constructs a framework for developing structured argumentation models and translations between models (given intertranslatability of models), building on the implementations of Dung's abstract argumentation frameworks discussed in Chapter 4 and the Carneades argumentation model discussed in Chapter 5. Given the implementations of the two models, we can exploit the translation given in Chapter 6 to derive and subsequently implement a translation from Carneades into AFs, obtaining one of the first implementations of a translation from a structured into an abstract argumentation model. A methodology to quickly test and formally prove desirable properties of such implementations using a theorem prover is furthermore provided, demonstrating Haskell implementations of correspondence properties and a formalisation of Dung's AFs into the Agda theorem prover, obtaining the first fully machine-checkable formalisation of an argumentation model. The final result is a verified pipeline from the structured model Carneades into existing efficient SAT-based implementations [56] of Dung's AFs. All work is open source, publicly available and immediately installable[54].

This chapter is structured as follows. Section 7.1 discusses a direct algorithmic translation from Carneades into Dung. In Section 7.2 we consider

[54]The implementations and formalisations are fully documented and can be found online together with a collection of additional examples: `http://www.cs.nott.ac.uk/~bmv/COMMA/`.

quick testing and complete formalisation of argumentation models and correctness properties. Section 7.3 discusses related work. Section 7.4 concludes, tying all strands of work together into one verified pipeline.

7.1 An algorithm and implementation for the translation of Carneades into Dung

Many of the structured approaches in argumentation can be translated into abstract models like Dung's AFs [122, 77, 88, 19, 22, 76]. In particular, it is known that Carneades can be translated into ASPIC$^+$ [77, 76], which in turn can be translated into AFs [145] (see Chapter 6). However, up until now, such translations, especially for models that are further removed from Dung's AFs, have rarely been *implemented* (see Section 7.3). We have taken two steps towards remedying this situation. Firstly, we give an algorithm for translating a structured model, Carneades, directly into an abstract model, Dung's AFs by deriving a translation based on the work in Chapter 6. Secondly, this translation is implemented and discussed in this section.

7.1.1 A practical algorithm for the translation of Carneades into Dung

Before introducing the algorithm, we need to define what is required of a translation from Carneades into Dung.

Evaluating a Carneades model yields two results: a set of applicable arguments and a set of acceptable conclusions (Section 3.4). The target AF thus needs to include arguments representing both. Our algorithm gradually builds up Dung arguments and an attack relation, by gradually translating the applicability and acceptability part of each Carneades argument.

The variable *generatedAF* is initialized with one argument, *defeater*, used to attack translated inapplicable arguments, together will all *assumptions* of the Carneades model. All Carneades arguments are then grouped by conclusions and subsequently sorted topologically based on their dependency graph. The third and final step first translates the applicability part for all arguments pro and con a conclusion c (based on the previously translated dependencies). After this, the acceptability part can be translated — all proof standards in Carneades, for some proposition c, depend on the applicability of arguments pro and con c — and both translated results put into *generatedAF*. This is repeated until *sortedArgs* is empty.

Algorithm 7.1. Algorithm for translation from Carneades into Dung's AFs

```
1. $generatedAF = \langle\{defeater\} \cup assumptions, \emptyset\rangle$.
2. $sortedArgs$ = Topological sort of $arguments$ on its dependency graph.
3. while $sortedArgs \neq [\,]$:
    (a) Pick the first argument in $sortedArgs$. Remove all arguments from
        $sortedArgs$ that have the same conclusion, $c$, and put them in
        $argSet$.
    (b) Translate applicability part of arguments in $argSet$, building on
        previous $generatedAF$, putting generated arguments/attacks in
        $tempAF$.
    (c) $argSet = \emptyset$.
    (d) Repeat (a) through (c) for the arguments for opposite conclusion
        $\overline{c}$.
    (e) Add all arguments for the applicability part from $tempAF$ to
        $generatedAF$.
    (f) Translate the acceptability part of $c$ and $\overline{c}$ based on arguments in
        $tempAF$. Add the results and $tempAF$ to $generatedAF$.
    (g) $tempAF = \langle\emptyset, \emptyset\rangle$.
```

Note that *arguments* refers to the set of arguments in a CAES.

More precisely, step 3.(b) requires us to add an argument to the AF for each argument for c.

If the translated premises of this argument are known *not* to hold, or if one the translated exceptions *is* known to hold, we add an attack from *defeater* to this argument. The acceptability step in 3.(f) adds arguments to the AF for the conclusion c and $\overline{c}$. Here the acceptability will again be translated by adding an attack from *defeater* to c if the proof standard is not upheld, given the translation of the applicability of c and $\overline{c}$.[55]

Using this algorithm we can get a one-one mapping from the union of arguments and conclusions to arguments in an AF, with the exception of one administrative node, *defeater*, that can easily be filtered out.

7.1.2 Step by step translation of an example

Figure 7.1 defines three arguments (leaving out weights and proof standards):

- an argument in favour of the proposition *murder* given that the propositions *kill* and *intent* hold

[55] A more intuitive translation could pick the con argument with the maximum weight instead, but this is less efficient.

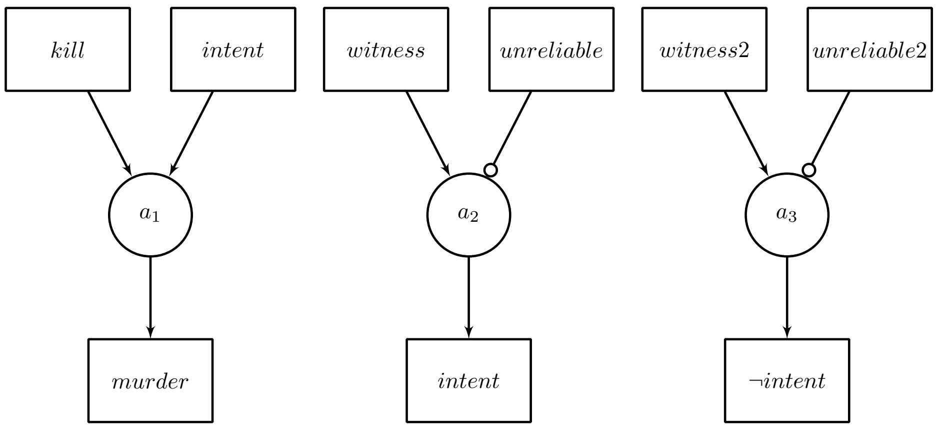


Figure 7.1: Three arguments in a murder case in Carneades

- an argument in favour of the proposition *intent* given that there is a *witness* and the exception (denoted by the circle) *unreliable* does not hold
- a self-defence argument from a second *witness2* that *intent* does not hold, given that the exception of that witness being *unreliable2* does not hold.

The set of propositions {*kill*, *witness*, *witness2*, *unreliable2*} are assumed.

Referring back to Definition 3.52, we can see that every proposition, including its negation, is present in the dependency graph for the set of three arguments defined above: see Figure 7.2. For reasons of presentation, we have left out the negations: all links and nodes are exactly the same for the contrary of each literal. The dependency graph makes it clear that it is necessary to translate the propositions *unreliable*, *unreliable2*, *witness* and *witness2* before *intent* and its two arguments (one pro and one con) can be translated. Figure 7.3 shows the resulting translation (all proposition names, including defeater, shortened to first letter).

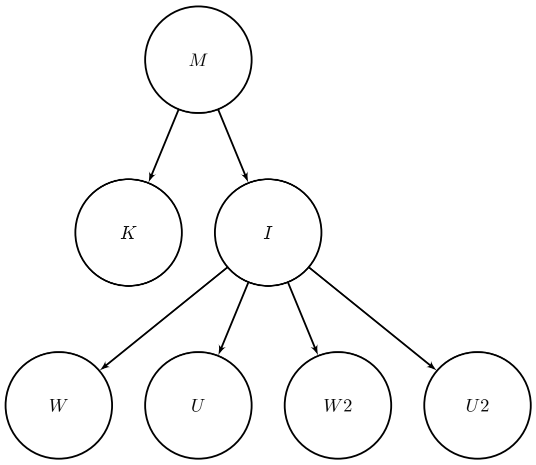


Figure 7.2: The dependency graph corresponding to the three arguments of Figure 7.1

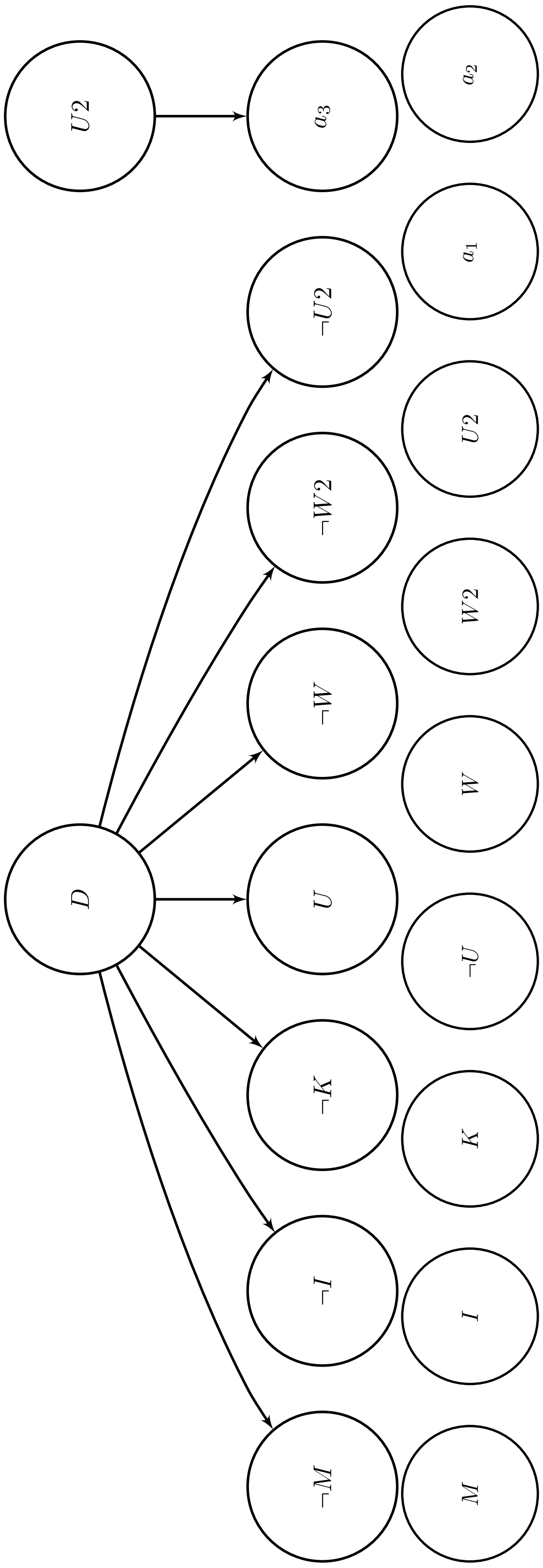


Figure 7.3: The Dung AF corresponding to the translation of the three arguments of Figure 7.1

7.1.3 Our implementation of the algorithm

To encode Carneades' arguments and propositions into the translated argumentation framework we could generate *String* labels from the arguments and propositions. However, using Haskell's mechanism to instantiate type arguments, we opt to instead instantiate the Dung AF, by using a union of the Carneades' arguments and propositions as the framework arguments.

```
type ConcreteArg = Either PropLiteral Argument
type ConcreteAF  = DungAF ConcreteArg
```

There are various advantages to this approach: arguments that are unique in Carneades will necessarily be unique in the AF without having to generate labels; evaluation of the resulting argumentation framework is straightforward, since the conclusions corresponding to the arguments in the AF are the conclusions of the actual arguments in the AF, and finallly, the implementation of correspondence properties and testing the resulting translation is made easy by having direct access to the corresponding Carneades arguments.

The result of the translation will be a *ConcreteAF*, but we will keep track of the translation status of an argument by means of a labelled argument and argumentation framework to enable a more efficient translation and to prevent having to encode support relations into Dung's AFs[56].

```
type LConcreteArg = (Bool, ConcreteArg)
type LConcreteAF = DungAF LConcreteArg
```

We can now proceed to discuss the implementations corresponding to the steps in Algorithm 7.1. The *defeater* (labelled) argument is an argument in the translated that is assumed true, and is used to attack arguments that do not uphold their proof standard or have unacceptable premises.

```
defeater :: LConcreteArg
defeater = (True, Left $ mkProp "defeater")
```

[56]The translation from Carneades to ASPIC$^+$ relies on ASPIC$^+$ inference rules and the argument construction in Algorithm 6.9 to implicitly construct support. Instead, following the the same structure of the Haskell functions below, one could implement a translation that delegates more of the evaluation to Dung's AFs by not keeping track of the status: conclusions of arguments would have defeaters for all its con arguments, while for the applicability of an argument one would create arguments for all the subsets of the premises, each defeating its strict subsets and each strict subset defeating the applicability of the argument.

The topological sort of the dependency graph is delegated to the *topsort′* function of the FGl library [59]. The result is a list, pairing a proposition with all its pro arguments, which then needs to be reversed to put the leafs of the dependency graph at the front.

```
topSort :: ArgSet → [(PropLiteral, [Argument])]
topSort g
   | cyclic g    = error "Argumentation graph is cyclic!"
   | otherwise = reverse $ topsort′ g
```

A propositional literal in Carneades that is part of the assumptions can be translated directly, by making it into a *True* labelled argument in the argument graph.

```
propToLArg :: PropLiteral → LConcreteArg
propToLArg p = (True, Left p)
```

The *translate* function translates an arbitrary *cycle-free* Carneades argument Evaluation Structure (CAES) into a Dung argumentation framework by topologically sorting the argument set. It combines the defeater argument and the assumptions lifted to arguments into an initial *LConcreteAF* and translates the sorted argument set to a *LConcreteAF* by calling the *argsToAF* function with the sorted set and the initial AF. The resulting AF is then stripped of its labels by mapping *snd* and the *stripAttack* function.

```
translate :: CAES → ConcreteAF
translate caes@(CAES (argSet, (assumptions, _), _))
 = AF (map snd args) (map stripAttack attacks)
   where AF args attacks =
     argsToAF (topSort argSet)
              caes
              (AF (defeater :
                     map propToLArg assumptions)
                  [])
stripAttack ::  (LConcreteArg, LConcreteArg) →
                (ConcreteArg, ConcreteArg)
stripAttack (a, b) = (snd a, snd b)
```

The *argsToAF* function corresponds to step 3. of Algorithm 7.1. There are two basic cases: if there are no more arguments to process, the translated AF is returned, or if there is a propositional literal left, but it is an assumption, it has already been translated and does not need to be considered. Otherwise, collect all pro and con arguments for p (con arguments are obtained by calling *conArgs*) and remove them from *argList*. The translation is then done in four steps. *transApps* is called to translate the applicability part of the pro and con arguments. *transAcc* is called to translate the acceptability of p and the opposite of p (note that the order of applicable arguments is switched for translating the acceptability of the opposite of p). The results of these four calls are collected and used in the recursive step of *argsToAF*, together with the still to process arguments, with the con arguments (*con*) removed from the *argList* by applying *delete*.

```
argsToAF :: [(PropLiteral, [Argument])] → CAES →
                LConcreteAF → LConcreteAF
argsToAF [] _ transAF = transAF
argsToAF (pro@(p, proArgs) : argList)
              caes@(CAES (_, (assumptions, _), _))
                (AF args defs)
 | p ∈ assumptions = argsToAF argList caes (AF args defs)
 | otherwise =
let con                    = conArgs p argList
    (proAppArgs, proDefs)  = transApps args pro
    (conAppArgs, conDefs)  = transApps args con
    (proAccArg, proDefs′)  = transAcc p proAppArgs conAppArgs caes
    (conAccArg, conDefs′)  = transAcc (negate p) conAppArgs proAppArgs caes
    argList′               = delete con argList
in argsToAF argList′ caes
   (AF (proAccArg : conAccArg : proAppArgs ++ conAppArgs ++ args)
       (proDefs′ ++ conDefs′ ++ proDefs ++ conDefs ++ defs))
```

The *conArgs* function corresponds to step 3.(d) of the algorithm, i.e., retrieving the arguments con a given proposition p. The *find* function returns the first element of a given list that satisfies a given condition ($\equiv$ *negate* p), returning *Nothing* if it fails. The *fromMaybe* function then turns a *Maybe* b into a b, by taking two parameters, a default value of type b in case of a *Nothing*, and a *Maybe* b. In case of the *Just* constructor, the *fromMaybe* functions returns the element inside it.

```
conArgs :: PropLiteral → [(PropLiteral, [Argument])] →
             (PropLiteral, [Argument])
conArgs p argList = fromMaybe (negate p, [])
                                (find ((≡ negate p) ∘ fst)
                                      argList)
```

The translation of applicability is done in two steps, This function takes two arguments, a list of already translated arguments (including the translated premises and exceptions) and a proposition paired with its to be translated arguments. It collects the results of the transApp function, which does the main work.

```
$transApps :: [LConcreteArg] \to (PropLiteral, [Argument]) \to$
             $([LConcreteArg], [(LConcreteArg, LConcreteArg)])$
$transApps\ tArgs\ (p, args) =$
  $\mathbf{let}\ tr = map\ (transApp\ tArgs\ p)\ args$
  $\mathbf{in}\ (map\ fst\ tr, concatMap\ snd\ tr)$
```

Given a list of already translated arguments and a propositional literal, an argument (pro the propositional literal) is translated into a Dung argument and a (possibly empty) list of attackers. An argument is immediately inapplicable if not all its premises have been labelled *True* or if one of its exceptions has been. In the second case, attackers are added to the AF for every exception. *either* takes two functions, a function taking a *Left a* and returning a *c* and a function taking a *Right b* into a *c*, applying the appropriate function to an *Either a b*.

```
$transApp :: [LConcreteArg] \to PropLiteral \to Argument \to$
            $(LConcreteArg, [(LConcreteArg, LConcreteArg)])$
$transApp\ tArgs\ p\ a@(Arg\ (prems, excs, c))$
 $\mid accProps\ tArgs\ `intersect`\ prems \not\equiv prems$
   $= ((False, Right\ a), [(defeater, (False, Right\ a))])$
 $\mid otherwise =$
  $\mathbf{let}\ acceptableExceptions = filter\ (\lambda(b, arg) \to$
                                          $b \wedge either\ (\in excs)$
                                                     $(const\ False)$
                                                     $arg)$
                                        $tArgs$
      $applicableArg$        $= (null\ acceptableExceptions, Right\ a)$
      $defeats$              $= map\ (\lambda argExc \to (argExc, applicableArg))$
                                        $acceptableExceptions$
  $\mathbf{in}\ (applicableArg, defeats)$
```

