Relating, implementing and formalising argumentation models using the Curry-Howard correspondence and other functional techniques

Bas van Gijzel

University of Nottingham

October 22, 2014
ISIS talk
Outline

1 Motivation of the methodology
   Overview of the specifications/implementations/formalisations

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

3 Verifying correctness of an implementation
   Verification using FP
   Theorem proving

4 Conclusions and future work
Outline

1 Motivation of the methodology
   - Overview of the specifications/implementations/formalisations

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

3 Verifying correctness of an implementation
   - Verification using FP
   - Theorem proving

4 Conclusions and future work
Argumentation

An example argument in the legal domain:

```
intent kill
a1 murder
```
An example argument in the legal domain:

\[
\begin{array}{c}
\text{intent} \\
\text{kill} \\
\end{array}
\rightarrow (a_1)
\begin{array}{c}
murder
\end{array}
\]

Or alternatively:

\[
\frac{\text{intent}}{\text{kill}} \quad (a_1)
\]

\[
\frac{\text{murder}}{}
\]
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
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:
How to implement an argumentation model

Two main ways to implement an argumentation model:
• Directly implement it into your favourite programming language;
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).
Implementations of abstract models

Status of implementations for abstract models, e.g. Dung’s AFs:

- A decent amount of well-documented and open source applications.
- Recent efforts to optimise the evaluation of AFs/ADFs using:
  - SAT-solvers
  - Answer-set programming
- A decent amount of other abstract models have been implemented through encodings into AFs. For instance ASPARTIX, DIAMOND and ArgSemSAT:
  - See: http://www.dbai.tuwien.ac.at/research/project/argumentation/systempage/
  - https://isysrv.informatik.uni-leipzig.de/diamond
  - http://tiny.cc/argsemsat
Implementations of abstract models

Status of implementations for abstract models, e.g. Dung’s AFs:

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

Status of implementations for abstract models, e.g. Dung’s AFs:

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

  • SAT-solvers
  • Answer-set programming

A decent amount of other abstract models have been implemented through encodings into AFs. For instance ASPARTIX, DIAMOND and ArgSemSAT:

See:
http://www.dbai.tuwien.ac.at/research/project/argumentation/systempage/
https://isysrv.informatik.uni-leipzig.de/diamond/
http://tiny.cc/argsemsat
Implementations of abstract models

Status of implementations for abstract models, e.g. Dung’s AFs:

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

Status of implementations for abstract models, e.g. Dung’s AFs:

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

Status of implementations for abstract models, e.g. Dung’s AFs:

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

Status of implementations for abstract models, e.g. Dung’s AFs:

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

For instance ASPARTIX, DIAMOND and ArgSemSAT:

See:
http://www.dbai.tuwien.ac.at/research/project/argumentation/systempage/
https://isysrv.informatik.uni-leipzig.de/diamond
http://tiny.cc/argsemsat
Implementations of structured models

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

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

Status of implementations for structured models: ASPIC

The same holds for various other models/projects.
Implementations of structured models

• Significant amount of implementations are unavailable and closed source: ASPIC (EU), ArguGrid, and many more...

Implementations of structured models

• Significant amount of implementations are unavailable and closed source: ASPIC (EU), ArguGrid, and many more...
  • Effort spent is practically lost,
Implementations of structured models

- Significant amount of implementations are unavailable and closed source: ASPIC (EU), ArguGrid, and many more...
  - Effort spent is practically lost,
  - Implementation techniques are non-reproducible.
Implementations of structured models

• Significant amount of implementations are unavailable and closed source: ASPIC (EU), ArguGrid, and many more...
  • Effort spent is practically lost,
  • Implementation techniques are non-reproducible.
• A few mature implementations (Carneades, TOAST, etc.), but:
Implementations of structured models

• Significant amount of implementations are unavailable and closed source: ASPIC (EU), ArguGrid, and many more...
  • Effort spent is practically lost,
  • Implementation techniques are non-reproducible.
• A few mature implementations (Carneades, TOAST, etc.), but:
  • Again often closed source,
Implementations of structured models

• Significant amount of implementations are unavailable and closed source: ASPIC (EU), ArguGrid, and many more...
  • Effort spent is practically lost,
  • Implementation techniques are non-reproducible.

