Towards a framework for the implementation and verification of translations between argumentation models

Bas van Gijzel Henrik Nilsson
University of Nottingham
{bmv,nhn}@cs.nott.ac.uk

Abstract

Argumentation theory is concerned with studying the nature of arguments in the most general sense, including for example scientific, legal, or even completely informal arguments. There are two main approaches. Abstract argumentation is completely generic, making no specific assumptions about the structure of arguments. Structured argumentation, on the other hand, does adopt a predetermined structure pertaining to the domain of discourse. Structured argumentation models have seen a recent surge, with new developments in both general frameworks and more domain-specific approaches. Yet, in contrast to the abstract approach, there is a distinct lack of *implementations* of structured argumentation models. We believe a key reason for this is the lack of suitable implementation frameworks. Building on previous work, this paper attempts to tackle this problem by applying functional programming techniques. We show how to implement one structured argumentation framework (Carneades) and one abstract framework (Dung) in this way, and then proceed to show how to implement a translation from the former into the latter, one of the first such implementations. Ultimately, we hope our work will evolve into a domain-specific language for implementation of argumentation frameworks. But even at this stage, the paper demonstrates the benefits of functional programming as a tool for argumentation theory.

Categories and Subject Descriptors I.2.3 [*Deduction and Theorem Proving*]: Nonmonotonic reasoning and belief revision

General Terms Theory, Verification

Keywords argumentation, functional programming, Haskell, domain specific language

1. Introduction

Argumentation theory is an interdisciplinary field studying how conclusions can be reached through logical reasoning in settings where the soundness of arguments might be subjective and arguments can be contradictory. There are two main approaches: the 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 a predetermined argument structure, more or less specific to domains such as legal or scientific argumentation. Structured argumentation models have seen a recent surge, with new developments in both general frameworks [3, 5, 27] and more domain-specific approaches [19, 20].

For the abstract approach, significant effort has been directed towards the construction of tools and efficient implementations; see [8] for a survey. The situation for the structured approach is markedly different: despite existing translations from structured into abstract argumentation frameworks [6, 17, 18, 25, 27], which in principle should allow abstract argumentation implementations to be leveraged, there is still a lack of implementations of structured models and their translations. Possible reasons include:

- Abstract argumentation is closely related to logic programming [12]. This facilitates developing intuitive implementations, closely aligned with the logical specification, that are also efficient [8]. 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 [29]. 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 [29] 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.
- Translations can be notoriously complex, both in implementation and in verification. Examples include the translation of Carneades into ASPIC$^+$ [17, 18] and the translation of abstract dialectical frameworks into Dung [6]. Both proofs are substantial and very technical, and thus hard to verify even for experts in the field.

This paper attempts to address the lack of implementations of structural argumentation frameworks by exploiting functional programming. We argue 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) [22, 23], allowing tailoring to specific requirements which is exactly what is needed to support particular structured argumentation frameworks. Our choice of programming language is Haskell. This is motivated by our previous work [16], where we managed to implement the Carneades argumentation model in a way that is easily understand-

IFL'13, August 28–30, 2013, Nijmegen, The Netherlands.

http://dx.doi.org/10.1145/2620678.2620688Reprinted from IFL'13, Implementation and Application of Functional Languages, August 28–30, 2013, Nijmegen, The Netherlands, pp. 93–103.

able to argumentation theorists with no prior Haskell knowledge. Further, two of the Cabal packages discussed in this paper (the Dung and CarneadesDSL package) have been used in AI for programming (e.g. a closed source natural language processor) and teaching[1]. We hope that our approach ultimately will result in an EDSL that is as suitable for implementing structured argumentation frameworks as logic programming is for implementing abstract ones.

The specific contributions of this paper are the following:

- We explain and implement two argumentation models: Dung's argumentation frameworks [12] (AFs), the standard abstract model; and Carneades [19, 20], a structured argumentation model used in the legal domain. All code for Dung's AFs is included and discussed. Essentially, the code is just a transliteration of standard definitions, allowing this paper to simultaneously serve as documentation and implementation. The implementation of Carneades is based on previous work [16].
- We provide one of the first *implementations* of a translation between argumentation models, fully documenting the techniques and making all work publicly available and reusable.
- We discuss the desired properties of such a translation, sketch their implementation in Haskell, and discuss how they might be formalised in a theorem prover.
- We discuss and provide (online) a formalisation of the implementation of Dung's AFs in a theorem prover, Agda, giving the first fully machine-checkable formalisation of an argumentation model, and showcasing the benefits of using a functional programming language as an initial implementation.

These contributions and the relationship between them are summarised in Figure 1, where solid lines and outlines indicate what is in the paper and dashed ones indicate future work.

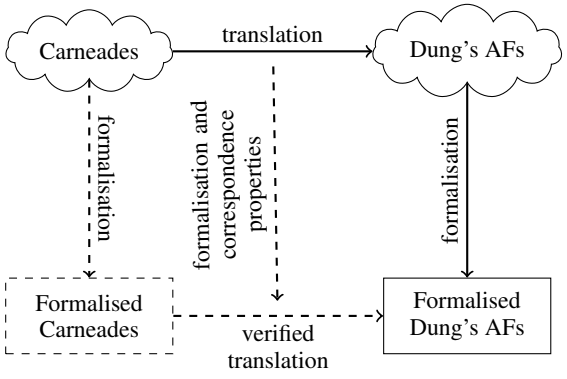

Figure 1. Overview of the contributions

The paper is structured as follows. Section 2 gives an introduction to Dung's abstract argumentation frameworks, along with an implementation in Haskell. Section 3 does the same for Carneades. In Section 4, we show how to implement a translation of Carneades into Dung's AFs. We also consider correctness properties. In Section 5 we discuss our formalisation of Dung's AFs in the theorem prover Agda, and what we gained from this. We conclude in Section 6 with a discussion of what we have learnt from this study and how to take the work further.

