A principled approach to the implementation of argumentation models

Bas van Gijzel

University of Nottingham

March 4, 2014
Outline

1. Introduction to argumentation
   - A perceived problem
   - A proposed solution

2. Implementing argumentation models using Haskell
   - Dung’s AFs
   - Carneades
     - Translation from Carneades into Dung’s AFs

3. Verifying correctness of an implementation
   - Implementation of properties
   - Proving of properties

4. Further/future work and conclusions
Outline

1. Introduction to argumentation
   A perceived problem
   A proposed solution

2. Implementing argumentation models using Haskell
   Dung’s AFs
   Carneades
   Translation from Carneades into Dung’s AFs

3. Verifying correctness of an implementation
   Implementation of properties
   Proving of properties

4. Further/future work and conclusions
Argumentation

An example argument in the legal domain:
An example argument in the legal domain:
Argumentation theory

Interdisciplinary area with various applications:
Argumentation theory

Interdisciplinary area with various applications:

• **Law:**
  Systems *modelling* legal problems/cases,
Argumentation theory

Interdisciplinary area with various applications:

• **Law:**
  Systems *modelling* legal problems/cases,

• **Decision making:**
  *Organising* information and source of *efficiency* in decision theory,
Argumentation theory

Interdisciplinary area with various applications:

- **Law:**
  Systems *modelling* legal problems/cases,

- **Decision making:**
  *Organising* information and source of *efficiency* in decision theory,

- **Communication theory/linguistics:**
  *Making* argumentation in existing texts *precise.*
Argumentation theory

Interdisciplinary area with various applications:

• **Law:**
  Systems *modelling* legal problems/cases,

• **Decision making:**
  Organising information and source of *efficiency* in decision theory,

• **Communication theory/linguistics:**
  Making argumentation in existing texts *precise*.

All these topics can give rise to different notions of argument and therefore different argumentation models.
Types of argumentation models

Two types of argumentation models:

• Abstract models
  - Abstract from the concrete structure of argument and the reasons of conflict between arguments,
  - Elegant and easy to understand, but impractical for directly modelling complex arguments.

• Structured models
  - Specify the nature of the argument construction and explicitly build up conflict relation(s).
  - Utilises domain-specific constructs to closely model actual argumentation problems.
Types of argumentation models

Two types of argumentation models:

• Abstract models
Types of argumentation models

Two types of argumentation models:

• Abstract models
  • Abstract from the concrete structure of argument and the reasons of conflict between arguments,
Types of argumentation models

Two types of argumentation models:

• **Abstract models**
  • Abstract from the concrete **structure of argument** and the **reasons of conflict** between arguments,
  • **Elegant** and **easy** to understand, but **impractical** for directly modelling complex arguments.
Types of argumentation models

Two types of argumentation models:

• **Abstract models**
  • Abstract from the concrete structure of argument and the reasons of conflict between arguments,
  • Elegant and easy to understand, but impractical for directly modelling complex arguments.

• **Structured models**
Two types of argumentation models:

- **Abstract models**
  - Abstract from the concrete structure of argument and the reasons of conflict between arguments,
  - Elegant and easy to understand, but impractical for directly modelling complex arguments.

- **Structured models**
  - Specify the nature of the argument construction and explicitly build up conflict relation(s).
Types of argumentation models

Two types of argumentation models:

• **Abstract models**
  • Abstract from the concrete **structure of argument** and the reasons of **conflict** between arguments,
  • **Elegant** and **easy** to understand, but **impractical** for directly modelling complex arguments.

• **Structured models**
  • **Specify** the nature of the **argument construction** and explicitly build up **conflict relation(s)**.
  • Utilises **domain-specific** constructs to **closely model** actual argumentation problems.
Abstract argumentation: Dung’s AFs

Dung’s (abstract) argumentation frameworks (AFs) are a gold standard of argumentation.
Abstract argumentation: Dung’s AFs

Dung’s (abstract) argumentation frameworks (AFs) are a gold standard of argumentation.

• Relatively simple data structures/algorithms (complexity still NP or higher for most problems)
Abstract argumentation: Dung’s AFs

Dung’s (abstract) argumentation frameworks (AFs) are a gold standard of argumentation.

- Relatively simple data structures/ algorithms (complexity still NP or higher for most problems)
- Has been used as a base for many other abstract models
Abstract argumentation: Dung’s AFs

Dung’s (abstract) argumentation frameworks (AFs) are a gold standard of argumentation.

• Relatively simple data structures/algorithms (complexity still NP or higher for most problems)
• Has been used as a base for many other abstract models
• A significant amount of models, including structured models, are instances of Dung’s model (are translatable to)
How to implement an argumentation model

Two main ways to implement an argumentation model:

• Directly implement it into your favourite programming language,
• Implement the translation, given a formal relation to another (implemented) simpler model (e.g. to Dung's AFs).
How to implement an argumentation model

Two main ways to implement an argumentation model:

- Directly implement it into your favourite programming language,
Two main ways to implement an argumentation model:

- **Directly implement** it into your favourite programming language,
- **Implement the translation**, given a formal relation to another (implemented) simpler model (e.g. to Dung’s AFs).
Outline

1. Introduction to argumentation
   A perceived problem
   A proposed solution

2. Implementing argumentation models using Haskell
   Dung’s AFs
   Carneades
   Translation from Carneades into Dung’s AFs

3. Verifying correctness of an implementation
   Implementation of properties
   Proving of properties

4. Further/future work and conclusions
Implementations of abstract models

State of implementations for Dung’s AFs:

- A decent amount of well-documented and open source applications.
- Recent efforts to optimise the evaluation of AFs using:
  - SAT-solvers
  - Answer-set programming
- A decent amount of other abstract models have been implemented through encodings into AFs. For instance CEGARTIX and Vispartix:
  - See: http://www.dbai.tuwien.ac.at/proj/argumentation/cegartix/
  - See: http://www.dbai.tuwien.ac.at/proj/argumentation/vispartix/
Implementations of abstract models

State of implementations for Dung’s AFs:

- A decent amount of well-documented and open source applications.
Implementations of abstract models

State of implementations for Dung’s AFs:

• A decent amount of well-documented and open source applications.
• Recent efforts to optimise the evaluation of AFs using:
Implementations of abstract models

State of implementations for Dung’s AFs:

• A decent amount of well-documented and open source applications.
• Recent efforts to optimise the evaluation of AFs using:
  • SAT-solvers
Implementations of abstract models

State of implementations for Dung’s AFs:

• A decent amount of *well-documented and open source* applications.