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

• Significant amount of implementations are unavailable and closed source: ASPIC (EU), ArguGrid, and many more...
  • Effort spent is practically lost,
  • Implementation techniques are non-reproducible.
• A few mature implementations (Carneades, TOAST, etc.), but:
  • Again often closed source,
  • Often not directly related to the actual mathematical model.

See:
http://carneades.github.io/carneades/Carneades/
http://www.arg.dundee.ac.uk/toast/
Implementations of translations

Implemented translations are even more rare:

• Situation is improving for abstract argumentation (Sylwia Polberg and others);
• For existing translations from structured models to AFs, however again a lack of implementations;
• Additionally:
  • Translations are complex and relatively ad-hoc
  • Proofs of correctness are complex
Implementations of translations

Implementations of translations are even more rare:

• Situation is improving for abstract argumentation (Sylwia Polberg and others);
Implementations of translations

Implementations of translations are even more rare:

• Situation is improving for abstract argumentation (Sylwia Polberg and others);
• For existing translations from structured models to AFs, however again a lack of implementations;
Implementations of translations

Implemented translations are even more rare:

• Situation is improving for abstract argumentation (Sylwia Polberg and others);
• For existing translations from structured models to AFs, however again a lack of implementations;
• Additionally:
Implementations of translations

Implementations of translations are even more rare:

- Situation is improving for abstract argumentation (Sylwia Polberg and others);
- For existing translations from structured models to AFs, however again a lack of implementations;
- Additionally:
  - Translations are complex and relatively ad-hoc
Implementations of translations

Implemented translations are even more rare:

• Situation is improving for abstract argumentation (Sylwia Polberg and others);
• For existing translations from structured models to AFs, however again a lack of implementations;
• Additionally:
  • Translations are complex and relatively ad-hoc
  • Proofs of correctness are complex
Problem statement

We need:

- publicly available and reproducible implementations/implementation methods;
Problem statement

We need:

• publicly available and reproducible implementations/implementation methods;
• further verification or even complete mechanical formalisation of translations/proofs
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)
A principled approach to solving this problem (1)

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:
A principled approach to solving this problem (1)

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 express more **general mathematics**

• **Data structures**
A principled approach to solving this problem (1)

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 express more general mathematics
- Data structures
- Able to easily verify or prove properties
A principled approach to solving this problem (1)

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 express more **general mathematics**
- **Data structures**
- Able to easily **verify** or **prove properties**

My suggestion: **functional programming**, in specific Haskell/Agda.
A principled approach to solving this problem (2)

• Provide implementation of Dung’s AFs and some structured models (Carneades, ASPIC⁺) in Haskell
A principled approach to solving this problem (2)

- Provide implementation of Dung’s AFs and some structured models (Carneades, ASPIC+) in Haskell
  - In a tutorial-like fashion;
A principled approach to solving this problem (2)

• Provide implementation of Dung’s AFs and some structured models (Carneades, ASPIC⁺) in Haskell
  • In a tutorial-like fashion;
  • Close to the actual mathematical definitions;
A principled approach to solving this problem (2)

- Provide implementation of Dung’s AFs and some structured models (Carneades, ASPIC⁺) in Haskell
  - In a tutorial-like fashion;
  - Close to the actual mathematical definitions;
  - With output usable by other existing efficient implementations.
A principled approach to solving this problem (2)

• Provide implementation of Dung’s AFs and some structured models (Carneades, ASPIC$^+$) in Haskell
  • In a tutorial-like fashion;
  • Close to the actual mathematical definitions;
  • With output usable by other existing efficient implementations.

• In the same fashion: implement a translation
A principled approach to solving this problem (2)

• Provide implementation of Dung’s AFs and some structured models (Carneades, ASPIC⁺) in Haskell
  • In a tutorial-like fashion;
  • Close to the actual mathematical definitions;
  • With output usable by other existing efficient implementations.

