A Computational Framework for Dialectical Reasoning

Pierre St-Vincent[1], Daniel Poulin[2] and Paul Bratley[1]

[2]Centre de recherche en droit public
poulind@droit.umontreal.ca

[1]Département d'informatique et de recherche opérationnelle
{stvincen, bratley}@iro.umontreal.ca

Université de Montréal
c.p. 6128, Succursale A
Montréal (Québec)
Canada H3C 3J7

Abstract

Dialectics are important not only in law but in every domain where knowledge is not certain; that is, everywhere assumptions must be made. After a review of recent advances in computational dialectics and related fields, we present the framework of a system for constructing dialectical arguments from a rule-based representation of law.

In this system, meta level reasoning serves to allow for multiple utilisations of the rules. At the object level, rules grouped in modules represent "ground" knowledge. At the meta level, modules contain meta level rules that query other modules, at the object level or at some meta level, for arguments. During the construction of arguments, meta level rules use a filtering mechanism that works like simple regular expressions. This mechanism selects lower level rules according to their contexts.

The object rules of the system are marked with interpretative contexts to permit varying points of view while maintaining an isomorphic representation of knowledge. The rules can be preceded by explicit negation, and the presence of contradictory rules allows conflicting arguments to be built. Examples are given and a discussion of future work concludes the paper.

1. Introduction

There is nothing new in a dialectical approach to knowledge. Since ancient times, the art of dialectics has been allied to that of logic to add conviction to an argument, rather than merely proving it formally. For Plato, dialectics was the art of discussion by question and answer; for Aristotle, arguments based on uncertain premisses, that could be debated and defeated, were dialectical in nature. Since Hegel dialectics has been seen rather as an approach to reasoning that recognizes the inseparable nature of contradictory elements that unite in a final synthesis. Logic and dialectics both support one another and oppose one another. Both aim to ensure correct ways of thinking, but while the former limits itself to formal truth, the latter tries to convince, to bring the hearer not merely to understand, but to agree.

Legal reasoning seems to partake more of dialectics than of logic. A lawyer does not feel himself confined to a single reading of his texts that he would propose in every case. Depending on his client, and depending on his objective, he may allow himself the necessary dialectic liberty of giving more weight to some point of view which, while not the most evident, is nevertheless plausible, and which, above all, advances the interests he represents. This is by no means to say that he is free to advance any view he chooses, no matter how unlikely, but that among several possible defensible lines of reasoning he will choose the one that serves his purposes. Sometimes it is possible to defend different readings of legal rules; at others different interpretations of the facts may give an advocate the required room for manœuvre; but he will never be reduced to silence. Despite being themselves trained in such dialectical exercises, lawyers are sometimes surprised to come across them in other disciplines, even those deemed to be among the most positivist of the sciences. Thus in cases concerning patents, for example, the most eminent jurists can be astonished to encounter an old friend: "How is it that, invariably, and in a discipline and a procedure that is supposed to be both scientific and objective, there are always two independent, eminent, and experienced experts who completely disagree with each other?" [Henderson 94, p.2] It is simple to explain this ubiquity of dialectics: no doubt it is true that whenever points of view diverge and arguments arise, then a dialectic process is inevitable. This simple truth must necessarily be taken into account in any attempt to model legal reasoning.

A number of pioneering systems that attempted to model legal reasoning were heavily influenced by logic [Smith 87, Susskind 87]; sometimes logic was even proposed as the natural language to use to express legal norms [Sergot 86]. Little by little, however, researchers began to explore other avenues that allow them to express more easily different approaches to such norms. Rissland and Ashley were among the first to understand the importance of being able to express the alternate lines of reasoning that characterize legal arguments. Their systems of case-based reasoning have always tried to model the different approaches accepted by the courts, whether in the area of commercial secrets, tax exemptions, or elsewhere [Rissland 87, Ashley 90].

Quite recently a number of approaches have been suggested that aim to go beyond the single-mindedness of the early systems, and to extend the dialectical capabilities of case-based reasoning systems to the wider class of legal expert systems in general. Many of the researchers involved have favoured using some form of default logic, while others rely on meta reasoning.

For instance, Prakken proposed a logical system involving the use of Reiter's normal defaults [Prakken 93]. It allows for ordering conflicting arguments by comparing them using, for instance, lex superior, lex posterior or lex specialis. When many defaults are applicable to derive an argument they can in turn be ordered, if necessary. This scheme is extended to combine multiple orderings and even the meta-level production of orderings.

Similarly Brewka [Brewka 94] uses a prioritized default logic based on Reiter's logic. It can implement explicit partial orders by defining an operator ("<") that can be used either in the formulas or in the defaults. Its scheme is thus very general. For example, it allows defaults to be specified in the ordering of other defaults.