• Recent efforts to *optimise* the evaluation of AFs using:
  • SAT-solvers
  • Answer-set programming

For instance CEGARTIX and Vispartix:

See: 
http://www.dbai.tuwien.ac.at/proj/argumentation/cegartix/ 
http://www.dbai.tuwien.ac.at/proj/argumentation/vispartix/
Implementations of abstract models

State of implementations for Dung’s AFs:

• A decent amount of well-documented and open source applications.
• Recent efforts to optimise the evaluation of AFs using:
  • SAT-solvers
  • Answer-set programming
• A decent amount of other abstract models have been implemented through encodings into AFs.
Implementations of abstract models

State of implementations for Dung’s AFs:

- A decent amount of well-documented and open source applications.
- Recent efforts to optimise the evaluation of AFs using:
  - SAT-solvers
  - Answer-set programming
- A decent amount of other abstract models have been implemented through encodings into AFs.

For instance CEGARTIX and Vispartix:

See:
http://www.dbai.tuwien.ac.at/proj/argumentation/cegartix/
http://www.dbai.tuwien.ac.at/proj/argumentation/vispartix/
Implementations of structured models

State of implementations for structured models:
Implementations of structured models

State of implementations for structured models: ASPIC
Implementations of structured models

State of implementations for structured models: ASPIC
Implementations of structured models

State of implementations for structured models: ARGUGRID
Implementations of structured models

State of implementations for structured models: ARGUGRID
Implementations of structured models

State of implementations for structured models: ASPIC+
Implementations of structured models

State of implementations for structured models: ASPIC+
Implementations of structured models

State of implementations for structured models: ASPIC+

However, no source code is available!
Implementations of structured models

State of implementations for structured models: Carneades
Implementations of structured models

State of implementations for structured models: Carneades

Carneades is an argument mapping and evaluation application, with a graphical user interface, and a software library for building applications supporting various argumentation tasks. [http://carneades.github.com/](http://carneades.github.com/)

- 2,175 commits
- 6 branches
- 7 releases
- 5 contributors

- fixed ring issue. site is now available at <host>:<port>
  - sekaiser authored a day ago
  - latest commit 146553b6d

- arguments
  - Added a README.txt file to the example LKIF arguments along with a nX...
  - a month ago

- config
  - Merge 'project' branch
  - 11 months ago

- doc
  - relicensces the project from EuPL to MPL 2.0
  - 2 days ago

- license
  - relicensces the project from EuPL to MPL 2.0
  - 2 days ago

- projects
  - make the loading of the MARKOS theories robust to (missing) files errors
  - 4 months ago

- schemas
  - Continued work on the GraphML export
  - 10 months ago

- src
  - fixed ring issue. site is now available at <host>:<port>
  - a day ago

- .gitignore
  - adds pom.xml file to .gitignore
  - 4 days ago

- INSTALL.txt
  - Continued writing the user manual
  - 1 year ago

- README
  - relicensces the project from EuPL to MPL 2.0
  - 2 days ago
Implementations of structured models

State of implementations for structured models: Carneades

Open source and mature, but implementation does not correspond to a mathematical model!
Implementations of structured models

In conclusion:

• Significant amount of implementations are unavailable and closed source:
  • Effort spent is practically lost,
  • Implementation techniques are non-reproducible.
• A few mature implementations, but:
  • Again often closed source,
  • Often not directly related to the actual mathematical model,
  • Sometimes not even correctly implemented!
  • No implemented translations as far as I am aware!
Implementations of structured models

In conclusion:

• Significant amount of implementations are unavailable and closed source:
  • Effort spent is practically lost,
Implementations of structured models

In conclusion:

- Significant amount of **implementations are unavailable** and **closed source**:
  - **Effort** spent is practically **lost**,
  - **Implementation techniques are non-reproducible**.
Implementations of structured models

In conclusion:

• Significant amount of implementations are unavailable and closed source:
  • Effort spent is practically lost,
  • Implementation techniques are non-reproducible.

• A few mature implementations, but:
Implementations of structured models

In conclusion:
• Significant amount of implementations are unavailable and closed source:
  • Effort spent is practically lost,
  • Implementation techniques are non-reproducible.
• A few mature implementations, but:
  • Again often closed source,
Implementations of structured models

In conclusion:

• Significant amount of implementations are unavailable and closed source:
  • Effort spent is practically lost,
  • Implementation techniques are non-reproducible.

• A few mature implementations, but:
  • Again often closed source,
  • Often not directly related to the actual mathematical model,
Implementations of structured models

In conclusion:

• Significant amount of implementations are unavailable and closed source:
  • Effort spent is practically lost,
  • Implementation techniques are non-reproducible.

• A few mature implementations, but:
  • Again often closed source,
  • Often not directly related to the actual mathematical model,
  • Sometimes not even correctly implemented!
Implementations of structured models

In conclusion:

• Significant amount of implementations are unavailable and closed source:
  • Effort spent is practically lost,
  • Implementation techniques are non-reproducible.

• A few mature implementations, but:
  • Again often closed source,
  • Often not directly related to the actual mathematical model,
  • Sometimes not even correctly implemented!

No implemented translations as far as I am aware!
Outline

1. Introduction to argumentation
   A perceived problem
   A proposed solution

2. Implementing argumentation models using Haskell
   Dung’s AFs
   Carneades
   Translation from Carneades into Dung’s AFs

3. Verifying correctness of an implementation
   Implementation of properties
   Proving of properties

4. Further/future work and conclusions
Argumentation and implementation language

Abstract argumentation can be implemented using:

• Logic programming, formally related to Dung's argumentation frameworks,
• Answer set programming, a natural candidate for calculating semantics (extensions).

Structured argumentation models need a similar language:

• Able to easily express general mathematics,
• Data structures.