• In the same fashion: implement a translation
• Provide quick verification by implementation of properties
A principled approach to solving this problem (2)

- Provide implementation of Dung’s AFs and some structured models (Carneades, ASPIC$^+$) in Haskell
  - In a tutorial-like fashion;
  - Close to the actual mathematical definitions;
  - With output usable by other existing efficient implementations.

- In the same fashion: implement a translation
- Provide quick verification by implementation of properties

Result: a methodology that allows for quick and clean implementing and initial testing of properties.
A principled approach to solving this problem (3)

Additionally:

• Provide mechanical formalisation of implementations and translation, using the theorem prover, Agda;
A principled approach to solving this problem (3)

Additionally:

• Provide mechanical formalisation of implementations and translation, using the theorem prover, Agda;
• Using a theorem prover based on the Curry-Howard correspondence:
A principled approach to solving this problem (3)

Additionally:

• Provide **mechanical formalisation** of implementations and translation, using the **theorem prover, Agda**;

• Using a **theorem prover** based on the **Curry-Howard correspondence**:
  • **Types** with accompanying **implementations(functions)**, correspond to **theorems** with accompanying **proofs**;
A principled approach to solving this problem (3)

Additionally:

• Provide mechanical formalisation of implementations and translation, using the theorem prover, Agda;
• Using a theorem prover based on the Curry-Howard correspondence:
  • Types with accompanying implementations (functions), correspond to theorems with accompanying proofs;
  • Meaning we get a mechanically verified formalisation and implementation in one.
A principled approach to solving this problem (3)

Additionally:

• Provide mechanical formalisation of implementations and translation, using the theorem prover, Agda;

• Using a theorem prover based on the Curry-Howard correspondence:
  • Types with accompanying implementations(functions), correspond to theorems with accompanying proofs;
  • Meaning we get a mechanically verified formalisation and implementation in one.

Result: a verified pipeline to translate models to an efficiently implemented model.
A principled approach to solving this problem (4)

Additionally:

\[\text{See } \text{http://www.cs.nott.ac.uk/~bmv/COMMA/}\]
A principled approach to solving this problem (4)

Additionally:

• All Haskell code will or has been published on Hackage/GitHub under an open source license, with:

\[\text{1} \text{See } \text{http://www.cs.nott.ac.uk/~bm/COMMA/}\]
A principled approach to solving this problem (4)

Additionally:

• All Haskell code will or has been published on Hackage/GitHub under an open source license, with:
  • accompanying installation instructions;
  • elaborate examples;
  • and appropriate documentation.

1See http://www.cs.nott.ac.uk/~bmv/COMMA/
A principled approach to solving this problem (4)

Additionally:

- All Haskell code will or has been published on Hackage/GitHub under an open source license, with:
  - accompanying installation instructions;
  - elaborate examples;
  - and appropriate documentation.

- Proofs in Agda are open source and publicly available.

---

1 See [http://www.cs.nott.ac.uk/~bmv/COMMA/](http://www.cs.nott.ac.uk/~bmv/COMMA/)
A principled approach to solving this problem (4)

Additionally:

• All Haskell code will or has been published on Hackage/GitHub under an open source license, with:
  • accompanying installation instructions;
  • elaborate examples;
  • and appropriate documentation.

• Proofs in Agda are open source and publicly available

I hope this helps to tackle the problem of unavailable implementations and lost programming methodology.¹

¹See http://www.cs.nott.ac.uk/~bmv/COMMA/
Outline

1 Motivation of the methodology
   Overview of the specifications/implementations/formalisations

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

3 Verifying correctness of an implementation
   Verification using FP
   Theorem proving

4 Conclusions and future work
Schematic overview of the work done

(1)
Schematic overview of the work done

Dung’s AFs

Generalised ASPIC+

Formalisation

Formalised Dung’s AFs

Translation
1. Motivation of the methodology
   Overview of the specifications/implementations/formalisations