This function expects the following arguments: a propositional literal at question, a list of pro arguments (labelled 'True', and thus acceptable in the current AF), a list of con arguments (acceptable in the current AF) and a CAES. The result will be an argument corresponding to the proposition and a list of attacks[57]. The *transAcc* function has a guard for every proof standard, mapping them only to *True* given that there is an applicable argument and the Haskell equivalent of the proof standard is met.

```
transAcc :: PropLiteral → [LConcreteArg] → [LConcreteArg] →
            CAES → (LConcreteArg, [(LConcreteArg, LConcreteArg)])
transAcc c [ ] conArgs caes = ((False, Left c), [(defeater, (False, Left c))])
transAcc c ((_, Left _) : proArgs) conArgs caes
   = error "Proposition in the list of applicable arguments"
transAcc c ((False, _) : proArgs) conArgs caes
   = transAcc c proArgs conArgs caes
transAcc c proArgs@((True, _) : proArgs')
            conArgs caes@(CAES (_, _, standard))
 | standard c ≡ Scintilla
   = ((True, Left c), [ ])
 | standard c ≡ Preponderance ∧
   maxWeight proArgs caes > maxWeight conArgs caes
      = ((True, Left c), [ ])
 | standard c ≡ ClearAndConvincing ∧
   maxWeight proArgs caes > α ∧
   maxWeight proArgs caes > maxWeight conArgs caes + β
   = ((True, Left c), [ ])
 | standard c ≡ BeyondReasonableDoubt ∧
   maxWeight proArgs caes > α ∧
   maxWeight proArgs caes > maxWeight conArgs caes + β ∧
   maxWeight conArgs caes < γ
   = ((True, Left c), [ ])
 | standard c ≡ DialecticalValidity ∧ null conArgs
   = ((True, Left c), [ ])
 | otherwise
   = ((False, Left c), [(defeater, (False, Left c))])
```

The *maxWeight* function determines the maximum weight of a list of applicable arguments (assumed to have the same conclusion).

```
maxWeight :: [LConcreteArg] → CAES → Double
```

[57]The translation simply uses the defeater argument to defeat arguments that do not meet the proof standard. Alternatively, one could make it explicit which con argument makes a pro argument for a conclusion unacceptable. This can be done by looking up the appropriate con arguments, checking the proof standard for each, and adding an attack from the con argument to the conclusion of the pro argument in the translated AF, if it does not meet the standard.

```
maxWeight as caes@(CAES (_, (_, argWeight), _))
  = foldl max 0 [argWeight a | (True, Right a) ← as]
```

Translation of the example CAES from Section 5.3.6. The following is the prettified output of the translation, where the five propositions in the middle are the assumptions and defeater. An extended version of this example, including weights and proof standards is also available online[58].

```
> translate caes
  AF [
    Left (True, "murder"),
    Left (False, "murder"),
    Right ["kill", "intent"]∼[] ⇒ "murder",
    Left (False, "intent"),
    Left (True, "intent"),
    Right ["witness2"]∼["unreliable2"] ⇒ "-intent",
    Right ["witness"]∼["unreliable"] ⇒ "intent",
    Left (True, "unreliable"),
    Left (False, "unreliable")
    ,
    Left (True, "defeater"),
    Left (True, "kill"),
    Left (True, "witness"),
    Left (True, "witness2"),
    Left (True, "unreliable2")
    ]
  [
    (Left (True, "defeater"), Left (True, "murder")),
    (Left (True, "defeater"), Left (False, "murder")),
    (Left (True, "defeater"),
     Right ["kill", "intent"]∼[] ⇒ "murder"),
    (Left (True, "defeater"), Left (False, "intent")),
    (Left (True, "defeater"), Left (True, "intent")),
    (Left (True, "unreliable2"),
     Right ["witness2"]∼["unreliable2"] ⇒ "-intent"),
    (Left (True, "defeater"), Left (True, "unreliable")),
    (Left (True, "defeater"), Left (False, "unreliable"))
  ]
```

[58]See: http://hackage.haskell.org/package/CarneadesIntoDung-1.0/docs/Language-CarneadesIntoDung-Examples.html

7.2 Verification of formal properties of implementations

This section discusses two approaches to verifying the correctness of an implementation. The first is property-based testing. Given implementations of key correctness properties, tools like QuickCheck [40] can usually quickly identify any problems by picking simple counter-examples from thousands of randomly generated test cases. The second approach takes this further by formally verifying the correctness of an implementation by means of a theorem prover.

7.2.1 Quick testing of properties

For the translation discussed in Section 7.1, we can refer to the definitions of the correspondence of applicability of arguments and acceptability of propositions, see Theorem 6.10, repeating it here for convenience:

Theorem 7.2. *Let C be a CAES, $\langle arguments, audience, standard\rangle$, $\mathcal{L}_{CAES}$ the propositional language used and let the argumentation framework corresponding to C be AF. Then the following holds:*

1. *An argument $a \in arguments$ is applicable in C iff there is an argument contained in the complete extension of AF with the corresponding conclusion arg_a.*

2. *A propositional literal $c \in \mathcal{L}_{CAES}$ is acceptable in C or $c \in assumptions$ iff there is an argument contained in the complete extension of AF with the corresponding conclusion c.*

Informally, the properties state that every argument and proposition in a CAES, after translation, will have a corresponding argument and keep the same acceptability status.

We will now sketch the implementations of the the correspondence properties in Haskell. The functions *corApp* and *corAcc* take a Carneades model and given that the translation function is a correct implementation, the Haskell implementation of correspondence of applicability and acceptability should always return *True*.

```
corApp :: CAES → Bool
corApp caes@(CAES (argSet, _, _)) =
  let transCAES = translate caes
      appArgs     = filter (`applicable` caes)
```

```
                              (getAllArgs argSet)
         transArgs   = stripRight (groundedExt transCAES)
      in fromList appArgs ≡ fromList transArgs
```

Here *transCAES* is the Carneades model after translation. *appArgs* are the applicable arguments in *caes* using the original definitions of applicability in the Carneades model. We then evaluate *transCAES* according to the grounded labelling (this is fine since the resulting AF is proven to be cycle-free (Proposition 3.16) and filter out the translated arguments using *stripRight* (discarding arguments representing propositions). The final line checks equality of the two (by making the lists into sets using *fromList*). A tool like QuickCheck can then be used to generate lots of random CAESs, and should *corApp* return *False* for any of them, a counter-example has been found. QuickCheck includes sophisticated infrastructure for tailoring the test case generation to work well also for complicated domains.

```
corAcc :: CAES → Bool
corAcc caes@(CAES (argSet, (assumptions, _), _)) =
  let transCAES = translate caes
      accProps    = filter (λc → c ‘acceptable‘ caes ∨
                                   c ∈ assumptions)
                              (getProps argSet)
      transProps  = stripLeft (delete
                        (Left (mkProp "defeater"))
                        (groundedExt transCAES))
  in fromList accProps ≡ fromList transProps
```

Two points were not manifest in our correspondence properties. First, we have to remove the administrative *defeater* node from our grounded extension. Second, in *corApp* and *corAcc* we remove propositions and arguments, respectively, by calling *stripRight* and *stripLeft*. Then, as expected, for our example in Section 7.1.2:

```
corApp caes ∧ corAcc caes
> True
```

Taking this further, we can use QuickCheck [40] (see Section 2.1.6) to automatically generate Carneades argument evaluation structures for us by defining appropriate *Arbitrary* instances for the data types[59].

[59]The *Arbitrary* instances are a bit convoluted given acyclicity of arguments and other restrictions on a CAES and are thus omitted.

As it happens, *corAcc* and *corApp* are already valid QuickCheck properties:

$> quickCheck\ (\lambda c \rightarrow corApp\ c \wedge corAcc\ c)$
$OK : passed\ 100\ tests.$

7.2.2 Complete formalisation in a theorem prover

The previous subsection illustrated how implementing key correctness properties can help find problems automatically. However, not finding problems does not guarantee the correctness of the code; guarantees can only by obtained through formal proofs.

One approach which allows us to unify our specification of an argumentation model with the implementation is to formalise the model into a theorem prover. We demonstrate this approach by formalising Dung's argumentation frameworks, up to grounded labelling, into Agda [128]. [60]. Agda is a programming language and interactive theorem prover based on Martin Löf type theory. Its syntax is very close to that of Haskell, making the step from implementation to complete formalisation relatively small. In Agda, through the Curry-Howard correspondence [93], types correspond to propositions and programs correspond to proofs. To prove a theorem is to implement a program having the corresponding type. This means that if we implement (prove) grounded semantics in Agda, we get some key results for free. First of all, Agda checks that all functions are terminating. Thus, because we successfully implemented the grounded semantics in Agda, we immediately know that our algorithm is terminating on all (finite) inputs. Further, as a labelling is part of the output, we have actually proven that the grounded extension always exists, verifying one of Dung's original results [48]. The correctness of these proofs are automatically checked by the Agda type checker and thus the correctness of the proofs only depends on the core implementation of Agda.

Finally, the technical nature of the mathematical properties proven in this formalisation, similar to the proofs of correspondence results between argumentation models, are not meant for an end-user of an actual implementation of the argumentation model. What we do gain, however, is a mechanically proven way to check that our standard algorithms are correct, which is especially useful in the case that the two languages are relatively close (as is the case for Haskell and Agda).

[60]The complete implementation is fully documented and open source and can be found online: `http://www.cs.nott.ac.uk/~bmv/Code/AF2.agda`

Agda formalisation

The Agda code in this section closely follows the Haskell implementation of Chapter 4. We start off with the definition of an argumentation framework (instantiated to *String*s) and labellings.

```
data DungAF (A : Set) : Set where
  AF : List A → List (A × A) → DungAF A
AbsArg = String
a : AbsArg
a = "A"
b : AbsArg
b = "B"
c : AbsArg
c = "C"
  -- an AF such that: A → B → C
AF₁ : DungAF AbsArg
AF₁ = AF (a :: b :: c :: []) ((a, b) :: (b, c) :: [])
  -- an AF such that: A ↔ B
AF₂ : DungAF AbsArg
AF₂ = AF (a :: b :: []) ((a, b) :: (b, a) :: [])
data Status : Set where
  In        : Status
  Out       : Status
  Undecided : Status
```

While the Agda code is intended to be structured similarly to the Haskell code, Agda does not (yet) have syntax for list comprehensions, which means list comprehensions need to be desugared, thereby complicating the Agda equivalents of *attacked* and *unattacked*.

```
unattacked : {A : Set} → (A → A → Bool) → List A
                       → DungAF A → A → Bool
unattacked _≡_ outs (AF _ def) arg =
  null
    (deleteFirstsBy _≡_
      (List.map proj₁
                (filter ((λ x → x ≡ arg) ∘ proj₂)
                        def)) outs)
attacked : {A : Set} → (A → A → Bool) → List A
```

```
                    → DungAF A → A → Bool
attacked _≡_ ins (AF _ def) arg =
  ¬ (null
    (intersectBy _≡_
      (List.map proj₁
               (filter ((λ x → x ≡ arg) ∘ proj₂)
                     def)) ins))
```

intersectBy and *deleteFirstBy*, are the Agda equivalents of *intersect* and (\\\\), where instead of writing *Eq a* ⇒ we need to explicitly supply an equality function (_ ≡ _)[61]. The functions $proj_1$ and $proj_2$ respectively take the first and second element of a pair.

Below is an attempt to formalise the grounded labelling function using Lists, returning a list of arguments with their respective statuses. The code is structured very similarly to the Haskell implementation in Section 4.2.2. *groundedL* takes an equality function between arguments *A*, an AF and returns a list of labelled arguments, by calling the helper function *groundedList*. Similar to the Haskell implementation, *groundedList* keeps track of three lists, respectively *ins*, *outs* and *args*, slowly assigning *args* to *ins* and *outs* depending on whether they are *attacked* or *unattacked*.

```
groundedL : {A : Set} → (A → A → Bool) →
            DungAF A → List (A × Status)
groundedL _≡_ (AF args def)
  = groundedList _≡_ [] [] args (AF args def)


groundedList : {A : Set} → (A → A → Bool) → List A →
               List A → List A → DungAF A →
               List (A × Status)
groundedList _      ins outs [] _
  = List.map (λ x → (x, In)) ins ++ List.map (λ x → (x, Out)) outs
groundedList _≡_ ins outs args af
  with filter   (unattacked _≡_ outs af) args |
       filter   (attacked _≡_ ins af)      args |
       filter   (unattacked _≡_ outs af) args ++
       filter   (attacked _≡_ ins af)      args
... | _    | _    | []
  = List.map (λ x → (x, In)) ins ++
```

[61]The source file corresponding to this subsection contains the complete definitions of *intersectBy* and *deleteFirstsBy*.

```
        List.map (λ x → (x, Out)) outs ++
        List.map (λ x → (x, Undecided)) args
... | ins′ | outs′ | (x :: xs)
    = groundedList  _≡_
                    (ins ++ ins′)
                    (outs ++ outs′)
                    (deleteFirstsBy _≡_ args (x :: xs))
                    af
```

Although this implementation would be correct in a Haskell like language, in Agda this implementation is not sufficient, with the termination checker marking it as possibly non-terminating. The problem here is the following: *ins*, *outs* keep track of the currently already status assigned arguments and slowly get bigger, while *args* contains the arguments that are still to be assigned a status (if possible) and is the argument that should be getting smaller to avoid infinite recursion. Given that the implementation of *deleteFirstsBy* is terminating and does actually remove elements, it will probably seem reasonable to the reader that *args* argument in the recursive call to *groundedList* should get smaller. However, this is certainly not a structural decrease and would therefore be rejected by the Agda compiler (this would be a problem in most other proof assistants as well).

In the following an implementation will be constructed that does structurally recurse, by keeping track of the number of elements in *args* and proving that this does get structurally smaller. This is achieved by making the three arguments *ins*, *outs* and *args* into *Vector*s that all have an explicit length. However, defining a *filter* function on *Vector*s is a problem: it is not clear how many elements would be filtered out, making the length of the resulting *Vector* unknown. Instead, we build on the *Find* datatype by Norell [128] (see Section 2.2), defining a *FindV* data type that will support the filtering of one or no elements, making the lengths of the resulting *Vector*s calculable at compile time.

The *Find* and *All* data types, can now be adapted to *Vector*s:

```
data FindV {A : Set} (p : A → Bool) : {n : ℕ} →
                Vec A n → Set where
  foundV    : {k : ℕ} {m : ℕ} (xs : Vec A k) (y : A) →
                satisfies p y → (ys : Vec A m) →
                FindV p (xs ++ y :: ys)
  notfoundV : {n : ℕ} {xs : Vec A n} →
                AllV (satisfies (¬ ∘ p)) xs → FindV p xs

infixr 30 _ : allV : _
```

```
data AllV {A : Set} (P : A → Set) : {n : ℕ} →
            Vec A n → Set where
allV[ ]   : AllV P [ ]
_:allV:_ : {x : A} {n : ℕ} {xs : Vec A n} → P x →
           AllV P xs → AllV P (x :: xs)
```

Although it is not explicit in the result for the *foundV* constructor, the length of the *Vector* is implicitly calculated and available statically.

Now the *find* function can be adapted to *Vector*s, allowing us to filter out one element at the time, while obtaining proof of what the resulting element and lists are:

```
findV : {A : Set} {n : ℕ} (p : A → Bool) (xs : Vec A n) → FindV
findV p [ ]         = notfoundV allV [ ]
findV p (x :: xs) with p x | inspect p x
... | true  | [prf] = foundV [ ] x (trueIsTrue prf) xs
... | false | _ with findV p xs
findV p (x :: ._)  | false | [prf] | foundV xs y py ys
   = foundV (x :: xs) y py ys
findV p (x :: xs)  | false | [prf] | notFoundV npxs
   = notfoundV (lemma (falseIsFalse prf) : allV : npxs)
```

We need a few simple lemmas on natural numbers to prove some obvious mathematical facts. The lemmas can be handled by an automatic ring-solver in the Agda library. This is achieved by giving a syntactical representation of the theorem and letting the RingSolver rewrite this. It is successful if the Agda compiler accepts *refl*. One lemma's implementation is provided, the rest can be found in the source code of this subsection.

```
lemma2 : {m n k l : ℕ} → (suc (m + n + (k + l))) →
                         (m + n + (k + suc l))
lemma2 {m} {n} {k} {l} =
  solve 4
    (λ m′ n′ k′ l′ →
      con 1 :+(m′ :+n′ :+(k′ :+l′)) :=
      m′ :+n′ :+(k′ :+(con 1 :+l′)))
    refl m n k l


lemma3 : {m n k l : ℕ} → (m + suc n + (k + l)) →
         (m + n + (k + suc l))
lemma4 : {a k l : ℕ} → a ≡ k + suc l → a ≡ suc (k + l)
```

$$\mathit{lemma5} : \{\, a\ k\ l : \mathbb{N} \} \to \mathit{suc}\ a \equiv k + \mathit{suc}\ l \to a \equiv k + l$$

We now have all the tools needed to be able to define the grounded labelling. Again we provide the definitions in two parts, a function *grounded* that calculates the grounded labelling given an equality function and an argumentation framework, and a helper function *grounded′* that takes three *Vector*s, the current *ins* and *outs* (starting empty), and the arguments to process (*args*), the AF (*af*), an equality on arguments (_ ≡ _) and a proof that there is number k that we can prove is equal to the length of *args*. By proving that a number k is equal to o, we can then pattern match on k, demonstrating that it gets structurally smaller in each recursive call.

```
grounded′ : { A : Set } → { m n o : ℕ } → (Σ ℕ λ k → k ≡ o) →
            (A → A → Bool) → Vec A m → Vec A n → Vec A o →
            DungAF A → Vec (A × Status) (m + n + o)
```

Base case: in case *args* is empty, then we have no more arguments to process and we can immediately return a mapping of *ins* and *outs*:

```
grounded′ _ _ ins outs [ ] _
  = (map (λ x → (x, In)) ins ++
      map (λ x → (x, Out)) outs) ++ [ ]
```

Otherwise, we try to find an unattacked or attacked argument, by matching on both *findV* expressions, using the *with* construct.

```
grounded′ _ _≡_ ins outs args af
  with findV (unattacked _≡_ (toList outs) af) args |
       findV (attacked   _≡_ (toList ins)  af) args
```

There are two impossible cases: the length of *args* is zero, while we did manage to respectively find an unattacked, or attacked element inside *args*. For both cases we have that the length of the vector is $o = k + \mathit{suc}\ l$, while we have proof that the length of the *Vector* is also equal to zero (zero, p). After applying lemma4 to rewrite o into *suc* _ (because $k + \mathit{suc}\ l \equiv \mathit{suc}\ (k + l)$), we can use the fact that $(\mathit{suc}\ \_) \not\equiv \mathit{zero}$, and define both cases using the absurd pattern, ().

```
grounded′ { o = .(k + suc l) } (zero, p) _≡_ _ _ ._ af
  | foundV { k } { l } _ _ _ _
  | _ with lemma4 { zero } { k } { l } p
... | ()
grounded′ { o = .(k + suc l) } (zero, p) _≡_ _ _ ._ af
```

```
        | notFoundV _
        | foundV {k} {l} _ _ _ _ with lemma4 {zero} {k} {l} p
    ... | ()
```

Now follow the two recursive cases, having found an unattacked/attacked element.

