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

Bas van Gijzel
Functional Programming Laboratory
School of Computer Science
University of Nottingham
bmv@cs.nott.ac.uk

ABSTRACT
In the last two decades the general interest in abstract argumentation as well as structured argumentation has surged. There has been a plethora of new argumentation models, from general frameworks to more domain specific ones. It has been shown that many of these models can be translated to Dung’s abstract argumentation frameworks. Considering the amount effort put into the optimisation of Dung’s AF’s, one would expect dozens of these translations to be implemented and running to make use of these efficient algorithms. However, this is not the case at present. By providing a tutorial implementation of Dung’s frameworks in Haskell, and formalising this implementation in a theorem prover, we aim to provide a solid base for the implementation and verification of other argumentation models, and very importantly formal translations between models.

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

General Terms
Theory, Verification

Keywords
argumentation, functional programming, Haskell, Agda, theorem proving, ACAI

1. INTRODUCTION
In the last two decades the general interest in abstract argumentation as well as structured argumentation has surged. There has been a plethora of new argumentation models, from general frameworks [18, 3, 5] to more domain specific ones [13, 12]. It has been shown that many of these models can be translated to Dung’s framework [18, 11, 10, 4, 17]. Considering the amount effort put into the optimisation of Dung’s AF’s [6], one would expect dozens of these translations to be implemented and running to make use of these efficient algorithms. However, this is not the case at present. There are a number of possible reasons:

• Most implementations of argumentation models are not publicly available (or the website is no longer available) and thus coupling of implementations is not that easily done.

• Translations can be notoriously complex, both in implementation and in verification. As a good example, consider the translation of Carneades to ASPIC++ [11, 10], or the translation of abstract dialectical frameworks to Dung [4]. Both proofs are at least a page long, and are hard to verify even by experts in the field.

This paper makes an initial push to solving this problem. We provide and discuss all the Haskell programming code of an implementation of Dung’s argumentation frameworks, containing a decent amount of the standard definitions, making this paper both documentation and implementation. Similar to our previous work [9] we provide our implementation as a public library1. Our choice of programming language, Haskell, is motivated by this previous attempt, where we managed to implement an argumentation model intuitively enough to be easily readable by an argumentation theorist with no previous knowledge of Haskell. After discussing the Haskell implementation, we discuss and provide (in the Appendix) a formalisation of this implementation in a theorem prover. Although we just give a sketch of how the actual translation of Carneades to Dung and its derived properties/proofs can be done we already provide the following contributions:

• an intuitive implementation of Dung’s AFs in Haskell, staying faithful to the actual mathematical definitions;
• to our knowledge, the first formalisation of an argumentation model in a theorem prover;
• a general methodology for implementing translations between frameworks and proving properties between them;
• an effort towards the first formalisation of a translation between argumentation models.

The paper is structured as follows. In Section 2 we give an introduction to Dung’s abstract argumentation frameworks, each time providing implementations of the definitions in functional programming language, Haskell. In Section 3 we discuss our formalisation of this implementation in a theorem prover, Agda, and what we can gain from this. In Section 4 we provide an introduction to Carneades, again give corresponding Haskell definitions and end with a sketch on how to encode useful properties between Carneades and Dung using insights from Section 3. We conclude in Section 5 with a discussion of what we have learnt from this initial study and how we can take this further.

1See http://hackage.haskell.org/package/Dung
2. AN IMPLEMENTATION OF AFS IN HASKELL

The abstract argument system, or argumentation framework (AF) as introduced by Dung [8] is a very simple, but general model that is able to capture various contemporary approaches to non-monotonic reasoning. It has also been the translation target for many modern structured argumentation models [18, 11, 10, 4, 17] that have been introduced later in the literature. In this section we will give a significant part of the standard definitions of Dung’s AFs, including an algorithm for the grounded semantics and show how these definitions can be almost immediately translated into (a slightly stylised version) of the functional programming language, Haskell. The purpose of this section is to show how a functional programming language such as Haskell can be used to quickly implement a prototype of an argumentation model and to set up the possibility for proving properties of this implementation. This section can also serve as a tutorial like introduction to the implementation of AFs up to grounded semantics.