[1] School of Informatics, University of Edinburgh: `http://www.inf.ed.ac.uk/teaching/courses/ailp/`

2. An implementation of AFs in Haskell

The abstract argument system, or argumentation frameworks (AFs), introduced by Dung [12] is a simple yet general model capable of capturing various contemporary approaches to non-monotonic reasoning. It has been the translation target for a number of modern structured argumentation models [6, 17, 18, 25, 27]. This section gives a large part of the standard definitions of Dung's AFs, including an algorithm for what is termed the *grounded semantics*, showing how these definitions can be transliterated into (a slightly stylised version of) Haskell[2]. The aim is to show how a functional programming language can be used to quickly implement a prototype of an argumentation model in a way that facilitates proving properties about the implementation. Additionally, the section constitutes a tutorial-like introduction to the implementation of AFs up to grounded semantics. As for our earlier work (Section 3), the code is available as a free library[3]. Those interested in a more complete introduction to AFs and alternative semantics can consult Baroni and Giacomin [1].

An abstract argumentation framework consists of a set of abstract arguments and a binary relation on this set representing *attack*: 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. They may come from any domain, including informal ones such as "if it rains you get wet". Further, no restrictions are imposed on the attack relation as such: what meanings that can be ascribed to an argumentation framework is instead wholly captured by the notion of *extensions* defined later (Definition 2.8). In particular, an attack relation is not assumed to be symmetric as an attacked argument does not necessarily constitute a counter-attack of the attacking argument. The relation is not even assumed to be anti-reflexive; i.e., self-contradicting arguments are not ruled out.

Definition 2.1 (Abstract argumentation framework)**.** An *abstract argumentation framework* is a tuple $\langle Args, Atk \rangle$, where $Args$ is a set of arguments and $Atk \subseteq Args \times Args$ is an arbitrary relation on $Args$ representing attack.

While the notion of argument is abstract, we need to assume that it is possible to determine if two arguments are the same or not. For now, we simply use a string to label arguments to that end.

$$\textbf{data}\ DungAF\ arg = AF\ [arg]\ [(arg, arg)]\ \textbf{deriving}\ (Show)$$

$$\textbf{type}\ AbsArg = String$$

Note that we use lists instead of sets for ease of presentation.

Example 2.2. 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 attacks C and A attacks B. Consequently A *reinstates* C as A attacks the attacker of C. This is formally captured by $AF_1 = \langle \{A, B, C\}, \{(A, B), (B, C)\} \rangle$; see Figure 2.

$$A \longrightarrow B \longrightarrow C$$

Figure 2. An (abstract) argumentation framework

[2] The source code for this Section is written in literate Haskell and can be run as is by a standard Haskell compiler.
`http://www.cs.nott.ac.uk/~bmv/Code/dunginhaskell_ifl.lhs`

[3] See `http://hackage.haskell.org/package/Dung`.

In Haskell:

```
a, b, c :: AbsArg
a = "A"
b = "B"
c = "C"

AF1 :: DungAF AbsArg
AF1 = AF [a, b, c] [(a, b), (b, c)]
```

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 $AF = \langle Args, Atk \rangle$.

Definition 2.3 (Set-attacks)**.** A set $S \subseteq Args$ of arguments attacks an argument $A \in Args$ iff there exists a $Y \in S$ such that $(Y, X) \in Atk$.

For example, in Figure 2, $\{A, B\}$ set-attacks C, because B attacks C and $B \in \{A, B\}$.

Definition 2.4 (Conflict-free)**.** A set $S \subseteq Args$ of arguments is called *conflict-free* iff there are no X, Y in S such that $(X, Y) \in Atk$.

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 2, $\{A, C\}$ is a conflict-free set.

Definition 2.5 (Acceptability)**.** 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 $(Y, X) \in Atk$ then there exists a $Z \in S$ for which $(Z, Y) \in Atk$.

An argument is acceptable (w.r.t. to some set S) if all its attackers are attacked in turn. (Note that although the acceptability is w.r.t. to a set S, *all* attackers are taken in account.) For example, in Figure 2, C is acceptable w.r.t. $\{A, B, C\}$, because A attacks the only attacker 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 2.6 (Characteristic function)**.** 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 2, $F_{AF}(\emptyset) = \{A\}$, $F_{AF}(\{A\}) = \{A, C\}$ and $F_{AF}(\{A, B, C\}) = \{A, C\}$.

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 attacks.

Definition 2.7 (Admissibility)**.** A conflict-free set of arguments S is admissible iff every argument X in S is acceptable with respect to S, i.e. $S \subseteq F_{AF}(S)$.

Note that not every conflict-free set is necessarily admissible. For example, in Figure 2, $\{C\}$ is conflict-free but is not an admissible set, since $(B, C) \in Atk$ and there is no argument in $\{C\}$ that defends from this attack.

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, for completeness, we define the four standard extensions as given by Dung [12]. However, we will only concern ourselves with one of these, the *grounded* extension, in the remainder of the paper.

Definition 2.8 (Extensions)**.** 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:

- *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 attacking all arguments in $Args \backslash S$.

As proven in Dung [12] F_{AF} is monotonic with respect to set inclusion. Furthermore, the grounded extension always exists and is unique, and there always exists a preferred extension. Alternatively, 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 attackers and those that are "completely defended". A *stable extension* is an extension that is able to attack 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 attacks.