The Vector we try to return is of the "wrong" length, so we need to rewrite it using the basic *lemma3* and substitute this value in the Vector constructor. We furthermore rewrite the proof of the length of o, using *lemma5* to become one smaller on both sides of the equation, guaranteeing the structural decrease on the recursive call.

```
    grounded′ {_} {m} {n} {o = .(k + suc l)}
              (suc a, p) _≡_ ins outs
              .(xs ++ y :: ys) af | foundV {k} {l} xs y _ ys | _
      = subst (Vec _) (lemma2 {m} {n} {k} {l})
                      (grounded′ (a, lemma5 {a} {k} {l} p)
                                 _≡_ (y :: ins) outs (xs ++ ys) af)
```

Similarly:

```
    grounded′ {_} {m} {n} {o = .(k + suc l)} (suc a, p) _≡_ ins outs
              .(xs ++ y :: ys) af |
              notFoundV _ | foundV {k} {l} xs y _ ys
      = subst (Vec _) (lemma3 {m} {n} {k} {l})
                      (grounded′ (a, lemma5 {a} {k} {l} p)
                                 _≡_ ins (y :: outs) (xs ++ ys) af)
```

Final case (fixpoint): we haven't found any unattacked/attacked element and thus are done:

```
    grounded′ _ _ ins outs args _ | notFoundV _ | notFoundV _
      = (map (λ x → (x, In)) ins
        ++ map (λ x → (x, Out)) outs)
        ++ map (λ x → (x, Undecided)) args
```

With the *grounded′* helper function defined, we can now define the actual grounded labelling function, which calls grounded' with the correct *Vector*s and lengths.

```
    grounded : {A : Set} → (A → A → Bool) → DungAF A →
               List (A × Status)
    grounded _≡_ (AF args def) = toList
```

```
  (grounded′ ((length (fromList args)), refl)
              _≡_ [] [] (fromList args) (AF args def))
```

length takes the implicit length argument of a *Vector*:

```
length : {A : Set} {n : ℕ} → Vec A n → ℕ
length {_} {n} _ = n
```

Then as expected:

```
testGrounded1 : List (AbsArg × Status)
testGrounded1 = grounded String._≡_ AF₁
testGrounded2 : List (AbsArg × Status)
testGrounded2 = grounded String._≡_ AF₂


> testGrounded1
  ("C", In) :: ("A", In) :: ("B", Out) :: []
> testGrounded2
  ("A", Undecided) :: ("B", Undecided) :: []
```

Finally, the grounded extension can again be defined through the grounded labelling by keeping the arguments with an *In* label:

```
groundedExt : {A : Set} → (A → A → Bool) → DungAF A → List A
groundedExt _≡_ (AF args def) =
  List.map proj₁
           (filter ((_≡_ In) ∘ proj₂)
                   (grounded _≡_ (AF args def)))
```

7.3 Related work

The most closely related work to the formalisation in this chapter is the Logic of Argumentation (LA) by Krause et al. [103]. Krause et al. develop a model of argumentation based on the Curry-Howard correspondence, representing arguments for a conclusion by lambda terms for a type. The lambda terms are furthermore generalised to contain free variables, allowing weights to be assigned by a context, giving a founded approach to argument aggregation by means of complete semi-lattices. Although LA strongly incorporates the connection between lambda calculus and intuitionistic logic, the implementation is not formalised in a theorem prover. As far as I am aware, the work in this chapter is the first formalisation of an argumentation model in a theorem prover.

Assumption-based argumentation[62] is very closely connected to logic programming, giving an interpretation of negation as failure, default logic and auto-epistemic logic inside its framework [50, 61]. Various frameworks, including assumption-based argumentation, Hunter and Besnard's classical approach to argumentation, Carneades and others have been translated into the ASPIC$^+$ framework [145, 121, 120, 122]. See Section 6.3 for further references. However, apart from translations between logic programming based approaches and assumption-based argumentation, it seems to be rare that translations are actually implemented.

The functional programming framework in this chapter attempts to alleviate some of these problems by providing a high level framework that is able to capture both abstract argumentation and logic programming based approaches (see Chapter 4), structured argumentation models (see Chapter 5) and translations from seemingly non-related structured models and abstract models (see Chapter 6 and this chapter).

7.4 Conclusions and future work

In this chapter we have shown that functional programming, specifically Haskell, is very suitable for the implementation of structured and abstract models of argumentation. We gave one of the first algorithmic translations between a structured and an abstract model of argumentation, implemented this, and showed how to quickly test key properties. We then took this further, taking our implementation of Dung's AFs into a theorem prover, proving termination and one of Dung's original results. Finally, we combine all this into a verified pipeline, starting from a Carneades input file, running it through our implementation of the translation, and outputting to a file that is readable by the existing efficient implementation ASPARTIX [56]. A demonstration can be found online[62].

Future work includes extending the work on the correctness of the pipeline to complete, automatically verified proofs through a theorem prover. This would require formalising Carneades and the translation from Carneades into Dung, and then formalising correspondence properties and rationality postulates[63]. Another interesting line of work would be to extend the formalisation

[62]See: `www.cs.nott.ac.uk/~bmv/CarneadesIntoDung/Demo/`.

[63]The work together with Tom Gordon and Douglas Walton [84] attacks the problem the other way around by immediately specifying the whole model and translation directly into Agda, instead of starting in Haskell. However, this work has not yet been published.

of Dung's argumentation frameworks in Agda to incorporate further definitions and theorems. This would require to reprove various of Dung's results, because his proofs in general are non-constructive.

Chapter 8

A general argumentation framework supporting weights and argument aggregation

This chapter introduces a general argumentation model supporting propagation/combination of weights and the aggregation/accrual of arguments, by extending ASPIC$^+$ [122] using techniques from the Logic of Argumentation [103]. The Logic of Argumentation, as introduced in Section 3.7.1, combines an "evidential category" with a content ordering and confidence measure to supply the user with a general method of aggregation on LA's arguments, i.e., typed lambda terms. The next sections will make these statements much more concrete by adapting the techniques given in LA to generalise the ASPIC$^+$ argumentation model with proof standards/burdens. This generalisation will allow us to apply the aggregation principles of LA, by adapting the concepts of the ordering, confidence measures and the implicit weight propagation into the ASPIC$^+$ system. This results in a system integrating proof standards, argument aggregation and weight propagation in a principled way.

We do not make any assumptions about the origin of the weights, but we do put sensible restrictions on the interaction of the given weights and on the aggregation of arguments build from the given weights, knowledge and rules. The intention is to provide a framework allowing instantiations depending on a specific domain; we provide an example instantiation of this framework in the context of the probabilistic domain.

Section 8.1 introduces an instantiation of the 2013 version of the ASPIC$^+$ argumentation framework [122]. Section 8.2 extends the ASPIC$^+$ instantiation to allow a derivation of an ordering of arguments based on its fallible content. Section 8.3 expands on the previous section, providing multiple, mathematically founded, methods of combining and propagating weights assigned to knowledge and rules. Section 8.4 combines the weight propagation with a general notion of argument accrual, allowing different types of accrual, given reasonable constraints. Section 8.5 provides an implementation of the developed framework by applying the approach introduced in Chapter 7. Section 8.6 discusses related work. Finally, Section 8.7 concludes and discusses future work.

8.1 An instantiation of ASPIC$^+$ (2013)

This section introduces an instantiation of the most recent version of ASPIC$^+$ [122], with some adapted definitions. Argumentation systems are defined in a similar way to ASPIC$^+$ (2010) (see Definition 3.30), however the naming convention of defeasible rules has been made explicit.

Definition 8.1 (Argumentation system (Def. 2 of [122]). An *argumentation system* is a tuple $AS = \langle \mathcal{L}, ^-, \mathcal{R}, n \rangle$ where

- $\mathcal{L}$ is a logical language,
- $^-$ is a function from $\mathcal{L}$ to $2^{\mathcal{L}}$, such that:
  - φ is a *contrary* of ψ if $\varphi \in \bar{\psi}, \psi \notin \bar{\varphi}$;
  - φ is a *contradictory* of ψ (denoted by '$\varphi = -\psi$'), if $\varphi \in \bar{\psi}, \psi \in \bar{\varphi}$;
- $\mathcal{R} = \mathcal{R}_s \cup \mathcal{R}_d$ is a set of strict ($\mathcal{R}_s$) and defeasible inference rules ($\mathcal{R}_d$) such that $\mathcal{R}_s \cap \mathcal{R}_d = \emptyset$,
- $n : \mathcal{R}_d \rightarrow \mathcal{L}$ is a *naming convention* for defeasible rules.

Knowledge bases are restricted to contain only necessary axioms and ordinary premises.

Definition 8.2 (Knowledge base (Adapted Def. 4 of [122])). A *knowledge base* in an argumentation system $\langle \mathcal{L}, ^-, \mathcal{R}, n \rangle$ is a set $\mathcal{K}$ where $\mathcal{K} \subseteq \mathcal{L}$ and $\mathcal{K}$ is partitioned into two subsets $\mathcal{K}_n$ (the *axioms*) and $\mathcal{K}_p$ (the *ordinary premises*).

The definitions of arguments extends Definition 3.32 of ASPIC$^+$ (2010) to contain a set of strict rules (*StRules*).

Definition 8.3 (Arguments (Def. 5 of [122])). An *argument* A on the basis of a knowledge base $\mathcal{K}$ in an argumentation system $\langle \mathcal{L}, ^-, \mathcal{R}, n \rangle$ is:

1. φ if $\varphi \in \mathcal{K}$ with:
   $Prem(A) = \{\varphi\}$,

$Conc(A) = \varphi$,
$Sub(A) = \{\varphi\}$,
$DefRules(A) = \emptyset$,
$StRules(A) = \emptyset$,
$TopRule(A) =$ undefined.

2. $A_1, \ldots, A_n \rightarrow \psi$ if $A_1, \ldots, A_n$ are arguments such that there exists a strict rule $Conc(A_1), \ldots, Conc(A_n) \rightarrow \psi$ in $\mathcal{R}_s$,
$Prem(A) = Prem(A_1) \cup \ldots \cup Prem(A_n)$,
$Conc(A) = \psi$,
$Sub(A) = Sub(A_1) \cup \ldots \cup Sub(A_n) \cup \{A\}$,
$DefRules(A) = DefRules(A_1) \cup \ldots \cup DefRules(A_n)$,
$StRules(A) = StRules(A_1) \cup \ldots \cup StRules(A_n) \cup \{Conc(A_1), \ldots, Conc(A_n) \rightarrow \psi\}$,

$TopRule(A) = Conc(A_1), \ldots, Conc(A_n) \rightarrow \psi$.

3. $A_1, \ldots, A_n \Rightarrow \psi$ if $A_1, \ldots, A_n$ are arguments such that there exists a defeasible rule $Conc(A_1), \ldots, Conc(A_n) \Rightarrow \psi$ in $\mathcal{R}_d$,
$Prem(A) = Prem(A_1) \cup \ldots \cup Prem(A_n)$,
$Conc(A) = \psi$,
$Sub(A) = Sub(A_1) \cup \ldots \cup Sub(A_n) \cup \{A\}$,
$DefRules(A) = DefRules(A_1) \cup \ldots \cup DefRules(A_n) \cup \{Conc(A_1), \ldots, Conc(A_n) \Rightarrow \psi\}$,
$StRules(A) = StRules(A_1) \cup \ldots \cup StRules(A_n)$,
$TopRule(A) = Conc(A_1), \ldots, Conc(A_n) \Rightarrow \psi$.

Furthermore, for any argument A, $Prem_n(A) = Prem(A) \cap \mathcal{K}_n$ and $Prem_p(A) = \mathcal{K}_p$.

Attacks in ASPIC$^+$ (2013) are redefined in terms of the naming convention (n).

Definition 8.4 (ASPIC$^+$ attacks (Def. 8 of [122]))**.** Let A and B be two arguments. A *attacks* B iff A *undercuts*, *rebuts* or *undermines* B, where:

- A *undercuts* B (on B') iff $Conc(A) \in \overline{n(r)}$ for some $B' \in Sub(B)$ of the form $B''_1, \ldots, B''_n \Rightarrow \psi$.
- A *rebuts* B (on B') iff $Conc(A) \in \overline{\varphi}$ for some $B' \in Sub(B)$ of the form $B''_1, \ldots, B''_n \Rightarrow \psi$. In such a case A *contrary-rebuts* B iff $Conc(A)$ is a contrary of φ.
- A *undermines* B (on B') iff $Conc(A) \in \overline{\varphi}$ for some $B' = \varphi$, $\varphi \in Prem_p(B)$. In such a case A *contrary-undermines* B iff $Conc(A)$ is a contrary of φ.

An argumentation theory and the corresponding structured argumentation frameworks are defined in terms of the previous definitions.

Definition 8.5 (Argumentation theory (Def. 10 of [122])). An *argumentation theory* is a tuple $AT = \langle AS, \mathcal{K} \rangle$ where AS is an argumentation system (Definition 8.1) and $\mathcal{K}$ is a knowledge base in AS (Definition 8.2).

Definition 8.6 (Structured argumentation frameworks (Def. 11 of [122])). Let AT be an argumentation theory $\langle AS, \mathcal{K} \rangle$.

A *structured argumentation framework* (SAF) defined by AT, is a triple $\langle \mathcal{A}, \mathcal{C}, \preceq \rangle$, where $\mathcal{A}$ is the set of all the finite arguments constructed from $\mathcal{K}$ in AS (henceforth called the set of arguments on the basis of AT), $\preceq$ is an ordering on A, and $(X, Y) \in \mathcal{C}$ iff X attacks Y.

Definition 8.7 (Well-defined SAFs (Def. 12 of [122])). Let $AT = \langle AS, \mathcal{K} \rangle$ be an argumentation theory $AS = \langle \mathcal{L}, ^-, \mathcal{R}, n \rangle$. We say that AT is:

- *closed under contraposition* iff for all $S \subseteq \mathcal{L}$, $s \in S$ and φ, if $S \vdash \varphi$ then $S \setminus \{s\} \cup \{-\varphi\} \vdash -s$.
- *closed under transposition* iff if $\varphi_1, \ldots, \varphi_n \to \psi \in \mathcal{R}_s$, then for $i = 1 \ldots n$, $\varphi_1, \varphi_{i-1}, -\psi, \varphi_{i+1}, \ldots, \varphi_n \to -\varphi_i \in \mathcal{R}_s$;
- *axiom consistent* iff $Cl_{\mathcal{R}_s}(\mathcal{K}_n)$ is consistent.
- *well formed* if whenever φ is a contrary of ψ then $\psi \notin \mathcal{K}_n$ and ψ is not the consequent of a strict rule.

If a SAF is defined by an AT that is axiom consistent, well formed and closed under contraposition or closed under transposition, then the SAF is said to be *well defined.*

The version of strict continuations of Modgil and Prakken [122] is not correct, allowing arguments with more premises and defeasible rules to be a strict continuation of another argument. Instead, we change their definition to be inductive, based on [124]:

Definition 8.8 (Strict continuation).

1. For all arguments A it holds that A is a strict continuation of $\{A\}$;

2. If $A_1, \ldots, A_n$ and $S_1, \ldots, S_n$ are such that all A_i are a strict continuation of S_i, then $A_1, \ldots, A_n \to p$ is a strict continuation of $S_1 \cup \ldots \cup S_n$.

3. Nothing else is a strict continuation of anything else.

Finally, we introduce *reasonable argument orderings.*

Definition 8.9 (Reasonable argument orderings (Def. 18 of [122]))**.** An argument ordering $\preceq$ is *reasonable* iff:

1. (a) $\forall A, B$, if A is strict and firm and B is plausible or defeasible, then $B \prec A$;

   (b) $\forall A, B$, if B is strict and firm then $B \not\prec A$;

   (c) $\forall A, A', B$ such that A' is a strict continuation of $\{A\}$, if $A \not\prec B$ then $A' \not\prec B$, and if $B \not\prec A$ then $B \not\prec A'$ (i.e., applying strict rules to a single argument's conclusion and possibly adding new axiom premises does not weaken, respectively strengthen, arguments).

2. Let $\{C_1, \ldots, \mathcal{C}_n\}$ be a finite subset of $\mathcal{A}$, and for $i = 1 \ldots n$, let $C^{+\backslash i}$ be some strict continuation of $\{C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n\}$. Then it is not the case that: $\forall i, C^{+\backslash i} \prec C_i$.

8.2 Content orderings

In this section we will discuss how to build up an ordering between arguments, called a *content ordering*, based on their defeasible content. The idea of a content ordering is based on the Logic of Argumentation [103].