We will not be able to fully discuss AFs nor can we handle Haskell syntax, so for a more in depth introduction to abstract argumentation see Baroni and Giacomin [1] and for Haskell one can consult one of the standard textbooks [14, 15].

Definition 1. Abstract argumentation framework An abstract argumentation framework is a tuple \( (\text{Args}, \text{Def}) \), such that \( \text{Args} \) is a set of arguments and \( \text{Def} \subseteq \text{Args} \times \text{Args} \) is a defeat relation on the arguments in \( \text{Args} \).

It is important to note that the type of argument is left abstract in the definition above, allowing for possible instantiations of argument by other frameworks such as ASPIRE ++ [18]. Several of the following definitions however, will need to have some notion of equality on arguments to be able to make the needed comparisons. For now, we use \text{Strings} to just label abstract arguments.

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

Note that we use lists instead of sets to allow for an easier presentation.

Example 1. An example (abstract) argumentation framework containing three arguments where the argument \( C \) reinstates the argument \( B \) by defeating its defeating argument \( B \) is captured by \( \text{AF}_1 = \{ (a, b, c), \{ (a, b), (b, c) \} \} \).

\[
A \Rightarrow B \Rightarrow C
\]

And in Haskell:

\[
\begin{align*}
a, b, c :: & \text{AbsArg} \\
a & = "A" \\
b & = "B" \\
c & = "C" \\
\text{AF}_1 :: & \text{DungAF AbsArg} \\
\text{AF}_1 = & \text{AF} [a, b, c] [(a, b), (b, c)]
\end{align*}
\]

We now give a few standard definitions for AFs such as the acceptability of arguments and admissibility of sets.

Definition 2. Let \( \text{AF} = (\text{Args}, \text{Def}) \) and \( \text{S} \subseteq \text{Args} \).

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

In our Haskell implementation of these definitions we resort to a few standard library functions: \( \text{null} \) is a function that takes a list as an argument and returns a \( \text{True} \) if it is empty and \( \text{False} \) otherwise, and \( \text{and} \) is the equivalent to \( \land \) on lists, while \( \text{or} \) is the equivalent to \( \lor \) on lists.

```haskell
setAttacks :: Eq arg => DungAF arg -> [arg] -> arg -> Bool
setAttacks (AF _ def) args arg
  = or [b == arg \| (a, b) <- def, a \in args]
conflictFree :: Eq arg => DungAF arg -> [arg] -> Bool
conflictFree (AF _ def) args
  = null [(a, b) \| (a, b) <- def, a \in args, b \in args]
acceptable :: Eq arg => DungAF arg -> arg -> [arg] -> Bool
acceptable af@(AF _ def) a args
  = and [setAttacks af args b \| (b, a') <- def, a \equiv a']
f :: Eq arg => DungAF arg -> [arg] -> [arg]
f af@(AF args' _) args
  = [a | a <- args', acceptable af a args]
fAF :: AbsArg -> [AbsArg]
fAZ = f AF
admissible :: Eq arg => DungAF arg -> [arg] -> Bool
admissible af args = args \subseteq f af args
```

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.