Our Haskell version of the first few definitions is a straightforward implementation of the mathematical definitions.

```
setAttacks :: Eq arg ⇒ DungAF arg → [arg] →
              arg → Bool
setAttacks (AF _ atk) args arg
  = or [y ≡ arg | (x, y) ← atk, x ∈ args]

conflictFree :: Eq arg ⇒ DungAF arg → [arg] → Bool
conflictFree (AF _ atk) args
  = null [(x, y) | (x, y) ← atk, x ∈ args, y ∈ args]

acceptable :: Eq arg ⇒ DungAF arg → arg →
              [arg] → Bool
acceptable af@(AF _ atk) x args
  = and [setAttacks af args y | (y, x') ← atk, x ≡ x']

f :: Eq arg ⇒ DungAF arg → [arg] → [arg]
f af@(AF args' _) args
  = [x | x ← args', acceptable af x args]

fAF1 :: [AbsArg] → [AbsArg]
fAF1 = f AF1

admissible :: Eq arg ⇒ DungAF arg → [arg] → Bool
admissible af args = conflictFree af args ∧
                     args ⊆ f af args
```

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 fAF1
> ["A", "C"]
```

The contexts *Eq arg* mean that these functions are only applicable to types of arguments for which equality has been defined.

Given an argumentation framework, we can determine which arguments are justified by applying an argumentation semantics. However, instead of the extension based approach, we will take

the *labelling-based* approach to grounded semantics. The labelling based approach is more commonly used in actual implementations. 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 Section 5).

Formally, the labelling approach is a generalisation of the extension-based approach [7], permitting further possibilities than arguments being either *in* or *out* of an extension; e.g. the state of an argument might be *undecided*. The grounded labelling can easily be mapped back to a grounded extension by simply discarding all arguments not labelled *in*. The algorithm below is commonly used for computing the grounded labelling [24]. However, our version is corrected and clarified slightly: the "if then" in the $\exists$ has been changed to "and", and we have made the set of unlabelled arguments explicit as the set of undecided arguments, undec.

We define a labelling $\mathcal{L}$ to be 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 $\mathrm{in}(\mathcal{L}), \mathrm{out}(\mathcal{L}), \mathrm{undec}(\mathcal{L})$ to refer to the first, second, and third field of a labelling $\mathcal{L}$. The variable i refers the ith step of calculating the labelling; its incrementation is handled implicitly in the algorithm.

Algorithm 2.9. Algorithm for grounded labelling (Algorithm 6.1 of [24])

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

The Haskell equivalent to a labelling:

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

For our Haskell implementation, we will first translate the two conditions for x containing quantifiers in line 3 and 4.

```
  -- if all attackers are Out
unattacked :: Eq arg ⇒ [arg] →
              DungAF arg → arg → Bool
unattacked outs (AF _ atk) arg =
  let attackers = [x | (x, y) ← atk, arg ≡ y]
  in null (attackers \\ outs)


  -- if there exists an attacker that is In
attacked :: Eq arg ⇒ [arg] →
            DungAF arg → arg → Bool
attacked ins (AF _ atk) arg =
  let attackers = [x | (x, y) ← atk, arg ≡ y]
  in ¬ (null (attackers `intersect` ins))
```

Our implementation consists of two parts. The main function for the grounded labelling, and a helper function implementing the actual algorithm. The latter has two additional accumulation arguments for keeping track of the Ins and Outs and an argument initially containing all arguments, which will decrease when arguments are labelled.

```
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 (unattacked outs af) undec
      newOuts = filter (attacked 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
```

Then as expected:

```
grounded AF1
> [("A", In), ("C", In), ("B", Out)]
```

Finally, the grounded extension can be defined by returning only those arguments that are labelled In from the grounded labelling.

```
groundedExt :: Eq arg ⇒ DungAF arg → [arg]
groundedExt af = [arg | (arg, In) ← grounded af]
```

3. An implementation of Carneades in Haskell

This section gives definitions and corresponding implementations in Haskell of the Carneades argumentation model [19, 20]. Carneades is a structured argumentation model designed to capture standards and burdens of proof. This section is largely based on previous work [16][4] and is mainly included in the interest of making the paper self-contained. However, the use of proof standards has been changed by relying on the definition of $PSName$ to allow for an easier translation.

3.1 Arguments

Carneades models arguments placed in favour of or against atomic propositions. 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.

Definition 3.1 (Carneades' Arguments)**.** 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 atomic proposition. 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 suffices 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)
```

The negation for a literal p, written $\overline{p}$ can then be defined:

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

[4] See `http://www.cs.nott.ac.uk/~bmv/CarneadesDSL` for the literate Haskell source code and Cabal package.

We choose to realise an *argument* as a newtype (to allow a manual equality instance) containing a tuple of two lists of propositions, its *premises* and its *exceptions*, and a proposition that denotes the *conclusion*:

$$\textbf{newtype}\ Argument = Arg\ ([PropLiteral], [PropLiteral], PropLiteral)$$

Arguments are considered equal if their premises, exceptions and conclusion are equal. Arguments are thus identified by their logical content. The equality instance for $Argument$ (omitted for brevity) takes this into account by comparing the lists as sets.

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 [4], we use a dependency graph to determine acyclicity of a set of arguments.

Definition 3.2 (Acyclic set of arguments)**.** 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 simplicity, only providing a check for acyclicity and a function to retrieve arguments pro a proposition. We use FGL [13] 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.

$$\begin{aligned}&\textbf{type}\ ArgSet = \ldots\\ &getArgs \quad :: PropLiteral \to ArgSet \to [Argument]\\ &checkCycle :: ArgSet \to Bool\end{aligned}$$

3.2 Carneades Argument Evaluation Structure

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

Definition 3.3 (Carneades Argument Evaluation Structure (CAES))**.** A *Carneades Argument Evaluation Structure* (CAES) is a triple

$$\langle arguments, audience, standard\rangle$$

where $arguments$ is an acyclic set of arguments, $audience$ is an audience as defined below (Def. 3.4), 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. The transliteration into Haskell is almost immediate

$$\textbf{newtype}\ CAES = CAES\ (ArgSet, Audience, PropStandard)$$

Definition 3.4 (Audience)**.** Let $\mathcal{L}$ be a propositional language. An *audience* is a tuple $\langle$*assumptions*, *weight*$\rangle$, where $assumptions \subset \mathcal{L}$ is a propositionally consistent set of literals (i.e., not containing both a literal and its negation) assumed to be acceptable to the audience and *weight* is a function mapping arguments to a real-valued weight in the range $[0, 1]$.

