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 an interdisciplinary field studying how conclusions can be reached through logical reasoning in settings where the soundness of arguments might be subjective and arguments can be contradictory. There are two main approaches: the structured approach giving a predetermined structure to arguments, including for example legal and scientific arguments, and the abstract approach making no specific assumptions about the form of arguments and thus being generally applicable. Structured argumentation models have seen a recent surge, with new developments in both general frameworks and more domain-specific approaches. For the abstract approach, a significant effort has been directed towards the construction of usable tools and efficient implementations. However, despite these tools and existing translations from structured into abstract argumentation frameworks in the literature, there is still a lack of implementations of the structured models and their translations, suggesting that there are different problems with the implementation of the structured counterpart.

Building on previous work, this paper attempts to tackle this problem by applying functional programming techniques. We first explain and implement two argumentation models: Dung's argumentation frameworks, the standard abstract model, and Carneades, a structured argumentation model used in the legal domain. Having expressed both in full concrete detail, yet in a sufficiently abstract way thanks to using functional programming, we provide one of the first *implementations* of a translation between argumentation models, fully documenting the techniques, making all work publicly available and reusable.

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 structured approach giving a predetermined structure to arguments, including for example legal and scientific arguments, and the abstract approach makes no specific assumptions about the form of arguments and thus being generally applicable. Structured argumentation models have seen a recent surge, with new developments in both general frameworks [3, 5, 23] and more domain-specific approaches [18, 19]. For the abstract approach, a significant effort has been directed towards the construction of usable tools and efficient implementations; see [7] for a survey. However, despite these tools and existing translations from structured into abstract argumentation frameworks in the literature [6, 16, 17, 22, 23], there is still a lack of *implementations* of the structured models and their translations, suggesting that there are different problems with the implementation of the structured counterpart.

We give a number of potential reasons:

- Abstract argumentation is closely related to logic programming [11], making it easier to develop implementations that are both intuitive and efficient [7]. In contrast, no mainstream, general-purpose language or paradigm provide an equally close fit for structured argumentation. There are a few implementations done in Java [25], but, Java being an imperative language, they are quite far removed from the logical specification thus making it significantly more difficult to verify whether the implementation is actually correct, rather than what is the case when specification and implementation are closely aligned.
- Most implementations of structured argumentation models are not publicly available. Simari [25] gives an overview of some of the structured argumentation models, but most implementations are now unavailable or closed source. In those cases code was never published, this means that the information regarding the techniques of the implementations has effectively 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$^+$ [16, 17] and the translation of abstract dialectical frameworks into Dung [6]. Both proofs are at least a page long and are hard to verify even for experts in the field.

This paper attempts to address this problem by applying functional programming techniques, using Haskell as our programming language. We argue that to simplify verification of structured argumentation frameworks and their translations, they should be implemented in a declarative language in such a way that the code is close to the actual mathematics. Due to their declarative nature, functional programming languages thus provide a good fit. A particular advantage of functional languages is their proven track record of being suitable for tailoring to specific needs through the approach

of Embedded Domain-Specific Languages (EDSL) [18, 19]. Our specific choice of programming language, Haskell, is further motivated by our previous work [15], where we managed to implement the Carneades argumentation model in such a way that is was easily readable by an argumentation theorist with no previous knowledge of Haskell. We thus hope that our approach ultimately could result in an EDSL that is as suitable for implementing structured argumentation frameworks as logic programming is for implementing abstract ones.

Our contributions are the following:

- We explain and implement two argumentation models: Dung's argumentation frameworks [11] (AFs), the standard abstract model, and Carneades [18, 19], a structured argumentation model used in the legal domain. We provide and discuss all the Haskell programming code of the implementation of Dung's AFs. The code is to a large extent a transliteration of standard definitions, allowing this paper to simultaneously serve documentation and implementation. Our discussion of Carneades is based on our previous work in [15].
- 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 an implementation of these properties in Haskell, and discuss a possible formalisation of these properties into 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.