Gordon has studied a different problem, namely identifying the legal points involved in an argument [Gordon 93]. His system uses another nonmonotonic logic, the "conditional entailment" of Geffner and Pearl [Geffner 92]. It can order conflicting rules either using an "automatic" measure of specificity or using defaults to encode the priorities. Gordon builds upon this logic to implement the "Pleadings Game" in which two players, the plaintiff and the defendant, argue about a legal problem while respecting Alexy's rules of rationality [Alexy 89]. A "formalization of Toulmin's theory of practical argumentation" the system allows for an identification of the issues of a problem.

Schobbens too has designed a logic that allows him to take account of the reliability of witnesses by ranking their testimony, and to consider the priority of higher-level laws (lex superior), and to make initial assumptions [Schobbens 93]. Hage also has presented a system able to solve conflicts between rules [Hage 93].

Sartor uses meta reasoning in logic programming. He wants to "extend formal methods outside the domain of deduction, to the moments of dialectical conflict – and therefore of choice and evaluation – which characterize legal and moral reasoning." [Sartor 94, p. 178] His system allows for justified preferences, exceptions, and so on. It uses contexts to "name" the rules and identify them. Following the Prolog execution scheme, the system builds arguments that can defeat one another depending on information represented by priority rules. In this way, priorities being represented by "normal" rules, can be defeated in their turn. Only "success" arguments are producible by this system; it does not implement so-called "negation by failure" and the system has only one level.

As far as meta reasoning is concerned, Hamfelt was the first to propose a system using several levels of knowledge to model the action of interpretative rules when applied to the substantive legal rules [Hamfelt 89]. His system is different from those where the role of the meta-rules is to implement mechanisms for control or introspection. Instead, the meta-rules (the rules of interpretation) are used to modify the rules at the lower level (the substantive legal rules). It remains to be seen whether it is possible to write meta-rules able to generate automatically legally acceptable variations of the substantive rules. Schild too has described a system using meta-knowledge to create rules. Here the generated rules are intended to implement substantive rules that use "vague" predicates. In this system, whenever such a predicate is not defined by other rules, a meta-rule is triggered to create new rules, based on comments found in the pertinent jurisprudence (the "obiter"). These new rules are supposed to apply to the factual situation being considered [Schild 93]. Poulin proposed using meta-rules in a more general way. In his system, the meta-rules represent four kinds of knowledge: general, procedural, adversarial and inferential. The rules of the field in question, relieved of these considerations, can then be expressed in a declarative fashion. More importantly, contradictory rules can now coexist at the object level. Conflicts between rules are resolved by meta-rules that model the interpretative techniques used by lawyers [Poulin 93a; 93b].

The following sections present a computational framework to implement the type of model proposed by Poulin. We shall be particularly interested in the production of coherent arguments in a setting where the object level rules may be contradictory. Throughout the following sections we give examples of rules and meta-rules that illustrate the computations involved. It is worth emphasising, to avoid misunderstanding, that these examples are intended to exercise the system being built or to illustrate a point, not to represent genuine legal situations.

2. Overview

The principal goal of the argument producing system presented here is to allow us to build experimental dialectical systems that can be easily modified to test different research hypotheses. The system produces *arguments*, that is, lines of reasoning to support a desired conclusion. For ease of modification, a fixed scheme is not in order. Thus we designed a programming language that is an extension of Prolog, and a runtime support system. Only the core of the language is described in this paper. Its implementation is now almost complete.

Arguments produced by the system are like proof trees augmented with contextual annotations. However in case of failure, where a proof tree would be empty and useless, an argument explains the reason for the failure and shows those parts of the proof that succeeded, if any.

The language is based on metaprogramming. This clears the way to using a purely declarative representation of object level knowledge, as proposed by Poulin [Poulin 93a]. On the other hand the production of arguments is mostly procedural, and the rules for this are placed at some meta level. There can be any number of meta levels. The rules themselves differ whether they are at some meta level or at the object level. At the object level the rules should mirror the if-then structure of the rules of law [Sergot 86]; further, they should be as isomorphic as possible to the sources [Bench-Capon 92]. At the meta levels the rules should allow us to implement the general, procedural, adversarial and inferential expertise of legal thinking and judicial practice [Poulin 93a, 93b, 93c].

Whether object or meta, rules are marked with contexts. The context of a rule can represent a variety of meta-knowledge. This includes:

- The factual basis of the knowledge represented by the rule. This basis may indentify such things as the source of law originating the rule. It could also indicate the means by which the knowledge was obtained, when and where it was obtained, and by whom. For instance, something might have been learned by eavesdropping on cellular phone frequencies.
- The theory of the particular field involved. In the legal field, this includes the interpretative method used in deriving the rule from some legal source [Wroblewski 88; Bergel 89; Du Pasquier 79].
- Claims that may or may not be true. This will be illustrated later.

