ESTRELLA	ESTRELLA IST-2004-027655 ***European project for Standardized Transparent Representations in order to Extend LegaL Accessibility Specific Targeted Research or Innovation Project*** Specific Targeted Research Project Information Society Technologies

Deliverable N°: 4.3

The reference LKIF inference engine

Version:	FINAL 1.0
Due date of Deliverable:	September 30, 2008
Actual submission date:	September 15, 2008
Start date of Project:	1 January 2006
Duration:	30 months
Project Coordinator:	Universiteit van Amsterdam (NL)
Lead contractor deliverable:	Fraunhofer FOKUS
Participating contractors:	Alma Mater Studiorum - Universita di Bologna (IT), University of Liverpool (UK), Fraunhofer Gesellschaft zur foerderung der angewandten forschung e.v. (DE), RuleWise b.v. (NL), RuleBurst (EUROPE) Ltd. (UK), knowledgeTools International GmbH (DE), Interaction Design Ltd. (UK), SOGEI - Societa Generale d'Informatica S.P.A. (IT), Ministro per le Riforme e le Innivazioni nella Publica Amministrazione (IT), Hungarian Tax and Financial Control Administration (HU), Budapesti Corvinus Egyetem (HU), Ministero dell'Economia e delle Finanze (IT), Consorzio Pisa Ricerche SCARL (IT)

Project funded by the European Community under the 6th Framework Programme

	Dissemination Level	
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Executive Summary

This ESTRELLA report is Deliverable D4.3, documenting the source code of the ESTRELLA *reference inference engine*, called Carneades, for the final version of the Legal Knowledge Interchange Format (LKIF), as it is defined in Deliverable D4.1. The previous version of this report was delivered as D1.6, as a specification of the APIs of the inference engine. Deliverable D1.5 consisted of the source code of the first version of the reference inference engine. The API document of Workpackage 4, Deliverable D4.4, now documents a programming language independent, Web Services API to this inference engine, using the SOAP protocol.

This reference inference engine consists of several modules, organized into the layered architecture:

Foundation. The foundation layer consists of modules for configuring the system for a particular installation, managing some basic data structures (tables, sets, heaps and streams), and for heuristically searching problem spaces.

Statement. The statement layer provides a module for comparing and decomposing statements, abstracting away syntactic details which are irrelevant for the higher layers, and a module implementing a unification algorithm, needed for implementing inference engines for logics with variables ranging over compound terms, such as first-order logic.

Argument. The argument layer provides modules for constructing, evaluating and visualizing proofs, called "argument graphs", and modules for applying argumentation schemes to search heuristically for argument graphs in which some goal statement is acceptable (i.e. a defensible presumption) or not acceptable.

Inference. The inference layer provides modules for generating arguments from ontologies, defeasible inference rules, precedent cases, and testimonial evidence. It also includes a module for compiling formulas of first-order logic into disjunctive normal form. Another module provides rulebases with an API to the Scheme programming language, allowing Scheme expressions to be embedded in defeasible inference rules (Section 5.2) and evaluated when searching for arguments. This module also provides methods for constructing arguments about standard LKIF predicates.

LKIF. Finally, the Legal Knowledge Interchange Format (LKIF) layer provides a module for importing and exporting rules and arguments encoded in LKIF.

The reference inference engine, Carneades, is written in functional style using the Scheme programming language Scheme [Dybvig, 2003]. Prior knowledge of

functional programming and Scheme is prerequisite for understanding this report. The Carneades system is open source software, available at `http://carneades.berlios.de`.

The Reference LKIF Inference Engine
ESTRELLA Deliverable D4.3

Thomas F. Gordon
Fraunhofer FOKUS, Berlin

Contents

ESTRELLA
p/a Faculty of Law
University of Amsterdam
PO Box 1030
1000BA Amsterdam
The Netherlands

tel: +31 20 525 3485/3490
fax: +31 20 525 3495
`http://www.estrellaproject.org`

Corresponding author:
Thomas F. Gordon
`thomas.gordon@fokus.fraunhofer.de`

Chapter 1

Introduction

This report documents the final version of the ESTRELLA *reference inference engine*, called Carneades, for the Legal Knowledge Interchange Format (LKIF), as it is specified in Deliverable D4.1. Carneades is open source software, available at `http://carneades.berlios.de`.

The report is intended to serve as a detailed reference manual documenting all of the data structures, functions and procedures of the reference inference engine. The target audience is programmer's who either want to use, modify or extend the code, or who want to understand the code well enough to be able to reimplement the system, perhaps in another programming language.

Prior knowledge of functional programming and, in particular, the Scheme programming language Scheme [Dybvig, 2003] is presumed.