The paper is structured as follows. In Section 2 we give an introduction to Dung's abstract argumentation frameworks, each time providing implementations of the definitions in functional programming language, Haskell. In Section 3 we provide an introduction to the Carneades model, again giving corresponding Haskell definitions. Section 4 discusses the implementation of a translation of Carneades into Dung's AFs and sketch an implementation of a few desired properties. In Section 5 we discuss the possibilities of formalising the implementation of the previous translation and briefly mention our formalisation of Dung's AFs in a theorem prover, Agda, and what we can gain from this. We conclude in Section 6 with a discussion of what we have learnt from this study and how we can take this further.

2. An implementation of AFs in Haskell

The abstract argument system, or argumentation frameworks (AFs) as introduced by Dung [11] is a very simple, yet general model that is able to capture various contemporary approaches to non-monotonic reasoning. It has also been the translation target for many modern structured argumentation models [6, 16, 17, 22, 23] that have been introduced later in the literature. This section gives a significant part of the standard definitions of Dung's AFs, including an algorithm for what is termed the *grounded semantics*, and show how these definitions can be almost immediately translated into (a slightly stylised version) of the functional programming language, Haskell[1]. The purpose of this section is to show how a functional programming language such as Haskell can be used to quickly implement a prototype of an argumentation model and in a way that is amenable to proving properties of this implementation. This section can also serve as a tutorial-like introduction to the implementation of AFs up to grounded semantics. The implementation is provided as a public library[2], in the same way as the code for our earlier work discussed in Section 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 and they may come from any domain, including informal ones such as "if it rains you get wet". Note that the attack relation is not assumed to be symmetric: an attacked argument does not necessarily constitute a counter-attack of the attacking argument.

Definition 2.1. Abstract argumentation framework An *abstract argumentation framework* is a tuple $\langle Args, Att\rangle$, such that $Args$ is a set of arguments and $Att \subseteq Args \times Args$ is an attack relation on the arguments in $Args$.

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 use $String$s to just label arguments.

$$\begin{aligned}&\textbf{data}\ DungAF\ arg = AF\ [arg]\ [(arg, arg)]\\&\quad \textbf{deriving}\ (Show)\\&\textbf{type}\ AbsArg = String\end{aligned}$$

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

Example 2.2. An example (abstract) argumentation framework containing three arguments where the argument C *reinstates* the argument A by attacking its attacking argument B is captured by $AF_1 = \langle\{A, B, C\}, \{(A, B), (B, C)\}\rangle$.

Figure 1. An (abstract) argumentation framework

And in Haskell:

$$\begin{aligned}&a, b, c :: AbsArg\\&a = \texttt{"A"}\\&b = \texttt{"B"}\\&c = \texttt{"C"}\\&AF_1 :: DungAF\ AbsArg\\&AF_1 = AF\ [a, b, c]\ [(a, b), (b, c)]\end{aligned}$$

We now quickly give a few standard definitions for AFs such as the acceptability of arguments and admissibility of sets. We will use an arbitrary but fixed argumentation framework $AF = \langle Args, Att\rangle$.

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 $(Y, X) \in Att$.

For example, in Figure 1, $\{A, B\}$ 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 $(X, Y) \in Att$.

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

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

Considering a set of arguments as a position an agent can take with regards to its knowledge, conflict-freeness is then often the minimal requirement for a reasonable position. For example, in Figure 1, $\{A, C\}$ would be 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 $Y \in S$: if $(Y, X) \in Att$ then there is a $Z \in S$ for which $(Z, Y) \in Att$.

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 1, $\{C\}$ is acceptable w.r.t. $\{A, B, C\}$, because A attacks the only attacker of C, i.e. B.

Dung defined the semantics of the argumentation frameworks by using the concept of *extensions* and the *characteristic function* of an AF. An extension is always a subset of $Args$, and can intuitively be seen as a set of arguments that are acceptable when taken together. We will just discuss the grounded extension, but for completeness we give the four standard semantics defined by Dung.

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 1, $F_{AF}(\emptyset) = \{A\}$, $F_{AF}(A) = \{A, C\}$ 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 1, $\{C\}$ is conflict-free but is not an admissible set, since $(B, C) \in Att$ and there is no argument in $\{C\}$ that defends from this attack.