This definition is captured by the following Haskell definitions:

$$\begin{aligned}&\textbf{type}\ Audience = (Assumptions, ArgWeight)\\ &\textbf{type}\ Assumptions = [PropLiteral]\\ &\textbf{type}\ ArgWeight = Argument \to Weight\\ &\textbf{type}\ Weight = Double\end{aligned}$$

Further, as each proposition is associated with a specific proof standard, we need a mapping from propositions to proof standards. We map to the name of the proof standard, instead of directly mapping to a function, allowing us in our translation from Carneades into Dung to compare proof standards. $psMap$ maps names to their respective functions.

$$\begin{aligned}&\textbf{type}\ PropStandard = PropLiteral \to PSName\\ &\textbf{data}\ PSName = Scintilla\\ &\quad \mid Preponderance \mid ClearAndConvincing\\ &\quad \mid BeyondReasonableDoubt \mid DialecticalValidity\\ &\textbf{deriving}\ Eq\\ \\ &psMap :: PSName \to ProofStandard\end{aligned}$$

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:

$$\textbf{type}\ ProofStandard = PropLiteral \to CAES \to Bool$$

This aggregation process will be defined in detail in the next section, but note that it is done relative to a specific CAES.

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

- for all $p \in P$, p is an assumption or [$\overline{p}$ is not an assumption and p is acceptable in C] and
- for all $e \in E$, e is not an assumption and [$\overline{e}$ is an assumption or e is not acceptable in C].

Definition 3.6 (Acceptability of propositions)**.** Given a CAES C, a proposition p is *acceptable* in C iff $(s\ p\ C)$ is $True$, where s is the proof standard for p.

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:

$$\begin{aligned}&applicable :: Argument \to CAES \to Bool\\ &applicable\ \ (Arg\ (prems, excns, \_))\\ &\qquad\qquad caes@(CAES\ (\_, (assumptions, \_), \_))\\ &\quad = and\ \$\ [p \in assumptions \vee\\ &\qquad\qquad (negate\ p \notin assumptions \wedge\\ &\qquad\qquad\ p\ `acceptable`\ caes) \mid p \leftarrow prems]\\ &\qquad\qquad +\!\!+\\ &\qquad\qquad [(e \notin assumptions) \wedge\\ &\qquad\qquad\ (negate\ e \in assumptions \vee\\ &\qquad\qquad\ \neg\ (e\ `acceptable`\ caes)) \mid e \leftarrow excns]\\ &acceptable :: PropLiteral \to CAES \to Bool\\ &acceptable\ p\ caes@(CAES\ (\_, \_, standard))\\ &\quad = s\ p\ caes\\ &\quad \textbf{where}\ s = psMap\ \$\ standard\ p\end{aligned}$$

3.4 Proof standards

Carneades predefines five proof standards, originating from the work of Freeman and Farley [14, 15]: *scintilla of evidence*, *pre-*

ponderance 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, we proceed to give the definitions directly in Haskell, as they really only are transliterations 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:

$$\begin{aligned}&scintilla :: ProofStandard\\&scintilla\ p\ caes@(CAES\ (g, \_, \_))\\&= any\ (`applicable`caes)\ (getArgs\ p\ g)\end{aligned}$$

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, we start by defining those notions:

$$\begin{aligned}&maxWeightApplicable :: [Argument] \to CAES \to Weight\\&maxWeightApplicable\ as\ caes@(CAES\ (\_, (\_, argWeight), \_))\\&\quad = foldl\ max\ 0\ [argWeight\ a \mid a \leftarrow as, a\ `applicable`\ caes]\\&maxWeightPro :: PropLiteral \to CAES \to Weight\\&maxWeightPro\ p\ caes@(CAES\ (g, \_, \_))\\&= maxWeightApplicable\ (getArgs\ p\ g)\ caes\\&maxWeightCon :: PropLiteral \to CAES \to Weight\\&maxWeightCon\ p\ caes@(CAES\ (g, \_, \_))\\&= maxWeightApplicable\ (getArgs\ (negate\ p)\ g)\ caes\end{aligned}$$

We can then define the proof standard preponderance:

$$\begin{aligned}&preponderance :: ProofStandard\\&preponderance\ p\ caes = maxWeightPro\ p\ caes >\\&\qquad\qquad maxWeightCon\ p\ caes\end{aligned}$$

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:

$$\begin{aligned}&clear\_and\_convincing :: ProofStandard\\&clear\_and\_convincing\ p\ caes\\&= \ (mwp > \alpha) \wedge (mwp - mwc > \beta)\\&\ \ \mathbf{where}\\&\quad mwp = maxWeightPro\ p\ caes\\&\quad mwc = maxWeightCon\ p\ caes\end{aligned}$$

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:

$$\begin{aligned}&beyond\_reasonable\_doubt :: ProofStandard\\&beyond\_reasonable\_doubt\ p\ caes\\&= clear\_and\_convincing\ p\ caes \wedge\\&\quad (maxWeightCon\ p\ caes < \gamma)\end{aligned}$$

Finally dialectical validity requires at least one applicable argument pro p and no applicable arguments con p:

$$\begin{aligned}&dialectical\_validity :: ProofStandard\\&dialectical\_validity\ p\ caes\\&\quad = scintilla\ p\ caes \wedge \neg\ (scintilla\ (negate\ p)\ caes)\end{aligned}$$

3.5 Convenience functions

We provide a set of functions to facilitate construction of propositions, arguments, argument sets and sets of assumptions. The functions $mkProp$, $mkArg$ and $mkAssumptions$ build the respective type in the obvious way, while performing a few sanity checks on the input. The function $mkArgSet$ constructs a graph based on a list of arguments, adding nodes for propositions and edges for the induced dependencies as determined by Definition 3.2.

$$\begin{aligned}&mkProp &&:: String \to PropLiteral\\&mkArg &&:: [String] \to [String] \to\\&&&\quad String \to Argument\\&mkArgSet &&:: [Argument] \to ArgSet\\&mkAssumptions &&:: [String] \to [PropLiteral]\end{aligned}$$

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:

$$\begin{aligned}&audience :: Audience\\&audience = (assumptions, weight)\end{aligned}$$

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:

$$\begin{aligned}&getArgs &&:: PropLiteral \to ArgSet \to\\&&&\quad [Argument]\\&getAllArgs &&:: ArgSet \to [Argument]\\&getProps &&:: ArgSet \to [PropLiteral]\\&applicableArgs &&:: CAES \to [Argument]\\&nonApplicableArgs &&:: CAES \to [Argument]\\&acceptableProps &&:: CAES \to [PropLiteral]\\&nonAcceptableProps &&:: CAES \to [PropLiteral]\end{aligned}$$

3.6 Implementing a CAES

This subsection shows how an argumentation theorist given the Carneades DSL developed in this section quickly and at a high level of abstraction can implement a Carneades argument evaluation structure and evaluate it.