My suggestion: functional programming, in specific Haskell.
Argumentation and implementation language

Abstract argumentation can be implemented using:

- Logic programming, formally related to Dung’s argumentation frameworks,
Abstract argumentation can be implemented using:

- **Logic programming**, formally related to Dung’s argumentation frameworks,
- **Answer set programming**, a natural candidate for calculating semantics (extensions).
Argumentation and implementation language

Abstract argumentation can be implemented using:

- Logic programming, formally related to Dung’s argumentation frameworks,
- Answer set programming, a natural candidate for calculating semantics (extensions).

Structured argumentation models need a similar language:
Argumentation and implementation language

Abstract argumentation can be implemented using:

- Logic programming, formally related to Dung’s argumentation frameworks,
- Answer set programming, a natural candidate for calculating semantics (extensions).

Structured argumentation models need a similar language:

- Able to easily express general mathematics,
- Data structures.
Abstract argumentation can be implemented using:

- Logic programming, formally related to Dung’s argumentation frameworks,
- Answer set programming, a natural candidate for calculating semantics (extensions).

Structured argumentation models need a similar language:

- Able to easily express general mathematics,
- Data structures.

My suggestion: functional programming, in specific Haskell.
Goals

One goal of my current work and PhD is to provide a framework for:
One goal of my current work and PhD is to provide a framework for:

- implementation methods and reproducible, open source implementations for:
One goal of my current work and PhD is to provide a framework for:

- implementation methods and reproducible, open source implementations for:
  - *Structured* and some abstract models;
One goal of my current work and PhD is to provide a framework for:

- **implementation methods and reproducible, open source implementations for:**
  - **Structured** and some abstract models;
  - **Translations** between argumentation models.
Goals

To take this further and provide a framework for:
Goals

To take this further and provide a framework for:

- quickly testing properties such as rationality postulates and theorems;
Goals

To take this further and provide a framework for:

• **quickly testing** properties such as **rationality postulates** and **theorems**;
• **quickly testing correctness of translations** through correspondence properties.
Goals

Then, to completely formalise this and make a framework for:
Goals

Then, to completely formalise this and make a framework for:

- proving properties such as rationality postulates and theorems;
Goals

Then, to completely formalise this and make a framework for:

• **proving** properties such as rationality postulates and theorems;

• **proving correctness of translations** through proving correspondence properties.
Goals

Then, to completely formalise this and make a framework for:

• **proving** properties such as *rationality postulates* and theorems;

• **proving correctness of translations** through proving correspondence properties.

Finally, to:
Then, to completely formalise this and make a framework for:

- proving properties such as rationality postulates and theorems;
- proving correctness of translations through proving correspondence properties.

Finally, to:

- Connect the implementation of Dung’s AFs to an optimised implementation using ASP or SAT.
Then, to completely **formalise** this and make a framework for:

- **proving** properties such as **rationality postulates** and theorems;
- **proving correctness of translations** through proving correspondence properties.

Finally, to:

- **Connect the implementation** of Dung’s AFs to an **optimised** implementation using ASP or SAT.

Result: a **verified way to translate** models to an **efficiently implemented** model.
Outline

1. Introduction to argumentation
   A perceived problem
   A proposed solution

2. Implementing argumentation models using Haskell
   Dung’s AFs
   Carneades
   Translation from Carneades into Dung’s AFs

3. Verifying correctness of an implementation
   Implementation of properties
   Proving of properties

4. Further/future work and conclusions
Outline

1. Introduction to argumentation
   A perceived problem
   A proposed solution

2. Implementing argumentation models using Haskell
   Dung’s AFs
   Carneades
   Translation from Carneades into Dung’s AFs

3. Verifying correctness of an implementation
   Implementation of properties
   Proving of properties

4. Further/future work and conclusions
Crash course: Haskell Data definitions

Haskell has algebraic data types, allowing pattern matching:

```haskell
data Bool = True | False

neg True = False
neg False = True

data Maybe a = Nothing | Just a

-- Safe division
divide :: Double → Double → Maybe Double
divide a 0 = Nothing
divide a b = Just (a / b)

divide 3 4 ⇒ Just 0.75
divide 3 0 ⇒ Nothing
```
Crash course: Haskell Data definitions

Haskell has algebraic data types, allowing pattern matching:

```haskell
data Bool = True | False

neg True = False
neg False = True
```

Data types with type parameters:

```haskell
data Maybe a = Nothing | Just a
```

Safe division:

```haskell
divide :: Double → Double → Maybe Double
divide a 0 = Nothing
divide a b = Just (a / b)
```

Example:

- `divide 3 4` results in `Just 0.75`.
- `divide 3 0` results in `Nothing`.
Crash course: Haskell Data definitions

Haskell has algebraic data types, allowing pattern matching:

```
data Bool = True | False

neg True = False
neg False = True
```

Data types with type parameters:
Crash course: Haskell Data definitions

Haskell has algebraic data types, allowing pattern matching:

```
data Bool = True | False

neg True = False
neg False = True
```

Data types with type parameters:

```
data Maybe a = Nothing | Just a
```
Crash course: Haskell Data definitions

Haskell has algebraic data types, allowing pattern matching:

```haskell
data Bool = True | False

neg True = False
neg False = True
```

Data types with type parameters:

```haskell
data Maybe a = Nothing | Just a
```

-- Safe division
```haskell
divide :: Double → Double → Maybe Double
divide a 0 = Nothing
divide a b = Just (a / b)
```
Crash course: Haskell Data definitions

Haskell has algebraic data types, allowing pattern matching:

```haskell
data Bool = True | False

neg True = False
neg False = True
```

Data types with type parameters:

```haskell
data Maybe a = Nothing | Just a
```

-- Safe division
```haskell
divide :: Double → Double → Maybe Double
divide a 0 = Nothing
divide a b = Just (a / b)
```

> divide 3 4 ⇒ Just 0.75
> divide 3 0 ⇒ Nothing
An abstract argumentation framework (AF) is a tuple $AF = \langle \text{Args}, \text{Att} \rangle$ such that:

- $\text{Args}$ is a set of (abstract) arguments,
- $\text{Att} \subseteq \text{Args} \times \text{Args}$.