Definition 2.8 (Extensions)**.** Given a conflict-free set of arguments S, argumentation framework AF, and if the domain of F is ordered with respect to set inclusion then:

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

As proven in Dung [11], given that that the domain of F is ordered w.r.t. to set inclusion, F is monotonic. 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.

```
setAttacks :: Eq arg ⇒ DungAF arg → [arg] →
              arg → Bool
setAttacks (AF _ att) args arg
  = or [y ≡ arg | (x, y) ← att, x ∈ args]
conflictFree :: Eq arg ⇒ DungAF arg → [arg] → Bool
conflictFree (AF _ att) args
  = null [(x, y) | (x, y) ← att, x ∈ args, y ∈ args]
acceptable :: Eq arg ⇒ DungAF arg → arg →
              [arg] → Bool
acceptable af@(AF _ att) x args
  = and [setAttacks af args y | (y, x') ← att, 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
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"]
```

Note that by the required $Eq\ arg \Rightarrow$, Haskell forces us to see that we need an equality on arguments to be able implement these functions.

Given an argumentation framework, we can determine which arguments are justified by applying an argumentation semantics. However, in contrast to the succinct extension based approach, we will take the *labelling-based* approach to grounded semantics. The labelling based approach is more commonly used in actual implementations and thus easier compare to existing implementations. Furthermore, this obviates the need to formalise fixed points, significantly reducing the amount of work needed when implementing everything in a theorem prover. Below we have given the commonly used algorithm for grounded labelling [21]. However, for correctness, the "if then" in the $\exists$ has been changed to "and". Although this small fix in the algorithm is obvious and to our knowledge has not been adopted in any actual implementation, it does make a case in point for formalisation of more complex mathematics such as a translation between argumentation models.

Algorithm 2.9. Algorithm for grounded labelling (Algorithm 6.1 of [21])

1. $\mathcal{L}_0 = (\emptyset, \emptyset, \emptyset)$
2. **repeat**
3. $\text{in}(\mathcal{L}_{i+1}) = \text{in}(\mathcal{L}_i) \cup \{x \mid x \text{ is not labelled in } \mathcal{L}_i,$ $\forall y : \text{if } y \mathcal{R} x \text{ then } y \in \text{out}(\mathcal{L}_i)\}$
4. $\text{out}(\mathcal{L}_{i+1}) = \text{out}(\mathcal{L}_i) \cup \{x \mid x \text{ is not labelled in } \mathcal{L}_i,$ $\exists y : y \mathcal{R} x \text{ and } y \in \text{in}(\mathcal{L}_{i+1})\}$
5. **until** $\mathcal{L}_{i+1} = \mathcal{L}_i$
6. $\mathcal{L}_\mathcal{G} = (\text{in}(\mathcal{L}_i), \text{out}(\mathcal{L}_i), \mathcal{A} - (\text{in}(\mathcal{L}_i) \cup \text{out}(\mathcal{L}_i)))$

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 _ att) arg =
  let attackers = [x | (x, y) ← att, 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 _ att) arg =
  let attackers = [x | (x, y) ← att, arg ≡ y]
  in ¬ (null (attackers `intersect` ins))
```

We split the implementation in two parts. A function for the grounded labelling which can immediately be applied to an AF, and a function actually implementing the algorithm, which has an additional two arguments that accumulate the Ins and Outs.

```
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 args af =
  let newIns  = filter (unattacked outs af) args
      newOuts = filter (attacked ins af)    args
  in if null (newIns ++ newOuts)
     then map (λx → (x, In)) ins
          ++ map (λx → (x, Out)) outs
          ++ map (λx → (x, Undecided)) args
     else grounded' (ins ++ newIns)
                    (outs ++ newOuts)
                    (args \\ (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 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

In this section we will give definitions and again corresponding implementations in Haskell of the Carneades argumentation model [18, 19], an argumentation model designed to capture standards and burdens of proof. This is largely based on previous work in [15][3]. We have included the majority of the previous work, to ensure that this paper is self-contained. We provide a shorter treatment of this implementation, together with one alteration: we have changed the use of proof standards by relying on the definition of $ProofStandardNamed$ to allow for an easier translation.