Given an argumentation framework, we can determine which arguments are justified by applying an argumentation semantics. We will take the \text{labelling-based} approach to grounded semantics (in contrast to extension-based). For correctness, the “if then” in the \( \exists \) has been changed to “and”. Although the fix here is obvious, it does make a case in point for formalisation of more complex mathematics such as a translation between argumentation models.

\[ \text{Algorithm 1. Algorithm for grounded labelling (Algorithm 6.1 of [16])} \]

1. \( \text{L}_0 = (\emptyset, \emptyset, \emptyset) \)
2. \( \text{repeat} \)
3. \( \text{in}(L_{i+1}) = \text{in}(L_i) \cup \{x \mid x \text{ is not labelled in } L_i\} \) \\
4. \( \text{out}(L_{i+1}) = \text{out}(L_i) \cup \{x \mid x \text{ is not labelled in } L_i\} \) \\
5. until \( L_{i+1} = L_i \) \\
6. \( \mathcal{L}_0 = (\text{in}(L_i), \text{out}(L_i), A - (\text{in}(L_i) \cup \text{out}(L_i))) \)

The Haskell equivalent to a labelling:

```haskell
data Status = In | Out | Undecided 

deriving (Eq, Show)
```

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

- if all attackers are \( \text{Out} \)
  unattacked :: Eq arg \( \Rightarrow \) [arg] \( \rightarrow \) 
  DungAF arg \( \rightarrow \) arg \( \rightarrow \) Bool 
  unattacked outs (AF \( \setminus \) def) arg = 
    let attackers = [\( a \mid (a, b) \leftrightarrow \text{def}, \arg \equiv b \)] 
    in null (attackers \( \setminus \) outs)

- if there exists an attacker that is \( \text{In} \)
  attacked :: Eq arg \( \Rightarrow \) [arg] \( \rightarrow \) 
  DungAF arg \( \rightarrow \) arg \( \rightarrow \) Bool 
  attacked ins (AF \( \setminus \) def) arg = 
    let attackers = [\( a \mid (a, b) \leftrightarrow \\text{def}, \arg \equiv b \)] 
    in \( \neg \) (null (attackers \( \setminus \) ins))

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

\[
\text{grounded} :: \text{Eq arg} \Rightarrow \text{DungAF arg} \Rightarrow \{(\text{arg, Status})\} \\
\text{grounded af}@\text{AF args} = \text{grounded}' \[\text{(arg, Status)}\] \arg af
\]

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

\[
\text{grounded'} \arg \text{ins outs arg} = \\
\text{let newIns = filter (unattacked outs af) arg} \\
\text{newOuts = filter (attacked ins af) arg} \\
\text{if null (newIns + newOuts)} \text{then} \\
\text{map (lambda x -> (x, Ins)) ins} \\
\text{map (lambda x -> (x, Outs)) outs} \\
\text{else grounded' (ins + newIns) newIns newOuts af}
\]

Then as expected:

\[
\text{grounded AF} > [("A", \text{In}), ("C", \text{In}), ("E", \text{Out})]
\]

Finally, the grounded extension can be defined by returning only those arguments that are \text{In} from the grounded labelling.

\[
\text{groundedExt} :: \text{Eq arg} \Rightarrow \text{DungAF arg} \rightarrow [\text{arg}] \\
\text{groundedExt af} = [\text{arg} | (\text{arg, In}) \leftrightarrow \text{grounded af}]
\]

3. FORMALISING THE IMPLEMENTATION IN A THEOREM PROVER

Now that we have been able to construct an implementation of Dung’s AFs, we can formalise this implementation into a theorem prover of our choosing. Agda is also functional in nature and very close to Haskell making it an obvious choice. Agda is a programming language and a theorem prover at the same time. Types with accompanying implementations (functions), correspond to theorems with accompanying proofs through the Curry-Howard correspondence, or the proofs-as-programs interpretation. This means that if we write an implementation/proof of grounded semantics in Agda we already gain a few nice results for free. All functions that are implemented are guaranteed to be terminating, which means that because we successfully implemented the grounded semantics, we immediately know that our algorithm is terminating on all (finite) inputs and because Agda will always give back a labelling, we also have proven that the grounded extension always exists! The correctness of these proofs are automatically checked by the Agda type checker and thus the correctness of the proofs only depends on the core implementation of Agda.