In other words a directed graph.
Definition of AFs

An abstract argumentation framework (AF) is a tuple $AF = \langle Args, Att \rangle$ such that:

- $Args$ is a set of (abstract) arguments,
- $Att \subseteq Args \times Args$.
Definition of AFs

An abstract argumentation framework (AF) is a tuple $AF = \langle \text{Args}, \text{Att} \rangle$ such that:

- $\text{Args}$ is a set of (abstract) arguments,
- $\text{Att} \subseteq \text{Args} \times \text{Args}$.

In other words a directed graph.
An abstract argumentation framework (AF) is a tuple $AF = \langle \text{Args}, \text{Att} \rangle$ such that:

- $\text{Args}$ is a set of (abstract) arguments,
- $\text{Att} \subseteq \text{Args} \times \text{Args}$.

In other words a directed graph.

\[ A \rightarrow B \rightarrow C \]
Given $AF = \langle \text{Args}, \text{Att} \rangle$
AFs in Haskell

Given $AF = \langle \text{Args}, \text{Att} \rangle$

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

Considering arguments as Strings:

```haskell
type AbsArg = String

A \to A \to A \to A
```

And in Haskell:

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

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

Given $AF = \langle \text{Args}, \text{Att} \rangle$

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

Considering arguments as **Strings**: 
AFs in Haskell

Given $AF = \langle \text{Args}, \text{Att} \rangle$

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

Considering arguments as Strings:

```haskell
type AbsArg = String
```

AFs in Haskell

Given $AF = \langle \text{Args}, \text{Att} \rangle$

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

Considering arguments as Strings:

```
type AbsArg = String
```

```
A --> B --> C
```

AFs in Haskell

Given \( AF = \langle \text{Args}, \text{Att} \rangle \)

\[
\text{data DungAF } \text{arg} = AF [\text{arg}] [(\text{arg}, \text{arg})]
\]

Considering arguments as Strings:

\[
\text{type AbsArg = String}
\]

\[
A \rightarrow B \rightarrow C
\]

And in Haskell:

\[
a, b, c :: \text{AbsArg} \\
a = "A" \\
b = "B" \\
c = "C"
\]

\[
AF_1 :: DungAF AbsArg \\
AF_1 = AF [a, b, c] [(a, b), (b, c)]
\]
Attacking with a set of arguments

Given $AF = \langle \text{Args}, \text{Att} \rangle$. 

Attacking with a set of arguments

Given \( AF = \langle \text{Args}, \text{Att} \rangle \).

A set \( S \subseteq \text{Args} \) of arguments \textbf{attacks} an argument \( A \in \text{Args} \)
Attacking with a set of arguments

Given $AF = \langle \text{Args}, \text{Att} \rangle$.

A set $S \subseteq \text{Args}$ of arguments attacks an argument $A \in \text{Args}$ iff there exists a $B \in S$ such that $(B, A) \in \text{Att}$.
Attacking with a set of arguments

Given $AF = \langle \text{Args, Att} \rangle$.

A set $S \subseteq \text{Args}$ of arguments attacks an argument $A \in \text{Args}$ iff there exists a $B \in S$ such that $(B,A) \in \text{Att}$.

In Haskell:

```
setAttacks :: Eq arg ⇒ DungAF arg → [arg] → arg → Bool
setAttacks (AF _ att) args arg
  = or [b ≡ arg | (a,b) ← att, a ∈ args]
```
Attacking with a set of arguments

Given $AF = \langle \text{Args}, \text{Att} \rangle$.

A set $S \subseteq \text{Args}$ of arguments attacks an argument $A \in \text{Args}$ iff there exists a $B \in S$ such that $(B, A) \in \text{Att}$.

In Haskell:

```haskell
setAttacks :: Eq arg ⇒ DungAF arg → [arg] → arg → Bool
setAttacks (AF _ att) args arg
  = or [b ≡ arg | (a, b) ← att, a ∈ args]
```

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 $AF = \langle \text{Args}, \text{Att} \rangle$. 
Conflict-freeness

Given $AF = \langle \text{Args}, \text{Att} \rangle$.

A set $S \subseteq \text{Args}$ of arguments is called conflict-free iff
Given $AF = \langle \text{Args}, \text{Att} \rangle$.

A set $S \subseteq \text{Args}$ of arguments is called conflict-free iff there is no $A, B \in S$ such that $(A, B) \in \text{Att}$.
Conflict-freeness

Given $AF = \langle \text{Args}, \text{Att} \rangle$.

A set $S \subseteq \text{Args}$ of arguments is called conflict-free iff there is no $A, B \in S$ such that $(A, B) \in \text{Att}$.

\[
\text{conflictFree} :: \text{Eq arg} \Rightarrow \text{DungAF arg} \rightarrow [\text{arg}] \rightarrow \text{Bool}
\]

\[
\text{conflictFree} (AF \_ att) s
\]

\[
= \text{null } [(a, b) | (a, b) \leftarrow \text{att}, a \in s, b \in s]
\]
State of implementation

Implementation is up to grounded semantics:
State of implementation

Implementation is up to **grounded semantics:**

- Available as a **Cabal package**;
Implementation is up to **grounded semantics**:

- Available as a **Cabal package**;
- Also documented as a **literate programming** paper;
Implementation is up to **grounded semantics**:

- Available as a [Cabal package](https://hackage.haskell.org/package);  
- Also documented as a literate programming paper;  
- Is currently used in the evaluation of NLP.
Outline

1 Introduction to argumentation
   A perceived problem
   A proposed solution

2 Implementing argumentation models using Haskell
   Dung’s AFs
   Carneades
   Translation from Carneades into Dung’s AFs

3 Verifying correctness of an implementation
   Implementation of properties
   Proving of properties

4 Further/future work and conclusions
Carneades

Carneades


- Structured argumentation
- Distinguishing feature: proof standards on a local level.
Pro and con arguments

Two types of arguments regarding a conclusion $c$:

- An argument with conclusion $c$ is called pro $c$.
- An argument for an opposite conclusion, $\neg c$, is called con $c$.