The example below considers a court case where there has been assumed to be a killing (*kill*). There are two witnesses (*witness* and *witness2*), of which the second is assumed to be unreliable (*unreliable2*). There are three arguments, the first says that a killing with intent implies a murder and the other two are statements from the two respective witnesses, implying the killer did or did not have the intent to kill.

$$\begin{aligned}arguments &= \{arg1, arg2, arg3\},\\assumptions &= \{kill, witness, witness2, unreliable2\},\\standard(intent) &= BeyondReasonableDoubt,\\standard(x) &= Scintilla, \text{ for any other proposition } x,\\\alpha &= 0.4,\ \beta = 0.3,\ \gamma = 0.2.\end{aligned}$$

Arguments and the argument graph are constructed by calling $mkArg$ and $mkArgSet$ respectively:

$$\begin{aligned}&arg1, arg2, arg3 :: Argument\\&arg1 = mkArg\ [\texttt{"kill"}, \texttt{"intent"}]\ [\,]\ \texttt{"murder"}\end{aligned}$$

```
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"]~["unreliable"] ⇒ "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
```

4. An implementation of a translation from Carneades into AFs

The abstract and structured argument approaches may seem very different. Yet translation from structured models into abstract ones is possible. In particular, it is known that Carneades can be translated into a framework called ASPIC$^+$ [17, 18], which in turn can be translated into Dung's AFs [27]. However, such translations have rarely if ever been *implemented*. We believe one reason for this is the lack of an appropriate programming methodology.

The preceding two sections showed that typed functional programming languages like Haskell can offer a compelling setting for implementation of argumentation frameworks. In the following, we take this one step further and show how the functional approach also is suitable for implementing *translation* between frameworks, here translation of Carneades directly into Dung's AFs. We thus see that functional programming can address all basic argumentation theory implementation needs. This in turn suggests that it might be feasible to refine what we have done into a general-purpose, argumentation theory (E)DSL, thereby addressing the lack of appropriate programming methodology mentioned above. This case becomes even more compelling when we consider formalisation in a theorem prover in the next section. All code is publicly available[5][6].

4.1 Translation structure

To avoid having to introduce the ASPIC$^+$ model, we use a derived translation based on the algorithm below, allowing us to directly translate Carneades into Dung's argumentation frameworks.

Algorithm 4.1. Algorithm for argument translation from Carneades into Dung's AFs (Adapted Definition 4.9 of [18])

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 applicability part of arguments in $argSet$, building on previously $generatedArgs$ and put the generated arguments in $tempArgs$.
   (c) $argSet = \emptyset$.
   (d) Repeat (a) through (c) for the arguments for the opposite conclusion $\overline{c}$.
   (e) Translate the acceptability part of c and $\overline{c}$ based on arguments in $tempArgs$. Add the results and $tempArgs$ to $generatedArgs$.
   (f) $tempArgs = \emptyset$.

Note that $arguments$ refers to the set of arguments in a CAES.

For our implementation we give the main functions or type signatures involved. If we look at the translation as given in van Gijzel and Prakken [18], we can make the following observation: When a CAES is translated into Dung's AFs (through ASPIC$^+$), the resulting framework includes arguments corresponding to Carneades propositions. However, it also includes arguments representing the original Carneades argument nodes. For our derived translation we take the same approach. Therefore, instead of labelling our abstract arguments with a *String*, we label arguments with their original propositional literal or original Carneades argument. For the

[5] For the complete translation see:
`http://www.cs.nott.ac.uk/~bmv/Code/translation_ifl.lhs`.

[6] For the Cabal package see:
`http://hackage.haskell.org/package/CarneadesIntoDung`.

translation of acceptability we need to keep track of the status of some of the previously translated arguments and thus we also use a (Boolean) labelled version of arguments. Then, using these two argument types we can define similarly instantiated AFs.

```
type ConcreteArg   = Either PropLiteral Argument
type LConcreteArg = (Bool, ConcreteArg)

type ConcreteAF   = DungAF ConcreteArg
type LConcreteAF = DungAF LConcreteArg
```

Before introducing the main translation function we discuss a few auxiliary definitions. First of all, arguments that do not hold up to their respective proof standard (e.g., an argument with clear and convincing evidence as a proof standard, but an argument weight less than α) needs to be translated in such a way that it is attacked in the corresponding AF. However, there is no attacker in Carneades in the sense required by an AF. We therefore introduce one additional argument called *defeater*. This is added to the AF as an unattacked argument and will attack other arguments that do not satisfy their assigned proof standard.

```
defeater :: LConcreteArg
defeater = (True, Left $ mkProp "defeater")
```

topSort is a topological sorting of the argument graph and corresponds to point 2 of Alg. 4.1. We implement our topological sorting in terms of FGL's implementation of this functionality. However, due to the definition of dependency graphs, we have to reverse the ordering. The cyclicity check is used to prohibit translation of a cyclic CAES as these do not have a semantics in the original Carneades model:

```
topSort :: ArgSet → [(PropLiteral, [Argument])]
topSort g
  | cyclic g
    = error "Argumentation graph is cyclic!"
  | otherwise
    = reverse (topsort' g)
```

propToLArg labels a proposition as *True* in the AF and stores the original proposition as its label:

```
propToLArg :: PropLiteral → LConcreteArg
propToLArg p = (True, Left p)
```

The main translation function initialises the translation process by mapping all assumptions to arguments, thereby supplying the *argsToAF* with the initially acceptable arguments. The result of *argsToAF* is the complete Dung framework, which is then stripped of its labels.

```
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)
          [])
```

argsToAF corresponds to point 3 of Alg. 4.1. There are three cases to handle. If there are no more arguments to process, the translated AF is returned:

```
argsToAF :: [(PropLiteral, [Argument])] → CAES →
            LConcreteAF → LConcreteAF
argsToAF [] _ transAF = transAF
```