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

3. Verifying correctness of an implementation
   Verification using FP
   Theorem proving

4. Conclusions and future work
Outline

1. Motivation of the methodology
   Overview of the specifications/implementations/formalisations

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

3. Verifying correctness of an implementation
   Verification using FP
   Theorem proving

4. Conclusions and future work
An abstract argumentation framework (AF) is a tuple $AF = \langle \text{Args}, \text{Att} \rangle$ such that:
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,
- $\text{Att} \subseteq \text{Args} \times \text{Args}$.
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.
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.

\[ A \rightarrow B \rightarrow C \]
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)]
  deriving (Show)
```

Considering arguments as Strings:

```haskell
type AbsArg = String

A -> B -> 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)]
deriving (Show)
```

Considering arguments as Strings:

```haskell
a, b, c :: AbsArg
a = "A"
b = "B"
c = "C"
AF_1 :: DungAF AbsArg
AF_1 = AF [a, b, c] [(a, b), (b, c)]
```
AFs in Haskell

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

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

Considering arguments as Strings:

\[
\text{type } \text{AbsArg} = \text{String}
\]
AFs in Haskell

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

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

Considering arguments as Strings:

```haskell
type AbsArg = String
```

![Diagram](A \longrightarrow B \longrightarrow C)
Given \( AF = \langle \text{Args}, \text{Att} \rangle \)

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

Considering arguments as \text{Strings} as \text{Strings}:

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

And in Haskell:

\[
a, b, c :: \text{AbsArg} \\
a = "A" \\
b = "B" \\
c = "C" \\
AF_1 :: \text{DungAF} \; \text{AbsArg} \\
AF_1 = AF \; [a, b, c] \; [(a, b), (b, c)]
\]
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 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 $AF = \langle \text{Args}, \text{Att} \rangle$.

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 ]
```
Attacking with a set of arguments

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

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

In Haskell:

\[
\begin{align*}
\text{setAttacks} & \:: \text{Eq arg} \Rightarrow \text{DungAF arg} \rightarrow [\text{arg}] \rightarrow \\
& \quad \text{arg} \rightarrow \text{Bool} \\
\text{setAttacks} \ (\text{AF } \text{– att} ) \ \text{args} \ \text{arg} \\
& = \text{or} \ [b \equiv \text{arg} \mid (a, b) \leftarrow \text{att}, a \in \text{args}]
\end{align*}
\]

Note that by the required \( \text{Eq arg} \Rightarrow \), Haskell forces us to see that we need an equality on arguments to be able implement these functions.
Conflict-freeness

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

Given $AF = \langle Args, Att \rangle$.

A set $S \subseteq Args$ of arguments is called conflict-free iff
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 \( (A, B) \in \text{Att} \).
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 $(A, B) \in \text{Att}$.

\[
\text{conflictFree} :: \text{Eq arg} \Rightarrow \text{DungAF arg} \rightarrow [\text{arg}] \rightarrow \text{Bool}
\]
\[
\text{conflictFree} (AF \_ \att) s = \text{null} \ [(a, b) | (a, b) \leftarrow \att, a \in s, b \in s]
\]
Overview of the implementation

The complete implementation includes: £

£See http://www.cs.nott.ac.uk/~bmv/Dung/
Overview of the implementation

The complete implementation includes: ²

- The four **standard semantics**: grounded, complete, preferred, stable;

Overview of the implementation

The complete implementation includes: ²

• The four **standard semantics**: grounded, complete, preferred, stable;

• **Semi-stable semantics**;

Overview of the implementation

The complete implementation includes: ²

• The four **standard semantics**: grounded, complete, preferred, stable;

• **Semi-stable** semantics;

• Command-line application allowing **parsing** and **output** to a standard format, to use an external **efficient** implementation;

²See http://www.cs.nott.ac.uk/~bmv/Dung/
Overview of the implementation

The complete implementation includes: ²

• The four standard semantics: grounded, complete, preferred, stable;
• Semi-stable semantics;
• Command-line application allowing parsing and output to a standard format, to use an external efficient implementation;
• Various implementations of formal properties, allowing verification by using Quickcheck.

