Argument & Computation 9 (2018) 91–119
DOI 10.3233/AAC-180039
IOS Press

Representing argumentation schemes with Constraint Handling Rules (CHR)

Thomas F. Gordon [a,*], Horst Friedrich [a] and Douglas Walton [b]
[a] *Fraunhofer FOKUS, Berlin, Germany*
E-mail: horst.friedrich@fokus.fraunhofer.de
[b] *University of Windsor, Windsor, Canada*
E-mail: dwalton@uwindsor.ca

Abstract. We present a high-level declarative programming language for representing argumentation schemes, where schemes represented in this language can be easily validated by domain experts, including developers of argumentation schemes in informal logic and philosophy, and serve as *executable specifications* for automatically constructing arguments, when applied to a set of assumptions. Since argumentation schemes are defeasible inference rules, both premises and conclusions of schemes can be second-order schema variables, i.e. without a fixed predicate symbol. Thus, while particular schemes can be and have been implemented in computer programs, in general argumentation schemes cannot be represented as executable specifications using logic programming languages based on first-order logic, such as Prolog. Moreover, even if the conclusion (head) of Prolog rules could be second-order variables, a depth-first, backward-chaining search strategy, as typically used in logic programming, would usually cause such programs to not terminate, since every goal would match the head of such a scheme, including all goals created by instantiating the body of the same scheme. The language for representing argumentation schemes presented here, for the purpose of automatically constructing arguments, uses Constraint Handling Rules (CHR), a declarative, Turing complete, forwards-chaining, rule-based programming language introduced by Thom Frühwirth in 1991. CHR is attractive for representing and implementing argumentation for several reasons, including: 1) Inference rules, rewrite rules, sequents, proof rules, and logical axioms can be directly written in CHR. 2) The execution of CHR rules can be interrupted and restarted at any time, with intermediate results approximating the final solution, and 3) Constraints can be input incrementally as they become known, during rule execution, without requiring recomputation. These three properties of CHR appear attractive for representing and implementing argumentation schemes. Since argumentation schemes are (defeasible) inference rules, the ability of CHR to represent inference rules directly would appear to be quite useful. The ability to stop the computation and produce approximate results is compatible with one objective of argumentation, to provide a principled method for supporting approximate reasoning with limited resources. Because argumentation typically takes place in dialogs, with evidence and arguments brought forward and asserted by the participants incrementally, during the course of the dialog, CHR's ability to handle new information, incrementally introduced during the computation, may be useful. This new rule language for representing argumentation schemes is validated by using it to represent twenty representative argumentation schemes.

Keywords: Argumentation schemes, argument generation, rule-based systems, Constraint Handling Rules, logic programming, data protection

1. Introduction

Argumentation schemes [25] serve at least two functions:

(1) They provide normative standards for critically evaluating arguments, by matching arguments to schemes to see if they fit acceptable patterns of argumentation, to identify missing premises, and to facilitate the asking of critical questions.

[*]Corresponding author. URL: http://www.tfgordon.de/contact/.

(2) They provide guidance for making (constructing, inventing, generating) good arguments in the first place, i.e. arguments that will satisfy the normative standards specified by the schemes.

Computational models of argument can model either or both of these functions of argumentation schemes. In this paper, we focus on the second function. Whereas some prior work on computational models of argumentation schemes consists of procedural programs for generating arguments for specific schemes, e.g [3], our aim is to develop a high-level declarative programming language for representing, ideally, *all* argumentation schemes, where schemes represented in this language can be easily validated by developers of argumentation schemes in informal logic and philosophy and serve as *executable specifications* for automatically constructing arguments, when applied to a set of assumptions.

Argumentation schemes [25] are defeasible inference rules. Most, like argument from expert opinion, are to some extent domain-dependent, because they include predicates intended to be interpreted in a particular, domain-dependent way. Some, like defeasible modus ponens, are more generic. Let's take a closer look at these two schemes.

First, here is a simplified version of the scheme for argument from expert witness testimony, which is the prototypical argumentation scheme most often used to introduce the concept of argumentation schemes.

1.1. Argument from expert witness testimony

Premises

- E is an expert
- E asserts P

Conclusions

- P

The expert witness scheme makes use of two domain-dependent predicates:

- is-an-expert/1
- asserts/2

The numbers indicate the arity of each predicate. In the presentation of the scheme, the predicates are shown using an infix notation close to natural language.

In the scheme, E and P are scheme variables. P is a *second-order* variable, ranging over propositions. Notice that the conclusion of the scheme is P. The conclusion does not mention a particular predicate. When the scheme is applied, P is instantiated with a particular proposition, with a particular predicate, and an argument for the proposition P is constructed. The argument constructed can be attacked in the usual ways, with a *rebuttal* (an argument for some proposition which cannot be accepted if P is accepted, for example $\neg P$), an *undercutter* (an argument against the applicability of this argument, for example an argument for E being biased), or a *premise defeater* (an argument for some proposition contrary to a premise, for example an argument con "E is an expert.").

The second argumentation scheme we want to discuss is defeasible modus ponens.

1.2. Defeasible modus ponens

Premises:

- if P then Q
- P

Conclusions

- (Presumably) Q

Defeasible modus ponens has the same form as modus ponens, except that the conclusion is only presumably true, rather than necessarily true. (The modality, "presumably", is made explicit in the example only for emphasis. The conclusions of all argumentation schemes are presumably true.) An argument constructed by instantiating this scheme can be attacked in the usual ways, for example by a rebuttal, an argument for $\neg Q$.

Notice that the major premise of defeasible modus ponens, "if P then Q.", is not an atomic proposition, but rather a compound proposition built by connecting two propositions, P and Q, using an "if-then" (implication) operator. Thus, P and Q here are once again second-order variables ranging over propositions. The conclusion of the defeasible modus ponens scheme is also a second-order variable, as is the conclusion of the expert witness testimony scheme. In addition, the defeasible modus ponens scheme has a minor premise, P, which is a second-order variable.

Argumentation schemes like these, with second-order variables, are quite common. It is particularly common for the conclusion of schemes to be a second-order variable, as in both of these examples. Other examples include schemes for argument from abduction, analogy, established rule, ethos, ignorance, position to know, and precedent.

We are aware of no *computational* models of argumentation schemes which are capable of automatically constructing (inventing, generating, deriving) arguments by instantiating second-order schemes such as these. Prior computational models of argumentation schemes are more limited. Either they are used to check whether existing arguments match the form of a given scheme, as in Aracauria [18], are restricted to propositional (fully instantiated) schemes, as in ArguMed [21], are not defined in sufficient detail to know whether second-order variables are supported, e.g. Pollock's OSCAR system [16], are mathematical models which leave too many details unspecified to be sufficient as a specification for implementing an inference engine, such as ASPIC+[1] [17], or are based on logic programming methods enabling only first-order argumentation schemes to be used to automatically construct arguments, such as Assumption-Based Argumentation (ABA) [8] and earlier versions of Carneades [10].[2]

To understand more clearly the difficulties in representing argumentation schemes using Horn clause logic, the subset of first-order logic used by logic programming languages such as Prolog, let us see how far we can get in representing the scheme for arguments from expert witness testimony in Prolog. Let us first represent, as Prolog "facts", the following assumptions about a case:

```
expert(john).
asserts(john, caused_by(global_warming, humans)).
```

[1]The TOAST implementation of ASPIC+ [19] is propositional. Its rules are fully instantiated axioms, with no variables.