The idea of a content ordering is that the order between two arguments is based on their *fallible content*. An argument would be lower in the ordering if it has more content that makes the argument weaker. The content orderings we develop in this section, is an alternative approach to *reasonable argument orderings* (Definition 8.9) deliberately violates the second strict continuation property (2.)).

First we define *content equal arguments*, an alternative to strict continuations of arguments, that does not look at how an arguments is constructed, but instead only looks at the logical content.

Definition 8.10 (Content equal arguments)**.** For any set of arguments $\{A_1, \ldots, A_n\}$, the argument A is *content equal* to another argument A' iff:

– $\mathit{Prem}_p(A) = \mathit{Prem}_p(A')$
  (i.e., the ordinary premises in A are exactly those of A');

– $\mathit{DefRules}(A) = \mathit{DefRules}(A')$
  (i.e., the defeasible rules in A are exactly those of A');

Example 8.11 (Motivation for content equal arguments)**.** Consider the following premises and rules:

$$
\begin{aligned}
\mathcal{K}_p &= \{p, t\},\\
\mathcal{R}_s &= \{q, r \rightarrow s\},\\
\mathcal{R}_d &= \{p \Rightarrow q;\ q \Rightarrow r;\ t \Rightarrow q\},\\
n &= \{(p \Rightarrow q, d_1), (q \Rightarrow r, d_2), (t \Rightarrow q, d_3)\}
\end{aligned}
$$

We can construct the following arguments from this:

$$
\begin{array}{ll}
A_1 : p & A_6 : A_5 \Rightarrow_{d_2} r\\
A_2 : t & A_7 : A_4, A_5 \Rightarrow_{d_3} q\\
A_3 : A_1 \Rightarrow_{d_1} q & A_8 : A_3, A_4 \rightarrow s\\
A_4 : A_3 \Rightarrow_{d_2} r & A_9 : A_5, A_6 \rightarrow s\\
A_5 : A_2 \Rightarrow_{d_3} q &
\end{array}
$$

In proof tree form:

$$
\dfrac{\dfrac{\dfrac{p}{q}\,(d_1)}{r}\,(d_2) \qquad \dfrac{t}{q}\,(d_3)}{s}
\qquad\qquad\qquad
\dfrac{\dfrac{\dfrac{t}{q}\,(d_3)}{r}\,(d_2) \qquad \dfrac{p}{q}\,(d_1)}{s}
$$

It seems reasonable to argue that both arguments should probably be equal in strength since they use the exact same defeasible rules and premises, but in a different order. However, although they are content equal, they are certainly not strict continuations of the same arguments/strict continuations.

In ASPIC$^+$, there are two types of fallible content, namely the defeasible rules and the non-axiom premises, which are instantiated to just ordinary premises (see Definition 8.2). We now provide three sensible interpretations that produce an appropriate ordering from this defeasible content, taking increasingly more of the structure of the argument in account.

8.2.1 Content orderings based on sets

We start with a content ordering that takes the fallible content of an argument to be a set, thereby ignoring any order or repetition.

Definition 8.12 (Content ordering of sets)**.** A *content ordering* of a set of arguments *Args* with language $\mathcal{L}$ is a function where $\mathit{content}(A) = \mathit{DefRules}(A) \cup \mathit{Prem}_p(A)$, for $A \in \mathit{Args}$ combined with the following derived order:

- $\mathit{content}(A) \supseteq \mathit{content}(B)$ implies $A \leqslant B$,

- $content(A) = \emptyset$ implies $B \leqslant A$, for all B.

Note that we define the content of an argument to be a set, with the subset relation being the ordering relation. The use of a set implies that the reuse of syntactically equal defeasible content, e.g. multiple uses of a ordinary premise or defeasible rule, will not weaken an argument as a whole. By taking this approach we avoid many of the counter-intuitive problems mentioned by Pollock [142]. We do not attempt to tackle semantical equality of logical content as this is generally undecidable. This definition furthermore ensures that a strict and firm argument will always be at least as strong as any other argument, satisfying a condition similar to that in Krause et al. [103], namely that all purely logical arguments should be the strongest (top) element in an ordering. [64]

Example 8.13 (Example content ordering)**.** Consider the following premises and rules:

$$\begin{aligned}
\mathcal{K}_n &= \{q\},\\
\mathcal{K}_p &= \{p, r\},\\
\mathcal{R}_s &= \{q, r \rightarrow t\},\\
\mathcal{R}_d &= \{p, q \Rightarrow s;\ s, t \Rightarrow u\},\\
n &= \{(p, q \Rightarrow s, d_1), (s, t \Rightarrow u, d_2)\}
\end{aligned}$$

We can construct the following arguments from this:

$$\begin{array}{ll}
A_1 : p & A_4 : A_1, A_2 \Rightarrow_{d_1} s\\
A_2 : q & A_5 : A_2, A_3 \Rightarrow_{d_2} t\\
A_3 : r & A_6 : A_4, A_5 \Rightarrow_{d_3} u
\end{array}$$

The defeasible content for these arguments would be the following:

$$\begin{aligned}
content(A_1) &= \{p\}\\
content(A_2) &= \emptyset\\
content(A_3) &= \{r\}\\
content(A_4) &= \{p, d_1\}\\
content(A_5) &= \{r\}\\
content(A_6) &= \{p, r, d_1, d_2\}
\end{aligned}$$

[64]Although the condition in Krause et al. is not directly stated for content orderings but for confidence measures, it does hold for content orderings since the map from a content ordering to a measure of confidence is a semi-lattice homomorphism, thereby preserving the top element.

Giving the following derived order:

$$\begin{aligned}
\mathit{content}(A) &\subseteq \mathit{content}(A), \text{ for all arguments } A\\
\mathit{content}(A_1) &\subset \mathit{content}(A_4) \subset \mathit{content}(A_6),\\
\mathit{content}(A_2) &\subset \mathit{content}(A'), \text{ for all other arguments } A'\\
\mathit{content}(A_3) &= \mathit{content}(A_5) \subset \mathit{content}(A_6)
\end{aligned}$$

The content of an argument is a set, so note that reuse of premises (or rules) in an argument does not influence the content ordering.

Our use of content ordering can be related to the previously defined notions of content equality and strict continuations.

Proposition 8.14.

1. *For all arguments A and B, we have that $\mathit{content}(A) = \mathit{content}(B)$ iff A is content equal to B;*
2. *For all A, with A as a strict continuation of $\{A_1, \ldots, A_n\}$, we have that $\mathit{content}(A) = \mathit{content}(A_1) \cup \ldots \cup \mathit{content}(A_n)$.*

Proof. 1. By definition.
2. If A is a strict continuation of $\{A_1, \ldots, A_n\}$, then by 2. of Definition 8.8 it must be the case that A is of the form $\{S_1, \ldots, S_n \rightarrow p\}$, where $\{S_1, \ldots, S_n\}$ are strict continuations. Then by Definition 8.3, we have that $\mathit{DefRules}(A) = \mathit{DefRules}(S_1) \cup \ldots \cup \mathit{DefRules}(S_n)$ and the same for $\mathit{Prem}(A)$. Given that for 1. of Definition 8.8 it holds trivially, it holds generally by induction. □

Proposition 8.15. *Any content ordering $\leqslant$ according to Definition 8.12 is a preorder.*

Proof. For any argument A we have $A \leqslant A$, because $\mathit{content}(A) \subseteq \mathit{content}(A)$. Similarly, by transitivity of $\subseteq$ on the *content* we have transitivity of $\leqslant$. □

From the concept of a content ordering we can go to a content based ordering, which is an extension of the content ordering allowing further defined preferences on the content of arguments, given that the resulting ordering is still a preorder.

Definition 8.16 (Content based ordering)**.** Let $\mathcal{A}$ be a set of arguments, and $\leqslant$ the content ordering on this set. Then an ordering $\preceq$ on $\mathcal{A}$ is a *content based ordering* iff for all arguments A, B and C:

- $A \leqslant B$ implies $A \preceq B$;

- $A \preceq B$ and $B \preceq C$ implies $A \preceq C$.

Proposition 8.17. *Any content based ordering $\preceq$ according to Definition 8.16 is a preorder.*

Proof. By definition, we have that because of the reflexivity of a content ordering, a content based ordering is reflexive. Transitivity holds by definition. □

Proposition 8.18. *An argument ordering $\preceq$ defined using Definition 8.16 is adheres to the first three principles of a reasonable argument ordering (Definition 8.9).*

Proof. 1.i) Assume A is strict and firm and B is plausible or defeasible, then $\mathit{content}(A) = \emptyset$ and $\mathit{content}(B) \neq \emptyset$. Thus, because $\mathit{content}(A) \subset \mathit{content}(B)$ we have $B \prec A$.

1.ii) Assume A is strict and firm, then $\mathit{content}(A) = \emptyset$. Thus, because for any B, $\mathit{content}(A) \subseteq \mathit{content}(B)$, we have that $A \preceq B$ and therefore $B \not\prec A$.

1.iii) Assume an argument A' that is a strict continuation of an argument A. By Proposition 8.14 we have that $\mathit{content}(A) = \mathit{content}(A')$. Thus for all B, if we have $A \not\prec B$, then by transitivity of $\preceq$ (Proposition 8.17) and the equality of content we have that $A' \not\prec B$.

□

We can now define an equivalence relation based on the content based orderings.

Definition 8.19 (Equivalence of arguments)**.** Given a set of arguments $\mathcal{A}$ and a content based ordering $\preceq$ on $\mathcal{A}$, then two arguments $A, B \in \mathcal{A}$ are equivalent iff $A \preceq B$ and $B \preceq A$.

Proposition 8.20. *The equivalence between arguments in Definition 8.19 gives rise to an equivalence relation on arguments.*

Proof. By Definition 8.16 $A \preceq A$, and therefore by Definition 8.19 any A is equivalent to itself. Transitivity holds by similar reasoning. Finally, Definition 8.19 defines the two arguments equivalent if they are symmetric. □

8.2.2 Content orderings based on multi-sets

Arguments in a domain with limited resources, for instance arguments based on actions which might have premises that are "used up", need to keep track of each occurrence of premises and defeasible rules. One way to do so is by means of a multiset (bag). This can be done by taking all sets and operators from Definition 3.32 and in the Definition below to be standard multisets and appropriate operations.

8.2.3 Content orderings based on sequences

Finally, we give a content ordering that also keeps the order of the applied rules and premises. Now we need to interpret Definition 8.3 to use sequences (lists), replacing every { and } with brackets [and], $\emptyset$ with the empty sequence [ ] and replace $\cup$ with concatenation $+\!\!+$. The definition of ordering of ASPIC$^+$ (see Definition 8.9) is most closely related to a content ordering based on sequences: the order in which arguments are built up matters (see also Example 8.11), however the order in which subarguments are added to a superargument does not.

8.3 Weight assignment, weight propagation and proof standards

In this section we will consider how can we sensibly assign weights to arguments in a further instantiation of the version of ASPIC$^+$ given in Section 8.1. This will be lead by the content orderings defined in Section 8.2. To do so we take the simplified version of an argumentation system of Prakken and Sartor [154] (Definition 3.65), restricting the argumentation system to propositional logic and standard negation. However, instead of directly defining an ordering on knowledge and defeasible rules, we instead map knowledge and defeasible rules to weights. We then take the previously defined content ordering (see Definition 8.12), and derive a content based ordering based on the extra information of the weights. From this we gain a normal ASPIC$^+$ argumentation system, while at the same time we can get a more founded approach to weight assignment, by allowing weights to be propagated.

8.3.1 Confidence measures

An argument that is constructed in ASPIC$^+$ by purely using necessary premises (axioms) and chaining strict inference rules, would not have any defeasible content. We would thus expect the weight of this argument to be maximal with regard to the range of weights. So the restriction we put on the weights is that weights are mapped to some preordered *bounded* interval $\mathcal{M}$. Some example bounded intervals would then be the unit interval on the reals, [0..1], the natural numbers joined with infinity, $\mathcal{N}^\infty$, or a set such as $\{-, \square, \bigcirc, +\}$, taking the reflexive closure of the order: $\{- < \square, - < \bigcirc, \square < +, \bigcirc < +, - < +\}$.

Definition 8.21 (Argumentation system)**.** An *argumentation system* is a tuple $AS = \langle \mathcal{L}, ^-, \mathcal{R}, \leqslant \rangle$ where

- $\mathcal{L}$ is a logical language closed under classical negation,
- $^-$ is a symmetric contrariness relation on $\mathcal{L}$ (p and $-p$ are said to be each other's *contradictories*),
- $\mathcal{R} = \mathcal{R}_s \cup \mathcal{R}_d$ is a set of strict ($\mathcal{R}_s$) and defeasible inference rules ($\mathcal{R}_d$) such that $\mathcal{R}_s \cap \mathcal{R}_d = \emptyset$,
- *weight* is a function from $r \in \mathcal{R}_d$ to $\mathcal{M}$.

The knowledge base is restrained to contain only axioms and premises. We replace the preference ordering again with a weight assignment to $\mathcal{M}$. It is fine to name the weight function on rules *weight* as well, since rules and knowledge are assumed to be disjoint within the logical language $\mathcal{L}$.

Definition 8.22 (Knowledge base)**.** A *knowledge base* in an argumentation system $\langle \mathcal{L}, ^-, \mathcal{R}, \leqslant \rangle$ is a pair $\langle \mathcal{K}, \textit{weight} \rangle$ where $\mathcal{K} \subseteq \mathcal{L}$ and *weight* is a function from $p \in \mathcal{K}_p$ to $\mathcal{M}$. Here $\mathcal{K}$ is partitioned into two subsets $\mathcal{K}_n$ (axioms) and $\mathcal{K}_p$ (ordinary premises).

Building on the definition of a content ordering, a *confidence measure* takes in defeasible content, maps it to a bounded set of weights, $\mathcal{M}$, while retaining properties of the derived ordering, implying a content-based ordering and finally combines it into a single value in $\mathcal{M}$. Thus, to derive the weight of an argument, we apply the content ordering, map its defeasible content to weights using the *weight* function, and then use a form of *propagation* and combination to derive a single weight in $\mathcal{M}$.

The combining process of weights should adhere to some very reasonable properties: the order in which we combine content should not matter, a strict argument should get the maximal weight and a strict argument should not weaken its superarguments (needed in the case of argument aggregation). The combining of weights can be achieved by applying an abstract operator $\bullet$ that adheres to these properties. This implies that $\mathcal{M}$, the operator $\bullet$ and the maximal element in $\mathcal{M}$, called $\top$, together form a *commutative monoid.*

Definition 8.23 (Confidence measure)**.** Let *Args* be a set of arguments and $\mathcal{L}$ the logical language used. Then a *confidence measure*, *cm*, is a function from a content ordering to a dictionary of weights, $\mathcal{M}$, such that the following properties hold:

1. $\langle \mathcal{M}, \bullet, \top \rangle$ is a commutative monoid, i.e.:

- $\forall a, b \in \mathcal{M}. a \bullet b \in \mathcal{M}$, (closure)
- $\forall a, b, c \in \mathcal{M}. (a \bullet b) \bullet c = a \bullet (b \bullet c)$, (associativity)
- $\forall a \in \mathcal{M}. \top \bullet a = a \bullet \top = a$, (identity)
- $\forall a, b \in \mathcal{M}. a \bullet b = b \bullet a$. (commutativity)

2. $cm : 2^{\mathcal{L}} \to \mathcal{M}$,
   $cm(\emptyset) = \top$,
   $cm(L) = weight(l_1) \bullet \ldots \bullet weight(l_n)$, with $l_i \in L$,
   $L_1 \subseteq L_2$ implies $cm(L_1) \leqslant cm(L_2)$.

The preservation of the content ordering guarantees that arguments which are weaker in content, will also be weaker in combined weight. The commutative monoid structure excludes unreasonable confidence measures, such as a mapping from defeasible content to $\{False, True\}$, mapping $\emptyset$ to $True$ and with $\downarrow$ (nor) as the $\bullet$ operator.

There are multiple valid instantiations of $\langle \mathcal{M}, \bullet, \top \rangle$ for a confidence measure, below are two useful examples.

Example 8.24 (Concrete dictionaries of weights)**.** Let *Args* be a set of arguments and $\mathcal{L}$ the corresponding logical language, then the following are two concrete instances of the commutative monoid $\mathcal{M}$:

- $\langle [0..1], *, 1 \rangle$,
- $\langle [0..1], minimum, 1 \rangle$.

Note that this only works if we assume the normal ordering on [0..1], otherwise preservation of the content ordering might be violated.

We can now modify the bop argumentation theories, as given before in Definition 3.72, to instead map weights using a content ordering and a confidence measure.

Definition 8.25 (General bop-argumentation theories (Adapted Def. 5.1 of [154]))**.** A *general bop argumentation theory* is a tuple $AT = \langle AS, KB, t, B, \mathcal{M}, w, \alpha, \beta, \gamma \rangle$ where AS is an argumentation system, KB is a knowledge base in AS as before, and

- $t \in \mathcal{L}$ (the main topic of the AT)
- $B \subseteq \mathcal{L}$ such that for no φ and $-\varphi$ are in B (we write $ebop(\varphi)$ iff $\varphi \in B$),
- $\mathcal{M}$ is a dictionary of weights,
- w is a *weight function* s.t.:
  - $w : \mathcal{A}_{AT} \to \mathcal{M}$,
  - $w = cm \circ content$.

- $\alpha, \beta, \gamma \in \mathcal{R}^+ \cup \{0\}$.

Note that the weight function, w, only depends on the content of A, avoiding circularity without having to explicitly define it.

Example 8.26 (Concrete bop-AT and weight assignment)**.** Let $\mathcal{M} = \langle [0..1], *, 1 \rangle$, be our dictionary of weights and assume a content ordering based on sets.