The objectives of ESTRELLA are summarized clearly on pages 3-4 of the Technical Annex:

> The primary business objective of the ESTRELLA project is to develop and validate an open, standards-based platform allowing public administrations to develop and deploy comprehensive legal knowledge management solutions, without becoming dependent on proprietary products of particular vendors. ESTRELLA will support, in an integrated way, both legal document management and legal knowledge systems, to provide a complete solution for improving the quality and efficiency of the determinative processes of public administration requiring the application of complex legislation and other legal sources. ESTRELLA will facilitate a market of interoperable components for legal knowledge systems, allowing public administrations and other users to freely choose among competing development environments, inference engines, and other tools.
>
> The main technical objectives of the ESTRELLA project are to develop a Legal Knowledge Interchange Format (LKIF), building upon emerging XML-based standards of the Semantic Web, including RDF and OWL, and Application Programmer Interfaces (APIs) for interacting with LKIF legal knowledge systems. The LKIF will apply the state of the art in the field of Artificial Intelligence and Law, taking into account business and application requirements. Existing Semantic Web initiatives are aimed at modeling concepts (OWL ontologies) and rules

(RuleML and SWRL). The LKIF will build on but go beyond this generic work to allow further kinds of legal knowledge to be modeled, including: meta-level rules for reasoning about rule priorities and exceptions, legal arguments, legal procedures, cases and case factors, values and principles. In addition, an OWL ontology of basic legal concepts, such as obligations, permissions, rights and powers, will be developed, which can be reused when modeling specific legal domain, such as tax law.

A reference inference engine and run-time environment capable of processing knowledge bases using all the features of the LKIF will be implemented and validated in the pilot applications. To achieve and demonstrate vendor neutrality and independence, translators between the LKIF format and the existing proprietary formats of LKBS vendors participating in the project (RuleWise, RuleBurst and knowledgeTools) will be developed and validated in pilot applications.

1.1 Overview of the Reference Inference Engine

This ESTRELLA reference inference engine consists of several modules, organized into the layered architecture shown in Figure 1.1

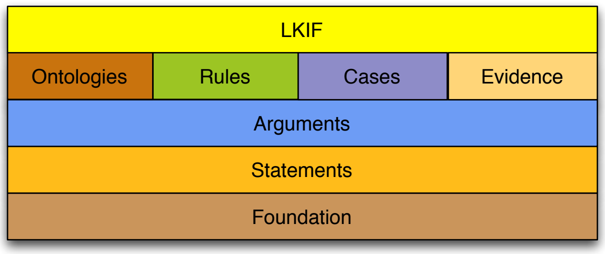

Figure 1.1: Module Layers

In this architecture, a module may make use of the services of another module in the same layer or any layer below it in the diagram. Conversely, a module does not depend on the services of any module at some higher level.

Since the higher layers build upon the lower layers, we will describe the lowest layers first:

Foundation. The foundation layer consists of modules for configuring the system for a particular installation, managing some basic data structures (tables, sets, heaps and possibly infinite sequences of data), and for heuristically searching problem spaces.

Statement. The statement layer provides a module for comparing and decomposing statements, abstracting away syntactic details which are irrelevant for the higher layers, and a module implementing a unification algorithm, needed for implementing inference engines for logics with variables ranging over compound terms, such as first-order logic.

Argument. The argument layer provides modules for constructing, evaluating and visualizing proofs, called "argument graphs", and modules for applying argumentation schemes to search heuristically for argument graphs in which some goal statement is acceptable (i.e. a defensible presumption) or not acceptable.

Inference. The inference layer provides modules for generating arguments from ontologies, defeasible inference rules, precedent cases, and testimonial evidence. It also includes a module for compiling formulas of first-order logic into disjunctive normal form. Another module provides rulebases with an API to the Scheme programming language, allowing Scheme expressions to be embedded in defeasible inference rules (Section 5.2) and evaluated when searching for arguments. This module also provides methods for constructing arguments about standard LKIF predicates.

LKIF. Finally, the Legal Knowledge Interchange Format (LKIF) layer provides a module for importing and exporting rules and arguments encoded in LKIF.

1.2 The Functional Style of the Reference Inference Engine

The rest of this report presents the modules of the reference inference engine in detail, with a chapter for each layer of the system architecture shown in Figure 1.1. The inferece engine has been implemented in a functional programming style using a functional programming language, Scheme [Dybvig, 2003]. The previous version of the inference engine, reported in D1.6, was implemented in the PLT dialect of Scheme [Project,]. In the meantime, the inference engine has been ported as to be fully compliant with the latest R6RS Scheme standard [1]

The purpose of the reference inference engine is to provide a clear, high-level research prototype implementation of LKIF which can serve as a guide for more efficient and robust, production-quality implementations.