[2]These earlier versions of Carneades allowed argumentation schemes with second-order variables to be represented and used to manually construct arguments, by filling in forms, and to check whether arguments correctly instantiate schemes, but not to automatically construct arguments from a set of assumptions.

Given these facts, the challenge is to represent the expert witness scheme as a single Prolog rule (Horn clause), in such a way that the following query can be proven by Prolog, answering `yes`:

```
?- caused_by(global_warming, humans).
```

The above representation of the assumptions already shows how one hurdle can be overcome. Although Horn clause logic is a subset of first-order logic, it is possible to represent second-order propositions about atomic formulas, such as global warming being caused by humans here, by reifying such atomic formulas as terms. So far, so good.

But how can the expert witness scheme be represented? Here is one approach, suggested to me by Trevor Bench-Capon but also used by ABA [8, 200–201]:

```
holds(P) :- asserts(E,P), expert(E).
```

The idea here is to represent the second-order conclusion, P, of the scheme with a first-order atomic formula, `holds(P)`, by introducing a unary `holds` predicate. This approach attempts to reduce general inference rules, with premises and second-order variables, to first-order axioms, i.e. inference rules with no premises and only first-order variables in the conclusion. That is, strictly speaking there are no premises in this Horn clause representation of the inference rule, because a Horn clause is a first-order formula. The `:-` symbol in the clause represents the material conditional logical connective, not a deducibility relation.

While axioms can be viewed as a basic form of inference rule, the attempt to represent more general inference rules using a `holds` predicate like this has severe limitations. Since we will want to be able to chain arguments together, by using argumentation schemes to construct arguments for the premises of other arguments, we need some way to convert atoms of the form `holds(P)` to `P`, so that premises can be matched (unified) with `P`. It may seem that one way to achieve this would be to add an additional rule for each predicate, as in the following examples:

```
expert(P) :- holds(expert(P)).
asserts(E,P) :- holds(asserts(E,P)).
caused_by(X,Y) :- holds(caused_by(X,Y)).
```

From a knowledge-representation point of view, this seems rather verbose and cumbersome, but presumably these additional rules could be generated automatically, using some kind of preprocessor. However this approach suffers from a more serious problem: Such a rule would need to be generated for *every* predicate in the application domain, and thus *every* goal would match the conclusion of *every* argumentation scheme with a second-order variable as its conclusion, due to Prolog's goal-directed, backwards-chaining control strategy, causing the search space to become infinite. Thus many (not all) queries will cause the inference engine to enter an endless loop and fail to terminate, depending on the order of facts and rules. Suppose, for example, the Prolog program consists only of the following rules, without any facts:

```
holds(P) :- asserts(E,P),expert(E).
expert(E) :- holds(expert(E)).
asserts(E,P) :- holds(asserts(E,P)).
caused_by(X,M) :- holds(caused_by(X,M)).
```

With these clauses, the following query causes an endless loop and runs out of stack space:

```
?- caused_by(global_warming,humans).
ERROR: Out of local stack
```

It is clear why this happens: The query causes an endless loop between the `holds` and `asserts` rules:

```
(1) caused_by(global_warming,humans)
(2) holds(caused_by(global_warming,humans))
(3) asserts(E,caused_by(global_warming,humans))
(4) holds(asserts(E,caused_by(global_warming,...)))
(5) ...
```

Since this encoding of argumentation schemes requires the definition of every predicate to have an additional `holds` rule, many queries will not terminate in this way, making this encoding useless in combination with Prolog's simple depth-first, backwards-chaining control strategy. Notice that the example query will not terminate no matter how the clauses of the program are ordered, because the program contains no facts. While the program can be made to terminate for this particular goal (query) by adding sufficient facts before the rules, this would not be a general solution to the control problem, not even for other queries using the same rules, because one cannot assume that there are sufficient facts to answer every query affirmatively.

The event calculus [14] uses a `holds` predicate for a similar but more limited purpose, for reasoning about the effects of actions using Prolog. It does not suffer from the control issues discussed here, because the `holds` predicate is used in a more focused way only for a subset of the predicates, called *fluents*, which are state-dependent in the domain model. These fluents are queried *only* using the `holds` predicate. They are never mapped to object-level predicates in the way suggested above.

There is one final and fatal problem with representing argumentation schemes directly in Prolog this way that is important to mention: no arguments are constructed! Thus there is no way to resolve conflicts among arguments, to balance arguments or to use the arguments to help understand or explain the results, for example using argument diagrams.

All of these problems might be overcome by writing a meta-interpreter for argumentation schemes in Prolog, but this would be using Prolog in its capacity as a general-purpose programming language, rather than as an inference engine for Horn clause logic. Some expert system shells, in particular APES [13], were implemented as meta-interpreters in Prolog. APES was able to generate explanations which can be viewed as arguments from the traces of rule applications [4]. However rules in APES were Horn clauses and could not represent argumentation schemes with second-order variables, for the reasons discussed above, and also did not generate counterarguments or use a structured model of argument to resolve attack relations among arguments. The alternative approach we investigate in this paper, using Constraint Handling Rules to represent argumentation schemes, can also use Prolog as an implementation language. Indeed several implementations of Constraint Handling Rules in Prolog exist and we make use of the one provided by SWI Prolog.

As suggested in the previous paragraph, this paper explores the idea of representing argumentation schemes using another kind of rule-based programming, Constraint Handling Rules, introduced by Thom Frühwirth in 1991 [9], to overcome all of the problems identified above by meeting the following requirements:

- Allow second-order variables in the premises and conclusions of schemes
- Not require additional rules for bringing second-order propositions down to the object-level.
- Generate arguments as output
- Guarantee termination

The rest of this article is organized as follows. Section 2 introduces Constraint Handling Rules, including examples. Section 3 shows one way to represent argumentation schemes using Constraint Handling Rules, in such as way as to generate arguments and overcome the other problems identified in this introduction. The section also briefly describes two implementations of this approach, one using the Constraint Handling Rules interpreter provided as a library by SWI Prolog and the second based on our custom implementation of Constraint Handling Rules in the Go programming language. Section 4 validates the rule language by demonstrating how to use it to represent twenty argumentation schemes, selected by Douglas Walton as being representative and widely used. Finally, Section 5 presents our conclusions and summarizes the main results.

2. Constraint handling rules

Constraint Handling Rules (CHR) is a *declarative*, forwards-chaining rule language originally developed by Thom Frühwirth in 1991 [9].[3] Forwards-chaining rule engines, based on production rules, have been used from the beginning in expert systems [5,6]. Production rules are condition-action rules, where the conditions of rules are matched against data structures in *working memory*, a conflict-resolution strategy is used to select a matching rule, and then the action of the selected rule is executed, possibly modifying the working memory and performing side effects, such as outputting data to a file. This process is then repeated until no rule matches the state of the working memory.