If there is a propositional literal left, but it is an assumption, it has already been translated and does not need to be considered.

```
argsToAF (pro@(p, proArgs) : argList)
         caes@(CAES (_, (assumptions, _), _))
         (AF args defs)
| p ∈ assumptions
  = argsToAF argList caes (AF args defs)
```

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. *trApps* is called to translate the applicability part of the pro and con arguments. *trAcc* is called to translate the acceptability of p and $\overline{p}$ (note that the order of applicable arguments is switched for translating the acceptability of $\overline{p}$). The results of these four calls are collected and used in the recursive step of *argsToAF*.

```
| otherwise =
  let con            = conArgs p argList
      (pApp, pDefs) = trApps args pro
      (cApp, cDefs) = trApps args con
      (pAcc, pDefs') = trAcc p pApp cApp caes
      (cAcc, cDefs') = trAcc (negate p) cApp pApp caes
      argList'       = delete con argList
  in argsToAF argList' caes
      (AF (pAcc : cAcc : pApp ++ cApp ++ args)
      (pDefs' ++ cDefs' ++ pDefs ++ cDefs ++ defs))
```

4.2 Translation of applicability

To translate applicability of an argument, we need to know how its premises and exceptions have been translated. Due to the topological ordering of our arguments, it is guaranteed that premises and exceptions will have been translated before the argument using them.

trApps 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. *trApps* collects the results of the *trApp* function, which does the main work.

```
trApps :: [LConcreteArg] → (PropLiteral, [Argument]) →
          ([LConcreteArg],
           [(LConcreteArg, LConcreteArg)])
trApps tArgs (p, args) =
  let tr = map (trApp tArgs p) args
  in (map fst tr, concatMap snd tr)
```

The translation of an applicable argument in *trApp* is done as follows. 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. Acceptable propositions are filtered out using *accProps* and compared to the premises of the to-be-translated argument. If a premise of the to be translated argument is present as an acceptable argument in the previously translated argument, then the argument has to be attacked in the resulting AF.

```
accProps :: [LConcreteArg] → [PropLiteral]
accExcs :: [LConcreteArg] → [PropLiteral] →
  [LConcreteArg]

trApp :: [LConcreteArg] → PropLiteral → Argument →
         (LConcreteArg, [(LConcreteArg, LConcreteArg)])
trApp tArgs p a@(Arg (prems, excs, c))
| accProps tArgs `intersect` prems ≢
  prems = ((False, Right a), [(defeater, (False, Right a))])
```

Acceptable exceptions are filtered out (as arguments) using the helper function $accExc$. Every acceptable exception will generate an attacker to the to be translated argument.

$$
\begin{aligned}
&\mid otherwise = \\
&\textbf{let } accE \ = accExcs\ tArgs\ excs \\
&\quad appArg = (null\ accE, Right\ a) \\
&\quad attacks = map\ (\lambda argExc \rightarrow (argExc, appArg))\ accE \\
&\textbf{in } (appArg, attacks)
\end{aligned}
$$

4.3 Translation of acceptability

By analysing the defined proof standards in Section 3.4, we can observe that the acceptability of a proposition depends on the existence of applicable arguments pro and con the proposition, and on the weights assigned to these applicable arguments. For the standards defined, only the existence of an applicable argument or the maximum weight of the pro and con arguments are relevant. Thus, we first define $maxWeight$, a function that determines the maximum weight of a list of applicable arguments (assumed to have the same conclusion).

$$
\begin{aligned}
&maxWeight :: [LConcreteArg] \rightarrow CAES \rightarrow Double \\
&maxWeight\ as\ caes@(CAES\ (\_, (\_, argWeight), \_)) \\
&= foldl\ max\ 0\ [argWeight\ a \mid (True, Right\ a) \leftarrow as]
\end{aligned}
$$

The main translation function for acceptability can then be defined. $trAcc$ expects the following arguments: the 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.

$$
\begin{aligned}
&trAcc :: PropLiteral \rightarrow [LConcreteArg] \rightarrow \\
&\qquad [LConcreteArg] \rightarrow CAES \rightarrow \\
&\qquad (LConcreteArg, \\
&\qquad\ [(LConcreteArg, LConcreteArg)])
\end{aligned}
$$

In the case there are no arguments in favour of c, it cannot be acceptable. Thus it is labelled false and paired with an attack that attacks c.

$$
\begin{aligned}
&trAcc\ c\ [\,]\ conArgs\ caes \\
&\quad = ((False, Left\ c), [(defeater, (False, Left\ c))])
\end{aligned}
$$

Now follow two cases: an impossible case, since $trAcc$ is only called with arguments, not with propositions, and a case where we match an argument which has been labelled $False$ in our translation. In the latter, we recurse to find an argument labelled $True$ instead.

$$
\begin{aligned}
&trAcc\ c\ ((\_, Left\ \_) : proArgs)\ conArgs\ caes \\
&\quad = error\ \texttt{"Proposition in list!"} \\
&trAcc\ c\ ((False, \_) : proArgs)\ conArgs\ caes \\
&\quad = trAcc\ c\ proArgs\ conArgs\ caes
\end{aligned}
$$