The mathematical properties proven here, similar to the proofs of correspondence results between argumentation models, are not meant for an end-user of an actual implementation of the argumentation model. What we do gain however, is a mechanically proven way to check that our standard algorithms are correct, which is especially useful in the case that the two languages are relatively close (as is the case for Haskell and Agda). Because a full treatment of the Agda code in combination with the Haskell programming language would unbalance this paper, we have instead included the relevant code in the Appendix3.

Although the Agda code is quite technical, this is not a need for concern. The mathematical properties proven here, similar to the proofs of correspondence results between argumentation models, are not meant for an end-user of an actual implementation of the argumentation model. What we do gain however, is a mechanically proven way to check that our standard algorithms are correct, which is especially useful in the case that the two languages are relatively close (as is the case for Haskell and Agda).

4. INTEGRATING OTHER FRAMEWORKS

The previous section talked about the formalisation of some basic definitions of AFs in Agda. However, given this initial effort, there is a lot space for interesting future work. In this section I will give definitions and again corresponding implementations in Haskell of the Carneades argumentation model [13, 12], an argumentation model designed to capture standards and burdens of proof. This is based on previous work in [9]. After that, I will sketch which useful properties of a possible translation between Carneades and Dung’s AFs we might want to prove, given that we again have a corresponding implementation (of Carneades) in the same theorem prover.

We will start with the definition of argument in Carneades.

Definition 3. Carneades’ Arguments Let \( \mathcal{L} \) be a propositional language. An argument is a tuple \((P, E, c)\) where

3For the complete Agda code see: http://www.cs.nott.ac.uk/~bmv/Code/AF2.agda
proposition) and to be pro its conclusion c (which may be a negative atomic proposition) and con the negation of c.

In Carneades all logical formulae are literals in propositional logic; i.e., all propositions are either positive or negative atoms. Taking atoms to be strings suffice in the following, and propositional literals can then be formed by pairing this atom with a Boolean to denote whether it is negated or not:

```haskell
type PropLiteral = (Bool, String)
```

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

```haskell
negate :: PropLiteral -> PropLiteral
negate (b, x) = (¬ b, x)
```

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

```haskell
data Argument = Arg ([PropLiteral], [PropLiteral], PropLiteral)
```

Arguments are considered equal if their premises, exceptions and conclusion are equal; thus arguments are identified by their logical content. The equality instance for Argument (omitted for brevity) takes this into account by comparing the lists as sets. A set of arguments determines how propositions depend on each other. Carneades requires that there are no cycles among these dependencies. Our implementation of this acyclic set will be considered abstract for simplicity, instead just providing functions to retrieve all arguments and all used propositions in the Argument set.

```haskell
type ArgSet = ...
getAllArgs :: ArgSet -> [Argument]
getProps :: ArgSet -> [PropLiteral]
```

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

**Definition 4.** Carneades Argument Evaluation Structure (CAES) A Carneades Argument Evaluation Structure (CAES) is a triple \( \langle \text{arguments}, \text{audience}, \text{standard} \rangle \), where arguments is an acyclic set of arguments, audience is an audience (see below), and standard is a total function mapping each proposition to its specific proof standard.

The transliteration into Haskell is almost immediate:

```haskell
newtype CAES = CAES (ArgSet, Audience, PropStandard)
```

We will skip most of the definitions of audience and proof standards. The only directly relevant part for the evaluation are the assumptions (of an audience) which is simply a consistent set of literals assumed to be acceptable, similar to axioms. In Haskell this is \([\text{PropLiteral}]\).

Two concepts central to the evaluation of a CAES are applicability of arguments, which arguments should be taken into account, and acceptability of propositions, which conclusions can be reached under the relevant proof standards, given the beliefs of a specific audience.