Consider the following premises and rules:

$$\begin{aligned}
\mathcal{K}_n &= \{q\}, \\
\mathcal{K}_p &= \{p, r\}, \\
\mathcal{R}_s &= \emptyset, \\
\mathcal{R}_d &= \{p, q \Rightarrow_{d_1} s;\ p, r, s \Rightarrow_{d_2} t\}, \\
\mathit{weight}(p) &= 0.3;\ \mathit{weight}(r) = 0.3, \\
\mathit{weight}(d_1) &= 0.9;\ \mathit{weight}(d_2) = 0.7;
\end{aligned}$$

We can construct the following arguments from this:

$$\begin{array}{ll}
A_1 : p & A_4 : A_1, A_2 \Rightarrow_{d_1} s \\
A_2 : q & A_5 : A_1, A_3, A_4 \Rightarrow_{d_2} t \\
A_3 : r &
\end{array}$$

The content ordering for these arguments would be the following:

$$\begin{aligned}
\mathit{content}(A_1) &= \{p\} \\
\mathit{content}(A_2) &= \emptyset \\
\mathit{content}(A_3) &= \{r\} \\
\mathit{content}(A_4) &= \{p, d_1\} \\
\mathit{content}(A_5) &= \{p, r, d_1, d_2\}
\end{aligned}$$

The content of an argument is a set, so note that reuse of premises (or rules) in an argument does not influence the content ordering. From the content ordering and using the weight function defined on our concrete dictionary of weights, we can apply the confidence measure:

$$\begin{aligned}
cm(\{p\}) &= \mathit{weight}(p) = 0.3 \\
cm(\emptyset) &= 1.0 \\
cm(\{r\}) &= 0.3 \\
cm(\{p, d_1\}) &= 0.3 * 0.9 = 0.27 \\
cm(\{p, r, d_1, d_2\}) &= 0.3 * 0.3 * 0.9 * 0.7 = 0.0567
\end{aligned}$$

Which immediately gives us to the weights of the arguments.

8.3.2 Attack and defeat

We adapt the notions of attack and rebuttal of [154] (see Definitions 3.69 and 3.70) while defining successful undermining in the obvious way.

Definition 8.27 (successful rebuttal under burden of persuasion (Def. 5.2 of [154]))**.** Argument A *successfully rebuts* argument B if A rebuts B on B' and

1. $\mathit{ibop}(\mathit{Conc}(A))$ and $w(A) > w(B') + \beta$; or else
2. $\mathit{dbop}(\mathit{Conc}(A))$ and $w(A) \nless w(B')$; or else
3. $w(A) + \beta \nless w(B')$.

Since we replaced assumptions by normal premises, an undermining attack does not always succeed, but instead, similar to rebutting attack, is dependent on the weights assigned.

Definition 8.28 (successful undermining under burden of persuasion)**.** Argument A *successfully undermines* argument B if A undermines B on φ and

1. $\mathit{ibop}(\mathit{Conc}(A))$ and $w(A) > \mathit{weight}(\varphi) + \beta$; or else
2. $\mathit{dbop}(\mathit{Conc}(A))$ and $w(A) \nless \mathit{weight}(\varphi)$; or else
3. $w(A) + \beta \nless \mathit{weight}(\varphi)$.

The previous notions can be combined in an overall definition of defeat:

Definition 8.29 (Defeat (Def. 3.21 of [145]))**.** Argument A *defeats* argument B iff A undercuts or successfully rebuts or successfully undermines B. Argument A *strictly defeats* argument B iff A *defeats* B and B does not defeat A.

Again we generate strict undercutters if $w(A) < \alpha$ and rules for any $B = B_1, \ldots, B_n \rightarrow \neg \mathit{Conc}(A)$ such that $w(B) \geqslant \gamma$.

This already integrates weight propagation and defeat into one framework.

8.4 Argument aggregation

In the version of ASPIC$^+$ by Prakken and Sartor [154] (see Section 3.5), multiple arguments for the same conclusion c are implicitly aggregated by the definition of successful rebuttal. Lets assume multiple arguments $A_1 \ldots A_n$ for the conclusion c with no undercuts or undermining attacks, and just rebuttals on the conclusion c. Then the only factor that matters in determining the successful defeat of an argument A_i is its weight, giving an implicit aggregation determined by the maximum weight. Namely if A_i is not successfully rebutted, then any other argument A_j with $\mathit{weight}(A_j) \geqslant \mathit{weight}(A_i)$ will also not be successfully rebutted.

Aggregation of arguments by taking the maximum is the sensible thing to do when we have no knowledge of the dependencies between arguments. However, if we know (some of) the dependencies we can make the weight of an aggregation of arguments possibly stronger than the arguments that are part of the aggregation. To do so we will make aggregation explicit in the language by providing an additional way to construct argument expanding on Definition 8.3.

Definition 8.30 (Arguments (continued from Definition 8.3))**.** An *argument* A on the basis of a knowledge base $\langle \mathcal{K}, \leqslant' \rangle$ in an argumentation system $\langle \mathcal{L}, ^-, \mathcal{R}, \leqslant \rangle$ is:

4. $A_1 \vee A_2 \vee \ldots \vee A_n \Rightarrow \psi$ if $A_1, \ldots, A_n$ are arguments given that $\mathit{Conc}(A_1) = \mathit{Conc}(A_2) = \ldots = \mathit{Conc}(A_n) = \psi$,
   $\mathit{Prem}(A) = \mathit{Prem}(A_1) \cup \ldots \cup \mathit{Prem}(A_n)$,
   $\mathit{Conc}(A) = \psi$,
   $\mathit{Sub}(A) = \mathit{Sub}(A_1) \cup \ldots \cup \mathit{Sub}(A_n) \cup \{A\}$,
   $\mathit{DefRules}(A) = \mathit{DefRules}(A_1) \cup \ldots \cup \mathit{DefRules}(A_n)$,
   $\mathit{TopRule}(A) = A_1 \vee A_2 \vee \ldots \vee A_n \Rightarrow \psi$

An aggregated argument for φ is built up by two or more arguments for φ and the inference step counts as a defeasible rule (extending the naming function n appropriately), allowing the conclusion to be rebutted.

8.4.1 Generalised content orderings

The defeasible content of an aggregated argument is now no longer just a set of logical formula, but a sequence of these sets. To handle this we will need to change our definition of content ordering accordingly.

Definition 8.31 (Generalised content ordering)**.** A *generalised content ordering* of a set of arguments *Args* with language $\mathcal{L}$ is a function such that:

- $\mathit{content} : \mathit{Args} \to D$ (where D is a distributive lattice of $2^{\mathcal{L}}$),
- $\mathit{content}(A) = \mathit{lift}(\psi)$, if $A : \psi$,
- $\mathit{content}(A) = \mathit{content}(A_1) \wedge \ldots \wedge \mathit{content}(A_n)$, if $A : A_1, \ldots A_n \to \psi$,
- $\mathit{content}(A) = \mathit{lift}(\mathit{TopRule}(A)) \wedge \mathit{content}(A_1) \wedge \ldots \wedge \mathit{content}(A_n)$, if $A : A_1, \ldots A_n \Rightarrow \psi$,
- $\mathit{content}(A) = \mathit{content}(A_1) \vee \ldots \vee \mathit{content}(A_n)$, if $A : A_1, \ldots A_n \Rightarrow \psi$.

Here *lift* is a function that takes a defeasible element and lifts it into the distributive lattice.

Example 8.32 (Concrete content ordering of an aggregated argument)**.** One concrete version of the generalised content ordering would be a set of sets. Given a set of arguments *Args* with language $\mathcal{L}$, let *content* be a function such that:

- $\mathit{content} : \mathit{Args} \rightarrow (2^{2^{\mathcal{L}}})$,
- $\mathit{content}(A) = \{\{\psi\}\}$, if $A : \psi$,
- $\mathit{content}(A) = \mathit{content}(A_1) \times \ldots \times \mathit{content}(A_n)$, if $A : A_1, \ldots A_n \rightarrow \psi$ (where $A_1 \times A_2$ is the set of cartesian products for all combinations of sets between two sets),
- $\mathit{content}(A) = \{\{\mathit{TopRule}(A)\}\} \times \mathit{content}(A_1) \times \ldots \times \mathit{content}(A_n)$, if $A : A_1, \ldots A_n \Rightarrow \psi$,
- $\mathit{content}(A) = \mathit{content}(A_1) \cup \ldots \cup \mathit{content}(A_n)$, if $A : A_1, \ldots A_n \Rrightarrow \psi$ (where $\cup$ is set concatenation).

We have the following derived order, where $\supseteq$ is now superset on sets of sets:

- $\mathit{content}(A) \supseteq \mathit{content}(B)$ iff $A \leqslant B$,
- $\mathit{content}(A) = \emptyset$ implies $B \leqslant A$, for all B.

The above definition again corresponds to a concrete instance of how LA orders arrows in the category of arguments using a join-semilattice (here $\emptyset$ is $\top$ and we use the above ordering).

Consider the following premises and rules:

$$\begin{aligned}
&\mathcal{K}_n = \emptyset,\\
&\mathcal{K}_p = \{a, c, d, g, i, h\},\\
&\mathcal{R}_s = \{a \rightarrow b;\ c \rightarrow b;\ b, d \rightarrow e;\ g, i \rightarrow e;\ e, h \rightarrow f\}\\
&\mathcal{R}_d = \emptyset,
\end{aligned}$$

From this we can construct the an argument containing multiple aggregations (an inference from multiple equal premises to an equal conclusion denotes an aggregation):

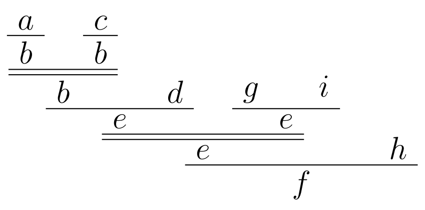


Figure 8.1: An aggregated argument for f

This argument contains several subarguments which can formally be written as follows:

$A_1 : a$

$A_2 : c$

$A_3 : d$

$A_4 : g$

$A_5 : i$

$A_6 : h$

$A_7 : A_1 \to b$

$A_8 : A_2 \to b$

$A_9 : A_7 \vee A_8 \Rightarrow b$

$A_{10} : A_9, A_3 \to e$

$A_{11} : A_4, A_5 \to e$

$A_{12} : A_{10} \vee A_{11} \Rightarrow e$

$A_{13} : A_{12}, A_6 \to f$

Now we can calculate the content of A_{13}.

$$\begin{aligned}
\mathit{content}(A_{13}) &= \mathit{content}(A_{12}) \times \mathit{content}(A_6) \\
&= \mathit{content}(A_{12}) \times \{\{h\}\} \\
&= (\mathit{content}(A_{10}) \cup (\mathit{content}(A_4) \times \mathit{content}(A_5))) \times \{\{h\}\} \\
&= (\mathit{content}(A_{10}) \cup (\{\{g\}\} \times \{\{i\}\})) \times \{\{h\}\} \\
&= (\mathit{content}(A_{10}) \cup (\{\{g, i\}\})) \times \{\{h\}\} \\
&= (\mathit{content}(A_{10}) \times \{\{h\}\}) \cup (\{\{g, i\}\} \times \{\{h\}\}) \qquad (1.) \\
&= (\mathit{content}(A_{10}) \times \{\{h\}\}) \cup \{\{g, i, h\}\} \\
&= (\mathit{content}(A_{10}) \times \{\{h\}\}) \cup \{\{g, i, h\}\} \\
&= (\mathit{content}(A_9) \times \mathit{content}(A_3) \times \{\{h\}\}) \cup \{\{g, i, h\}\} \\
&= (\mathit{content}(A_9) \times \{\{d\}\} \times \{\{h\}\}) \cup \{\{g, i, h\}\} \\
&= (\mathit{content}(A_9) \times \{\{d, h\}\}) \cup \{\{g, i, h\}\} \\
&= ((\mathit{content}(A_7) \cup \mathit{content}(A_8)) \times \{\{d, h\}\}) \cup \{\{g, i, h\}\} \\
&= ((\mathit{content}(A_1) \cup \mathit{content}(A_2)) \times \{\{d, h\}\}) \cup \{\{g, i, h\}\} \\
&= (\{\{a\}\} \cup \{\{c\}\}) \times \{\{d, h\}\}) \cup \{\{g, i, h\}\} \\
&= (\{\{a\}, \{c\}\} \times \{\{d, h\}\}) \cup \{\{g, i, h\}\} \\
&= \{\{a, d, h\}, \{c, d, h\}\} \cup \{\{g, i, h\}\} \\
&= \{\{a, d, h\}, \{c, d, h\}, \{g, i, h\}\}
\end{aligned}$$

Step (1.) is achieved by distributivity of $\times$ over $\cup$.

8.4.2 Probabilistic confidence measures

We now define a *probabilistic confidence measure* to give weights to our generalised content ordering. First, because we assume a probabilistic domain, we can take our weight of dictionaries, $\langle \mathcal{M}, \bullet, \top \rangle$ to be $\langle [0..1], *, 1 \rangle$. Then, the probability that an argument without aggregations is true, is the probability of all its defeasible rules and premises multiplied together. For aggregations however, we use the addition rule for probabilities implicitly performing analysis on which assumptions are shared syntactically. Remembering that the addition rule for probabilities is $p(X \vee Y) = p(X) + p(Y) - p(X \wedge Y)$, we have:

Definition 8.33 (Probabilistic confidence measure). Let *Args* be a set of arguments, $\mathcal{L}$ the logical language and let $\langle [0..1], *, 1 \rangle$ with 0 as e be the dictionary of weights. Then a *probabilistic confidence measure*, *cm*, is a function from a generalised content ordering to [0..1], such that the following properties hold:

- $cm(\top) = 1$,
- $cm(L_1 \vee L_2) = cm(L_1) + cm(L_2) - cm(L_1 \wedge L_2)$,
- $cm(L) = \prod_{l_i \in L} weight(l_i)$, for non-disjunctive L.

The probabilistic confidence measure is domain-independent and therefore assumes that all propositions are dependent[65].

Example 8.34 (Concrete probabilistic confidence measure on a concrete content ordering)**.** A concrete probabilistic confidence measure is a concrete version of a probabilistic confidence measure, working on concrete orderings as define in Example 8.32. Let *Args* be a set of arguments, $\mathcal{L}$ the logical language and let $\langle [0..1], *, 1 \rangle$ with 0 as e be the dictionary of weights. Then a concrete probabilistic confidence measure is a function *cm* from a concrete content ordering to [0..1], such that the following properties hold:

- $cm : (2^{2^{\mathcal{L}}}) \rightarrow [0..1]$
- $cm(\{\emptyset\}) = 1$,
- $cm(\{L_1, L_2, \ldots L_n\}) = cm(\{L_1\}) + cm(\{L_2 \ldots L_n\}) - cm(\{L_1 \times L_2 \ldots, L_1 \times L_n\})$, with $\times$ normal cartesian product of sets,
- $cm(\{L\}) = \prod_{l_i \in L} weight(l_i)$:

Continuing from Example 8.32, assume the following weights:

$$weight(a) = 0.6;\ weight(c) = 0.3;\ weight(d) = 0.8,$$
$$weight(g) = 0.4;\ weight(i) = 0.5;\ weight(h) = 0.9.$$

[65]It would be possible to add domain-specific information about independencies of propositions.

We can now calculate the probability/weight of argument A_{13}:

$$
\begin{aligned}
w(A_{13}) &= cm(content(A_{13})) \\
&= cm(\{\{a,d,h\},\{c,d,h\},\{g,i,h\}\}) \\
&= cm(\{\{a,d,h\}\}) + cm(\{\{c,d,h\},\{g,i,h\}\}) \\
&\qquad\qquad - cm(\{\{a,d,h\}\times\{c,d,h\},\{a,d,h\}\times\{g,i,h\}\}) \\
&= cm(\{\{a,d,h\}\}) + cm(\{\{c,d,h\},\{g,i,h\}\}) \\
&\qquad\qquad - cm(\{\{a,c,d,h\},\{a,d,g,i,h\}\}) \\
&= weight(a) * weight(d) * weight(h) + \ldots - \ldots \\
&= 0.6 * 0.8 * 0.9 + \ldots - \ldots \\
&= 0.432 + cm(\{\{c,d,h\},\{g,i,h\}\}) - \ldots \\
&= 0.432 + (cm(\{\{c,d,h\}\}) + cm(\{\{g,i,h\}\}) \\
&\qquad - cm(\{\{c,d,h\}\times\{g,i,h\}\})) - \ldots \\
&= 0.432 + (cm(\{\{c,d,h\}\}) + cm(\{\{g,i,h\}\}) \\
&\qquad - cm(\{\{c,d,g,i,h\}\})) - \ldots \\
&= 0.432 + (0.3 * 0.8 * 0.9 + 0.4 * 0.5 * 0.9 - 0.3 * 0.8 * 0.4 * 0.5 * 0.9) - \ldots \\
&= 0.432 + (0.3 * 0.8 * 0.9 + 0.4 * 0.5 * 0.9 - 0.3 * 0.8 * 0.4 * 0.5 * 0.9) - \ldots \\
&= 0.432 + (0.216 + 0.18 - 0.0432) - \ldots \\
&= 0.432 + 0.3528 - \ldots \\
&= 0.7848 - cm(\{\{a,c,d,h\},\{a,d,g,i,h\}\}) \\
&= 0.7848 - (cm(\{\{a,c,d,h\}\}) + cm(\{\{a,d,g,i,h\}\}) - \\
&\qquad\qquad cm(\{\{a,c,d,h\}\times\{a,d,g,i,h\}\})) - \\
&= 0.7848 - (cm(\{\{a,c,d,h\}\}) + cm(\{\{a,d,g,i,h\}\}) \\
&\qquad\qquad cm(\{\{a,c,g,d,i,h\}\})) \\
&= 0.7848 - (0.6 * 0.3 * 0.8 * 0.9 + 0.6 * 0.8 * 0.4 * 0.5 * 0.9 - \\
&\qquad\qquad 0.6 * 0.3 * 0.8 * 0.4 * 0.5 * 0.9) \\
&= 0.7848 - (0.1296 + 0.0864 - 0.02592) \\
&= 0.7848 - 0.19008 \\
&= 0.59472
\end{aligned}
$$

This integrates probabilistic weights, weight propagation, argument aggregation, proof standards, and defeat into one framework.

8.5 Implementation of content orderings, weight propagation and argument aggregation

This section describes the implementation of the previously defined notions of content orderings, weight propagation, argument aggregation and (probabilistic) content measures. Definitions corresponding to defeats and argument generation have not been fully implemented.