Aggregation of pro and con is done through proof standards.
Pro and con arguments

Two types of arguments regarding a conclusion $c$:

- An argument with conclusion $c$ is called \textit{pro} $c$, 

Two types of arguments regarding a conclusion $c$:

- An argument with conclusion $c$ is called pro $c$,
- An argument for an opposite conclusion, $\overline{c}$, is called con $c$.
Pro and con arguments

Two types of arguments regarding a conclusion $c$:

• An argument with conclusion $c$ is called $\text{pro } c$,
• An argument for an opposite conclusion, $\overline{c}$, is called $\text{con } c$.

Aggregation of $\text{pro}$ and $\text{con}$ is done through proof standards.
A propositional language \( \mathcal{L} \).
An argument \( \langle P, E, c \rangle \) has 3 parts:
A propositional language $\mathcal{L}$.
An argument $\langle P, E, c \rangle$ has 3 parts:

- premises, $P \subseteq \mathcal{L}$,
- exceptions, $E \subseteq \mathcal{L}$,
- conclusion, $c \in \mathcal{L}$.

All being propositional literals.
A **propositional language** \( \mathcal{L} \).

An argument \( \langle P, E, c \rangle \) has 3 parts:

- **premises**, \( P \subseteq \mathcal{L} \),
- **exceptions**, \( E \subseteq \mathcal{L} \),
- **conclusion**, \( c \in \mathcal{L} \).

All being **propositional literals**.

```haskell
type Proposition = (Bool, String)
data Argument = Arg [Proposition] [Proposition] Proposition
```
Arguments in Carneades consist of a two step inference:

- **Applicability** of an argument.
- **Acceptability** of the conclusion $c$. 
Applicability

```
Applicability

```

![Diagram showing the relationship between intent, kill, witness, unreliable, witness2, and unreliable2.]

- **intent**
- **kill**
- **witness**
- **unreliable**
- **witness2**
- **unreliable2**

Nodes labeled as $a_1$, $a_2$, and $a_3$ connect to these categories, indicating their applicability or relevance in the context.
An argument $\langle P, E, c \rangle$ is applicable in a CAES iff:

- $p \in P$ implies $p \in \text{assumptions}$ or $p \not\in \text{assumptions}$ and $p$ acceptable.
- $e \in E$ implies $e \not\in \text{assumptions}$ and $e \in \text{assumptions}$ or $e$ not acceptable.
Applicability of arguments

An argument $\langle P, E, c \rangle$ is applicable in a CAES iff:

- $p \in P$ implies $p \in \text{assumptions}$ or $[\overline{p} \notin \text{assumptions} \text{ and } p \text{ acceptable}]$. 
Applicability of arguments

An argument $\langle P, E, c \rangle$ is applicable in a CAES iff:

- $p \in P$ implies $p \in \text{assumptions}$ or $[\overline{p} \notin \text{assumptions}$ and $p$ acceptable].
- $e \in E$ implies $e \notin \text{assumptions}$ and $[\overline{e} \in \text{assumptions}$ or $e$ not acceptable].
Applicability in Haskell

\[
\text{applicable} :: \text{Argument} \rightarrow \text{CAES} \rightarrow \text{Bool} \\
\text{applicable} (\text{Arg} (\text{prems}, \text{excns}, _)) \\
\quad \text{caes}@(\text{CAES} (_, (\text{assumptions}, _), _)) \\
= \text{and} (\ [p \in \text{assumptions} \lor \\
\quad (\text{negate} \ p \notin \text{assumptions} \land \\
\quad \ p \text{ 'acceptable' caes}) \mid p \leftarrow \text{prems}])
\]
applicable :: Argument → CAES → Bool
applicable (Arg (prems, excns, _))
    caes@(CAES (_, (assumptions, _), _))
= and ( [p ∈ assumptions ∨
            (negate p ∉ assumptions ∧
             p 'acceptable' caes) | p ← prems ]
      ++
      [(e ∉ assumptions) ∧
       (negate e ∈ assumptions ∨
        ¬(e 'acceptable' caes)) | e ← excns ])

Applicability in Haskell
Acceptability

\[
\begin{align*}
\text{intent} & \quad \text{kill} \\
\text{a}_1 & \quad 0.8 \\
\text{murder} & \\
\text{witness} & \quad \text{unreliable} \\
\text{a}_2 & \quad 0.3 \\
\text{intent} & \\
\text{witness}_2 & \quad \text{unreliable}_2 \\
\text{a}_3 & \quad 0.3 \\
\sim\text{intent} & 
\end{align*}
\]
Acceptability in Haskell

Given a CAES $C = \langle \text{arguments, audience, standard} \rangle$.
A literal $p$ is acceptable in $C$ iff its proof standard returns $true$. 

Acceptability in Haskell

Given a CAES $C = \langle \text{arguments, audience, standard} \rangle$.
A literal $p$ is **acceptable** in $C$ iff its proof standard returns *true*.

\[
\begin{align*}
\textbf{type} & \quad \text{ProofStandard} = \text{Proposition} \rightarrow \text{CAES} \rightarrow \text{Bool} \\
\textbf{type} & \quad \text{AssignStandard} = \text{Proposition} \rightarrow \text{ProofStandard}
\end{align*}
\]
Acceptability in Haskell

Given a CAES $C = \langle \text{arguments}, \text{audience}, \text{standard} \rangle$. A literal $p$ is **acceptable** in $C$ iff its proof standard returns $true$.

```haskell
type ProofStandard = Proposition -> CAES -> Bool
type AssignStandard = Proposition -> ProofStandard

acceptable :: Proposition -> CAES -> Bool
acceptable p caes @
  (CAES (_, _, standard))
  = s p caes
  where s = standard p
```
Complete implementation and domain specific language for Gordon and Walton(2009):
State of implementation

Complete implementation and domain specific language for Gordon and Walton (2009):

- Available as a Cabal package;
State of implementation

Complete implementation and domain specific language for Gordon and Walton (2009):

- Available as a Cabal package;
- Also documented as a literate programming paper;
State of implementation

Complete implementation and domain specific language for Gordon and Walton (2009):

- Available as a Cabal package;
- Also documented as a literate programming paper;
- Is currently used in a university course in Edinburgh by Alan Smaill (students have to extend my implementation).
Outline

1 Introduction to argumentation
   A perceived problem
   A proposed solution

2 Implementing argumentation models using Haskell
   Dung’s AFs
   Carneades
   Translation from Carneades into Dung’s AFs