**Definition 5.** Applicability of arguments Given a set of arguments and a set of assumptions (in an audience) in a CAES \( C \), then an argument \( a = \langle P, E, c \rangle \) is applicable iff

- \( p \in P \) implies \( p \) is an assumption or \( \overline{p} \) is not an assumption and \( p \) is acceptable in \( C \)
- \( e \in E \) implies \( e \) is not an assumption and \( \overline{e} \) is an assumption or \( e \) is not acceptable in \( C \).

**Definition 6.** Acceptability of propositions Given a CAES \( C \), a proposition \( p \) is acceptable in \( C \) iff \( \langle s, p, C \rangle \) is true, where \( s \) is the proof standard for \( p \).

Note that these two definitions in general are mutually dependent because acceptability depends on proof standards, and most sensible proof standards depend on the applicability of arguments. This is the reason that Carneades restricts the set of arguments to be acyclic.

```haskell
applicable :: Argument -> CAES -> Bool
applicable (Arg (prems, excns, _)) = caes @ caes (⟨⟨ assumptions, _, _ ⟩⟩) = (⟨⟨ (p ∈ assumptions) ∨ \( p \) ‘acceptable’ caes | p ← prems) ↓ (e ∈ assumptions) ↓ (e ‘acceptable’ caes | e ← excns ) ])
where
  x ⊥ y = ¬ (x ∨ y)
acceptable :: PropLiteral -> CAES -> Bool
acceptable c caes @ (CAES (⟨⟨ _, _ standard ⟩⟩)) = c ‘acceptable’ caes
where s = standard c
```

One of the shortcomings in the discussed version of Carneades is that it is not able to handle cycles. The translation given in [11, 10] attempts to alleviate this problem, by providing a translation to ASPIC\(^+\) which is known to generate AFs [18] and then providing semantics to cycle-containing structures by delegating those to Dung’s semantics.

Then given that we already have some Haskell data type for ASPIC\(^+\) arguments:

```haskell
data ASPICPlusArgument = ...
```

We can define a Dung AF instantiated with concrete ASPIC\(^+\) arguments.

```haskell
type ASPICAF = DungAF ASPICPlusArgument
```

Given this, we have the necessary ingredients to sketch how a translation function and the properties we would like to prove look like. As a first step we assume we have already defined a translation function, and also have access to two functions that are able to retrieve the corresponding Dung argument to a Carneades Argument and PropLiteral. The type signatures are given below:

```haskell
translate :: CAES -> DungAF ASPICPlusArgument
arg2dung :: DungAF ASPICPlusArgument -> Argument -> ASPICPlusArgument
prop2dung :: DungAF ASPICPlusArgument -> PropLiteral -> ASPICPlusArgument
```

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

For the translation function here, we can refer to existing definitions of the correspondence of applicability of arguments and acceptability of propositions (Theorem 4.10 of [11]).

**Theorem 1.** Let C be a CAES, (arguments, audience, standard), $\mathcal{L}_{\text{CAES}}$ the propositional language used and let the argumentation framework corresponding to $C$ be AF. Then the following holds:

1. An argument $a \in \text{arguments}$ is applicable in $C$ if there is an argument contained in the complete extension of AF with the corresponding conclusion $a_r$.

2. A propositional literal $c \in \mathcal{L}_{\text{CAES}}$ is acceptable in $C$ or $c \in \text{assumptions}$ if there is an argument contained in the complete extension of AF with the corresponding conclusion $c$.

Informally, the properties state that every argument and proposition in a CAES, after translation, will have a corresponding argument and keep the same acceptability status. I will now sketch the implementation of these properties in Haskell. If the translation function is a correct implementation, the Haskell implementation of the correspondence will be acceptable/unacceptable.

Let $\text{CorApp}$ be a function that corresponds to the argumentation framework corresponding to $C$ be AF. Then the following holds:

1. An argument $a \in \text{arguments}$ is applicable in $C$ if there is an argument contained in the complete extension of AF with the corresponding conclusion $a_r$.