8.5.1 Arguments

ASPIC$^+$ uses a propositional language, so we can assume premises to be atomic propositions in propositional logic; i.e., all propositions are either positive or negative literals (multiple times negated literals can just be reduced back to single or non-negated literals). Taking literals to be strings suffice in the following, and propositions can then be formed by pairing a literal with a Boolean to denote whether it is positive or negative:

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

Defeasible rules should be able to be attacked and have to given names which are also in $\mathcal{L}$. We thus give defeasible rules *String* labels, which are assumed to be disjoint from propositions.

```
type Label = String
data StrictRule = SR {
  premSR :: [Proposition],
  concSR :: Proposition}
deriving (Eq, Show)
data DefeasibleRule = DR {
  lblDR :: Label,
  premDR :: [Proposition],
  concDR :: Proposition}
deriving (Eq, Show)
```

An argument is then a proposition, a defeasible/strict/ argument containing a list of subarguments and a defeasible/strict rule or an accrual argument containing a list of subarguments for the same conclusion.

```
data Argument = A Proposition
              | DefArg [Argument] DefeasibleRule
              | StrictArg [Argument] StrictRule
```

```
                        | AggrArg [Argument] Proposition
    deriving Eq
```

Prem, *Conc*, *Sub*, *DefRules*, *TopRule* follow the structure of arguments as defined in Definition 8.3 and 8.30. We additionally define the set of conclusions, *Concs*, and the set of premises, *Prems*.

```
prem :: Argument → [Proposition]
prem (A c)               = [c]
prem (DefArg args _)     = unionMap prem args
prem (StrictArg args _)  = unionMap prem args
prem (AggrArg args _)    = unionMap prem args

conc :: Argument → Proposition
conc (A c)                    = c
conc (DefArg _ (DR _ _ c))    = c
conc (StrictArg _ (SR _ c))   = c
conc (AggrArg _ c)            = c

sub :: Argument → [Argument]
sub a@(A _)               = [a]
sub a@(DefArg args _)     = unionMap sub args `union` [a]
sub a@(StrictArg args _)  = unionMap sub args `union` [a]
sub a@(AggrArg args _)    = unionMap sub args `union` [a]

defrules :: Argument → [DefeasibleRule]
defrules (A _)               = []
defrules (DefArg args d)     = unionMap defrules args `union` [d]
defrules (StrictArg args _)  = unionMap defrules args
defrules (AggrArg args _)    = unionMap defrules args

toprule :: Argument → Either DefeasibleRule StrictRule
toprule (A _)             = undefined
toprule (DefArg _ d)      = Left d
toprule (StrictArg _ s)   = Right s
toprule (AggrArg _ _)     = undefined

concs :: Argument → [Proposition]
concs = nub ∘ map conc ∘ sub

prems :: Argument → [Proposition]
prems = nub ∘ concatMap prem ∘ sub
```

unionMap is a *concatMap* on lists that applies union instead of ++[66].

[66]For the sake of presentation we use lists as sets throughout the implementation.

```
unions :: Eq a ⇒ [[a]] → [a]
unions = foldr union []

unionMap :: Eq b ⇒ (a → [b]) → [a] → [b]
unionMap f xs = unions $ map f xs
```

8.5.2 Argumentation system and argumentation theory

We assume a concrete content ordering, based on sets of sets (represented by lists of lists) and as a confidence measure we take $\langle M, \bullet, \top \rangle$ to be $\langle [0..1], *, 1 \rangle$.

```
type M = Double

instance Monoid M where
  mempty = 1.0
  mappend = (∗)
```

Although our implementation will never make use it, Haskell technically still allows the value of M to fall outside of the unit interval.

The knowledge base corresponding to Definition 8.22 can then be defined:

```
data KnowledgeBase = KB {
  kn :: [Proposition],            -- necessary axioms
  kp :: [Proposition],            -- premises
  weightK :: Proposition → M      -- weight function,
                                  -- mapping knowledge to a weight
                                  -- (replaces ⩽)
  }
```

The rule base in the argumentation system from Definition 8.21 can be implemented similarly:

```
data RuleBase = RB {
  rs :: [StrictRule],                -- strict rules
  rd :: [DefeasibleRule],            -- defeasible rules
  weightR :: DefeasibleRule → M      -- weight function, mapping
                                     -- defeasible rules to a weight
                                     -- (replaces ⩽′)
  }
```

Combining the *RuleBase* and *KnowledgeBase*, we can now define the argumentation theory (Definition 8.25):

```
data ArgumentationTheory = AT {
  rb :: RuleBase,
  kb :: KnowledgeBase,
  t :: Proposition,             -- topic of the conversation
  beliefs :: [Proposition],     -- Beliefs
  w_at :: [Argument] → M,
  α, β, γ :: M
  }
```

8.5.3 General and concrete content (based) orderings

We first define a general content ordering, by defining the the type of the distributive lattice. Our first attempt is a datatype that has two sets, each a subset of the logical language $\mathcal{L}$, respectively containing premises and rules.

```
data D = D [Proposition] [DefeasibleRule]
  deriving Show
```

The concrete content ordering corresponding to Definition 8.12, for an argument A is then $DefRules(A) \cup Prem(A)$. In Haskell using the D datatype:

```
contentOrdering :: Argument → D
contentOrdering a = D (prems a) (defrules a)
```

Exploiting the fact we already defined our M to be a *Monoid* we can easily define our concrete confidence measure:

```
confidenceMeasure :: D → M
confidenceMeasure (D ps r) = mconcat $ map weightRule r
                               ++ map weightPrem ps
```

Then the standard weight function for any bop-argumentation theory extended with weight propagation (and no argument aggregation) is:

```
w :: Argument → M
w = confidenceMeasure ∘ contentOrdering
```

8.5.4 Aggregation and the probabilistic confidence measure

Generalising the implementation of the content ordering to handle aggregation we now take a set of D to be our distributive lattice (represented as a list of Ds). We first define $\wedge$ on sets, by defining the *andd* function which takes together two elements of D giving back a D:

$$\begin{aligned}&\mathit{andd} :: D \to D \to D\\ &\mathit{andd}\ (D\ \mathit{prems}\ \mathit{lbls})\ (D\ \mathit{prems}'\ \mathit{lbls}') = D\ (\mathit{prems}\ \text{`}\mathit{union}\text{`}\ \mathit{prems}')\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{lbls}\ \text{`}\mathit{union}\text{`}\ \mathit{lbls}')\end{aligned}$$

Then the cartesian product of sets of sets ($\wedge$ for sets of sets):

$$\begin{aligned}&\mathit{crosses} :: [[D]] \to [D]\\ &\mathit{crosses} = \mathit{foldr}\ (\lambda \mathit{ds}\ \mathit{ds}' \to [\mathit{andd}\ d\ d' \mid d \leftarrow \mathit{ds}, d' \leftarrow \mathit{ds}'])\ [D\ [\,]\ [\,]]\end{aligned}$$

So now we can define our probabilistic content ordering:

$$\begin{aligned}&\mathit{probD2} :: \mathit{Argument} \to [D]\\ &\mathit{probD2}\ (A\ c) &&= [D\ [c]\ [\,]] \quad \text{-- equivalent to } \mathit{lift}\ c\\ &\mathit{probD2}\ (\mathit{DefArg}\ \mathit{args}\ d) &&= \mathit{crosses}\ ([D\ [\,]\ [d]] : \mathit{map}\ \mathit{probD2}\ \mathit{args})\\ &\mathit{probD2}\ (\mathit{StrictArg}\ \mathit{args}\ \_) &&= \mathit{crosses}\ (\mathit{map}\ \mathit{probD2}\ \mathit{args})\\ &\mathit{probD2}\ (\mathit{AggrArg}\ \mathit{args}\ c) &&= \mathit{concatMap}\ \mathit{probD2}\ \mathit{args}\end{aligned}$$

And our probabilistic confidence measure:

$$\begin{aligned}&p :: [D] \to M\\ &p\ [\,] &&= 0.0\\ &p\ (d : \mathit{xs}) &&= \mathit{confidenceMeasure}\ d + p\ \mathit{xs} - p\ (\mathit{map}\ (\mathit{andd}\ d)\ \mathit{xs})\end{aligned}$$

Again immediately giving us our weight function for any bop-argumentation theory extended with weight propagation and aggregation:

$$\begin{aligned}&\mathit{probW} :: \mathit{Argument} \to M\\ &\mathit{probW} = p \circ \mathit{probD2}\end{aligned}$$

8.5.5 Example

Given the implementation of content orderings and the probabilistic confidence measure, we can give the full implementation of the Example 8.32 and 8.34:

$$\begin{aligned}&\mathit{kn1} :: [\mathit{Proposition}]\\ &\mathit{kn1} = [\,]\\ &\mathit{kp1} :: [\mathit{Proposition}]\\ &\mathit{kp1} = \mathit{mkAssumptions}\ [\texttt{"a"}, \texttt{"c"}, \texttt{"d"}, \texttt{"g"}, \texttt{"i"}, \texttt{"h"}]\end{aligned}$$

```
kb1 :: KnowledgeBase
kb1 = KB kn1 kp1 weightPrem

weightPrem :: Proposition → Double
weightPrem (True, "a") = 0.6
weightPrem (True, "c") = 0.3
weightPrem (True, "d") = 0.8
weightPrem (True, "g") = 0.4
weightPrem (True, "i") = 0.5
weightPrem (True, "h") = 0.9
weightPrem _           = error "no weight assigned"

weightRule :: DefeasibleRule → Double
weightRule = error "No defeasible rules"

rb1 :: RuleBase
rb1 = RB rs1 rd1 weightRule

rd1 :: [DefeasibleRule]
rd1 = []

r1, r2, r3, r4, r5 :: StrictRule
r1 = mkStrictRule ["a"] "b"
r2 = mkStrictRule ["c"] "b"
r3 = mkStrictRule ["b", "d"] "e"
r4 = mkStrictRule ["g", "i"] "e"
r5 = mkStrictRule ["e", "h"] "f"

rs1 :: [StrictRule]
rs1 = [r1, r2, r3, r4, r5]

aggrArg1, aggrArg2, aggrArg3, aggrArg4, aggrArg5, aggrArg6 :: Argument
aggrArg1 = A (mkProp "a")
aggrArg2 = A (mkProp "c")
aggrArg3 = A (mkProp "d")
aggrArg4 = A (mkProp "g")
aggrArg5 = A (mkProp "i")
aggrArg6 = A (mkProp "h")

  -- a → b
aggrArg7 :: Argument
aggrArg7 = mkStrictArg [aggrArg1] r1

  -- c → b
aggrArg8 :: Argument
aggrArg8 = mkStrictArg [aggrArg2] r2

  -- a → b ∨ c → b
```

```
aggrArg9 :: Argument
aggrArg9 = mkAggrArg [aggrArg7, aggrArg8]
  -- (a → b ∨ c → b) → e
aggrArg10 :: Argument
aggrArg10 = mkStrictArg [aggrArg9, aggrArg3] r3
  -- g,i → e
aggrArg11 :: Argument
aggrArg11 = mkStrictArg [aggrArg4, aggrArg5] r4
  -- ((a → b ∨ c → b) → e) ∨ g,i → e
aggrArg12 :: Argument
aggrArg12 = mkAggrArg [aggrArg10, aggrArg11]
  -- (((a → b ∨ c → b) → e) ∨ g,i → e), h → f
aggrArg13 :: Argument
aggrArg13 = mkStrictArg [aggrArg12, aggrArg6] r5
```

And as expected:

```
probD2 aggrArg13
 > [D [(True, "a"), (True, "d"), (True, "h")] []
   , D [(True, "c"), (True, "d"), (True, "h")] []
   , D [(True, "g"), (True, "i"), (True, "h")] []]
probW aggrArg13
 > 0.59472
```

8.6 Related work

The content (based) ordering, confidence measure and argument aggregation from this chapter are based on the concepts applied in the Logic of Argumentation [103]. The ideas are combined with two different versions of ASPIC$^+$ [154, 122].

Pollock [142, 138, 141], and the derived work from Liang and Wei [109] use variable degrees of justification, but justify a conservative approach of only using the max operator for combining the weights. This is justified in the context of arguments with no contrary evidence or additional information about the source of defeasibility. Alternatively, when considering diminishes and accrual, you might want the contrary evidence to influence the decision by letting the con-arguments accrue more strongly than the pro-arguments.

8.7 Conclusions and future work

This chapter discussed a general framework that extends ASPIC$^+$ [154, 122] with multiple features:

- *content orderings* that construct a preference relation over arguments by comparing the defeasible content of arguments;
- *content based orderings* that build on content orderings, by allowing additional sources of preferences, while keeping a pre-order;
- *confidence measures* that map content orderings to a content based ordering, by assigning weights to individual premises and defeasible rules, combining them into a single weight;
- a mathematically founded approach to combining weights, using *commutative monoids*;
- *argument aggregation* on any content-based ordering, by making use of a distributive lattice;
- *probabilistic confidence measures*, a special case of confidence measures and argument aggregation, that takes a syntactical approach to calculating probabilities of arguments.

The general framework, with the above features has been implemented in Haskell.

Future work could look further into using Dempster-Shafer in an argumentation context [173], investigating which combination operators could be applied to the framework developed in this chapter. A further motivation of our approach could be achieved by showing how the principles of accrual by Prakken [144] and the rationality postulates by Caminada and Amgoud [29], see Section 3.6, are adhered to.

Chapter 9

Conclusion

This chapter gives a summary of the achieved work and discusses directions for future work.

9.1 Summary

The objectives of this thesis were two-fold: to construct a general framework for relating, implementing, and formally verifying argumentation models and translations between them; and secondly, to produce several use cases that demonstrate the general usefulness and applicability of the approach. To achieve these goals:

- it was demonstrated that functional programming, and the Haskell language in particular, are suitable for implementing abstract argumentation models and complex algorithms in that domain, providing an intuitive implementation that is close the original mathematical specifications of both a library and command-line application (Chapter 4);
- it was shown that Haskell can also be applied to implement a structured argumentation model, Carneades. The implementation is closely related to the mathematical specification and furthermore usable as a domain specific language (Chapter 5);
- a translation from Carneades into ASPIC$^+$ was developed, taking any CAES and translating it into an ASPIC$^+$ argumentation theory, thereby relating it to Dung's argumentation framework through the existing translation from ASPIC$^+$ into Dung [145]. Important properties of the translation were proved to hold and algorithms for generating arguments were developed (Chapter 6);

- a translation from Carneades into ASPIC$^+$ was developed, taking any CAES and translating it into an ASPIC$^+$ argumentation theory, thereby relating it to Dung's argumentation framework through the existing translation from ASPIC$^+$ into Dung [145]. Important properties of the translation were proved to hold and algorithms for generating arguments were developed (Chapter 6);
- a translation from Carneades into AFs was derived, furthermore providing an implementation of the definitions and the correspondence properties (Chapter 7);
- the implementation of AFs in Haskell was formalised into a theorem prover, possibly providing the first mechanical formalisation of an argumentation model (Chapter 7);
- a general framework was constructed, combining the implementations and formalisation into a verified pipeline that is able to take a Carneades argument evaluation structure, translate it and evaluate it with existing efficient SAT-solvers (Chapter 7);
- a generalisation of the ASPIC$^+$ framework, incorporating weight propagation, argument accrual and probabilistic weights was defined and furthermore implemented using the previously defined framework (Chapter 8).

A significant part of the objective of the thesis has been fulfilled, having been able to define the general framework and various use cases that apply the framework. However, a complete formalisation of Carneades, and the translation of Carneades into AFs, proved to be out of scope for this thesis.

9.2 Future work

9.2.1 Incorporate ASPIC$^+$ and efficient argument generation into a general framework

ASPIC$^+$ has proven to be a good translation target for various other structured argumentation models (see Chapter 6). However, although ASPIC$^+$ is a good foundation for translating argumentation models, the lack of concrete argument generation algorithms, particularly techniques that allow for fast algorithm generation for sub-cases of the framework, force you to write a specialised argument generation algorithm yourself each time (see Chapter 6 for the specialised argument generation for Carneades into Dung to avoid exponential blowup).

9.2.2 Extending the formalisation of Dung's AFs in Agda

The formalisation of Dung's AFs [48] in Agda discussed in Chapter 7 formalises AFs up to grounded semantics. An interesting direction would be to try to reprove some of Dung's original result in a constructive setting. It would be furthermore be worthwhile to verify that the approach taken to verify the grounded semantics is generalisable to the other standard semantics of Dung.

9.2.3 Formalising Carneades and the translation from Carneades into Dung in a theorem prover

To complete the overview picture given in Section 1.1, I would need to formalise Carneades and the translation from Carneades into Dung into Agda. An initial attempt has been made with Tom Gordon [84], but the technical nature of the formalisations makes it hard to publish the materials in an argumentation oriented venue and it has therefore not been published yet. Future work would thus be to complete the formalisation by proving properties of the formalisation and to write it such that it is understandable to an average argumentation theorist.

9.2.4 Rationality postulates and rules of aggregation for Chapter 8

An obvious extension to the work done in Chapter 8, is to show that after generalising ASPIC$^+$ with proof standards/burdens, that the rationality postulates (see Section 3.6) are still satisfied. It would furthermore be interesting to see which of the principles of accrual as formulated by Prakken [144] will hold in our extended system. Achieving this would show that the aggregation techniques as formulated in LA are principled and a good approach to take.

9.2.5 Extending categorical interpretation of argumentation to include negation

The Logic of Argumentation [103] is an extension of the interpretation of typed lambda calculus in category theory through the Curry-Howard-Lambek correspondence [104]. One problem Ambler mentions more explicitly in his previous and more technical paper [1], is the lack of proper handling of negation in his system. The version of the Logic of Argumentation as given by Krause et al. [103], partially solves this problem by handling this on the meta-level using non-numerical qualifications of argument: {*certain*, *confirmed*, *probable*, *plausible*, *supported*, *open*}, but this is not really satisfying in practice, especially not categorically. Interestingly, the generalisation in Chapter 8 seems to handle negation in a principled way and most importantly, numerically, through the use of proof standards. A possible research question to ask here is thus:

> How can we take back the notion of negation from Chapter 8 and convert it into a categorical interpretation of argumentation?