3 Verifying correctness of an implementation
   Implementation of properties
   Proving of properties

4 Further/future work and conclusions
Translation from Carneades into Dung’s AFs

My previous work has shown that:

• Carneades can be translated into ASPIC+
• which is known to generate AFs
• while keeping all important concepts - correspondence properties

However, given the complexity of ASPIC+, to more clearly demonstrate the translation and verification work:

• I derived a direct translation from Carneades into Dung,
• and developed an algorithm for generating the AFs.
Translation from Carneades into Dung’s AFs

My previous work has shown that:

- Carneades can be translated into ASPIC+
Translation from Carneades into Dung’s AFs

My previous work has shown that:

- Carneades can be translated into ASPIC+
- which is known to generate AFs
Translation from Carneades into Dung’s AFs

My previous work has shown that:

• Carneades can be translated into ASPIC+
• which is known to generate AFs
• while keeping all important concepts - correspondence properties
Translation from Carneades into Dung’s AFs

My previous work has shown that:

• Carneades can be translated into ASPIC+
• which is known to generate AFs
• while keeping all important concepts - correspondence properties

However, given the complexity of ASPIC+, to more clearly demonstrate the translation and verification work:
Translation from Carneades into Dung’s AFs

My previous work has shown that:
  • Carneades can be translated into ASPIC+
  • which is known to generate AFs
  • while keeping all important concepts - correspondence properties

However, given the complexity of ASPIC+, to more clearly demonstrate the translation and verification work:
  • I derived a direct translation from Carneades into Dung,
Translation from Carneades into Dung’s AFs

My previous work has shown that:

- Carneades can be translated into ASPIC+
- which is known to generate AFs
- while keeping all important concepts - correspondence properties

However, given the complexity of ASPIC+, to more clearly demonstrate the translation and verification work:

- I derived a direct translation from Carneades into Dung,
- and developed an algorithm for generating the AFs.
Translation from Carneades into Dung’s AFs in Haskell (1)

Short Haskell technicality:
Translation from Carneades into Dung’s AFs in Haskell (1)

Short Haskell technicality:

```
data Either a b = Left a | Right b
```
Translation from Carneades into Dung’s AFs in Haskell (1)

Short Haskell technicality:

```haskell
data Either a b = Left a | Right b
```

Carneades arguments in Dung using instantiation:
Translation from Carneades into Dung’s AFs in Haskell (1)

Short Haskell technicality:

```haskell
data Either a b = Left a | Right b
```

Carneades arguments in Dung using instantiation:

```haskell
type ConcreteArg = Either PropLiteral Argument
type ConcreteAF = DungAF ConcreteArg
```
Translation from Carneades into Dung’s AFs in Haskell (1)

Short Haskell technicality:

```haskell
data Either a b = Left a | Right b
```

Carneades arguments in Dung using instantiation:

```haskell
type ConcreteArg = Either PropLiteral Argument

type ConcreteAF = DungAF ConcreteArg
```

For efficiency we keep track of the status some of the arguments.
Translation from Carneades into Dung’s AFs in Haskell (1)

Short Haskell technicality:

```haskell
data Either a b = Left a | Right b
```

Carneades arguments in Dung using instantiation:

```haskell
type ConcreteArg = Either PropLiteral Argument

type ConcreteAF = DungAF ConcreteArg
```

For efficiency we keep track of the status some of the arguments. Labelled version:
Translation from Carneades into Dung’s AFs in Haskell (1)

Short Haskell technicality:

```haskell
data Either a b = Left a | Right b
```

Carneades arguments in Dung using instantiation:

```haskell
type ConcreteArg = Either PropLiteral Argument

type ConcreteAF = DungAF ConcreteArg
```

For efficiency we keep track of the status some of the arguments. Labelled version:

```haskell
type LConcreteArg = (Bool, ConcreteArg)

type LConcreteAF = DungAF LConcreteArg
```
Translation from Carneades into Dung’s AFs in Haskell (2)

Just a flavour. For translation of assumptions:
Translation from Carneades into Dung’s AFs in Haskell (2)

Just a flavour. For translation of assumptions:

\[ propToLArg :: PropLiteral \to ConcreteArg \]
\[ propToLArg \ p = \text{Left } p \]
The main translation function:
Translation from Carneades into Dung’s AFs in Haskell (3)

The main translation function:

\[
\text{translate} :: \text{CAES} \rightarrow \text{ConcreteAF} \\
\text{translate caes}(\text{CAES}(\text{argSet},(\text{assumptions},_),_)) = \text{AF}(\text{map snd args})(\text{map stripAttack attacks})
\]

where

\[
\text{AF args attacks} = \\
\text{argsToAF} \\
(\text{topSort argSet}) \text{ caes} \\
(\text{AF (defeater : map propToLArg assumptions)}) []
\]

- \text{topSort} topologically sorts the dependency graph,
Translation from Carneades into Dung’s AFs in Haskell (3)

The main translation function:

```haskell
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)
        [])
```

- `topSort` topologically sorts the dependency graph,
- `defeater` is the only administrative node, used for exceptions,
Translation from Carneades into Dung’s AFs in Haskell (3)

The main translation function:

\[
\text{translate} :: \text{CAES} \rightarrow \text{ConcreteAF}
\]

\[
\text{translate caes@}(\text{CAES (argSet, (assumptions, _), _)}) = \text{AF (map snd args) (map stripAttack attacks)}
\]

where

\[
\text{AF args attacks} = \text{argsToAF (topSort argSet) caes (AF (defeater : map propToLArg assumptions)} [[])
\]

- \text{topSort}\  \text{topologically sorts the dependency graph},
- \text{defeater} \  \text{is the only administrative node, used for exceptions},
- \text{argsToAF} \  \text{translates the arguments}.
Outline

1. Introduction to argumentation
   - A perceived problem
   - A proposed solution

2. Implementing argumentation models using Haskell
   - Dung’s AFs
   - Carneades
   - Translation from Carneades into Dung’s AFs