While production rule systems have been widely and successfully used for expert systems and implementing so-called "business rules", they do not have a *declarative* semantics. Declarative programming languages are used to describe *what* the problem is, rather than a procedure or algorithm stating *how* to solve the problem. Declarative programming languages, or rather interpreters or compilers for these languages, are clever enough to figure out how to solve the problem on their own, from a description of the problem. Declarative programming languages are typically based on well-founded theories of mathematical functions and/or logic. Prolog [7], based on the Horn clause subset of first-order logic, is perhaps the most prominent of these declarative languages.

One of the achievements of CHR is to realize a forwards-chaining rule language, similar to production rule languages, but with a declarative semantics. CHR is so-named, because the language was initially intended to be used to implement constraint solvers. A constraint solver takes as input a set of relationships among variables, called constraints, and derives further information about these variables. Early constraint solvers were for particular domains, for example propositional constraints over Boolean variables, or equations and inequalities over integers. CHR is more general purpose. It enables constraint solvers for a variety of domains to be specified, using rules.

To make this clearer, let us take a look at the standard example used to illustrate CHR, which defines rules for partial orderings:

```
reflexivity  @ X leq X <=> true.
```

[3]See also the CHR homepage at https://dtai.cs.kuleuven.be/CHR/.

```
antisymmetry @ X leq Y, Y leq X <=> X = Y.
transitivity @ X leq Y, Y leq Z ==> X leq Z.
idempotence  @ X leq Y \ X leq Y <=> true.
```

The predicate `leq` is intended to mean "less than or equal to". The words to the left of the `@` symbol in these four rules are identifiers, naming the rules. The `reflexivity`, `antisymmetry`, and `transitivity` rules specify the axioms of partial orderings, in a form close to their usual expression in mathematics. The first rule, for reflexivity, states that for all x, $x = x$. The `idempotence` rule allows the second instance of `X leq Y` to be deleted from the constraint store, since it is redundant.

There are three kinds of rules in CHR: simplification, propagation and "simpagation' rules, where simpagation rules are a hybrid kind of rule combining the features of simplification and propagation rules. All three kinds of rules are illustrated in the example. The reflexivity and antisymmetry rules are simplification rules; the transitivity rule is a propagation rule; and the idempotence rule is a simpagation rule.

Operationally, CHR rules are applied to a multiset of constraints (similar to Prolog facts) in a data structure called the *constraint store*, which serves the same function as the *working memory* in production rule systems. Since the constraint store is a multiset, the same fact may occur multiple times in the store.

When simplification rules, such as the reflexivity and antisymmetry rules, are applied ("fired"), the constraints matching the patterns on the left-hand side of the `<=>` symbol, called the *head* of the rule, are replaced by the constraints on the right-hand side, called the *body* of the rule. For people familiar with Prolog, this terminology may be somewhat confusing, because in Prolog the head of a rule represents its conclusion and the body its antecedents, opposite the convention of CHR. (Moreover, unlike Prolog, a CHR rule may have multiple conclusions.) But CHR's use of the terms "head" and "body" is nonetheless consistent with how these terms are used in Prolog, because in both languages atoms are matched against patterns in the head and, if the match is successful, replaced by atoms on the right. The difference is that Prolog is a goal-directed, backwards-chaining language, which reduces a goal by replacing it with new goals, whereas CHR, on the other hand, as a forwards-chaining rule language, applies rules to derive constraints, adding them to the constraint store.

Simplification and simpagation rules also *delete* constraints from the constraint store. Simplification rules replace the constraints matching the head of the rule with the constraints matching the body of the rule. Simpagation rules, similarly, replace the constraints to the right of the backslash symbol, `\`, in the head of the rule, with the constraints on the right. The matching constraints to the left of the backslash symbol in the head are not deleted.

It might seem counterintuitive at first that a declarative language is allowed to delete constraints from the store. But in CHR this is done in principled way, in a way which does not change the meaning of the constraints in the store. Simplification rules and simpagation rules are used to simplify constraints, as their names suggest, by replacing constraints matching the head with fewer constraints having the same meaning. Consider the idempotence rule, for example. Since the constraint store is a multiset, it may contain duplicate, redundant constraints. The idempotence rule simplifies the constraint store by removing duplicate constraints of the form `X leq Y`.

In addition to heads and bodies, CHR rules may also include, in so-called "guards", further *built-in* constraints. Which built-in constraints are available depends on the particular implementation of CHR. Guards are not illustrated here.

To get an idea of how the CHR inference engine works, let us see what CHR derives when applying the rules defining partial orderings above to the following "query", i.e. giving the initial state of the constraint store:

```
leq(A,B)
leq(B,C)
leq(C,A)
```

First, the transitivity propagation rule is fired and adds `leq(A,C)` to the store. Next, the antisymmetry simplification rule is fired, causing `leq(A,C)` and `leq(C,A)` to be removed and replaced by `A=C`. CHR has built-in support for equality reasoning, which is then used to derive `leq(C,B)` from `leq(A,B)`. Now the antisymmetry simplification rule is applied to `leq(C,B)` and `leq(B,C)`, causing these constraints to be replaced with `B=C`. No further rules can be applied, so the process terminates and returns the constraint store with `A=C` and `B=C`. Thus, CHR was able to infer that all three variables are equal.

In addition to supporting forwards-chaining, CHR has some other properties which may be desirable, depending on the application:

- Turing completeness: Any computable function can be represented using CHR rules.
- Every algorithm can be implemented in CHR with the best known time and space complexity [20].
- CHR rules can be executed concurrently [15].
- The execution of CHR rules can be interrupted and restarted at any time, with intermediate results approximating the final solution.
- Constraints can be input incrementally as they become known, during rule execution, without requiring recomputation.
- Inference rules, rewrite rules, sequents, proof rules, and logical axioms can be directly written in CHR [1].

The last three properties, in particular, appear attractive for representing and implementing argumentation schemes. Argumentation typically takes place in dialogs, with evidence and arguments brought forward and asserted by the participants incrementally, during the course of the dialog. It would be useful if CHR could be used to incrementally and efficiently construct arguments from evidence during dialogs. Moreover, since argumentation schemes are (defeasible) inference rules, the ability of CHR to represent inference rules directly would appear to be quite useful.

3. Representation and implementation of argumentation schemes

In this section we show how to represent argumentation schemes using CHR rules, and present an overview of the implementation of the component for generating arguments with argumentation schemes represented using CHR in this way, provided by Version 4 of the Carneades argumentation system.[4]

First we need a notation for argumentation schemes. Let us use the syntax for schemes we have developed for Carneades 4, which is based on the YAML markup language,[5] which in turn is syntactic

[4]https://carneades.github.io/Carneades/
[5]http://yaml.org/

sugar for JSON,[6] to make it easier to read and write. Here is a version of the scheme for arguments from expert opinion using this concrete syntax:

```
id: expert_opinion
meta:
  title: Argument from Expert Opinion
  source: >
    Douglas Walton, Appeal to Expert Opinion,
    The Pennsylvania University Press,
    University Park, Albany, 1997, p.211-225.
variables: [E,D,P]
premises:
  - expert(E,D)
  - in_domain(P,D)
  - asserts(E,P)
exceptions:
  - untrustworthy(E)
  - inconsistent_with_other_experts(P)
assumptions:
  - based_on_evidence(asserts(E,P))
conclusions:
  - P
```

This representation of argumentation schemes is, we claim, very high level and quite close to the usual way schemes are represented in informal logic. In our experience, informal logicians are able to read, understand and validate schemes represented in this form.