A program is a set of *modules*, some at the object level, others at meta levels. Using modules permits us to structure the rule bases by grouping related rules; this grouping restricts the search for matching rules and renders it more manageable for the programmer. It also generalizes the rule bases by allowing us to reuse common parts in different situations.

Object level modules (hereafter simply "object modules") begin with `begin_object` and end with `end_object` statements, thus:

```
begin_object(sample_object).
   ...
end_object.
```

Object level modules contain only object level rules.

Meta level modules (hereafter simply "meta modules") usually take other modules as parameters. As illustrated in the example below, they begin with `begin_meta` statements where their formal parameters are declared:

```
begin_meta(sample_meta(parameter1, parameter2)).
   ...
end_meta.
```

The language implements a subtask management architecture [van Harmelen 89]. Meta modules do their own inferencing. When necessary they call other modules, either meta or object, asking them to supply arguments. The called modules then build and return arguments to their callers.
The modules that constitute a rule base, a "program" so to speak, thus form an oriented graph where the links are the requests and the nodes are the modules themselves. Cycles may appear in this graph when requests make recursive calls. The starting point of the rule base is a special module, named `top`, that contains all the entry points. Since `top` has no parameters its requests can resolve at run-time all the module references encountered. This is how module expressions containing formal parameters are translated to expressions containing only the names of meta modules or object modules.

The entry points are shown in menus so the user can initiate his own requests. These entry points are defined with `user` rules like the following:

```
begin_meta(top).
   /* somewhere inside "top" we could have this rule: */
   user("menu entry label") :-
       some_context => some_request in some_module.
end_meta.
```

Choosing the menu item `menu entry label` begins execution of the request. The system will then construct arguments for the chosen goal, putting questions to the user when it encounters "user askable" facts. The resulting argument trees, whether successes or failures, are finally drawn on screen.

For the experienced user a top level is also available that allows immediate execution of any acceptable request.

3. The object Level Rules

At the object level the rules are like pure Prolog clauses augmented by contexts and explicit negation. The body of a rule consists of conjunctions and disjunctions of goals which can be explicitly negated. The head of a rule consists of two parts, a context and a goal, separated by the '`=>`' operator:

```
  context => goal  :-    conjunction-disjunction.
 |___________|_____|    |_________________________|

      head                         body
```

Any Prolog term, whether ground (i.e. variable free) or not, can be used as a context. For example, suppose the following facts and rules are in three different modules (the syntactic elements that delimit the modules are not shown):

```
/* module "claims", fact f1: */
claims(nixon) => pacifist(nixon).
```

```
/* module "peacocks_trial", fact f2: */
testimony("Miss Jameson", date(24, april, 1910)) =>
   on(butcher_knife, floor_of(kitchen)).
/* module "unemployment_obj", rule r3: */
interp_context(art(28,4,b),
              textual(ordinary_meaning),
              source(cub(45))) =>
   spouse_moving_exception(CLAIMANT, SPOUSE) :-
      military(SPOUSE),
      new_military_posting(SPOUSE).
```

The first fact is an instance of a simple claim, namely that "Nixon claims he is a pacifist". The second fact shows that "The butcher's knife was on the kitchen floor" is part of the testimony given by Miss Jameson on April 24th 1910. It gives an example of a factual context, i.e. that the rule comes from someone's testimony. This implies that the fact `on(butcher_knife, floor_of(kitchen))` can be used in inferences as long as Miss Jameson's testimony is legally valid. Rule `r3` (adapted from [Poulin 93b]) is accompanied by its interpretative context: its source is Canadian Unemployment Board decision number 45 and this decision follows a textual reading [MacCormick 91] of article 28)4¬b of the Unemployment Act, based on the ordinary meaning of words.

Contexts can also be used to represent the experts' opinions with which a rule is associated [Freeman 94]. For instance, suppose we have two divergent opinions as to whether or not a Greek sculpture is a forgery (here `r5` uses the explicit negation operator "¬"):

```
theory_of(professor(alpha)) =>
   forgery(korê) :-
      color(marble, pink),
      classical(proportions),
      sharp(chisels).
theory_of(professor(beta)) =>
   ¬ forgery(korê) :-                /* r5 */
      color(marble, pink),
      classical(proportions),
      sharp(chisels),
      origin(athens).
```

The first rule reads: "According to Professor Alpha, a korê is a forgery if its marble is pink, its proportions are classical and it has been sculpted with sharp chisels," while the second reads: "According to Professor Beta, a korê is not a forgery if its marble is pink, its proportions are classical and it has been sculpted with sharp chisels BUT it originates from Athens."