2. A propositional literal $c \in \mathcal{L}_{\text{CAES}}$ is acceptable in $C$ or $c \in \text{assumptions}$ if there is an argument contained in the complete extension of AF with the corresponding conclusion $c$.

Finally, we gave a description of a translation function and its required properties, showing how we can make proper use of the formalisations. Such a formalisation would give us the means to translate between argumentation models in a verified manner.

The initial results are encouraging, despite that we haven’t formalised an actual translation yet. The relatively successful formalisation of Dung’s argumentation frameworks suggest that the implementation and formalisation of a translation is not far off. It is important to note that our approach is not necessarily meant to give the final implementation of a model. The intended use of this approach is for quick prototyping/testing of argumentation models, followed by an implementation and verification of a translation between models, delegating the actual evaluation of arguments to an optimised implementation.

Instead of translating between argumentation models, we can also choose to translate to a specific format, such as a file format or a general format such as the Argument Interchange Format [7, 19]. Especially the recent work on giving a logical specification to the AIF [2] would be a good application for a theorem prover.

6. REFERENCES


APPENDIX

A. AGDA IMPLEMENTATION

We give our Agda code in the following pages, supplied with documentation.

The Agda code is structured similarly to the Haskell code, but with a few main differences. Agda does not (yet) have syntax for list comprehensions, which means list comprehension need to be desugared. We need a few simple lemmas to prove some obvious mathematical facts, but these can all be handled by an automatic solver in the Agda library. To simplify the proof, instead of filtering out all attacked and unattacked at once, we filter out one or none at the time, corresponding to the foundV and notFoundV constructors of the Find datatype. Finally, the implementation uses Vectors (lists with a fixed size), so that the size of the list is always known at compile time. This is useful because to prove that the grounded function terminates, we need to prove that the length of the arguments that still need to be labelled is structurally decreasing.

data DungAF (A : Set) : Set where
  AF : List A → List (A × A) → DungAF A
  AbsArg = String
  a : AbsArg
  a = "A"
  b : AbsArg
  b = "B"
  c : AbsArg
  c = "C"

  -- an AF such that: A → B → C
  AF_1 : DungAF AbsArg
  AF_1 = AF (a :: b :: c :: []) ((a, b) :: (b, c) :: [])

  -- an AF such that: A ↔ B
  AF_2 : DungAF AbsArg
  AF_2 = AF (a :: b :: []) ((a, b) :: (b, a) :: [])

  -- The Agda equivalent to a labelling:
data Status : Set where
  In : Status
  Out : Status
  Undecided : Status

  -- intersectBy and deleteFirstsBy, are the Agda equivalents of intersect and (\x)\n  -- (from a certain AF) that are already considered,
  -- compute whether an argument (arg) is attacked.
  unattacked : \A : Set\ → \(A → A → Bool\) → List A → DungAF A → A → Bool
  unattacked _≡_ outs (AF _def) arg =
    null
    (deleteFirstsBy _≡_
     (List.map proj1
      (filter ((λ x → x ≡ arg) ◦ proj2)
        def)) outs)

  -- Given an equality function, a list of arguments
  -- (from a certain AF) that are already considered In,
  -- compute whether an argument (arg) is attacked.
  attacked : \A : Set\ → \(A → A → Bool\) → List A → DungAF A → A → Bool
  attacked _≡_ ins (AF _def) arg =
    (null
     (intersectBy _≡_
      (List.map proj1
       (filter ((λ x → x ≡ arg) ◦ proj2)
         def)) ins))

  -- False as the empty (or impossible) Type
  data False : Set where
    -- True as the Unit Type (type with only one value)
    record True : Set where
      -- (using a record allows Agda to
      -- automatically infer the only allowed value)
      record True : Set where
        -- going from a decidable predicate to a Type
        isTrue : Bool → Set
        isTrue true = True
        isTrue false = False
-- going from a decidable predicate to a Type
isFalse : Bool → Set
isFalse true = False
isFalse false = True