There are a few things to notice about this syntax. First, the schema variables are declared explicitly. This may seem burdensome, but is useful for checking for misspelled variables in schemes, among other purposes. Second, as proposed in [11], the two types of critical questions are represented by exceptions and assumptions. Thirdly, argumentation schemes may now have more than one conclusion, though there is only one in this example. This change was motivated by the desire to support the full CHR rule language. There can be multiple conclusions in the body of CHR rules. But it has the further advantage of reducing the number of schemes required when several conclusions can be derived from the same premises. Fourthly, note that this rule is an example of a scheme having a second-order variable as its conclusion, `S` here. Finally, the example shows how arbitrary metadata about the scheme can be expressed. The various metadata properties, such as `title` and `source` in this example, are not predefined and can be freely selected.

We now show, by way of this example, how argumentation schemes are translated into CHR rules. The expert opinion scheme is translated into the following rule:

```
expert_opinion @
    expert(W,D),
    in_domain(S,D),
    asserts(W,S)
```

[6]http://json.org/

```
    ==>
    S,
    based_on_evidence(asserts(W,S)),
    argument(expert_opinion,[W,D,S]).
```

As illustrated here, each argumentation scheme is translated into a single CHR rule, in this example a propagation rule, with the same name (identifier). The premises of the scheme are translated into constraints in the head of the rule. The conclusions of the scheme are translated into constraints in the body of the rule. Moreover, each of the assumptions of the scheme are also added to the body of the CHR rule, allowing them to be used to derive further information by applying other rules. (The assumptions can be questioned and retracted later, when evaluating the arguments constructed.) Finally, an additional constraint is added to the end of the body of the rule, of the form `argument(<id>,[<variable>,...])`, to keep a record of the argument to be generated when applying the scheme.

Notice that the exceptions of a scheme are not translated and do not appear in the resulting CHR rule. To understand how exceptions are handled, we first need to explain the steps in the process for generating and evaluating arguments:

(1) The argumentation schemes are translated into CHR rules, as illustrated above.
(2) A set of assumptions, represented as ground atomic formulas, are translated into CHR constraints and added to the initial state of the constraint store.
(3) The CHR inference engine is run, repeatedly applying the rules to the constraint store until no rules match or until the `fail` constraint, signaling failure, is derived.
(4) The argument constraints in the store, i.e. the constraints of the form

    `argument(<id>,[<variable>,...])`

    are then translated into Carneades arguments and added to the argument graph.
(5) Assumptions of arguments are added to the assumptions of the argument graph.
(6) For each argument constructed by translating an argument constraint, undercutting arguments are constructed and also added to the argument graph for any exceptions of the applied scheme. The appropriate scheme is retrieved using the identifier of the scheme in the argument constraint.
(7) Finally, the arguments are evaluated, using the formal model of structured argument in [12], to weigh and balance the arguments, resolve attack relations among arguments and label the statements in the argument graph `in`, `out`, or `undecided`.

In addition to supporting multiple conclusions in schemes, we have extended argumentation schemes in further ways, in order to support the full expressiveness of Constraint Handling Rules. To illustrate one of these extensions, supporting simplification, here is a reconstruction of the CHR rules for partial orders, represented as argumentation schemes:

```
- id: reflexivity
   variables: [X]
   deletions:
     - leq(X,X)
   conclusions:
     - true
 - id: antisymmetry
   variables: [X,Y]
   deletions:
     - leq(X,Y)
     - leq(Y,X)
   conclusions:
     - X=Y

 - id: transitivity
   variables: [X,Y,Z]
   premises:
     - leq(X,Y)
     - leq(Y,Z)
   conclusions:
     - leq(X,Z)

 - id: idempotence
   variables: [X,Y]
   premises:
     - leq(X,Y)
   deletions:
     - leq(X,Y)
   conclusions:
     - true
```

All of the premises which are to be deleted from the constraint store when the scheme is applied are listed in a `deletions` block of the scheme. Thus, similar to CHR simpagation rules, argumentation schemes here combine the features of CHR simplification and propagation rules.

One caveat is order: Although all CHR *rules* can be expressed in Carneades, it is not possible in Carneades to formulate the *query* needed to reproduce the example presented in Section 2. This is because CHR queries are represented by Carneades assumptions and assumptions are restricted to ground atomic formulas. This restriction assures that all arguments are fully instantiated. That is all premises, conclusions, exceptions and assumptions of arguments are assured to be ground atomic formulas.

Carneades can be configured to use one of two different implementations of CHR for generating arguments from argumentation schemes using the method presented above: the implementation of CHR which comes pre-installed with SWI Prolog,[7] and a new implementation of CHR in the Go programming language, by the second author of this paper.[8] This new implementation of CHR is still under development but nearing completion. While we do not expect it to have the performance and maturity of the SWI Prolog implementation, it offers several advantages for Carneades: it eliminates a dependency on another system, making it easier to install and administer a Carneades server, and it enables us to experiment with CHR extensions. Two extensions have already been implemented:

[7] http://www.swi-prolog.org/
[8] https://github.com/hfried/GoCHR

(1) Since CHR is Turing complete, termination of CHR programs cannot in general be guaranteed. Our implementation of CHR allows the user to set a maximum number of rule firings, to assure termination within roughly predictable time limits, and returns the arguments constructed before the limit was reached. The engine can be restarted to generate further arguments. This is very much in line with the purpose and spirit of argumentation, as a rational method for problem solving and decision-making when information is inconsistent or incomplete.
(2) The second example argumentation scheme in the introduction, for defeasible modus ponens, cannot be implemented using the SWI Prolog version of CHR. While it allows second-order variables in the body (conclusion) of rules, it does not allow them in the head (premises). We are not sure whether this is a limitation of the SWI Prolog implementation of CHR, or the CHR specification. Either way, our implementation of CHR removes this restriction and allows second-order variables in both the head and body of rules, enabling defeasible modus ponens to be represented.

4. Validation

In this section we attempt to validate the rule language for argumentation schemes and demonstrate its expressiveness by using it represent twenty argumentation schemes, mostly from [25], selected on the basis of their representativeness and wide-use in practice. To allow the reader to evaluate for him- or herself the adequacy of the representations, we present the original formulation alongside our representation for each scheme. When the original source of a scheme is not [25], the source text will be referenced.

The schemes are presented in alphabetical order. To save space, the declarations of the predicates of the language are not presented. The complete source code, including the missing declarations, is available on Github.[9]

4.1. Abductive argumentation scheme

Premises

- D is a set of data or supposed facts in a case.
- Each one of a set of accounts A_1 , $A_2, \ldots, A_n$ is successful in explaining D.
- A_i is the account that explains D most successfully.