The following cases correspond to each proof standard:

$$
\begin{aligned}
&trAcc\ c\ proArgs@((True, \_) : proArgs')\ conArgs \\
&\qquad caes@(CAES\ (\_, \_, standard)) \\
&\mid standard\ c \equiv Scintilla \\
&\quad = ((True, Left\ c), [\,]) \\
&\\
&\mid standard\ c \equiv Preponderance \wedge \\
&\quad maxWeight\ proArgs\ caes > maxWeight\ conArgs\ caes \\
&\quad = ((True, Left\ c), [\,]) \\
&\\
&\mid standard\ c \equiv ClearAndConvincing \wedge \\
&\quad maxWeight\ proArgs\ caes > \alpha \wedge \\
&\quad maxWeight\ proArgs\ caes > maxWeight\ conArgs\ caes + \beta \\
&\quad = ((True, Left\ c), [\,]) \\
&\\
&\mid standard\ c \equiv BeyondReasonableDoubt \wedge \\
&\quad maxWeight\ proArgs\ caes > \alpha \wedge \\
&\quad maxWeight\ proArgs\ caes > maxWeight\ conArgs\ caes + \beta \wedge \\
&\quad maxWeight\ conArgs\ caes < \gamma \\
&\quad = ((True, Left\ c), [\,]) \\
&\\
&\mid standard\ c \equiv DialecticalValidity \wedge \\
&\quad null\ conArgs \\
&\quad = ((True, Left\ c), [\,])
\end{aligned}
$$

In the final case, the proof standard assigned to the proposition was not satisfied.

$$
\mid otherwise = ((False, Left\ c), [(defeater, (False, Left\ c))])
$$

4.4 Translation properties

Given a translation function, we would like to convince ourselves that the translation is actually correct. To do so, we would want to prove properties that are commonly expected of a translation functions in argumentation theory, namely that arguments and propositions that were acceptable/unacceptable in the original model, after translation into the other model, are identifiable and will still be acceptable/unacceptable. These conditions are commonly called correspondence properties.

For the translation function here, we can refer to existing definitions of the correspondence of applicability of arguments and acceptability of propositions (Theorem 4.10 of [18]).

Theorem 4.1. *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 implementation of these properties in Haskell. If the translation function is a correct implementation, the Haskell implementation of the correspondence properties should always return $True$. However to constitute an actual (mechanised) proof we would need to convert the translation and the implementation of the correspondence properties in Haskell to a theorem prover like Agda.

$$
\begin{aligned}
&corApp :: CAES \rightarrow Bool \\
&corApp\ caes@(CAES\ (argSet, \_, \_)) = \\
&\quad \textbf{let } transCAES = translate\ caes \\
&\qquad appArgs \ = filter\ (`applicable`caes) \\
&\qquad\qquad\qquad (getAllArgs\ argSet) \\
&\qquad transArgs \ = stripRight\ \$\ groundedExt\ transCAES \\
&\quad \textbf{in } fromList\ appArgs \equiv fromList\ transArgs \\
&\\
&corAcc :: CAES \rightarrow Bool \\
&corAcc\ caes@(CAES\ (argSet, (assumptions, \_), \_)) = \\
&\quad \textbf{let } transCAES = translate\ caes \\
&\qquad accProps \ = filter\ (\lambda c \rightarrow c\ `acceptable`\ caes \vee \\
&\qquad\qquad\qquad\qquad c \in assumptions) \\
&\qquad\qquad\qquad (getProps\ argSet) \\
&\qquad transProps \ = stripLeft\ \$\ delete
\end{aligned}
$$

$$(Left\ \$\ mkProp\ \texttt{"defeater"})$$
$$(groundedExt\ transCAES)$$
$$\textbf{in}\ fromList\ accProps \equiv fromList\ transProps$$

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 3.6:

$$corApp\ caes \wedge corAcc\ caes$$
$$> True$$

5. Formalising Dung's AFs in a theorem prover

Translations between argumentation models can be notoriously complex. We hope that the preceding sections have convinced the reader that using a language like Haskell already makes this problem significantly easier. Similar advantages hold for the functional implementations of argumentation models, ensuring that the specification and implementation of the semantics are closely aligned. However in the case of the implementation of translations, given the complexity of proofs of correctness, and the difficulty even for experts of the field to check this work, we believe that mechanical formalisation of translations and their correctness proofs also have significant benefits. This section constitutes a first step to this goal by providing, to our knowledge, the first formalisation of an argumentation model in a theorem prover. Our main contribution is the formalisation of the Haskell functions as outlined in Section 2, leaving further proofs of properties for future work. The work done in this section corresponds to the box "Formalised Dung's AFs" in Figure 1.

Below we sketch a part of our formalisation of the grounded extension, giving type signature and describing the function definitions. Our formalisation has been done in Agda [26], a dependently typed programming language and interactive theorem prover. We have chosen Agda due to its functional nature and because its syntax is very close to the syntax of Haskell. Because a full treatment of the Agda code would unbalance the paper, we have made the remaining code available online[7].

In our initial formalisation attempted we tried to translate the $grounded'$ labelling function from Section 2 into Agda. The main change was to translate the typeclass statement $Eq \Rightarrow a$ to the passing of an equality function on A.

-- grounded' from our Haskell implementation

$$grounded' ::\ Eq\ a \Rightarrow [a] \to [a] \to [a] \to DungAF\ a \to [(a, Status)]$$

-- corresponding Agda type

$$groundedList : \{A : Set\} \to (A \to A \to Bool) \to List\ A \to List\ A \to List\ A \to DungAF\ A \to List\ (A \times Status)$$

However, taking this naive approach is not sufficient. The algorithm 2.9 works with three sets of arguments. The sets *in* and *out* keep track of the arguments that already have had their status determined. They get bigger as the algorithm proceeds. On the other hand, the set of as yet unlabelled/undecided arguments, initially labelled *undec*, does get smaller as newly labelled arguments are removed. Once no further arguments can be labelled, the recursion ends. However, because we do not directly match on the structure of the list *undec* it is not structurally decreasing. This in turn means that Agda's termination checker marks the function as possibly non-terminating.