Which argument will be preferred depends on the meta level used. In cases like this, where opinions are given, the most specific argument is not necessarily the best.

We can generalize from ground contexts, used in the preceding examples, by using variables. For instance, Nixon's claim `f1` in our first example could be generalized as follows with the variable `EVERYBODY`:

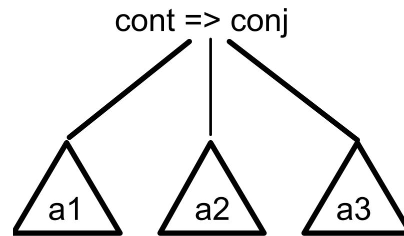
```
claims(EVERYBODY) => pacifist(nixon).
```

This means, obviously, that everybody claims that Nixon is a pacifist.

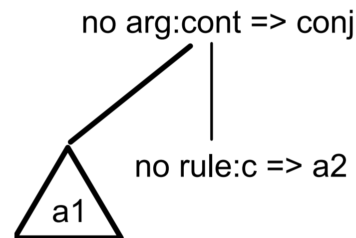
The rules produce arguments much like proof trees. (For a meta interpreter that explains its reasoning whether in case of success or failure, see [Yalçinalp 89]). In case of success the conjunct:

```
cont => conj :- a1, a2, a3.            /* r10 */
```

will produce the following argument, where the subtrees produced by `a1, a2` and `a3` are shown inside triangles and "cont => conj" is the root marked with its context:



In case of failure the same conjunct will produce an argument showing the parts that have succeeded and the cause of the failure, following the execution scheme of Prolog. For instance, suppose `a1` succeeds but that there are no rules for `a2`, so `a2` fails. We would then obtain the following argument, where the tags `no arg` (for "`NO ARGument`") and `no rule` identify failure nodes:



By inspecting this tree an experienced user can see which parts have been successful up to some point and what is the cause of the failure. As the tree suggests, there is a compromise made here: whereas it shows the success of `a1`, no effort is made to determine the eventual success of `a3`. This is a compromise between exploring all possible paths to maximize the argument trees, and annoying the user with questions that cannot possibly lead to success.

So-called negation by failure is implemented by reversing failed arguments. Such failures then naturally become successes of their counterparts.

While this section has concentrated on object level rules, meta level rules produce argument trees in a similar way, as we will see in the following section.

4. Meta level rules, Requests and Filters

Meta level rules have the same general structure as object level rules. The difference lies in the fact that their bodies are conjunctions and disjunctions of *requests* instead of "ordinary", Prolog-like goals. Requests implement the search for arguments. They are calls made from meta level modules to other modules asking them to provide arguments in favour of some specific goal. The requests use *filtering expressions* while constructing arguments to choose between available rules according to their contexts. The important point is that arguments are not selected by the filtering expressions after their production but at each step during it.

In our system requests are like the "demo" predicate usual in meta programming since Bowen and Kowalski's early work [Bowen 82]. A request consists of three parts, namely a filtering expression, a goal and a module expression:

```
   cont(1), cont(2) =>  goal       in  meta(some_module).

filtering expression    goal          module expression
```

When we place a request in a meta level rule it looks like this:

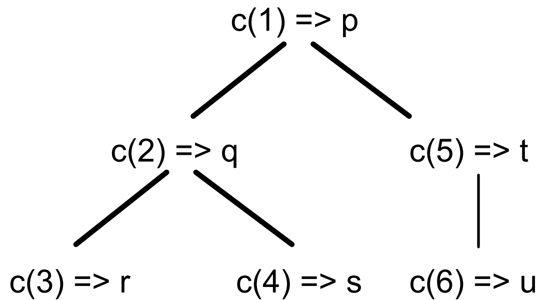
```
rule_context => rule_name :-
   request_filtering_exp => goal in module.
```

Filtering expressions (or "filters" for short) are like simple regular expressions without parentheses. The absence of parentheses implies that they are free of nested loops that would unduly complicate their use and implementation. The main difference between regular expressions and filtering expressions lies in the presence of Prolog variables in the latter. This means that unifications can occur when they are used. Where regular expressions accept strings of characters, filtering expressions accept sequences of contexts of arguments under construction.

Define the *context sequences* of an argument A to be:

$Cs_i(A)$ = the sequence of contexts along the i-th branch of A

where the i-th branch is the path from root to i-th leaf (counted from left to right). For example, let A be the following argument which has three branches.