3.1 Arguments

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.

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.

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

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)
```

The negation for a literal p, written $\overline{p}$ is then given as follows:

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

We chose 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*:

```
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 $Argument$ (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 [12] 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 = ...
getArgs    :: PropLiteral → ArgSet → [Argument]
checkCycle :: ArgSet → Bool
```

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

```
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 by the audience and *weight* is a function mapping arguments to a real-valued weight in the range $[0, 1]$.

This definition is captured by the following Haskell definitions:

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

Further, as each proposition is associated with a specific proof standard, we need a mapping from propositions to proof standards:

```
type PropStandard = PropLiteral → ProofStandardNamed
```

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

newtype ProofStandardNamed =
P (String, PropLiteral → CAES → Bool)

instance Eq ProofStandardNamed where
P (l1, _) ≡ P (l2, _) = l1 ≡ l2
```

We also introduce a named version of the proof standard, together with an equality instance, so that later in the translation step we can check the used proof standard. 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

- $p \in P$ 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 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:

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

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

3.4 Proof standards

Carneades predefines five proof standards, originating from the work of Freeman and Farley [13, 14]: *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, we proceed to give the definitions directly 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, we start by defining those notions:

```
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
```

We can then define the proof standard preponderance:

```
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)
```

3.5 Convenience functions

We provide a set of functions to facilitate construction of propositions, arguments, argument sets and sets of assumptions.

```
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]
```

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 as well.

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

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

```
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") =
  P ("beyond_reasonable_doubt",
     beyond_reasonable_doubt)
standard _             =
  P ("scintilla", 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
```

4. An implementation from Carneades into AFs

In the previous two sections we have seen both an implementation of Dung's argumentation frameworks and an implementation of Carneades, a structured argumentation model. There is an existing translation between the two models [16, 17], which provides a means to translate Carneades into the ASPIC$^+$ [23] structured argumentation model, which is known to generate Dung's AFs [23], thereby providing an indirect translation from Carneades into Dung's argumentation frameworks.

In this Section, we discuss an implementation that indeed translated a CAES from the Carneades argumentation model into

Dung's argumentation frameworks. However, to avoid having to introduce the ASPIC$^+$ model, we instead 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 generation (Adapted Definition 4.9 of [17])

1. $generatedArgs = choiceArgs = \emptyset$.
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*, using one argument per premise out of $choiceArgs$ as subarguments, and put the generated arguments in $tempArgs$.
   (c) Add all arguments from $tempArgs$ to $argSet$.
   (d) If present, pick one acceptable argument in $tempArgs$ that has the conclusion c and add it to $choiceArgs$.
   (e) $argSet = tempArgs = \emptyset$.

For our implementation we give the main functions or type signatures involved[4]. If we look at the translation as given in van Gijzel and Prakken [17], we can make the following observation: when a CAES is translated into Dung (through ASPIC$^+$) we end up with arguments in our Dung framework for the propositions which were in $\mathcal{L}$ (from Carneades), but also with arguments with represent the original Carneades argument nodes. Therefore, our arguments, instead of just having a $String$ labelled arguments, will contain both label and propositional literal (either standard proposition or a proposition associated with an argument). Then, using this argument type, we can define a similarly instantiated AF.

$$\textbf{type } Label = String$$
$$\textbf{type } ConcreteArg = (Label, PropLiteral)$$
$$\textbf{type } ConcreteAF = DungAF\ ConcreteArg$$

The main translation function calls functions to translate the assumptions and the argument graph in a CAES, by calling two helper functions $propToArg$ and $argsToAF$.