Conclusions

- Therefore, A_i is the most plausible hypothesis in the case.

```
id: abduction
variables: [S,T,H]
premises:
  - observed(S)
  - explanation(T,S)
  - in(T,H)
conclusions:
  - H
```

[9]https://github.com/carneades/carneades-4/blob/master/examples/AGs/YAML/walton.yml

```
exceptions:
  - more_coherent_explanation(T,S)
```

4.2. Argument from analogy

Premises:

- Generally case C_1 is similar to case C_2
- A is true in case C_1

Conclusions

- A is true in case C_2

Critical Questions

- Are there differences between C_1 and C_2 that would tend to undermine the force of the similarity cited?
- Is A true (false) in C_1?
- Is there some other case C_3 that is also similar to C_1, but in which A is false (true)?

```
id: analogy
variables: [C,A]
premises:
  - similar_case(C)
  - in_case(A,C)
conclusions:
  - A
exceptions:
  - relevant_differences(C)
  - more_on_point(A,C)
```

In our reconstruction of the scheme from analogy, an implicit current case is compared to the precedent case, C. The first premise checks whether the precedent is similar to the current case, and the second premise checks that a given statement, A, is true in the precedent case. The first exception, modelling the first critical question, asks whether there are relevant differences between the current case and the precedent. The second exception asks whether there is a more on point case than C in which A is not true. The second critical question is not modeled explicitly, since it merely reiterates the second premise, asking whether the statement is really true in the precedent.

4.3. Argument from appearance

Premises

- This object looks like it could be classified under verbal category C.

Conclusions

- Therefore this object can be classified under verbal category C.

Critical Questions

- Could the appearance of its looking like it could be classified under C be misleading for some reason?
- Although it may look like it can be classified under C, could there be grounds for indicating that it might be more justifiable to classify it under another category D?

```
id: appearance
variables: [O,C]
premises:
  - looks_like(O,C)
conclusions:
  - instance(O,C)
```

4.4. Argument from cause to effect

Premises

- Generally, if A occurs, then B will (might) occur.
- In this case, A occurs (might occur).

Conclusions

- Therefore, in this case, B will (might occur).

Critical Questions

- How strong is the causal generalization (if it is true at all)?
- Is the evidence cited (if there is any) strong enough to warrant the generalization as stated?
- Are there other factors that would or will interfere with or counteract the production of the effect in this case?

```
id: cause_to_effect
variables: [A,B]
premises:
  - causes(A,B)
  - has_occurred(A)
conclusions:
  - will_occur(B)
exceptions:
  - interference(A)
```

The first two critical questions are not explicitly included in the reconstruction, because they both attack the first premise, rather than articulating exceptions (undercutters) or assumptions. Premise attacks and rebuttals are modeled with separate argumentation schemes, rather than by exceptions and assumptions of a scheme.

4.5. *Argument from correlation to cause*

Premises

- There is a positive correlation between events E_1 and E_2

Conclusions

- E_1 causes E_2

```
id: correlation_to_cause
variables: [E1,E2]
premises:
  - correlated(E1,E2)
conclusions:
  - causes(E1,E2)
assumptions:
  - explanatory_theory(E1,E2)
exceptions:
  - causes_both(E1,E2)
```

4.6. *Argument from defeasible modus ponens*

Premises

- $A \Rightarrow B$
- A

Conclusions

- B

```
id: defeasible_modus_ponens
variables: [A,B]
premises:
  - implies(A,B)
  - A
conclusions:
  - B
```

In the original version, the $\Rightarrow$ symbol denotes a defeasible conditional, not a strict (material) conditional. Similarly, in the reconstruction the `implies` predicate also is intended to denote a defeasible conditional.

4.7. *Argument from definition to verbal classification*

Premises

- A fits definition D
- For all x, if A fits definition D, then x can be classified as having property G.

Conclusions

- *A* has property *G*

Critical Questions

- What evidence is there that *D* is an adequate definition, in light of other possible alternative definitions that might exclude *A*'s having *G*?
- Is the verbal classification in the classification premise based merely on a stipulative or biased definition that is subject to doubt?

```
id: definition_to_verbal_classification
variables: [O,G,D]
premises:
  - satisfies_definition(O,D)
  - classified_as(D,G)
conclusions:
  - instance(O,G)
exceptions:
  - inadequate_definition(D,G)
```

4.8. Argument from established rule

The version of the argument from established rule we use here is from [22]. No critical questions were formulated for this version of the scheme.

Premises

- If rule *R* applies to facts *F* in case *C*, conclusion *A* follows.
- Rule *R* applies to facts *F* in case *C*.

Conclusions

- In case *C*, conclusion *A* follows.

```
id: established_rule
variables: [C,R]
premises:
  - has_conclusion(R,A)
  - applicable(R)
conclusions:
  - A
assumptions:
  - valid(R)
```

In our reconstruction of the scheme for argument from established rule, the facts and case have been abstracted away, using an `applicable` predicate, which is intended to mean that the rule *R* applies to the facts of the current case. Notice that we have added an assumption for questioning the validity of the rule, which was not formulated in the original version.

4.9. Ethotic argument

Premises

- If x is a person of good (bad) moral character, then what x says should be accepted as more plausible (rejected as less plausible).
- a is a person of good (bad) moral character.

Conclusions

- Therefore, what x says should be accepted as more plausible (rejected as less plausible).

Critical Questions

- Is a a person of good (bad) moral character?
- Is character relevant in the dialogue?
- Is the weight of presumption claimed strongly enough warranted by the evidence given?

```
- id: ethotic1
  variables: [P,S]
  premises:
    - asserts(P,S)
    - good_moral_character(P)
  conclusions:
    - S
  assumptions:
    - character_is_relevant(S)

  id: ethotic2
  variables: [P,S]
  premises:
    - asserts(P,S)
    - bad_moral_character(P)
  conclusions:
    -  ¬S
  assumptions:
    - character_is_relevant(S)
```

The first scheme for ethotic arguments, `ethotic1`, has a second-order variable, S, as its conclusion. The conclusion $\neg S$, of the second scheme, `ethotic2`, however, is, a first-order atomic proposition, with $\neg$ being a unary predicate symbol, not a logical operator. Neither CHR nor Carneades has a built-in negation operator, but in Carneades all pairs of atomic propositions of the form P and $\neg P$ can be declared, in a single declaration, to be logical complements, which cannot both be accepted (in) in the same argument graph. If this is done, `ethotic1` and `ethotic2` can be used to construct rebuttals.

4.10. Argument from expert opinion

Premises

- Source E is an expert in subject domain S containing proposition A.
- E asserts that proposition A is true.

Conclusions

- *A* is true.

Critical Questions

- How credible is *E* as an expert source?
- Is *E* an expert in the field that *A* is in?
- What did *E* assert that implies *A*?
- Is *E* personally reliable as a source?
- Is *A* consistent with what other experts assert?
- Is *E*'s assertion based on evidence?

```
id: expert_opinion
variables: [W,D,S]
premises:
  - expert(W,D)
  - in_domain(S,D)
  - asserts(W,S)
conclusions:
  - S
exceptions:
  - untrustworthy(W)
  - inconsistent_with_other_experts(S)
assumptions:
  - based_on_evidence(asserts(W,S))
```

4.11. Argument from ignorance

This version of the scheme from ignorance is in the 2008 compendium [25], but the two critical questions of the scheme were added later, in [24].

Premises

- If *A* were true, then *A* would be known to be true.
- It is not the case that *A* is known to be true.

Conclusions

- Therefore, *A* is not true.

Critical Questions

- How deep has the search been?
- How deep does the search need to be in order to prove the conclusion that A is false to the required standard of proof in the investigation?

```
id: ignorance
variables: [A]
premises:
  - would_be_known(A)
  - ¬known(A)
conclusions:
  - ¬A
exceptions:
  - uninvestigated(A)
```

In the reconstruction, the two critical questions are modeled by a single exception, where the `uninvestigated` predicate is intended to mean that the statement A has not been *sufficiently* investigated.