3. Verifying correctness of an implementation
   - Implementation of properties
   - Proving of properties

4. Further/future work and conclusions
Outline

1. Introduction to argumentation
   A perceived problem
   A proposed solution

2. Implementing argumentation models using Haskell
   Dung’s AFs
   Carneades
   Translation from Carneades into Dung’s AFs

3. Verifying correctness of an implementation
   Implementation of properties
   Proving of properties

4. Further/future work and conclusions
Correspondence properties

Correspondence properties keep desired properties, such as:

```haskell
corApp :: CAES → Bool
corApp caes @ (CAES (argSet, , , )) =
  let
    transCAES = translate caes
    appArgs = filter ('applicable' caes)
      (getAllArgs argSet)
    transArgs = stripRight (groundedExt transCAES)
  in
    fromList appArgs ≡ fromList transArgs
```

True
Correspondence properties

Correspondence properties keep desired properties, such as:

• **Applicability** of arguments
Correspondence properties

Correspondence properties keep desired properties, such as:

- **Applicability** of arguments
- **Acceptability** of conclusions
Correspondence properties

Correspondence properties keep desired properties, such as:

- Applicability of arguments
- Acceptability of conclusions

Applicability of arguments in Haskell:
Correspondence properties

Correspondence properties keep desired properties, such as:

- Applicability of arguments
- Acceptability of conclusions

**Applicability** of arguments in Haskell:

```haskell
corApp :: CAES → Bool
corApp caes@(CAES (argSet, _, _)) =
  let transCAES = translate caes
      appArgs = filter ('applicable' caes)
                    (getAllArgs argSet)
      transArgs = stripRight (groundedExt transCAES)
    in fromList appArgs ≡ fromList transArgs
```
Correspondence properties

Correspondence properties keep desired properties, such as:

- Applicability of arguments
- Acceptability of conclusions

Applicability of arguments in Haskell:

```haskell
corApp :: CAES -> Bool
corApp caes@(CAES (argSet, _, _)) =
  let transCAES = translate caes
      appArgs = filter ('applicable' caes)
                 (getAllArgs argSet)
      transArgs = stripRight (groundedExt transCAES)
      in fromList appArgs == fromList transArgs
```

```
> corApp caes
True
```
Outline

1 Introduction to argumentation
   A perceived problem
   A proposed solution

2 Implementing argumentation models using Haskell
   Dung’s AFs
   Carneades
   Translation from Carneades into Dung’s AFs

3 Verifying correctness of an implementation
   Implementation of properties
   Proving of properties

4 Further/future work and conclusions
Dung’s model formalised in a theorem prover (1)

Formalised Dung’s AFs in a theorem prover: Formalisation written in Agda, a dependently typed functional programming language, similar to Haskell:
Dung’s model formalised in a theorem prover (1)

Formalised Dung’s AFs in a theorem prover: Formalisation written in Agda, a dependently typed functional programming language, similar to Haskell:

• **Formalised** the same set of functions and data types in Agda,

• **Given a finite AF, proved** termination, existence and uniqueness of grounded labelling,
Dung’s model formalised in a theorem prover (2)

Type of the grounded labelling in Haskell:
Dung’s model formalised in a theorem prover (2)

Type of the grounded labelling in Haskell:

$$grounded' :: Eq a \Rightarrow [a] \rightarrow [a] \rightarrow [a] \rightarrow DungAF a \rightarrow [(a, Status)]$$
Dung’s model formalised in a theorem prover (2)

Type of the grounded labelling in Haskell:

\[
\text{grounded' :: Eq } a \Rightarrow [a] \rightarrow [a] \rightarrow [a] \rightarrow \text{DungAF } a \rightarrow [(a, Status)]
\]

Corresponding Agda type:
Dung’s model formalised in a theorem prover (2)

Type of the grounded labelling in Haskell:

\[
grounded' :: \text{Eq } a \Rightarrow [a] \rightarrow [a] \rightarrow [a] \rightarrow \text{DungAF } a \rightarrow [(a, \text{Status})]
\]

Corresponding Agda type:

\[
groundedList : \{A : \text{Set}\} \rightarrow (A \rightarrow A \rightarrow \text{Bool}) \rightarrow
\text{List } A \rightarrow \text{List } A \rightarrow \text{List } A \rightarrow
\text{DungAF } A \rightarrow \text{List } (A \times \text{Status})
\]
Dung’s model formalised in a theorem prover (3)

Again:
Dung’s model formalised in a theorem prover (3)

Again:

\[
groundedList : \{ A : Set \} \to (A \to A \to \text{Bool}) \to \\
\quad List A \to List A \to List A \to \\
\quad DungAF A \to List (A \times \text{Status})
\]
Dung’s model formalised in a theorem prover (3)

Again:

\[
groundedList : \{ A : \text{Set} \} \rightarrow (A \rightarrow A \rightarrow \text{Bool}) \rightarrow \\
List A \rightarrow List A \rightarrow List A \rightarrow \\
DungAF A \rightarrow List (A \times \text{Status})
\]

Terminating function:
Dung’s model formalised in a theorem prover (3)

Again:

\[
groundedList : \{ A : Set \} \rightarrow (A \rightarrow A \rightarrow \text{Bool}) \rightarrow \\
                List A \rightarrow List A \rightarrow List A \rightarrow \\
                DungAF A \rightarrow List (A \times \text{Status})
\]

Terminating function:

\[
grounded' : \{ A : Set \} \rightarrow \{ m n o : \mathbb{N} \} \rightarrow \\
        (\sum \mathbb{N} \lambda k \rightarrow k \equiv o) \rightarrow (A \rightarrow A \rightarrow \text{Bool}) \rightarrow \\
        \text{Vec} A m \rightarrow \text{Vec} A n \rightarrow \text{Vec} A o \rightarrow \\
        DungAF A \rightarrow \text{Vec} (A \times \text{Status}) \\
        (m + n + o)
\]
Introduction to argumentation
   A perceived problem
   A proposed solution

Implementing argumentation models using Haskell
   Dung’s AFs
   Carneades
   Translation from Carneades into Dung’s AFs

Verifying correctness of an implementation
   Implementation of properties
   Proving of properties

Further/future work and conclusions
Overview of work discussed (1)

• Large parts of Dung’s definitions have been implemented in Haskell,
Overview of work discussed (1)