Appendix A

Tools for the implementation of argumentation models

The structured approach to argumentation has seen an increase in 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 chapter discusses which methods are taken in the thesis to alleviate this problem, i.e., it will be demonstrated how the functional programming language Haskell can naturally express mathematical definitions and it will be sketched how a theorem prover, Agda, can verify such a Haskell implementation. Furthermore, methods are provided for the streamlining the documenting of code, showing how *literate programming* allows an implementer of an argumentation model to write formal definition, implementation, and documentation in one file. All code in the thesis has been made open source, publicly available and reusable.

A.1 Programming methodology

A.1.1 Functional programming

When looking at recent developments in abstract argumentation [35], we can see that answer set programming (ASP) and Prolog have played 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 [48] in a very natural way, making it possible to make the code partly self-documenting.

For structured argumentation, a truly convincing implementation language has yet to emerge. There are various implementations done in Java [164, 174], but they are quite far removed from the logical specification making it significantly harder to verify whether the implementation is actually correct. Instead, this thesis applies a functional programming approach, using the programming language Haskell [114]. 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 [96], while additionally simplifying future verification of such implementations.

For example, the definition of admissibility in an argumentation framework (see also Section 3.2) can be implemented almost directly in Haskell.

Definition A.1 (Admissibility)**.** A conflict-free set of arguments S is admissible iff every argument X in S is acceptable with respect to S, i.e. $S \subseteq F_{AF}(S)$.

The type of Haskell definition of *admissible* demonstrates that most of Dung's definitions, including admissibility, are parameterised by a given AF and furthermore that to implement most of the definitions, we need a notion of equality on arguments.

```
admissible :: Eq arg ⇒ DungAF arg → [arg] → Bool
admissible af args = conflictFree af args ∧
                     args ⊆ f af args
```

A.1.2 Formalisation in a theorem prover

Given the complexity of some of the structured models and translation, it is not trivial to verify the correctness of an implementation. One way to achieve this beyond the proofs done on paper, is to formalise the implementation through an interactive theorem prover, such as Agda. 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 Agda very natural.

Below is an inductive data type that builds up a proof that says that a list satisfies a predicate P for all its elements. That is, we can inductively prove that a list satisfies a predicate for all of its elements, by returning the trivial proof given that *xs* is empty, or by showing that it holds for the head of the list x and a proof that P holds for all its elements in the tail of the list.

```
data All {A : Set} (P : A → Set) : List A → Set where
  all[ ]   : All P [ ]
  _:all:_ : forall {x xs} → P x → All P xs → All P (x :: xs)
```

We can then prove that all elements in a given list are even by defining inductively what it means to be even.

```
data isEven : Nat → Set where
  evZero    : isEven 0
  evSucSuc : {n : Nat} → isEven n → isEven (suc (suc n))
allEx0 : All isEven [ ]
allEx0 = all[ ]
allEx1 : All isEven (4 :: 2 :: [ ])
allEx1 = evSucSuc (evSucSuc evZero) :all:
              evSucSuc evZero :all:
                all[ ]
```

In Chapter 7 an Agda formalisation of Dung's AFs will be discussed.

A.2 Tools

A.2.1 Literate programming

Although implementations of structured argumentation models often have appropriate user instructions, it is not common that such an implementation also documents its methods of implementation aside from the algorithms used. To make this process more attractive, literate programming [102] is employed, 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 LaTeXand Haskell code, while still being readable by a standard Haskell compiler. Additionally, the tool called lhs2tex [113] provides the user with the automatic typesetting of Haskell code within a Literate Haskell file, generating appropriate LaTeXcode. This ensures that the documentation is kept up to date along with the programming code. All sections in the thesis containing code are written using literate programming, with source code provided for each individual section. The source code of these chapters contain all definitions, including definitions left out for brevity, together with explanations. These literate programming files can immediately be loaded into the Haskell compiler.

A.2.2 Open source software and public repositories

As discussed in Chapter 1, most implementations of structured argumentation models are not publicly available (any more) or are closed source. I believe that to progress the knowledge of implementing techniques for (structured) argumentation models, implementations should be made publicly available. All implementations in the thesis have been made available through the standard Haskell package repository called Hackage[67], providing source files and automatic generation of HTML documentation of the API. Code has also been put on the public git repository website, GitHub[68] [79]. The public availability and documentation of the implementation has attracted other people to contribute as well, e.g. Chapter 5 has been extended by a student, Stefan Sabev, and Chapter 4 has been used by other PhD students and companies.

[67]`http://hackage.haskell.org/`

[68]`https://github.com/nebasuke/thesis`

Appendix B

Minor technical contributions to existing argumentation models and algorithms

The thesis also contains a variety of smaller technical contributions, generalising or fixing formal specifications already existing in literature. In particular:

- The revised version of Definition 3.5 for ASPIC$^+$ (2010) (see just below Definition 3.42) fixes handling of issue premises, by defining arguments with issue premises to be not acceptable at all (instead of excluding them from extensions).

- Definitions in Section 3.4 have been split into a stage-specific part and a part with dialogical notions. The stage-specific part is largely inspired by Brewka and Gordon [22], while the phrasing of the definitions for the dialogical notion is new.

- Definition 3.50 has an additional constraint, requiring the intersection of premises and exceptions of an argument to be empty, in line with the definition of Brewka and Gordon [22].

- Definition 3.51 constraints the *weight* function to be total. Evaluation of arguments without weights, with exception of the scintilla of evidence standard, was undefined. I have therefore assumed this to be incorrect.

- Definition 3.52 defines the acyclicity of arguments by defining a dependency graph, based on Brewka and Gordon [22], thereby fixing the problems with the original definition using chains (see Definition 3.52). This thesis further adapts Brewka and Gordon's definition to refer to the opposite conclusion using $\overline{p}$, instead of the incorrect $\neg p$.

- Definition 3.55 fixes a small technical error in the original definition by Gordon and Walton [86]. The function *standard* returning a proof standard was instead directly called with the arguments for a proof standard.

- Definition 3.56 rephrases Brewka and Gordon's [22] Definition 3 to instead use an audience.

- Definition 3.72 adds an inverted proof burdens and changes the definition of B to define a default burden of proof (*ebop* is not used in the framework, and assumed to be incorrect).

- Definition 3.74 defines on which conditions an argument A successfully rebuts an argument B (on B'), depending on the assignment of an explicit or implicit proof burden. The original definition in Prakken and Sartor is not strictly correct, e.g. allowing arguments with an inverted proof burden to still satisfy the second or last rule. I have provided a corrected version, fixing the ordering of rules.

Most of these errors were discovered by the implementation and formalisation efforts, confirming that logical specification on its own is not enough to guarantee correctness.

Bibliography

[1] Simon Ambler. A categorical approach to the semantics of argumentation. *Mathematical Structures in Computer Science*, 6(2):167–188, 1996.

[2] Leila Amgoud. A unified setting for inference and decision: An argumentation-based approach. In *Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence*, pages 26–33. AUAI Press, 2005.

[3] Leila Amgoud, Lianne Bodenstaff, Martin Caminada, Peter McBurney, Simon Parsons, Henry Prakken, Jelle van Veenen, and Gerard A. W. Vreeswijk. Final review and report on formal argumentation system. Deliverable D2.6, ASPIC IST-FP6-002307, 2006.

[4] Leila Amgoud and Claudette Cayrol. On the acceptability of arguments in preference-based argumentation. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pages 1–7. Morgan Kaufmann Publishers Inc., 1998.

[5] Leila Amgoud and Claudette Cayrol. A reasoning model based on the production of acceptable arguments. *Annals of Mathematics and Artificial Intelligence*, 34(1-3):197–215, 2002.

[6] Leila Amgoud, Caroline Devred, and Marie-Christine Lagasquie-Schiex. A constrained argumentation system for practical reasoning. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '08, pages 429–436, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[7] Katie Atkinson and Trevor J. M. Bench-Capon. Argumentation and standards of proof. In *Proceedings of the 11th International Conference on Artificial Intelligence and Law (ICAIL-07)*, pages 107–116, New York, NY, USA, 2007. ACM.

[8] John Backus. Can programming be liberated from the Von Neumann style?: A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[9] Pietro Baroni, Federico Cerutti, Massimiliano Giacomin, and Giovanni Guida. AFRA: Argumentation framework with recursive attacks. *International Journal of Approximate Reasoning*, 52(1):19 – 37, 2011. Tenth European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU 2009).

[10] Pietro Baroni and Massimiliano Giacomin. On principle-based evaluation of extension-based argumentation semantics. *Artificial Intelligence*, 171:675–700, July 2007.

[11] Pietro Baroni and Massimiliano Giacomin. Semantics of abstract argument systems. In Guillermo R. Simari and Iyad Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 25–44. Springer US, 2009.

[12] Trevor J. M. Bench-Capon. Persuasion in practical argument using value-based argumentation frameworks. *Journal of Logic and Computation*, 13(3):429–448, 2003.

[13] Trevor J. M. Bench-Capon, Sylvie Doutre, and Paul E. Dunne. Audiences in argumentation frameworks. *Artificial Intelligence*, 171:42–71, January 2007.

[14] Philippe Besnard and Anthony Hunter. *Elements of Argumentation*. The MIT Press, 2008.

[15] Floris Bex, Sanjay Modgil, Henry Prakken, and Chris Reed. On logical specifications of the Argument Interchange Format. *Journal of Logic and Computation*, pages 951–989, 2012.

[16] Richard Bird and Philip Wadler. *Introduction to functional programming using Haskell*. Prentice Hall Europe Hemel Hempstead, UK, 1998.

[17] 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.

[18] Ana Bove and Peter Dybjer. Dependent types at work. In Ana Bove, Lus Barbosa, Alberto Pardo, and Jorge Pinto, editors, *Language Engineering and Rigorous Software Development*, volume 5520 of *Lecture Notes in Computer Science*, pages 57–99. Springer Berlin / Heidelberg, 2009.

[19] 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.

[20] Gerhard Brewka and Thomas Eiter. Argumentation context systems: A framework for abstract group argumentation. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 5753 of *Lecture Notes in Computer Science*, pages 44–57. Springer Berlin Heidelberg, 2009.

[21] Gerhard Brewka, Stefan Ellmauthaler, Hannes Strass, Johannes Peter Wallner, and Stefan Woltran. Abstract dialectical frameworks revisited. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 803–809. AAAI Press, 2013.

[22] Gerhard Brewka and Thomas F. Gordon. Carneades and abstract dialectical frameworks: A reconstruction. In Massimiliano Giacomin and Guillermo R. Simari, editors, *Computational Models of Argument. Proceedings of COMMA 2010*, pages 3–12, Amsterdam etc, 2010. IOS Press 2010.

[23] 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.

[24] Daniel Bryant and Paul Krause. A review of current defeasible reasoning implementations. *Knowledge Engineering Review*, 23:227–260, September 2008.

[25] Maximiliano Celmo Budán, Mauro Gómez Lucero, Carlos Iván Chesñevar, and Guillermo R. Simari. Modelling time and reliability in structured argumentation frameworks. In *Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2012.

[26] Martin Caminada. On the issue of reinstatement in argumentation. In *Logics in Artificial Intelligence*, pages 111–123. Springer, 2006.

[27] Martin Caminada. Semi-stable semantics. In *Proceedings of the First International Conference on Computational Models of Argument (COMMA 2006)*, pages 121–130. IOS Press, 2006.

[28] Martin Caminada. An algorithm for computing semi-stable semantics. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 222–234. Springer, 2007.

[29] Martin Caminada and Leila Amgoud. On the evaluation of argumentation formalisms. *Artificial Intelligence*, 171:286–310, April 2007.

[30] Martin Caminada and Massimiliano Giacomin. Introducing the special issue on 20 years of argument-based inference. *J. Log. Comput.*, 25(2):243–249, 2015.

[31] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

[32] Claudette Cayrol and Marie-Christine Lagasquie-Schiex. Bipolar abstract argumentation systems. In Guillermo R. Simari and Iyad Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 65–84. Springer US, 2009.

[33] Claudette Cayrol and Marie-Christine Lagasquie-Schiex. Bipolarity in argumentation graphs: Towards a better understanding. *International Journal of Approximate Reasoning*, 54(7):876 – 899, 2013. Special issue: Uncertainty in Artificial Intelligence and Databases.

[34] Federico Cerutti, Paul E. Dunne, Massimiliano Giacomin, and Mauro Vallati. A SAT-based approach for computing extensions in abstract argumentation. In *2nd International Workshop on Theory and Applications of Formal Argumentation (TAFA-13)*. Springer, 2013.

[35] Günther Charwat, Wolfgang Dvořák, 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.

[36] Günther Charwat, Wolfgang Dvořák, Sarah Alice Gaggl, Johannes Peter Wallner, and Stefan Woltran. Methods for solving reasoning problems in abstract argumentation – A survey. *Artificial Intelligence*, 220:28–63, 2015.

[37] Günther Charwat, Johannes Peter Wallner, and Stefan Woltran. Utilizing ASP for generating and visualizing argumentation frameworks. In *Proceedings of the Fifth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2012)*, pages 51–65, 2012.

[38] Carlos Iván Chesñevar, Ana Gabriela Maguitman, and Ronald Prescott Loui. Logical models of argument. *ACM Computing Surveys (CSUR)*, 32(4):337–383, December 2000.

[39] Carlos Iván Chesñevar, Jarred McGinnis, Sanjay Modgil, Iyad Rahwan, Chris Reed, Guillermo R. Simari, Matthew South, Gerard A. W. Vreeswijk, and Steven Willmott. Towards an Argument Interchange Format. *Knowledge Engineering Review*, 21(4):293–316, 2006.

[40] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[41] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (2nd ed.)*. MIT Press and McGraw-Hill, 2001.

[42] Roy L. Crole. *Categories for Types.* Cambridge Mathematical Textbooks. Cambridge University Press, 1993. xvii+335 pages, ISBN 0521450926HB, 0521457017PB.

[43] Haskell Brooks Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584, 1934.

[44] Haskell Brooks Curry, Robert Feys, William Craig, J. Roger Hindley, and Jonathan P Seldin. *Combinatory logic*, volume 2. North-Holland Amsterdam, 1972.

[45] Arthur P. Dempster. Upper and lower probabilities induced by a multivalued mapping. *The annals of mathematical statistics*, pages 325–339, 1967.

[46] Edsger W. Dijkstra. Recursive programming. *Numerische Mathematik*, 2:312–318, 1960.

[47] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning and logic programming. *IJCAI*, 93:852–857, 1993.

[48] 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.

[49] Phan Minh Dung, Robert A. Kowalski, and Francesca Toni. Assumption-based argumentation. In Iyad Rahwan and Guillermo R. Simari, editors, *Argumentation in Artificial Intelligence*, pages 199–218. Springer, Berlin, 2009.

[50] Phan Minh Dung, Paolo Mancarella, and Francesca Toni. Argumentation-based proof procedures for credulous and sceptical non-monotonic reasoning. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2408 of *Lecture Notes in Computer Science*, pages 289–310. Springer Berlin Heidelberg, 2002.

[51] Phan Minh Dung, Paolo Mancarella, and Francesca Toni. A dialectic procedure for sceptical, assumption-based argumentation. In *Proceedings of the First International Conference on Computational Models of Argument (COMMA 2006)*, pages 145–156. IOS Press, 2006.

[52] Phan Minh Dung, Paolo Mancarella, and Francesca Toni. Computing ideal sceptical argumentation. *Artificial Intelligence*, 171:642–674, July 2007.

[53] Paul. E. Dunne. *The complexity of Boolean networks.* Academic Press Professional, Inc., San Diego, CA, USA, 1988.

[54] Paul E. Dunne, Anthony Hunter, Peter McBurney, Simon Parsons, and Michael Wooldridge. Weighted argument systems: Basic definitions, algorithms, and complexity results. *Artificial Intelligence*, 175(2):457 – 486, 2011.

[55] Wolfgang Dvořák, Sarah Alice Gaggl, Johannes Peter Wallner, and Stefan Woltran. Making use of advances in answer-set programming for abstract argumentation systems. In *Applications of Declarative Programming and Knowledge Management*, pages 114–133. Springer, 2013.

[56] Wolfgang Dvořák, Matti Järvisalo, Johannes Peter Wallner, and Stefan Woltran. Complexity-sensitive decision procedures for abstract argumentation. *Artificial Intelligence*, 206:53–78, 2014.

[57] Uwe Egly, Sarah Alice Gaggl, and Stefan Woltran. Answer-set programming encodings for argumentation frameworks. *Argument and Computation*, 1(2):147–177, 2010.

[58] Stefan Ellmauthaler. Abstract dialectical frameworks: properties, complexity, and implementation. Master's thesis, TU Vienna, 2012.

[59] Martin Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(5):467–492, September 2001.

[60] Martin Erwig and Eric Walkingshaw. Causal Reasoning with Neuron Diagrams. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 101–108, 2010.

[61] Kave Eshghi and Robert A. Kowalski. Abduction compared with negation by failure. In *ICLP*, volume 89, pages 234–255, 1989.

[62] Arthur M. Farley and Kathleen Freeman. Burden of proof in legal argumentation. In *Proceedings of the 5th International Conference on Artificial Intelligence and Law (ICAIL-05)*, pages 156–164, New York, NY, USA, 1995. ACM.

[63] Kathleen Freeman and Arthur M. Farley. A model of argumentation and its application to legal reasoning. *Artificial Intelligence and Law*, 4:163–197, 1996.

[64] Dov M. Gabbay. Fibring argumentation frames. *Studia Logica*, 93(2-3):231–295, 2009.

[65] Dov M. Gabbay. *Meta-logical Investigations in Argumentation Networks*. College Publications, 2013.

[66] Dov M. Gabbay and Odinaldo Rodrigues. Probabilistic argumentation: An equational approach. *Logica Universalis*, 9(3):345–382, 2015.

[67] Alejandro J. García and Guillermo R. Simari. Defeasible logic programming: An argumentative approach. *Theory Pract. Log. Program.*, 4(2):95–138, January 2004.

[68] Dorian Gartner and Francesca Toni. CaSAPI: A system for credulous and sceptical argumentation. In *Proc. Workshop on Argumentation for Non-monotonic Reasoning*, pages 80–95, 2007.