```
translate :: CAES → ConcreteAF
translate caes@(CAES (argset,
                      (assumptions, weights),
                      standard)) =
  AF args attacks
where args = map propToArg assumptions ++ args'
  AF args' attacks = argsToAF (topSort argset)
                              caes (AF [] [])
```

$topSort$ is a topological sorting of the argument graph. We implement our topological sorting by using FGL's implementation of it. However, due to the definition of dependency graphs, we have to reverse the ordering.

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

A proposition can readily be translated to an argument in an AF.

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

```
propToArg :: PropLiteral → ConcreteArg
propToArg p@(True, l)  = (l, p)
propToArg p@(False, l) = ("not " ++ l, p)

argsToAF :: [(PropLiteral, [Argument])] →
            CAES → ConcreteAF →
            ConcreteAF
```

Given the translation, we need to be able to map our translated arguments and propositions to the original arguments and propositions in a CAES.

```
arg2dung :: ConcreteAF →
            Argument → ConcreteArg
prop2dung :: ConcreteAF →
             PropLiteral → ConcreteArg
```

Given a translation function, we can talk about the properties we would need to be able 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 [17]).

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. I 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.

```
corApp :: CAES → Bool
corApp caes@(CAES (argset, _, _)) =
  let translatedCAES = translate caes
      applicableArgs = filter (`applicable` caes)
                              (getAllArgs argset)
      corDungArgs    = map (arg2dung translatedCAES)
                           applicableArgs
  in corDungArgs ≡ groundedExt translatedCAES
corAcc :: CAES → Bool
corAcc caes@(CAES (argset, _, _)) =
  let translatedCAES = translate caes
      acceptableProps = filter (`acceptable` caes)
                               (getProps argset)
      corDungArgs     = map (prop2dung translatedCAES)
                            acceptableProps
  in corDungArgs ≡ groundedExt translatedCAES
```

5. Formalising Dung's AFs in a theorem prover

Translations between argumentation models can be notoriously complex. We hope that the previous sections have convinced the reader that using Haskell as a programming language 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 for even 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.

In Section 2 we were able to construct an implementation of Dung's AFs in Haskell. Given that Agda is also functional in nature and very close to Haskell in syntax, it is an obvious choice. Agda is a programming language and a theorem prover at the same time. Types with accompanying implementations (functions), correspond to theorems with accompanying proofs through the Curry-Howard correspondence [9, 10, 20], or the proofs-as-programs interpretation. This means that if we write an implementation/proof of grounded semantics in Agda we already gain a few nice results for free. Firstly, 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 and because Agda will always give back a labelling, we also have proven that the grounded extension always exists, verifying one of Dung's original results [11]. 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.

The mathematical properties proven in such a 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). Because a full treatment of the Agda code is not the focus of this paper, we have instead made the relevant code available online[5].

6. Conclusions

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 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 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 efficient implementations as given in [7].

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 [8, 24]. Especially the recent work on giving a logical specification to the AIF [2] would be a good application for a theorem prover.

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

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. URL `http://dx.doi.org/10.1007/978-0-387-98197-0_2`.

[2] F. Bex, S. Modgil, H. Prakken, and C. Reed. On logical specifications of the Argument Interchange Format. *Journal of Logic and Computation*, 2012. . URL `http://logcom.oxfordjournals.org/content/early/2012/08/03/logcom.exs033.abstract`.

[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] 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. URL `http://www.dbai.tuwien.ac.at/research/report/dbai-tr-2013-82.pdf`.

[8] 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. URL `http://www.journals.cambridge.org/abstract_S0269888906001044`.

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

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

[11] 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. .

[12] M. Erwig. Inductive graphs and functional graph algorithms. *Journal Functional Programming*, 11(5):467–492, Sept. 2001. ISSN 0956-7968. . URL `http://dx.doi.org/10.1017/S0956796801004075`.

[13] 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. .

[14] 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. URL `http://dx.doi.org/10.1007/BF00118492`. 10.1007/BF00118492.

[15] 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.

[16] 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.

[17] 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. . URL `http://www.tandfonline.com/doi/abs/10.1080/19462166.2012.661766`.

[18] 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.

[19] 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. .

[20] 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.

[21] 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. . URL `http://dx.doi.org/10.1007/978-0-387-98197-0_6`.

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

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

[24] 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. URL `http://dx.doi.org/10.1007/978-0-387-98197-0_19`.

[25] 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. URL `http://dl.acm.org/citation.cfm?id=2050266.2050276`.