• Large parts of Dung’s definitions have been implemented in Haskell,
• High-level code close to the mathematical definitions:
Overview of work discussed (1)

• Large parts of Dung’s definitions have been implemented in Haskell,
• High-level code close to the mathematical definitions:
  • Allowing greater understanding of the implementation,
Overview of work discussed (1)

• Large parts of Dung’s definitions have been implemented in Haskell,

• High-level code close to the mathematical definitions:
  • Allowing greater understanding of the implementation,
  • Easier realisation of existing/future translations,
Overview of work discussed (1)

- Large parts of Dung’s definitions have been implemented in Haskell,
- High-level code close to the mathematical definitions:
  - Allowing greater understanding of the implementation,
  - Easier realisation of existing/future translations,
  - Written in a notation closely related to the actual use.
- Implemented Carneades in Haskell,
Overview of work discussed (1)

• Large parts of Dung’s definitions have been implemented in Haskell,
• High-level code close to the mathematical definitions:
  • Allowing greater understanding of the implementation,
  • Easier realisation of existing/future translations,
  • Written in a notation closely related to the actual use.
• Implemented Carneades in Haskell,
• Implemented a translation from Carneades to Dung in Haskell and implemented correspondence properties,
Overview of work discussed (1)

• Large parts of Dung’s definitions have been implemented in Haskell,

• High-level code close to the mathematical definitions:
  • Allowing greater understanding of the implementation,
  • Easier realisation of existing/future translations,
  • Written in a notation closely related to the actual use.

• Implemented Carneades in Haskell,

• Implemented a translation from Carneades to Dung in Haskell and implemented correspondence properties,

• Formalisation of the Dung implementation into a theorem prover, Agda:
Overview of work discussed (1)

- Large parts of Dung’s definitions have been implemented in Haskell,
- High-level code close to the mathematical definitions:
  - Allowing greater understanding of the implementation,
  - Easier realisation of existing/future translations,
  - Written in a notation closely related to the actual use.
- Implemented Carneades in Haskell,
- Implemented a translation from Carneades to Dung in Haskell and implemented correspondence properties,
- Formalisation of the Dung implementation into a theorem prover, Agda:
  - Easier formalisation of existing/future translations,
Overview of work discussed (1)

• Large parts of Dung’s definitions have been implemented in Haskell,
• High-level code close to the mathematical definitions:
  • Allowing greater understanding of the implementation,
  • Easier realisation of existing/future translations,
  • Written in a notation closely related to the actual use.
• Implemented Carneades in Haskell,
• Implemented a translation from Carneades to Dung in Haskell and implemented correspondence properties,
• Formalisation of the Dung implementation into a theorem prover, Agda:
  • Easier formalisation of existing/future translations,
  • A better understanding of the meaning of some of the complexer argumentation models.
Overview of work discussed (2)

- All code is available as literate Haskell/Agda,
Overview of work discussed (2)

- All code is available as literate Haskell/Agda,
  - Paper and even the slides can be loaded into the compiler
Overview of work discussed (2)

- All code is available as literate Haskell/Agda,
  - Paper and even the slides can be loaded into the compiler
- Cabalised and uploaded the Dung implementation to Hackage,
Overview of work discussed (2)

• All code is available as literate Haskell/Agda,
  • Paper and even the slides can be loaded into the compiler
• Cabalised and uploaded the Dung implementation to Hackage,
• Cabalised and uploaded the Carneades implementation to Hackage,
Overview of work discussed (2)

• All code is available as literate Haskell/Agda,
  • Paper and even the slides can be loaded into the compiler
• Cabalised and uploaded the Dung implementation to Hackage,
• Cabalised and uploaded the Carneades implementation to Hackage,
• Cabalised and uploaded the translation from Carneades into Dung implementation to Hackage,
Overview of work discussed (2)

- All code is available as literate Haskell/Agda,
  - Paper and even the slides can be loaded into the compiler
- Cabalised and uploaded the Dung implementation to Hackage,
- Cabalised and uploaded the Carneades implementation to Hackage,
- Cabalised and uploaded the translation from Carneades into Dung implementation to Hackage,
- Installation instructions (hopefully) usable for non-experienced programmers.
Further work done

I have also done further work related to previously discussed work:
Further work done

I have also done further work related to previously discussed work:

• Implementation of propositional ASPIC+,
Further work done

I have also done further work related to previously discussed work:

• Implementation of propositional ASPIC+,
• Generalisation of ASPIC+ to weights, weight propagation and accrual, also based on the Logic of Argumentation,
Further work done

I have also done further work related to previously discussed work:

• Implementation of propositional ASPIC+,
• Generalisation of ASPIC+ to weights, weight propagation and accrual, also based on the Logic of Argumentation,
• An implementation of the above.
Future work (1)

• Implement further set of semantics for Dung’s AFs,
Future work (1)

- **Implement** further set of semantics for Dung’s AFs,
- **Formalisation** of Carneades’ definitions,
Future work (1)

• **Implement** further set of semantics for Dung’s AFs,
• **Formalisation** of Carneades’ definitions,
• **Further formalisation** of Dung’s definitions and theorems:
Future work (1)

- Implement further set of semantics for Dung’s AFs,
- Formalisation of Carneades’ definitions,
- Further formalisation of Dung’s definitions and theorems:
- Formalisation of the translation from Carneades to Dung.
Future work (1)

• **Implement** further set of semantics for Dung’s AFs,
• **Formalisation** of Carneades’ definitions,
• **Further formalisation** of Dung’s definitions and theorems:
  • **Formalisation** of the translation from Carneades to Dung.
• **Connect the implementation** of Dung’s AFs to an **optimised** implementation using ASP or SAT
Future work (2)

• Finish up the implementation of propositional ASPIC+,
Future work (2)

- Finish up the **implementation** of propositional ASPIC+,
- Finish up the **generalisation** of ASPIC+:
Future work (2)

- Finish up the implementation of propositional ASPIC+,
- Finish up the generalisation of ASPIC+:
  - Finish up the implementation of it:
Future work (2)

- Finish up the implementation of propositional ASPIC+,
- Finish up the generalisation of ASPIC+:
  - Finish up the implementation of it:
  - Possibly prove some accrual principles.