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

-- from decidable predicate to predicate on types
satisfies : \{ A : Set \} → (A → Bool) → A → Set
satisfies p x = isTrue (p x)

-- A simple lemma to transform to transform isFalse to isTrue
lemma : \{ x : Bool \} → isFalse x → isTrue (¬ x)
lemma true refl = true
lemma false refl = false

infixr 30 _ ; allV : _
data AllV { A : Set } { P : A → Set } : \{ n : N \} → Vec A n → Set where
allV [] = AllV P []
_ ; allV : \{ a : A \} : \{ n : N \} \{ xs : Vec A n \} → P x → AllV P xs → AllV P (x :: xs)
data FindV { A : Set } { P : A → Bool } : \{ n : N \} → Vec A n → Set where
foundV : \{ k : N \} \{ m : N \} \{ xs : Vec A k \} (y : A) → satisfies p y → (ys : Vec A m) → FindV p (xs ++ y :: ys)
notfoundV : \{ k : N \} \{ npxs : Vec A n \} → AllV (satisfies (¬ ◦ p)) xs → FindV p xs

findV : \{ A : Set \} \{ n : N \} \{ p : A → Bool \} (xs : Vec A n) → FindV
findV p [] = notfoundV allV []
findV p (x :: xs) with p x | inspect p x
findV p (x :: xs) | true \{ [prf] \} = foundV [] (trueIsTrue prf) xs
findV p (x :: xs) | false \{ [prf] \} = foundV [x : trueIsFalse prf] xs

findV p (xs :: ys) | false \{ [prf] \} | notfoundV npxs = notfoundV (lemma (falseIsFalse prf) : allV : npxs)

-- Simple arithmetic lemmas automatically solved by the RingSolver
-- (This is done by giving a syntactical representation of the theorem
-- and letting the RingSolver rewrite this. This is succesful since
-- I was able to use refl)
lemma2 : \{ m n k l : N \} → (suc \( m + n + (k + l) \)) → (m + n + (k + suc l))
lemma2 \{ m \} \{ n \} \{ k \} \{ l \} = solve 4
\( \lambda m\:
\n\con 1
\m' + (m' + n' + (k' + l'))
\)
\( m' + n' + (k' + (con 1 + l'))
\)
refl m n k l

lemma3 : \{ m n k l : N \} → (m + suc n + (k + l)) → (m + n + (k + suc l))
lemma3 \{ m \} \{ n \} \{ k \} \{ l \} = solve 4
\( \lambda m\:
\n\con 1
\m' + (con 1 + n' + (k' + l'))
\)
\( m' + n' + (k' + (con 1 + l'))
\)
refl m n k l

lemma4 : \{ a k l : N \} → a ≡ k + suc l → a ≡ suc (k + l)
lemma4 \{ a \} \{ k \} \{ l \} \{ p \} = trans p
(solve 2 \( \lambda k' l' \ → k' + (con 1 + l') \)
\( k + l' \)
)
refl k l

lemma5 : \{ a k l : N \} → suc a ≡ k + suc l → a ≡ k + l
lemma5 \{ a \} \{ k \} \{ l \} \{ p \} = cong pred
(lemma4 \{ suc a \} \{ k \} \{ l \} \{ p \})