²See http://www.cs.nott.ac.uk/~bmv/Dung/
Outline

1. Motivation of the methodology
   Overview of the specifications/implementations/formalisations

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

3. Verifying correctness of an implementation
   Verification using FP
   Theorem proving

4. Conclusions and future work

- Structured argumentation
- Distinguishing feature: proof standards on a local level.
Two types of arguments regarding a conclusion $c$: 

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

Aggregation of pro and con is done through proof standards.
Two types of arguments regarding a conclusion $c$:

- An argument with conclusion $c$ is called **pro $c$**, 

**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, $\overline{c}$, is called **con $c$**.
Pro and con arguments

Two types of arguments regarding a conclusion $c$:

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

Aggregation of pro and con is done through proof standards.
Arguments in Carneades in Haskell

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, $P \subset \mathcal{L}$,
- exceptions, $E \subset \mathcal{L}$,
- conclusion, $c \in \mathcal{L}$. 
A propositional language $\mathcal{L}$. An argument $\langle P, E, c \rangle$ has 3 parts:

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

```haskell
type Proposition = (Boolean, 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

- intent
- kill
- witness
- unreliable
- witness2
- unreliable2
An argument $\langle P, E, c \rangle$ is applicable in a CAES iff

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

Applicability of arguments
Applicability in Haskell

\[
\text{applicable} :: \text{Argument} \to \text{CAES} \to \text{Bool}
\]

\[
\text{applicable} \left( \text{Arg} \left( \text{prems}, \text{excns}, \_ \right) \right)
\]

\[
\text{caes} @ (\text{CAES} (\_, (\text{assumptions}, \_), \_))
\]

\[
= \text{and} \left[ p \in \text{assumptions} \lor 
\right.
\]

\[
\left( \text{negate} \ p \not\in \text{assumptions} \land
\right.
\]

\[
p \ '\text{acceptable'} \ caes \right) | p \leftarrow \text{prems} \left] \right.
\]

\[
++
\]

\[
\left[ (e \not\in \text{assumptions}) \land
\right.
\]

\[
\left( \text{negate} \ e \in \text{assumptions} \lor
\right.
\]

\[
\neg \left( e \ '\text{acceptable'} \ caes \right) \right) | e \leftarrow \text{excns} \right]
\]
Acceptability
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 $\text{true}$.

\[
\text{type } \text{ProofStandard} = \text{Proposition} \rightarrow \text{CAES} \rightarrow \text{Bool}
\]
\[
\text{type } \text{AssignStandard} = \text{Proposition} \rightarrow \text{ProofStandard}
\]
Acceptability in Haskell

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

\begin{align*}
\text{type} & \quad \text{ProofStandard} = \text{Proposition} \rightarrow \text{CAES} \rightarrow \text{Bool} \\
\text{type} & \quad \text{AssignStandard} = \text{Proposition} \rightarrow \text{ProofStandard}
\end{align*}

\begin{align*}
\text{acceptable} :: \text{Proposition} \rightarrow \text{CAES} \rightarrow \text{Bool} \\
\text{acceptable } p & \text{ caes@} \\
(\text{CAES } (\_ , \_ , \text{standard})) \\
& = s \ p \ \text{caes} \\
\text{where} & \quad s = \text{standard } p
\end{align*}
State of implementation

Complete implementation and domain specific language for Gordon and Walton (2009): ³

³See http://www.cs.nott.ac.uk/~bmv/CarneadesDSL/
State of implementation

Complete implementation and domain specific language for Gordon and Walton (2009): ³

• Available as a Cabal package;

³See http://www.cs.nott.ac.uk/~bmv/CarneadesDSL/
State of implementation

Complete implementation and domain specific language for Gordon and Walton (2009): ³

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

³See http://www.cs.nott.ac.uk/~bmv/CarneadesDSL/
State of implementation

Complete implementation and domain specific language for Gordon and Walton (2009): ³

- 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);

³See http://www.cs.nott.ac.uk/~bmv/CarneadesDSL/
State of implementation

Complete implementation and domain specific language for Gordon and Walton (2009): ³

- 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);
- Mechanical formalisation in Agda is in progress (with Tom Gordon) and going well.

³See http://www.cs.nott.ac.uk/~bmv/CarneadesDSL/
Outline

1 Motivation of the methodology
   Overview of the specifications/implementations/formalisations

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

3 Verifying correctness of an implementation
   Verification using FP
   Theorem proving

4 Conclusions and future work
My previous work has shown that:

- Carneades can be translated into ASPIC$^+$

Translation from Carneades into Dung’s 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
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+ (especially for formalisation), 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+ (especially for formalisation), 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+ (especially for formalisation), 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:

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

Short Haskell technicality:

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

Carneades arguments in AFs 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 AFs 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 AFs using instantiation:

```haskell
type ConcreteArg = Either PropLiteral Argument
type ConcreteAF = DungAF ConcreteArg
```

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

Short Haskell technicality:

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

Carneades arguments in AFs using instantiation:

```
type ConcreteArg = Either PropLiteral Argument

type ConcreteAF = DungAF ConcreteArg
```

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

```
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 (axioms):
Translation from Carneades into Dung’s AFs in Haskell (2)

Just a flavour. For translation of assumptions (axioms):

\[ propToLArg :: PropLiteral \to LConcreteArg \]

\[ propToLArg \ p = (True, \text{Left } p) \]
Translation from Carneades into Dung’s AFs in Haskell (3)

The formulation of Carneades hides some subtleties:
Translation from Carneades into Dung’s AFs in Haskell (3)

The formulation of Carneades hides some subtleties:

4 different things:

• Carneades arguments and their implementation;
Translation from Carneades into Dung’s AFs in Haskell (3)

The formulation of Carneades hides some subtleties:

4 different things:

• Carneades arguments and their implementation;
• Carneades argument set and its implementation;
Translation from Carneades into Dung’s AFs in Haskell (3)

The formulation of Carneades hides some subtleties:

4 different things:
- Carneades arguments and their implementation;
- Carneades argument set and its implementation;
- Carneades dependency graph and its implementation;
Translation from Carneades into Dung’s AFs in Haskell (3)

The formulation of Carneades hides some **subtleties**:

4 different things:

- Carneades arguments and their implementation;
- Carneades argument set and its implementation;
- Carneades dependency graph and its implementation;
- Resulting Dung AF and its implementation.
Translation from Carneades into Dung’s AFs in Haskell (4)

The main translation function:
Translation from Carneades into Dung’s AFs in Haskell (4)

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;
- `argsToAF` translates the arguments.
Outline

1 Motivation of the methodology
   Overview of the specifications/implementations/formalisations

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

3 Verifying correctness of an implementation
   Verification using FP
   Theorem proving

4 Conclusions and future work
Outline

1 Motivation of the methodology
   Overview of the specifications/implementations/formalisations

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

3 Verifying correctness of an implementation
   Verification using FP
   Theorem proving

4 Conclusions and future work
Implementation of properties

Theorem (Correspondence of applicability)

Let $C$ be a carneades argument evaluation structure, $\langle \text{arguments, audience, standard} \rangle$, $\mathcal{L}_{\text{CAES}}$ the propositional language used and let the argumentation framework corresponding to $C$ be $\text{AF}$.
Theorem (Correspondence of applicability)

Let \( C \) be a carneades argument evaluation structure, 
\( \langle \text{arguments}, \text{audience}, \text{standard} \rangle \), \( \mathcal{L}_{\text{CAES}} \) the propositional language used and let the argumentation framework corresponding to \( C \) be \( \text{AF} \). Then the following holds: An argument \( a \in \text{arguments} \) is applicable in \( C \) iff there is an argument contained in the complete extension of \( \text{AF} \) with the corresponding conclusion \( \text{arg}_a \) in an \( \text{AF} \).
Implementation of properties

Theorem (Correspondence of applicability)