[7] For the complete Agda code see:
`http://www.cs.nott.ac.uk/~bmv/Code/AF2.agda`

To resolve this problem we need to prove to the Agda termination checker that the size of *undec* is decreasing. We have solved this problem by first changing all lists to *Vector*s, thereby keeping track of the size of all the arguments, and secondly, by providing a proof that there is a certain number equal to the size of *undec* which is structurally decreasing. We give the type signature below.

$$grounded' : \{A : Set\} \to \{m\ n\ o : \mathbb{N}\} \to (\Sigma\ \mathbb{N}\ \lambda\ k \to k \equiv o) \to (A \to A \to Bool) \to Vec\ A\ m \to Vec\ A\ n \to Vec\ A\ o \to DungAF\ A \to Vec\ (A \times Status)\ (m + n + o)$$

Types with accompanying implementations (functions), correspond to theorems with accompanying proofs through the Curry-Howard correspondence [10, 11, 21], or the proofs-as-programs interpretation. This means that if we write an implementation/proof of grounded semantics in Agda, we get some key results for free. First of all, functions that are implemented are guaranteed to be terminating, which means that because we successfully implemented the grounded semantics, we immediately know that our algorithm is terminating on all (finite) inputs. Further, because Agda will always return a labelling, we also have proven that the grounded extension always exists, verifying one of Dung's original results [12]. 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).

6. Conclusions and future work

In this paper we have discussed a significant part of Dung's argumentation frameworks, the Carneades argumentation model and a translation from Carneades into AFs. We have shown that Haskell can provide a short and intuitive implementation, while keeping true to the original mathematical definitions. We have discussed the merits of formalising such an implementation in Agda, showing that is feasible to formalise an implementation of an argumentation model into a theorem prover. Finally, we gave a sketch of the required properties of a translation function, hinting at what to formalise in a theorem prover, with the ultimate goal to give us the means to translate between argumentation models in a verified manner.

The initial results are encouraging, despite that we haven't formalised an actual translation in a theorem prover yet. The successful implementation of a translation and the formalisation of Dung's argumentation frameworks suggest that the formalisation of a translation is not far off. It is important to note that our approach is not necessarily meant to give the final implementation of a model. The intended use of this approach is for quick prototyping/testing of argumentation models, followed by an implementation and verification of a translation between models, delegating the actual evaluation of arguments to an optimised implementation. Near future work is thus to connect the (verified) translations to existing efficient implementations as given in [8].

Instead of translating between argumentation models, we can also choose to translate to a specific format, such as a file format or a general format such as the Argument Interchange Format [9, 28]. Especially the recent work on giving a logical specification to the AIF [2] would be a good application for a theorem prover.

Acknowledgments

The authors would like to thank the anonymous reviewers for many useful comments and suggestions that helped improving the paper.

References

[1] P. Baroni and M. Giacomin. Semantics of abstract argument systems. In G. Simari and I. Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 25–44. Springer US, 2009. ISBN 978-0-387-98197-0.

[2] F. Bex, S. Modgil, H. Prakken, and C. Reed. On logical specifications of the Argument Interchange Format. *Journal of Logic and Computation*, 2012.

[3] A. Bondarenko, P. M. Dung, R. A. Kowalski, and F. Toni. An abstract, argumentation-theoretic framework for default reasoning. *Artificial Intelligence*, 93:63–101, 1997.

[4] G. Brewka and T. F. Gordon. Carneades and abstract dialectical frameworks: A reconstruction. In M. Giacomin and G. R. Simari, editors, *Computational Models of Argument. Proceedings of COMMA 2010*, pages 3–12, Amsterdam etc, 2010. IOS Press 2010.

[5] G. Brewka and S. 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.

[6] G. Brewka, P. E. Dunne, and S. 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.

[7] M. Caminada. An algorithm for computing semi-stable semantics. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 222–234. Springer, 2007.

[8] G. Charwat, W. Dvorák, S. A. Gaggl, J. P. Wallner, and S. Woltran. Implementing abstract argumentation - a survey. Technical Report DBAI-TR-2013-82, Vienna University of Technology, 2013.

[9] C. Chesñevar, J. McGinnis, S. Modgil, I. Rahwan, C. Reed, G. Simari, M. South, G. Vreeswijk, and S. Willmott. Towards an argument interchange format. *The Knowledge Engineering Review*, 21(4):293–316, 2006.

[10] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11): 584, 1934.

[11] H. B. Curry, R. Feys, W. Craig, J. R. Hindley, and J. P. Seldin. *Combinatory logic*, volume 2. North-Holland Amsterdam, 1972.

[12] P. M. 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. ISSN 0004-3702.

[13] M. Erwig. Inductive graphs and functional graph algorithms. *Journal Functional Programming*, 11(5):467–492, Sept. 2001. ISSN 0956-7968.

[14] A. M. Farley and K. 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. ISBN 0-89791-758-8.

[15] K. Freeman and A. M. Farley. A model of argumentation and its application to legal reasoning. *Artificial Intelligence and Law*, 4:163–197, 1996. ISSN 0924-8463.

[16] B. van Gijzel and H. 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.

[17] B. van Gijzel and H. Prakken. Relating Carneades with abstract argumentation. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-11)*, pages 1113–1119, 2011.

[18] B. van Gijzel and H. Prakken. Relating Carneades with abstract argumentation via the ASPIC$^+$ framework for structured argumentation. *Argument & Computation*, 3(1):21–47, 2012.

[19] T. F. Gordon and D. Walton. Proof burdens and standards. In G. Simari and I. Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 239–258. Springer US, 2009. ISBN 978-0-387-98197-0.

[20] T. F. Gordon, H. Prakken, and D. Walton. The Carneades model of argument and burden of proof. *Artificial Intelligence*, 171(10-15):875–896, 2007. ISSN 0004-3702.

[21] W. A. Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[22] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.

[23] P. Hudak. Modular domain specific languages and tools. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 134–142. IEEE, 1998.

[24] S. Modgil and M. Caminada. Proof theories and algorithms for abstract argumentation frameworks. In G. Simari and I. Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 105–129. Springer US, 2009. ISBN 978-0-387-98196-3.

[25] S. Modgil and H. Prakken. A general account of argumentation with preferences. *Artificial Intelligence*, 2012.

[26] U. 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. ISBN 978-1-60558-420-1.

[27] H. Prakken. An abstract framework for argumentation with structured arguments. *Argument & Computation*, 1:93–124, 2010.

[28] I. Rahwan and C. Reed. The argument interchange format. In G. Simari and I. Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 383–402. Springer US, 2009. ISBN 978-0-387-98197-0.

[29] G. 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. ISBN 978-3-642-23962-5.