Its context sequences are: $Cs_1(A) = [c(1), c(2), c(3)]$, $Cs_2(A) = [c(1), c(2), c(4)]$, and $Cs_3(A) = [c(1), c(5), c(6)]$.

Now a filtering expression accepts a complete argument tree if it accepts all its context sequences. In detail, filters accept context sequences as follows:

- A single term filter accepts a single context that unifies with it after proper variable substitution. Consider for instance the requests in the first column of the following table. They accept the rules with contexts shown in second column, and the resulting unifications are shown in the third:

Request	context => goal	Unifications
`contexte(1) => p in m`	`contexte(1) => p`	none
`c(X, X) => p in m`	`c(2, Y) => p`	`X=2` and `Y=2`
`VAR => p in m`	`npq => p`	`VAR=npq`

  Unifications made when an single term filter accepts a context are carried on to succeeding steps of accepting a branch.

- A Prolog list of single term filters such as `[a, b, c]` accepts contexts that unify with any one of the elements of the list. Thus lists implement alternatives between contexts.

- A single term filter or a list of single term filters may be preceded by these operators, with the meanings given:

  "*" the element may be repeated zero or more times ;
  "+" the element must be repeated at least once ;
  "?" the element is optional.

- A sequence of the preceding filters accepts context sequences accepted by each element in turn. Sequence elements are separated by commas.

- A context sequence is completely accepted by a filter when the last element of the sequence has been accepted by the last element of the filter.

- As a special case, the "always free variable" denoted by "`$`" is a single term filter that never unifies and can be used as a wildcard.

A few examples will illustrate filtering expressions:

- The request:
  ```
  interp_context(ART, CONT, SOURCE), *$ => p in m
  ```
  accepts arguments having roots unifying with "`interp_context(ART, CONT, SOURCE)`" and having any branch.

- ```
  theory_of(X), *$, fact => p in m
  ```
  accepts arguments having roots unifying with `theory_of(X)` and branches ending with `fact`.

- ```
  theory_of(X), *$, opinion_of(X)
           => p in m
  ```
  accepts arguments having roots unifying with

theory_of(X) and branches ending with the opinion of same.

- *$, fact => p in m
  accepts arguments with branches ending with fact

Module expressions define the modules in which a request searches for arguments. The simplest module expressions are the names of object modules and the formal parameters of meta modules. These are combined with the names of meta modules. For instance, in the module expression meta_module(some_module) we have some_module, the name of an object module, and meta_module, the name of a meta module of arity 1.

The exact place where an argument is constructed depends on the module expressions it meets. For instance, suppose we have object modules civil_obj and criminal_obj and the following meta modules:

```
begin_meta(meta_judge).
  ...
   /* rule1 */
   textual => responsible :- textual => faulty in
      meta_law(civil_obj).
   /* rule2 */
   textual => guilty :- textual => faulty in
      meta_law(criminal_obj).
end_meta.
begin_meta(meta_law(obj_law)).
  ...
   textual => faulty :- textual => negligent in obj_law.
end_meta.
```

Then the execution of rule1 will be followed by the request textual => faulty made to civil_obj while rule2 will be followed by the same request, this time addressed to criminal_obj. We see that the formal parameter of meta_law is effectively replaced at execution time by the actual parameter of the calling request.

For another example, the following request obtains all the arguments for the goal spouse_moving_exception in the module unemployment_obj:

```
/* module "unemployment_meta", rule meta_sme */
meta_interp_context(ART, CONT, SOURCE) =>
   meta_spouse_move_excpt(CLAIMANT, SPOUSE) :-
      /* the filter: */
     interp_context(ART, CONT, SOURCE), $* =>
      /* the goal:   */
     spouse_moving_exception(CLAIMANT, SPOUSE) in
      /* the module: */
     unemployment_obj.
```

In meta_spouse_move_excpt, all the rules for spouse_moving_exception in unemployment_obj with interpretative contexts unifying with interp_context(ART, CONT, SOURCE) will be used. Thus if interp_context(ART, CONT, SOURCE) is called with ART, CONT and SOURCE free, then the request will eventually backtrack on all available contexts for spouse_moving_exception. The variables will convey context information from the called module to the caller. This example also shows the use of the catch-all filter *$ which accepts any branch of any length.