4.12. Argument from negative consequences

Premises

- If A is brought about, then bad consequences will occur.

Conclusions

- Therefore, A should not be brought about.

Critical Questions

- How strong is the likelihood that the cited consequences will (may, must) occur?
- What evidence supports the claim that the cited consequences will (may, must) occur, and is it sufficient to support the strength of the claim adequately?
- Are there other opposite consequences (bad as opposed to good, for example) that should be taken into account?

```
id: negative_consequences
variables: [A,G]
premises:
  - brings_about(A,G)
  - bad(G)
conclusions:
  - ¬should_be_performed(A)
```

The critical questions are not modeled as exceptions or assumptions in the reconstruction. The first critical question asks about the weight of the argument. Weighing arguments is built in to the model of structured argument we are applying [12] and applies to all schemes, without the need for explicit exceptions or assumptions. We have interpreted the second critical question as merely challenging the first premise. Every premise can be questioned in our model of structured argument, without needing to explicitly enumerate critical questions for each premise. Alternatively, this critical question could have been interpreted as meaning that the premise is actually an assumption, not requiring proof until the question is asked. The final critical question asks whether there are any rebuttals. Rebuttals too are handled directly by the model of structured argument, without needing to add an explicit critical question asking for possible rebuttals to each scheme.

4.13. Argument from position to know

Premises

- Source a is in position to know about things in a certain subject domain S containing proposition A.
- a asserts that A is true (false).

Conclusions

- A is true (false).

Critical Questions

- Is a in position to know whether A is true (false)?
- Is a an honest (trustworthy, reliable) source?
- Did a assert that A is true (false)?

```
id: position_to_know
variables: [W,S,A]
premises:
  - position_to_know(W,S)
  - in_domain(A,S)
  - asserts(W,A)
conclusions:
  - A
exceptions:
  - dishonest(W)
```

In the reconstruction, the source is denoted by W (for witness) instead of a, because schema variables must begin with an uppercase letter and the variable A is already used, to denote the statement being asserted. The first premise of the source version is split into two premises in the reconstruction. The first states that W is in a position to know things in the domain S. The second states that A is in the domain S.

Only one of the critical questions is explicitly modeled in the reconstruction, with an exception asking whether W is dishonest, modeling the second critical question in the source version. The other two critical questions simply ask whether the premises are true.

4.14. Argument from positive consequences

Premises

- If A is brought about, good consequences will plausibly occur.

Conclusions

- Therefore, A should be brought about.

Critical Questions

- How strong is the likelihood that the cited consequences will (may, must) occur?

- What evidence supports the claim that the cited consequences will (may, must) occur, and is it sufficient to support the strength of the claim adequately?
- Are there other opposite consequences (bad as opposed to good, for example) that should be taken into account?

```
id: positive_consequences
variables: [A,G]
premises:
  - brings_about(A,G)
  - good(G)
conclusions:
  - should_be_performed(A)
```

The critical questions of the scheme for argument from positive consequences are the same as for the scheme for arguments from negative consequences. The reasons for not including exceptions or assumptions for these critical questions in the reconstruction are the same.

4.15. Argument from practical reasoning

Here we present two schemes for practical reasoning, from Atkinson and Bench-Capon [2] and Walton et al. [25]. We show how to represent both, simply because they have both been influential in the literature.

Atkinson and Bench-Capon's value-based version is:

In the current circumstances R
We should perform action A
Which will result in new circumstances S
Which will realise goal G
Which will promote some value V.

Notice that Atkinson and Bench-Capon do not distinguish premises and conclusions of the scheme. They do however list critical questions:

- Are the believed circumstances true?
- Assuming the circumstances, does the action have the stated consequences?
- Assuming the circumstances and that the action has the stated consequences, will the action bring about the desired goal?
- Does the goal realise the value stated?
- Are there alternative ways of realising the same consequences?
- Are there alternative ways of realising the same goal?
- Are there alternative ways of promoting the same value?
- Does doing the action have a side effect which demotes the value?
- Does doing the action have a side effect which demotes some other value?
- Does doing the action promote some other value?
- Does doing the action preclude some other action which would promote some other value?
- Are the circumstances as described possible?
- Is the action possible?
- Are the consequences as described possible?

- Can the desired goal be realised?
- Is the value indeed a legitimate value?

Here is are reconstruction of Atkinson and Bench-Capon's scheme for practical reasoning:

```
id: value_based_practical_reasoning
variables: [A,S1,S2,G,V]
premises:
  - current_circumstances(S1)
  - would_bring_about(A,S1,S2)
  - would_be_realized(G,S2)
  - would_promote_value(G,V)
conclusions:
  - should_be_performed(A)
assumptions:
  - legitimate_value(V)
  - worthy_goal(G)
  - possible(A)
exceptions:
  - bring_about_more_effectively(S2,A)
  - realize_more_effectively(G,A)
  - promote_more_effectively(V,A)
  - side_effects(A,S1)
```

Now, here is Walton's scheme for instrumental practical reasoning, called "practical inference" in [25]:

Premises

- I have a goal G.
- Carrying out this action A is a means to realize G.

Conclusions

- Therefore, I ought (practically speaking) to carry out this action A.

Critical Questions

- What other goals that I have that might conflict with G should be considered?
- What alternative actions to my bringing about A that would also bring about G should be considered?
- Among bringing about A and these alternative actions, which is arguably the most efficient?
- What grounds are there for arguing that it is practically possible for me to bring about A?
- What consequences of my bringing about A should also be taken into account?

```
id: instrumental_practical_reasoning
variables: [A,S1,S2,G]
premises:
  - current_circumstances(S1)
  - would_bring_about(A,S1,S2)
  - would_be_realized(G,S2)
conclusions:
  - should_be_performed(A)
assumptions:
  - possible(A)
  - possible(G)
exceptions:
  - bring_about_more_effectively(S2,A)
  - realize_more_effectively(G,A)
  - intervening_actions(A,G)
  - side_effects(A,S1)
  - incompatible_goal(G)
```

4.16. *Argument from precedent*

The version of argument from precedent we have chosen is from [23]. The critical questions included here are new. They were not explicated in [23].

Premises

- C_1 is a previously decided case.
- In case C_1, rule R was applied and produced finding F.
- C_2 is a new case that has not yet been decided.
- C_2 is similar to C_1 in relevant respects.

Conclusions

- Rule R should be applied to C_2 and produce finding F.

Critical Questions

- There relevant differences between C_1 and C_2.
- Rule R is not applicable in C_2.

```
id: precedent
variables: [F,C,R]
premises:
  - similar_case(C)
  - rule_of_case(R,C)
  - has_conclusion(R,F)
conclusions:
  - F
exceptions:
  - relevant_differences(R,F)
  - inapplicable_rule(R)
```

In our reconstruction of the argument from precedent, the second case, C_2, representing the current case, is left implicit. This is because we want the conclusion to be that F is true, rather than F is true in case C_2, so that the argument can be used to support arguments having F as a premise. Since there is only one case mentioned in the reconstruction, we have renamed C_1 to C. We have reduced the four premises to three, by combining the first and fourth premises into a single premise, the first in the reconstruction, meaning that C is a previously decided case which is similar to the current case.