Let $C$ be a carneades argument evaluation structure, $\langle \text{arguments, audience, standard} \rangle$, $\mathcal{L}_{CAES}$ the propositional language used and let the argumentation framework corresponding to $C$ be $AF$. Then the following holds: An argument $a \in \text{arguments}$ is applicable in $C$ iff there is an argument contained in the complete extension of $AF$ with the corresponding conclusion $\text{arg}_a$ in an $AF$.

$$
corApp :: \text{CAES} \rightarrow \text{Bool}
corApp caes@(CAES (\text{argSet, } _, _)) =
\begin{align*}
\text{let } & \text{transCAES } = \text{translate caes} \\
& \text{appArgs } = \text{filter ('applicable'caes)} \\
& \quad (\text{getAllArgs argSet}) \\
& \text{transArgs } = \text{stripRight (groundedExt transCAES)} \\
\text{in } & \text{fromList appArgs } \equiv \text{fromList transArgs}
\end{align*}
$$
Implementation of properties (2)

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

This is a property than can be verified using QuickCheck!
Implementation of properties (2)

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
Implementation of properties (2)

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

> corApp caes
True

This is a property than can be verified using QuickCheck!
1 Motivation of the methodology
   Overview of the specifications/implementations/formalisations

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

3 Verifying correctness of an implementation
   Verification using FP
   Theorem proving

4 Conclusions and future work
Dung’s model formalised in a theorem prover

Formalised Dung’s AFs in a theorem prover:
Dung’s model formalised in a theorem prover

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

• Formalised a subset of the Haskell implementation in Agda;
Formalised Dung’s AFs in a theorem prover: Formalisation written in Agda, a dependently typed functional programming language, syntax similar to Haskell:

- Formalised a subset of the Haskell implementation in Agda;
- Given a finite AF, proved termination, existence and uniqueness of the grounded labelling.
Grounded labelling takes three \textit{lists of arguments}:
Code example of the formalisation of Dung’s AFs (1)

Grounded labelling takes three lists of arguments:

- Ins
- Outs
- Unlabelled arguments (initially all)
Grounded labelling takes three lists of arguments:

- Ins
- Outs
- Unlabelled arguments (initially all)

\[ \textit{groundedList} : \{ A : \text{Set} \} \rightarrow \]
\[ \text{List } A \rightarrow \text{List } A \rightarrow \text{List } A \rightarrow \]
\[ \text{DungAF } A \rightarrow \text{List } (A \times \text{Status}) \]
Code example of the formalisation of Dung’s AFs (2)

Grounded labelling of an AF:
Grounded labelling of an AF:

\[
grounded' : \{ A : Set \} \rightarrow \{ m n o : \mathbb{N} \} \rightarrow \\
... \rightarrow \\
Vec A m \rightarrow Vec A n \rightarrow Vec A o \rightarrow \\
DungAF A \rightarrow Vec (A \times Status) \\
\quad (m + n + o)
\]
Grounded labelling of an AF:

\[
grounded' : \{ A : \text{Set} \} \rightarrow \{ m \ n \ o : \mathbb{N} \} \rightarrow
\left( \sum \mathbb{N} \lambda \ k \rightarrow k \equiv o \right) \rightarrow
\text{Vec} \ A \ m \rightarrow \text{Vec} \ A \ n \rightarrow \text{Vec} \ A \ o \rightarrow
\text{DungAF} \ A \rightarrow \text{Vec} \ (A \times \text{Status})
\left( m + n + o \right)
\]
1 Motivation of the methodology
   Overview of the specifications/implementations/formalisations

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

3 Verifying correctness of an implementation
   Verification using FP
   Theorem proving

4 Conclusions and future work
The functional programming approach has shown to work very well for:

Conclusions (1)
The **functional programming** approach has shown to **work very well** for:

- implementation of abstract argumentation;
The functional programming approach has shown to work very well for:

• implementation of abstract argumentation;
• implementation of structured argumentation, and for DSLs;
Conclusions (1)

The **functional programming** approach has shown to **work very well** for:

- **implementation of abstract argumentation**;
- **implementation of structured argumentation**, and for DSLs;
- **implementation of translations between models**;
The functional programming approach has shown to work very well for:

- implementation of abstract argumentation;
- implementation of structured argumentation, and for DSLs;
- implementation of translations between models;
- quick verification of all three.
Conclusions (2)

The *theorem proving* approach has **large gains**, but can be a significant effort:

- formalisation of Dung's AFs up to grounded semantics is very manageable;
- but further theorem proving is hard:
  - existing proofs are often non-constructive, making formalisation a big effort;
  - structure of translations and their proofs are relatively ad-hoc.