For our final example we turn to the "canonical" problem of representing priorities between competing applicable laws. For instance, "lex superior" and "lex posteriores" are two ways to choose between conflicting legal rules .
Suppose that at the object level two hypothetical building regulations are applicable. The first is a general but old law to be applied nation-wide (the National Building Code, nat_bc for short), while the second is a recent municipal regulation (the Municipal Building Regulation, mun_br for short). The object level also contains the "recent/superior" relationships between the two. Each rule is accompanied by its textual description, placed after the htext ("Help TEXT") operator:

```
begin_object(bcodes).      % Object level containing
                           % the building codes
   nat_bc =>
   min_distance(5, bungalow)
     htext
   "According to the National Building Code the
        minimal distance
        between two bungalows is 5 meters.".

   mun_br =>
   min_distance(3, bungalow)
     htext
   "According to the Municipal Building Regulation the
        minimal distance
        between two bungalows is 3 meters.".

   fact =>
   more_recent(mun_br, nat_bc)
     htext
   "The Municipal-BR is more recent than
        the National-BC".

   fact =>
   superior(nat_bc, mun_br)
     htext
   "The National-BC is superior to the
        Municipal-BR".
end_module.
```

The meta module priorities(o) uses "recent/superior" information in some object level module o to build arguments for a given predicate P in this same module:

```
begin_meta(priorities(o)).
   lex_superior(SuperLaw) => P :-
       (fact => superior(SuperLaw, _OtherLaw) in o),
       (SuperLaw => P in o)
     htext
       "This rule implements lex superior for a given
        predicate P in a some module o. It unifies
        its context parameter with the context of
        the superior law.".
```

```
lex_posterior(RecentLaw) => P :-
    (fact => more_recent(RecentLaw, _OtherLaw) in o),
 (RecentLaw => P in o)
   htext
    "This rule implements lex posterior for a given
    predicate P in a some module o. It unifies its
    context parameter with the context of the
    more recent law.".
end_module.
```

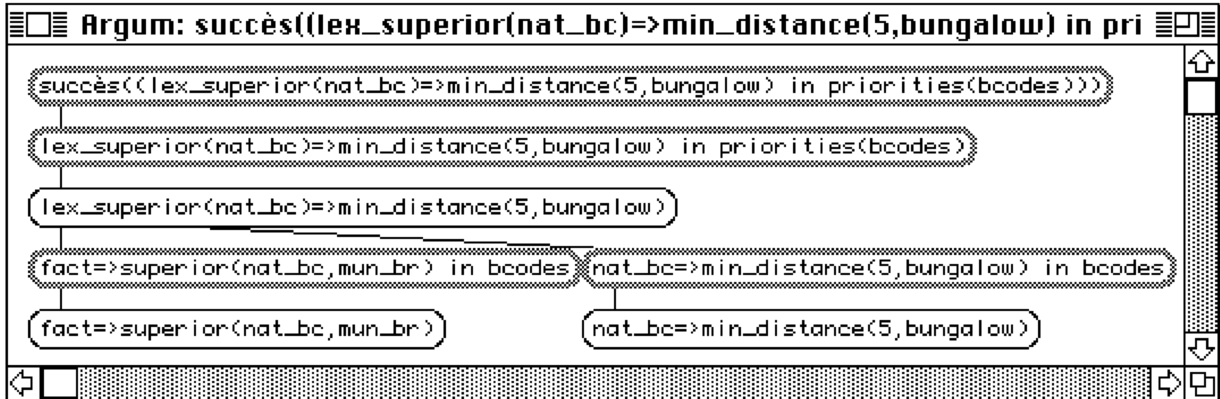


Figure 1 : Lex Superior Argument

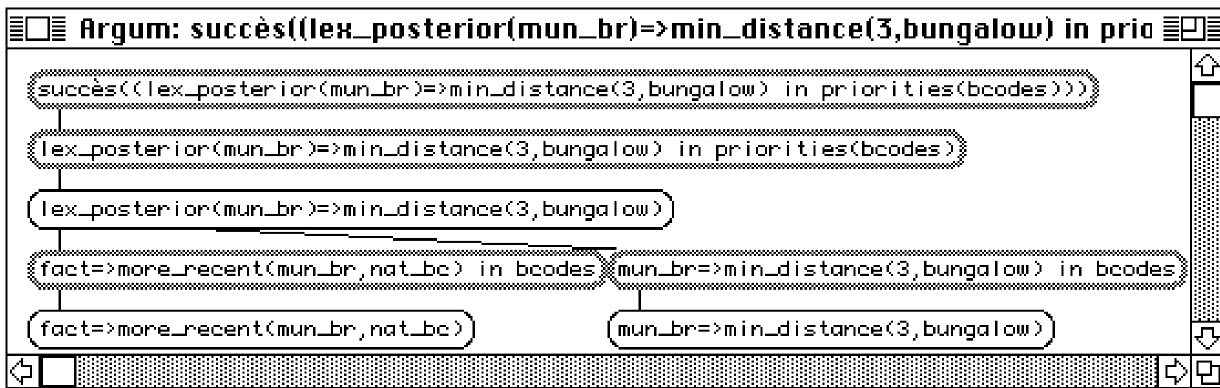


Figure 2 : Lex Posterior Argument

Now suppose we execute the following request for a "lex superior" solution at the toplevel :

```
?- do(lex_superior(SuperLaw) => min_distance(X,B) in
     priorities(bcodes)).
```