4.17. Slippery slope arguments

Premises

- A_0 is up for consideration as a proposal that seems initially like something that should be brought about.
- Bringing up A_0 would plausibly lead (in the given circumstances, as far as we know) to A_1, which would in turn plausibly lead to A_2, and so forth, through the sequence $A_2, \dots, A_n$.
- A_n is a horrible (disastrous, bad) outcome. Conclusion: A_0 should not be brought about.

Conclusions

- A_0 should not be brought about.

Critical Questions

- What intervening propositions in the sequence linking up A_0 with A_n. are actually given?
- What other steps are required to fill in the sequence of events, to make it plausible?
- What are the weakest links in the sequence, where specific critical questions should be asked on whether one event will really lead to another?

```
id: slippery_slope1
variables: [A,E]
premises:
  - would_realize(A,E)
  - horrible_costs(E)
conclusions:
  - negative_consequences(A)
```

```
id: slippery_slope2
variables: [E1,E2]
premises:
  - causes(E1,E2)
  - horrible_costs(E2)
conclusions:
  - horrible_costs(E1)
```

Notice that our reconstruction of the slippery slope scheme splits the scheme into two, where the second scheme represents an inductive step in a recursive argument for proving that events which cause events with horrible costs also have, indirectly, horrible costs. The base case in such a recursive argument would be covered by facts or assumptions stating that some particular events have horrible costs.

The reconstruction does not explicitly model the critical questions, using exceptions and assumptions. The first two critical questions are handled by the recursive (second) scheme, which chains together a sequence of events. The third critical question merely attacks the causality premise of the second scheme. Premise attacks are built into the model of structured argument and do not require critical questions to be expressed or represented.

4.18. Argument from sunk costs

The version of the scheme for argument from sunk costs below is from [25], except for the critical questions, which are new. No critical questions for the scheme were formulated in [25].

In the following scheme, let t_1 be the time of the proponent's commitment to a certain action (pre-commitment) and t_2 be the time of proponent's confrontation with the decision of whether to carry out the pre-commitment or not.

Premises

- There is a choice at t_2 between A and $\neg A$.
- At t_2 I am precommitted to A because of what I did or committed myself to at t_1.

Conclusions

- Therefore, I should choose A.

Critical Questions

- Is there some hope of completion of the course of action?
- Should the projected future losses of continuing this course of action outweigh the value of my commitment to continuing with it?
- Are my prior commitments important enough to warrant continuing this course of action, even though continuing might not lead to success in the future?
- Are cost benefit calculations applicable?

```
id: sunk_costs
variables: [A,C]
premises:
  - sunk_costs(A,C)
  - too_high_to_waste(C)
conclusions:
  - should_be_performed(A)
assumptions:
  - feasible(A)
exceptions:
  - future_losses_outweigh(A)
```

In the reconstruction, we have not modeled the first premise, which states there is a choice to be made between A and $\neg A$. This premise only makes explicit that there is a choice to be made between accepting and not accepting the conclusion of the scheme. Such premise could be added to every scheme, but is not really necessary or useful, since this choice is built-in to the underlying framework or logic of structured argumentation. The second premise, about being precommitted to the action, has been split into two in the reconstruction, one fixing the amount of the sunk costs, C, and the other claiming that these costs are too high to waste.

The first critical question, about there being some hope to complete the course of action, has been modeled in the reconstruction as an assumption, `feasible(A)`, which is intended to mean that it is still feasible to complete the course of action. The other critical questions have been combined into a single exception in the reconstruction, `future_losses_outweigh(A)`, which is intended to mean that projected future losses outweigh the sunk costs and the value of completing the project, in a cost-benefit analysis.

4.19. Argument from verbal classification

Premises

- a has property F.
- For all x, if x has property F, then x can be classified as having property G.

Conclusions

- a has property G.

Critical Questions

- What evidence is there that a definitely has property F, as opposed to evidence indicating room for doubt about whether it should be so classified?
- Is the verbal classification in the classification premise based merely on an assumption about word usage that is subject to doubt?

```
id: verbal_classification
variables: [O,F,G]
premises:
  - instance(O,F)
  - subclass(F,G)
conclusions:
  - instance(O,G)
```

4.20. Argument from witness testimony

Premises

- Witness W is in a position to know whether A is true or not.
- Witness W is telling the truth (as W knows it).
- Witness W states that A is true (false).

Conclusions

- A is true (false).

Critical Questions

- Is what the witness said internally consistent?

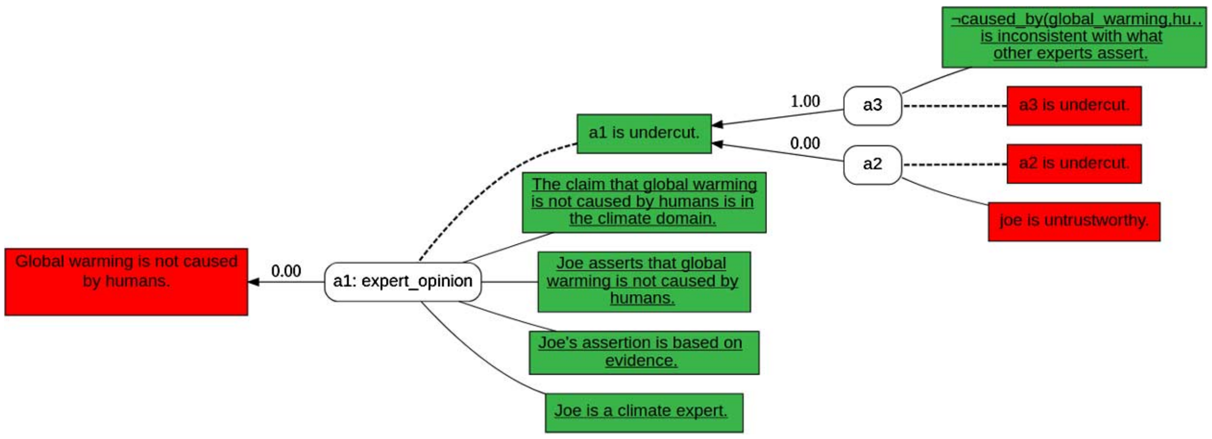
¬caused_by(global_warming,hu...
is inconsistent with what
other experts assert.
a3
1.00
a3 is undercut.
a1 is undercut.
0.00
a2
a2 is undercut.
joe is untrustworthy.
The claim that global warming is not caused by humans is in the climate domain.
Global warming is not caused by humans.
0.00
a1: expert_opinion
Joe asserts that global warming is not caused by humans.
Joe's assertion is based on evidence.
Joe is a climate expert.

Fig. 1. Global warming example.