[69] Bas van Gijzel. Relating proof standards and abstract argumentation. Master's thesis, Utrecht University, June 2011.

[70] Bas van Gijzel. Tools for the implementation of argumentation models. In Andrew V. Jones and Nicholas Ng, editors, *2013 Imperial College*

Computing Student Workshop, volume 35 of *OpenAccess Series in Informatics (OASIcs)*, pages 43–48, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[71] Bas van Gijzel. Dungell: A reference implementation of Dung's argumentation frameworks in Haskell. In *Online system descriptions of the First International Competition on Computational Models of Argumentation (ICCMA 2015)*, 2015. `http://argumentationcompetition.org/2015/solvers.html`.

[72] 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.

[73] Bas van Gijzel and Henrik Nilsson. Towards a framework for the implementation and verification of translations between argumentation models. In *Proceedings of ACAI Summer School 2013*, 2013.

[74] Bas van Gijzel and Henrik Nilsson. A principled approach to the implementation of argumentation models. In *Proceedings of the Fifth International Conference on Computational Models of Argument (COMMA 2014)*, pages 293–300. IOS Press, 2014.

[75] Bas van Gijzel and Henrik Nilsson. Towards a framework for the implementation and verification of translations between argumentation models. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages*, IFL '13, pages 93:93–93:103, New York, NY, USA, 2014. ACM.

[76] 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.

[77] 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.

[78] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[79] Github. `https://github.com/`. Accessed August 15, 2015.

[80] Thomas F. Gordon. An overview of the Carneades Argumentation Support System. In Chris Tindale and Chris Reed, editors, *Dialectics, Dialogue and Argumentation. An Examination of Douglas Walton's Theories of Reasoning*, pages 145–156. College Publications, 2010.

[81] Thomas F. Gordon. Personal communication, 2012.

[82] Thomas F. Gordon. Introducing the Carneades web application. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Law*, pages 243–244. ACM, 2013.

[83] 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.

[84] Thomas F. Gordon, Bas van Gijzel, and Douglas Walton. Formalizing cyclic argument graphs with type theory. Unpublished manuscript, 2015.

[85] Thomas F. Gordon and Douglas Walton. Legal reasoning with argumentation schemes. In *Proceedings of the 12th International Conference on Artificial Intelligence and Law (ICAIL-09)*, pages 137–146, New York, NY, USA, 2009. ACM.

[86] Thomas F. Gordon and Douglas Walton. Proof burdens and standards. In Guillermo R. Simari and Iyad Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 239–258. Springer US, 2009.

[87] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification, The 3rd Edition.* Addison-Wesley Professional, 2005.

[88] Guido Governatori. On the relationship between Carneades and Defeasible Logic. In Tom van Engers, editor, *Proceedings of the 13th International Conference on Artificial Intelligence and Law (ICAIL 2011).* ACM Press, 2011.

[89] Guido Governatori, Michael J. Maher, Grigoris Antoniou, and David Billington. Argumentation semantics for defeasible logics. In *Proceedings of the 6th Pacific Rim international conference on Artificial intelligence*, PRICAI'00, pages 27–37, Berlin, Heidelberg, 2000. Springer-Verlag.

[90] Matthias Grabmair, Thomas F. Gordon, and Douglas Walton. Probabilistic semantics for the Carneades argument model using Bayesian networks. In *Proceedings of the 2010 conference on Computational Models of Argument: Proceedings of COMMA 2010*, pages 255–266, Amsterdam, The Netherlands, The Netherlands, 2010. IOS Press.

[91] Diana Grooters and Henry Prakken. Combining paraconsistent logic with argumentation. In Simon Parsons, Nir Oren, Chris Reed, and Federico Cerutti, editors, *Computational Models of Argument - Proceedings of COMMA 2014, Atholl Palace Hotel, Scottish Highlands, UK, September 9-12, 2014*, volume 266 of *Frontiers in Artificial Intelligence and Applications*, pages 301–312. IOS Press, 2014.

[92] Cordelia Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138, 1996.

[93] William A. Howard. The formulae-as-types notion of construction. *To H.B. Curry: Essays on Combinatory logic, Lambda Calculus and Formalism*, 44:479–490, 1980.

[94] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

[95] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 134–142. IEEE, 1998.

[96] John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, April 1989.

[97] Anthony Hunter. A probabilistic approach to modelling uncertain logical arguments. *International Journal of Approximate Reasoning*, 54(1):47 – 81, 2013.

[98] Anthony Hunter and Matthias Thimm. Probabilistic argumentation with epistemic extensions and incomplete information. *arXiv*, abs/1405.3376, 2014.

[99] Graham Hutton. *Programming in Haskell*. Cambridge University Press, January 2007.

[100] Simon L. Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[101] Jeroen Keppens. Argument diagram extraction from evidential Bayesian networks. *Artificial Intelligence and Law*, 20:109–143, 2012.

[102] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[103] Paul Krause, Simon Ambler, Morten Elvang-Gøransson, and John Fox. A logic of argumentation for reasoning under uncertainty. *Computational Intelligence*, 11:113–131, 1995.

[104] Joachim Lambek and Philip J. Scott. *Introduction to higher order categorical logic*. Cambridge University Press, New York, NY, USA, 1986.

[105] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[106] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Universiteit Utrecht, 2001.

[107] David Lewis. Postscripts to 'Causation'. In *Philosphical Papers, Vol. II*, pages 196–210. Oxford University Press, 1986.

[108] Hengfei Li, Nir Oren, and Timothy J. Norman. Probabilistic argumentation frameworks. In Sanjay Modgil, Nir Oren, and Francesca Toni, editors, *Theory and Applications of Formal Argumentation*, volume 7132 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2012.

[109] QingYin Liang and Bin Wei. An argumentation model of evidential reasoning with variable degrees of justification. In Burkhard Schäfer, editor, *Legal Knowledge and Information Systems - JURIX 2012: The Twenty-Fifth Annual Conference, University of Amsterdam, The Netherlands, 17-19 December 2012*, volume 250 of *Frontiers in Artificial Intelligence and Applications*, pages 71–80. IOS Press, 2012.

[110] Fangzhen Lin and Yoav Shoham. Argument systems. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pages 245–255. Morgan Kaufmann Publishers Inc., 1989.

[111] Miran Lipovača. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011.

[112] John Wylie Lloyd. *Foundations of Logic Programming.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 1993.

[113] Andres Löh. lhs2tex. `http://www.andres-loeh.de/lhs2tex/`. Accessed July 10, 2013.

[114] Simon Marlow et al. Haskell 2010 language report. *URL http://www.haskell.org/onlinereport/haskell2010*, 2010.

[115] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[116] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*, pages 167–184, Upper Saddle River, NJ, USA, 1985. Prentice-Hall, Inc.

[117] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[118] Sanjay Modgil. *Revisiting abstract argumentation frameworks*, volume 8306 LNAI of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 1–15. Springer-Verlag Berlin Heidelberg, 2014.

[119] Sanjay Modgil and Martin Caminada. Proof theories and algorithms for abstract argumentation frameworks. In Guillermo R. Simari and Iyad Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 105–129. Springer US, 2009.

[120] Sanjay Modgil and Henry Prakken. Reasoning about preferences in structured extended argumentation frameworks. In Massimiliano Giacomin and Guillermo R. Simari, editors, *Computational Models of Argument. Proceedings of COMMA 2010*, pages 347–358, Amsterdam etc, 2010. IOS Press 2010.

[121] Sanjay Modgil and Henry Prakken. Revisiting preferences and argumentation. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-11)*, pages 1021–1026, 2011.

[122] Sanjay Modgil and Henry Prakken. A general account of argumentation with preferences. *Artificial Intelligence*, 195:361–397, 2013.

[123] Sanjay Modgil and Henry Prakken. The ASPIC$^+$ framework for structured argumentation: a tutorial. *Argument & Computation*, 5(1):31–62, 2014.

[124] Sanjay Modgil and Henry Prakken. Personal communication (email), 2015.

[125] Victor Noël and Antonis Kakas. Gorgias-c: Extending argumentation with constraint solving. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 5753 of *Lecture Notes in Computer Science*, pages 535–541. Springer Berlin Heidelberg, 2009.

[126] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*, volume 7 of *Int. Series of Monographs on Computer Science*. Oxford, 1990.

[127] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[128] Ulf Norell. Dependently typed programming in Agda. In *Proceedings of the 4th international workshop on Types in language design and implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM.

[129] Ulf Norell, Nils Anders Danielsson, and Andreas Abel. Agda. `http://wiki.portal.chalmers.se/agda/`. Accessed October 18, 2015.

[130] Donald Nute. Defeasible Logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 3, pages 353–395. Oxford University Press, 1994.

[131] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. *Real World Haskell: Code You Can Believe In*. O'Reilly Media, Inc., 2008.

[132] Nicolas Oury and Wouter Swierstra. The power of Pi. In James Hook and Peter Thiemann, editors, *ICFP*, pages 39–50. ACM, 2008.

[133] Jeff B. Paris. *The Uncertain Reasoner's Companion: A Mathematical Perspective*. Cambridge University Press, New York, NY, USA, 1994.

[134] Judea Pearl. Bayesian networks: A model of self-activated memory for evidential reasoning. In *Proceedings of the 7th Conference of the Cognitive Science Society, University of California, Irvine*, pages 329–334, 1985.

[135] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[136] John L. Pollock. *Knowledge and justification.* Princeton University Press, Princeton, 1974.

[137] John L. Pollock. Defeasible reasoning. *Cognitive Science*, 11(4):481–518, 1987.

[138] John L. Pollock. Justification and defeat. *Artificial Intelligence*, 67(2):377–407, June 1994.

[139] John L. Pollock. *Cognitive carpentry: A blueprint for how to build a person.* Mit Press, 1995.

[140] John L. Pollock. Defeasible reasoning and degrees of justification II. Unpublished manuscript. Available from: `http://johnpollock.us/ftp/PAPERS/Degrees.pdf`, 2002.

[141] John L. Pollock. A recursive semantics for defeasible reasoning. *Argumentation in Artificial Intelligence*, pages 173–197, 2009.

[142] John L. Pollock. Defeasible reasoning and degrees of justification. *Argument and Computation*, 1(1):7–22, 2010.

[143] David Poole. On the comparison of theories: Preferring the most specific explanation. In *IJCAI*, volume 85, pages 144–147, 1985.

[144] Henry Prakken. A study of accrual of arguments, with applications to evidential reasoning. In *Proceedings of the 10th International Conference on Artificial Intelligence and Law (ICAIL-05)*, pages 85–94, New York, NY, USA, 2005. ACM.

[145] Henry Prakken. An abstract framework for argumentation with structured arguments. *Argument & Computation*, 1:93–124, 2010.

[146] Henry Prakken. Some reflections on two current trends in formal argumentation. In *Logic Programs, Norms and Action*, pages 249–272. Springer, 2012.

[147] Henry Prakken. On support relations in abstract argumentation as abstractions of inferential relations. In *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*, volume 263, pages 735–740. IOS Press, 2014.

[148] Henry Prakken and Giovanni Sartor. A dialectical model of assessing conflicting arguments in legal reasoning. In *Logical Models of Legal Argumentation*, pages 175–211. Springer, 1996.

[149] Henry Prakken and Giovanni Sartor. Argument-based extended logic programming with defeasible priorities. *Journal of applied non-classical logics*, 7(1-2):25–75, 1997.

[150] Henry Prakken and Giovanni Sartor. Presumptions and burdens of proof. In Tom van Engers, editor, *Legal Knowledge and Information Systems. JURIX 2006: The 19th Annual Conference*, pages 21–30. IOS Press, 2006.

[151] Henry Prakken and Giovanni Sartor. Formalising arguments about the burden of persuasion. In *Proceedings of the 11th International Conference on Artificial Intelligence and Law (ICAIL-07)*, pages 97–106, New York, NY, USA, 2007. ACM.

[152] Henry Prakken and Giovanni Sartor. More on presumptions and burdens of proof. In Enrico Francesconi, Giovanni Sartor, and Daniela Tiscornia, editors, *Legal Knowledge and Information Systems. JURIX 2008 The 21st Annual Conference*, pages 176–185, Amsterdam, The Netherlands, 2008. IOS Press.

[153] Henry Prakken and Giovanni Sartor. A logical analysis of burdens of proof. In Hendrik Kaptein, Henry Prakken, and Bart Verheij, editors, *Legal Evidence and Proof: Statistics, Stories, Logic*, Applied Legal Philosophy Series, pages 223–253. Farnham: Ashgate Publishing, 2009.

[154] Henry Prakken and Giovanni Sartor. On modelling burdens and standards of proof in structured argumentation. In Katie Atkinson, editor, *Legal Knowledge and Information Systems. JURIX 2011: The Twenty-Fourth Annual Conference*, pages 83–92. Amsterdam etc, IOS Press (2011), 2011.

[155] Henry Prakken and Gerard A. W. Vreeswijk. Logics for defeasible argumentation. *Handbook of Philosophical Logic*, 4(5):219–318, 2002.

[156] Iyad Rahwan and Chris Reed. The Argument Interchange Format. In Guillermo R. Simari and Iyad Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 383–402. Springer US, 2009.

[157] Iyad Rahwan and Guillermo R. Simari. *Argumentation in Artificial Intelligence.* Springer Publishing Company, Incorporated, 1st edition, 2009.

[158] Chris Reed and Glenn Rowe. Araucaria: Software for argument analysis, diagramming and representation. *International Journal on Artificial Intelligence Tools*, 13(04):961–979, 2004.

[159] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1-2):81–132, 1980.

[160] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.

[161] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[162] John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74, 2002.

[163] Glenn Shafer. *A mathematical theory of evidence.* Princeton University Press, 1976.

[164] 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.

[165] Guillermo R. Simari and Ronald P. Loui. A mathematical treatment of defeasible reasoning and its implementation. *Artificial intelligence*, 53(2-3):125–157, 1992.

[166] Mark Snaith and Chris Reed. TOAST: Online ASPIC$^+$ implementation. In Bart Verheij, Stefan Szeider, and Stefan Woltran, editors, *COMMA*, volume 245 of *Frontiers in Artificial Intelligence and Applications*, pages 509–510. IOS Press, 2012.

[167] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism.* Elsevier, 2006.

[168] Hannes Strass. Approximating operators and semantics for abstract dialectical frameworks. *Artificial Intelligence*, 205:39–70, 2013.

[169] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[170] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[171] Wouter Swierstra. *A functional specification of effects.* PhD thesis, University of Nottingham, 2009.

[172] Wouter Swierstra and Thorsten Altenkirch. Beauty in the beast. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 25–36. ACM, 2007.

[173] Yuqing Tang, Nir Oren, Simon Parsons, and Katia Sycara. Dempster-Shafer argument schemes. In *Proceedings of ArgMAS 2013*, page to appear, 2013.

[174] Matthias Thimm. Tweety: A comprehensive collection of Java libraries for logical aspects of artificial intelligence and knowledge representation. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR'14)*, July 2014.

[175] Matthias Thimm and Serena Villata. International Competition on Computational Models of Argumentation 2015. `http://argumentationcompetition.org/2015/`.

[176] Matthias Thimm and Serena Villata, editors. *Unofficial proceedings of the International Competition on Computational Models of Argumentation 2015.* Arxiv, 2015. `http://arxiv.org/abs/1510.05373`.

[177] Simon Thompson. *Haskell: the craft of functional programming*, volume 2. Addison-Wesley, 1999.

[178] Sjoerd T. Timmer, John-Jules Ch. Meyer, Henry Prakken, Silja Renooij, and Bart Verheij. Explaining legal Bayesian networks using support graphs. In *Legal Knowledge and Information Systems. JURIX 2015: The Twenty-eighth Annual Conference*, pages 121–130, 2015.

[179] Sjoerd T. Timmer, John-Jules Ch. Meyer, Henry Prakken, Silja Renooij, and Bart Verheij. A structure-guided approach to capturing Bayesian reasoning about legal evidence in argumentation. In Ted Sichelman and Katie Atkinson, editors, *Proceedings of the 15th International Conference on AI and Law*, pages 109–118. ACM Press, 2015.

[180] Francesca Toni, Mary Grammatikou, Stella Kafetzoglou, Leonidas Lymberopoulos, Symeon Papavassileiou, Dorian Gaertner, Maxime Morge, Stefano Bromuri, Jarred McGinnis, Kostas Stathis, Vasa Curcin, Moustafa Ghanem, and Li Guo. The ArguGRID Platform: An overview. In Jörn Altmann, Dirk Neumann, and Thomas Fahringer, editors, *Grid Economics and Business Models*, volume 5206 of *Lecture*

Notes in Computer Science, pages 217–225. Springer Berlin Heidelberg, 2008.

[181] Stephen E. Toulmin. *The Uses of Argument.* Cambridge University Press, July 2003.

[182] Bart Verheij. Two approaches to dialectical argumentation: admissible sets and argumentation stages. In *Proceedings of the Eighth Dutch Conference on Artificial Intelligence (NAIC '96)*, pages 357–368, 1996.

[183] Bart Verheij. Dialectical argumentation with argumentation schemes: An approach to legal logic. *Artificial Intelligence Law*, 11(2-3):167–195, January 2003.

[184] Gerard A. W. Vreeswijk. *Studies in Defeasible Argumentation.* Doctoral dissertation Free University Amsterdam, 1993.

[185] Gerard A. W. Vreeswijk. Abstract argumentation systems. *Artificial Intelligence*, 90(1-2):225–279, 1997.

[186] Gerard A. W. Vreeswijk. Argumentation in Bayesian belief networks. In *Proceedings of the First international conference on Argumentation in Multi-Agent Systems*, ArgMAS'04, pages 111–129, Berlin, Heidelberg, 2005. Springer-Verlag.

[187] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.

[188] Philip Wadler. Propositions as types. *Communications of the ACM*, 2015.

[189] Eric Walkingshaw and Martin Erwig. A DSEL for Studying and Explaining Causation. In *IFIP Working Conference on Domain Specific Languages (DSL'11)*, pages 143–167, 2011.

[190] Niklaus Wirth. The programming language Pascal. *Acta informatica*, 1(1):35–63, 1971.