We obtain the following output:

```
SuperLaw = nat_bc, X = 5, B = bungalow
```

And the "lex superior" argument tree is produced (See Figure 1)

The gray nodes represent the execution of requests. The plain nodes take place for rules and facts. and conjunct nodes are connected to the same parent. The third node from top shows the unification between the variable `P` in the rule for `lex_superior` and the goal `min_distance(5, bungalow)` in the request.

If we then execute this request for "lex posterior":

```
?- do(lex_posterior(RecentLaw) => min_distance(X,B) in
     priorities(bcodes)).
```

We obtain:

```
RecentLaw = mun_br, X = 3, B = bungalow
```

And the "lex posterior" argument is built (See Figure 2).

We could continue to multiply our examples. For instance, our system uses both negation by failure and explicit negation. The use we make of these two concepts, and the relation between them, will be described in another article.

5. Conclusion

The system we have described is based on the following premisses.

- An expert system must be able to use the expertise peculiar to its particular application area. In law, this means it must be able to interpret legal rules, not simply apply them blindly.
- A legal expert system must be able to develop arguments both for and against a given point of view.
- Object level rules must be stated in a declaratory fashion.

From these considerations we were inevitably driven to use a meta level architecture. The system we have built can have any number of meta levels; its structure is modular, it incorporates filters to handle contextual information, and it produces proof trees (arguments) that take account of differing interpretations of the object level rules. The control mechanisms are explicit: using the same object level rules, we can fix a goal and have the system produce arguments supporting it, and also arguments against it.

The system is implemented for Macintosh computers using AAIS Prolog. The first working prototype, which had only one rudimentary meta level, was neverthless adequate to allow us to test the general feasibility and desirability of our ideas. A second version is now almost complete. Among other things, this new version already allows us to learn more about how to use the various kinds of negation. We also intend to explore how not just rules, but facts too, can be "interpreted" according to the contexts in which they occur: how reliable they are, whether they are admissible in evidence, and so on. Finally we intend in the course of 1995 to implement a complete example taken from a genuine area of law to see how well the system performs on a real application.

Acknowledgements

The work described in this paper is supported by the Social Sciences and Humanities Research Council of Canada, and by Quebec's *Fonds pour la Formation de Chercheurs et l'Aide à la Recherche*..

References

Alexy 89 Alexy, Robert, *A Theory of Legal Argumentation*, Oxford, at the Clarendon Press, 1989.

Ashley 90 Ashley, K.D., *Modeling Legal Argument*, MIT Press, Cambridge, Mass., 1990.

Bench-Capon 92 Bench-Capon, T.J.M. and F. Coenen, «Isomorphism and legal knowledge based systems», *Artificial Intelligence and Law*, vol. 1 (1992), pp. 65-86.

Bergel 89 Bergel, Jean-Louis, *Théorie générale du Droit*, coll. Méthodes du Droit, Dalloz, Paris, 2nd edition 1989.

Bowen 82 Bowen, Kenneth A. and R.A. Kowalski, «Amalgamating language and metalanguage in logic programming», in *Logic Programming (APIC Studies in Data Processing 16)*, Academic Press, 1982.

Brewka 94 Brewka, G., «Reasoning about Priorities in Default Logic», *Proceedings of the Twelfth National Conference on Artificial Intelligence,* Seattle, 1994, AAAI Press/MIT Press, Menlo Park, vol. 2, pp. 940-945.

Du Pasquier 88 Du Pasquier, Claude, *Introduction à la théorie générale et à la philosophie du Droit*, Delachaux et Niestlé, Paris, 6th edition 1988.

Freeman 94 Freeman, Kathleen, *Toward Formalizing Dialectical Argumentation*, PhD Thesis, CIS-TR-94-19, Dept of Computer Science, University of Oregon, 1994.

Gordon 93 Gordon, Thomas F., «The Pleadings Game – Formalizing Procedural Justice», *Proceedings of the Fourth International Conference on Artificial Intelligence and Law*, Vrije Universiteit, Amsterdam, 1993, ACM Press, New York, pp. 10-19.

Geffner 93 Geffner, H. and J. Pearl, «Conditional Entailment: Bridging Two Approaches to Default Reasoning»,*Artificial Intelligence*, 53, (2-3), pp. 209-244, 1992.