Part of this problem can be solved immediately, by using constructive mathematics/type theory for specifications and proofs of argumentation models.

The remaining research of my PhD will hopefully determine the merits of this approach more precisely.
Conclusions (2)

The theorem proving approach has large gains, but can be a significant effort:

- formalisation of Dung’s AFs up to grounded semantics is very manageable;

Part of this problem can be solved immediately, by using constructive mathematics/type theory for specifications and proofs of argumentation models. The remaining research of my PhD will hopefully determine the merits of this approach more precisely.
Conclusions (2)

The *theorem proving* approach has *large gains*, but can be a significant effort:

- **formalisation** of Dung’s *AFs up to grounded semantics* is very manageable;
- but further *theorem proving* is *hard*:
Conclusions (2)

The theorem proving approach has large gains, but can be a significant effort:

- **formalisation** of Dung’s AFs up to grounded semantics is very manageable;
- but further **theorem proving** is hard:
  - existing proofs are often non-constructive, making formalisation a big effort;
Conclusions (2)

The theorem proving approach has large gains, but can be a significant effort:

- formalisation of Dung’s AFs up to grounded semantics is very manageable;
- but further theorem proving is hard:
  - existing proofs are often non-constructive, making formalisation a big effort;
  - structure of translations and their proofs are relatively ad-hoc.

Part of this problem can be solved immediately, by using constructive mathematics/type theory for specifications and proofs of argumentation models. The remaining research of my PhD will hopefully determine the merits of this approach more precisely.
Conclusions (2)

The theorem proving approach has large gains, but can be a significant effort:

- **formalisation** of Dung’s AFs up to grounded semantics is very manageable;
- but further **theorem proving** is hard:
  - existing proofs are often non-constructive, making formalisation a big effort;
  - structure of translations and their proofs are relatively ad-hoc.

Part of this problem can be solved immediately, by using constructive mathematics/type theory for specifications and proofs of argumentation models.
Conclusions (2)

The theorem proving approach has large gains, but can be a significant effort:

- formalisation of Dung’s AFs up to grounded semantics is very manageable;
- but further theorem proving is hard:
  - existing proofs are often non-constructive, making formalisation a big effort;
  - structure of translations and their proofs are relatively ad-hoc.

Part of this problem can be solved immediately, by using constructive mathematics/type theory for specifications and proofs of argumentation models.

The remaining research of my PhD will hopefully determine the merits of this approach more precisely.
Future work

• Further formalisation of Dung’s AFs:
Future work

• Further formalisation of Dung’s AFs:
  • Basic theorems,
Future work

- Further formalisation of Dung’s AFs:
  - Basic theorems,
  - Formalise grounded semantics further.
Future work

- Further formalisation of Dung’s AFs:
  - Basic theorems,
  - Formalise grounded semantics further.
- Proofs for the formalisation of Carneades,
Future work

• Further formalisation of Dung’s AFs:
  • Basic theorems,
  • Formalise grounded semantics further.
• Proofs for the formalisation of Carneades,
• Proofs for the formalisation of cumulative arguments,
Future work

• Further **formalisation** of Dung’s AFs:
  • **Basic theorems**,  
  • Formalise **grounded semantics** further.
• Proofs for the formalisation of Carneades,
• Proofs for the formalisation of **cumulative arguments**,  
• **Proving** correspondence properties,
Future work

- Further formalisation of Dung’s AFs:
  - Basic theorems,
  - Formalise grounded semantics further.
- Proofs for the formalisation of Carneades,
- Proofs for the formalisation of cumulative arguments,
- Proving correspondence properties,
- Generalise ASPIC$^+$ and use it as a test for our methodology.