- Is what the witness said consistent with the known facts of the case (based on evidence apart from what the witness testified to)?
- Is what the witness said consistent with what other witnesses have (independently) testified to?
- Is there some kind of bias that can be attributed to the account given by the witness?
- How plausible is the statement *A* asserted by the witness?

```
id: witness_testimony
variables: [W,D,S]
premises:
  - position_to_know(W,D)
  - in_domain(D,S)
  - believes(W,S)
  - asserts(W,S)
conclusions:
  - S
assumptions:
  - internally_consistent(S)
exceptions:
  - inconsistent_with_known_facts(S)
  - inconsistent_with_other_witnesses(S)
  - biased(W)
  - implausible(S)
```

Figure 1 shows a simple example of how these schemes can be used by Carneades 4 to automatically generate, evaluate and visualize an argument graph, given the following assumptions:

```
- expert(joe, climate)
- asserts(joe,¬caused_by(global_warming,humans))
- in_domain(¬caused_by(global_warming, humans), climate)
- inconsistent_with_other_experts(¬caused_by(global_warming, humans))
```

5. Conclusions

Our experiments with using Constraint Handling Rules (CHR) to represent argumentation schemes for the purpose of generating arguments have been encouraging.

We have successfully implemented twenty representative argumentation schemes [25], including their critical questions.[10] Nine of these twenty schemes, about half, have conclusions which are second-order variables. Only one of the schemes, defeasible modus ponens, has a second-order variable as a premise. Our implementation of CHR has been extended to allow second-order variables in the premises of schemes.

Using CHR as a foundation for implementing argumentation schemes provided us with an opportunity to extend the concept of an argumentation scheme in various ways, to make it possible to represent any CHR rule as an argumentation scheme. This method for representing and implementing argumentation schemes inherits all of the attractive features of CHR, including Turing completeness, the possibility of concurrent execution, support for stopping and restarting computation at any time, with intermediate results available for use, and support for inputting further information incrementally during dialogues and other argumentation processes.

Conversely, the synthesis of CHR and argumentation provided by Carneades provides additional benefits not provided by CHR alone. CHR has no concept of negation. Carneades issues can model negation or, more generally, a set of conflicting positions of issues. Moreover, CHR provides no built-in support for defeasible reasoning. We use CHR to generate pro and con arguments, which are then evaluated in a post-process, using a model of structured argument, to support defeasible reasoning by weighing and balancing arguments and resolving attack relations among arguments. Most importantly, our system produces arguments which can be used to explain and understand CHR inferences, for example by visualizing the arguments in argument maps.

While the method presented here for generating arguments using Constraint Handling Rules was developed for the latest version of the Carneades model of structured argument [12], it can be adapted for use in any model of argument in which arguments are constructed by instantiating argumentation schemes. We leave it for future research by others to adapt the method to other models of structured argument.

Acknowledgements

This work was partially funded by the Social Sciences and Humanities Research Council of Canada (SSHRC), in the Carneades Argumentation System project (Insight Grant 435-2012-0104), and by Microsoft, in the DUCK project.

References

[1] S. Abdennadher, T. Frühwirth and C. Holzbaur, Introduction to the special issue on constraint handling rules, *Theory and Practice of Logic Programming* **5**(4–5) (2005), 401–402. doi:10.1017/S1471068405002346.

[2] K. Atkinson and T.J.M. Bench-Capon, Practical reasoning as presumptive argumentation using action based alternating transition systems, *Artificial Intelligence* **171**(10–15) (2007), 855–874. doi:10.1016/j.artint.2007.04.009.

[10] https://github.com/carneades/carneades-4/blob/master/examples/AGs/YAML/walton.yml

[3] T. Bench-Capon, K. Atkinson and A. Wyner, Using argumentation to structure e-participation in policy making, in: *Transactions on Large-Scale Data-and Knowledge-Centered Systems XVIII*, Springer, 2015, pp. 1–29.
[4] T. Bench-Capon and M. Sergot, Toward a rule-based representation of open texture in law, in: *Computer Power and Legal Language*, C. Walther, ed., 1988, pp. 39–60.
[5] L. Brownston, R. Farrell, E. Kant and N. Martin, *Programming in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley Longman, Boston, 1985.
[6] B.G. Buchanan and E.H. Shortliffe (eds), *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley, Reading, Mass., 1984.
[7] W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer-Verlag, 1981.
[8] P.M. Dung, R.A. Kowalski and F. Toni, Assumption-based argumentation, in: *Argumentation in Artificial Intelligence*, I. Rahwan and G.R. Simari, eds, Springer, 2009, pp. 199–218. doi:10.1007/978-0-387-98197-0_10.
[9] T. Frühwirth, *Constraint Handling Rules*, Cambridge University Press, 2009. doi:10.1017/CBO9780511609886.
[10] T.F. Gordon, Constructing arguments with a computational model of an argumentation scheme for legal rules – Interpreting legal rules as reasoning policies, in: *Proceedings of the Eleventh International Conference on Artificial Intelligence and Law*, ACM Press, 2007, pp. 117–121.
[11] T.F. Gordon, H. Prakken and D. Walton, The Carneades model of argument and burden of proof, *Artificial Intelligence* **171**(10–11) (2007), 875–896. doi:10.1016/j.artint.2007.04.010.
[12] T.F. Gordon and D. Walton, Formalizing balancing arguments, in: *Proceeding of the 2016 Conference on Computational Models of Argument (COMMA 2016)*, IOS Press, 2016, pp. 327–338. doi:10.3233/978-1-61499-686-6-327.
[13] P. Hammond, APES: A user manual, Technical report, 1983.
[14] R. Kowalski and M. Sergot, A logic-based calculus of events, *New Generation Computing* **4** (1986), 67–95. doi:10.1007/BF03037383.
[15] E.S.L. Lam and M. Sulzmann, A concurrent constraint handling rules implementation in Haskell with software transactional memory, in: *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, ACM, New York, NY, USA, 2007, pp. 19–24. ISBN 978-1-59593-690-5. doi:10.1145/1248648.1248653.
[16] J.L. Pollock, Defeasible reasoning in OSCAR, The Computing Research Repository, 2000, https://arxiv.org/pdf/cs/0003012.
[17] H. Prakken, An abstract framework for argumentation with structured arguments, *Argument & Computation* **1** (2010), 93–124. doi:10.1080/19462160903564592.
[18] C. Reed and G. Rowe, Araucaria: Software for puzzles in argument diagramming and XML, Technical Report, Department of Applied Computing, University of Dundee, 2001.
[19] M. Snaith and C. Reed, TOAST: Online ASPIC+ implementation, *COMMA* **245** (2012), 509–510.
[20] J. Sneyers, T. Schrijvers and B. Demoen, The computational power and complexity of Constraint Handling Rules, *ACM Trans. Program. Lang. Syst.* **31**(2) (2009), 8:1–8:42, ISSN 0164-0925. doi:10.1145/1462166.1462169.
[21] B. Verheij, *Virtual Arguments*, TMC Asser Press, The Hague, 2005. doi:10.1007/978-90-6704-661-9.
[22] D. Walton, An overview of the use of argumentation schemes in case modeling, in: *Modelling Legal Cases*, K. Atkinson, ed., Huygens Editorial, 2009, pp. 77–89.
[23] D. Walton, Similarity, precedent and argument from analogy, *Artificial Intelligence and Law* **18**(3) (2010), 217–246. doi:10.1007/s10506-010-9102-z.
[24] D. Walton, Argument mining by applying argumentation schemes, *Studies in Logic* **4**(1) (2011), 38–64.
[25] D. Walton, C. Reed and F. Macagno, *Argumentation Schemes*, Cambridge University Press, 2008. doi:10.1017/CBO9780511802034.