Hage 93 Hage, Jaap, «Monological reason-based logic», *Proceedings of the Fourth International Conference on Artificial Intelligence and Law*, Vrije Universiteit, Amsterdam, 1993, ACM Press, New York, pp. 30-39.

Hamfelt 89 Andreas Hamfelt and Jonas Barklund, «Metalevels in legal knowledge and their runnable representation in logic», in *Pre-Proceedings of the Third International Conference Logica Informatica Diritto – Expert Systems in Law*; November 1989, Florence, vol 2, pp. 557-576.

Henderson 94Henderson, Gordon F., «An introduction to patent law», *Patent Law in Canada*, Carswell, Scarborough, Ontario, 1994, pp. 1-14.

MacCormick 91 MacCormick, D.N. and R.S. Summers, *Interpreting Statutes: A Comparative Study*, Dartmouth, Aldershot, 1991

Poulin 93a Poulin, Daniel, *Interprétation et systèmes experts en droit écrit*, Mémoire de maîtrise, Faculté de droit, Université de Montréal, Montréal, 1993.

Poulin 93b Poulin, Daniel, Paul Bratley, Jacques Frémont and Ejan Mackaay, «Legal interpretation in expert systems», *Proceedings of the Fourth International Conference on Artificial Intelligence and Law*, Vrije Universiteit, Amsterdam, 1993, ACM Press, New York, p. 90-99.

Poulin 93c Poulin, Daniel, Pierre St-Vincent and Paul Bratley, «Contradiction and confirmation», *Proc. Intl Conf. on Database and Expert Systems Applications — DEXA '93,* Prague, September 1993 (*Lecture Nores in Computer Science* 720), Springer-Verlag, Berlin, 1993, pp. 502-513.

Prakken 93 Prakken, Henry, *Logical Tools for modelling Legal Argument*, Phd Thesis, Vrije Universiteit, Amsterdam, 1993.

Rissland 87 Rissland, E.L. and K. Ashley, «A case-based system for trade secrets law», *Proceedings of the First International Conference on Artificial Intelligence and Law*, Northeastern University, Boston, 1987, ACM Press, New York, pp. 60-66.

Robert 86 «Dialectique», *Le grand Robert de la langue française*, Éditions Le Robert, Paris, 1986, p. 508.

Sartor 93 Sartor, Giovanni, «A simple compational model for nonmonotonic and adversarial legal reasoning», *Proceedings of the Fourth International Conference on Artificial Intelligence and Law*, Vrije Universiteit, Amsterdam, 1993, ACM Press, New York, pp. 192-201.

Sartor 94 Sartor, Giovanni, «A Formal Model of Legal Argumentation», *Ratio Juris*, vol. 7, no. 2, july 1994, pp. 177-211.

Schild 93 Schild, Uri J. and Shai Herzog, «The use of meta-rules in rule-based legal computer systems», *Proceedings of the Fourth International Conference on Artificial Intelligence and Law*, Vrije Universiteit, Amsterdam, 1993, ACM Press, New York, pp. 100-109.

Schobbens 93 Schobbens, Pierre-Yves, «A logic for legal hierarchies», *Proceedings of the Fourth International Conference on Artificial Intelligence and Law*, Vrije Universiteit, Amsterdam, 1993, ACM Press, New York, pp. 272-281.

Sergot 86 Sergot, M.J., F. Sadri, R.A. Kowalski, F. Kriwaczek, P. Hammond and H.T. Cory, «The British Nationality Act as logic program», *Communications of ACM*, vol. 29 (1986), pp. 370-386.

Smith 87 Smith, J.C. and C. Deedman, «The application of expert systems technology to case-based law», *Proceedings of the First International Conference on Artificial Intelligence and Law*, Boston, 1987, ACM Press, New York, pp. 85-93.

Susskind 87 Susskind, R.E., *Expert Systems in Law : A Jurisprudential Inquiry*, Oxford, at the Clarendon Press, 1987.

van Harmelen 89 van Harmelen, F., «A classification of meta-level architectures», in *Meta-Programming in Logic Programming*, eds H. Abramson and M.H. Rogers, MIT Press, Cambridge,Mass., 1989

Wróblewski 88 Wróblewski, J., «Interprétation», in A.J. Arneaud (ed.), *Dictionnaire encyclopédique de théorie et de sociologie du droit*, Story-Scientia, Paris, 1988, pp. 199-201.

Yalçinalp 89 Yalçinalp, L.U and L. Sterling, «An integrated interpreter for explaining Prolog's successes and failures», in *Meta-Programming in Logic Programming*, ed. H. Abramson and M.H. Rogers, MIT Press, Cambridge, Mass., 1989.