-- length of Vectors
length : \{ A : Set \} \{ n : N \} → Vec A n → N
length \( _ = n \)
-- grounded' helper function for grounded defined below (using Vectors)
-- grounded' takes 3 Vectors. The current ins and outs (starting empty),
-- the arguments to process (args), a Dung AF (af)
-- a predicate on the arguments allowing for comparison (≤_≈)
-- and a proof that there is a number equal to the length of args (o)
grounded' : {A : Set} → {m n o : ℕ} → (Σ N λ k → k ≡ o) → (A → A → Bool)
  → Vec A m → Vec A n → Vec A o → DungAF A → Vec (A × Status) (m + n + o)
  → Base case:
  -- We have no more arguments to process.
  grounded' ≤_≈ ins outs [] ≡ (map (λ x → (x, In))) ins ++ map (λ x → (x, Out)) outs ++ []
  -- Inductive cases:
  -- Otherwise we can possibly find an unattached/attacked argument
grounded' ≤_≡ ins outs args af with findV (unattached ≤_≡ (toList outs) af) args |
  findV (attached ≤_≡ (toList ins) af) args
  -- Two impossible cases (!) means that there is no valid constructor:
  -- The length of args is zero, while we did manage to find an unattached/attacked element
  -- Thus we use lemma4 to rewrite o so we can match on suc using lemma4 (and suc_ ≠ zero)
grounded' {o = .(k + suc l)} {zero, p} ≤_≡ ins outs af with
  foundV {k} {l} \ustria{p} ≤_≡ af with lemma4 {zero} {k} {l} p
... | ()
grounded' {o = .(k + suc l)} {zero, p} ≤_≡ ins outs af with
  notFoundV _ \ustria{p} ≤_≡ with lemma4 {zero} {k} {l} p
... | ()
  -- Two cases:
  -- We have found an unattached/attacked element.
  -- The Vector we try to return is of the "wrong" length, so we need to rewrite it using the basic lemma3
  -- and substitute this value in the Vector constructor
  -- Similarly we need to rewrite the proof of the length of o, using lemma5
grounded' \ustria{m} \ustria{n} \ustria{o = .(k + suc l)} \ustria{a, p} ≤_≡ ins outs
  \ustria{(xs ++ ys :: ys)} af with
  foundV {k} {l} \ustria{x} y \ustria{ys} \ustria{p} ≡ subst (Vec _\ustria{p}) (lemma2 \ustria{m} \ustria{n} {k} {l})
  (grounded' \ustria{a, lemma5 \ustria{a}} {k} {l} p \ustria{y :: ins} outs (xs ++ ys) af)
grounded' \ustria{m} \ustria{n} \ustria{o = .(k + suc l)} \ustria{a, p} ≤_≡ ins outs
  \ustria{(xs ++ ys :: ys)} af with
  notFoundV _ \ustria{p} \ustria{y :: ins (y :: outs)} (xs ++ ys) af
  (grounded' \ustria{a, lemma5 \ustria{a}} {k} {l} p \ustria{y :: ins (y :: outs)} (xs ++ ys) af)
  -- Final case (fixpoint):
  -- We haven’t found any unattached/attacked element and thus are done.
grounded' ≤_≡ ins outs args _ \ustria{notFoundV _ \ustria{notFoundV _ ≡ (map (λ x → (x, In))) ins}}
  \ustria{# map (λ x → (x, Out)) outs}
  \ustria{# map (λ x → (x, Undecided)) args}

-- The actual grounded labelling function.
-- This function calls grounded with the correct Vectors and lengths.
grounded : {A : Set} → (A → A → Bool) → DungAF A → List (A × Status)
grounded ≤_≡ (AF args def) = toList
  (grounded' ((length (fromList args), refl)), refl)
  \ustria{[ [] ] (fromList args) (AF args def))}

testGrounded1 : List (AbsArg × Status)
testGrounded1 = grounded String ≤_≡ _ AF1
  -- ("C", In) :: ("A", In) :: ("B", Out) :: []
testGrounded2 : List (AbsArg × Status)
testGrounded2 = grounded String ≤_≡ _ AF2
  -- ("A", Undecided) :: ("B", Undecided) :: []

-- Defining the grounded extension using the grounded labelling is trivial,
-- we just need to keep all the arguments with an In label
groundedExt : {A : Set} → (A → A → Bool) → DungAF A → List A
groundedExt ≤_≡ (AF args def) = List.map proj1 ((filter (≤_≡ In) proj2) (grounded ≤_≡ (AF args def))))