[1]Revised 6 Report on the Algorithmic Language Scheme. `http://www.r6rs.org/`

Chapter 2

Foundational Layer

The foundation layer consists of modules for configuring the system for a particular installation, managing some basic data structures (tables, sets, heaps and streams), and for heuristically searching problem spaces.

2.1 Config

The `config` module defines a number of values which may need to be modified to configure the system for a particular installation. Currently it consists of four values:

`dot` The full pathname to the `dot` command for translating files in the dot language into various graphic formats. [1] The command must accept the file name of the dot file to be viewed as its first argument, and require no further arguments.

`preferred-graphic-format` A file suffix string, choosing the preferred graphics format to be used for displaying argument graphs. Any format supported by the `dot` command may be provided, such as `"png"`, `"ps"`, or `"svg"`.

`viewer` The full pathname to a command for viewing files in the chosen `preferred-graphic-format`.

`tmpdir` The full pathname to a directory for creating temporary files.

2.2 Table

A table is a data structure mapping *keys* to *values*. Keys may be any object comparable using the `equal?` predicate.

`(make-table)` makes an empty table.

`(table? object)` is a predicate for testing whether `object` is a table. Returns a boolean value.

`(insert table key value)` returns a table in which `table` has been extended by associating `key` with `value`.

[1]For more information about dot see `http://www.graphviz.org`.

`(lookup table key default)` returns the value of `key` in `table`, or `default`, if no object is associated with the key in the table.

`(keys table)` returns a list of the keys in `table`.

`(values table)` returns a list of the values in `table`.

`(filter table predicate)` returns a list of `(key . value)` pairs for all of the `(key. value)` pairs in the table which satisfy the `predicate`, which must be a function mapping pairs to boolean values.

`(filter-keys table predicate)` returns a list of keys for all of the `(key. value)` pairs in the table which satisfy the `predicate`, which must be a function mapping pairs to boolean values.

`(filter-values table predicate)` returns a list of values for all of the `(key. value)` pairs in the table which satisfy the `predicate`, which must be a function mapping pairs to boolean values.

2.3 Set

The `set` module provides a functional implementation of the mathematical concept of a set.[2]

`(empty-set =?)` returns an empty set which compares elements to members of the set using the provided equality predicate `=?`.

`((singleton =?) object)` returns a singleton set containing `object` which compares elements to members of the set using the provided equality predicate `=?`.

`((list->set =?) list)` converts the `list` to a set.

`(set->list set)` converts the `set` to a list.

`(empty? set)` is a predicate which tests whether the given `set` is empty.

`(select set)` returns some element of the `set` but raises an error if the set is empty.

`(filter predicate set)` returns a list of members of `set` which satisfy `predicate`.

`(any? predicate set)` tests whether any member of `set` satisfies `predicate`.

`(every? predicate set)` tests whether every member of `set` satisfies `predicate`.

`((adjoin =?) object set)` returns the set resulting from adding `object` to `set`.

[2]This functional implementation of sets in Scheme is based on code place in the public domain by Darius Bacon. See `http://www.accesscom.com/~darius`.

`((union =?) set1 set2)` returns the union of `set1` and `set2`, comparing their elements using the equality predicate `=?`.

`((intersection =?) set1 set2)` returns the intersection of `set1` and `set2`, comparing their elements using the equality predicate `=?`.

`((disjoint? =?) set1 set2)` tests whether `set1` and `set2` are disjoint, i.e. have no members in common, using the equality predicate `=?` to compare members.

`((difference =?) set1 set2)` returns a set consisting of the members of `set1` which are not in `set2`, using the equality predicate `=?` to compare members.

`((subset? =?) set1 set2)` tests whether `set1` is a subset of `set2`, using the equality predicate `=?` to compare members.

2.4 Heap

A heap is a data structure which maintains a partially ordered set of objects. Objects are sorted by *inserting* them into a heap. An item on 'top' of the heap, i.e. a least element in the partial ordering, can be removed from the heap.[3]

`(make-empty <=)` returns an empty heap which will use the `<=` binary predicate to order elements in the heap.

`(empty? heap)` tests whether `heap` contains no elements.

`(singleton <= object)` returns a heap containing just `object` which uses the predicate `<=` to order elements.

`(insert heap object)` returns the heap constructed by inserting, nondestructively, `object` into `heap`.

`(list->heap <= list)` returns a heap containing all of the members of `list`, sorted by `<=`.

`(find-min heap)` returns a top, i.e. smallest, member of `heap`, but raises an error if `heap` is empty.

`(delete-min heap)` returns the heap constructed by removing, nondestructively, an element on the top of `heap`.

`(merge heap1 heap2)` returns a heap containing the union of the members of `heap1` and `heap2`, sorted using the ordering predicate of `heap1`.

`(merge-heap-pairs <= list)` returns the heap constructed by merging all of the heaps in a list of heaps.

[3]This heap module is based on the implementation in the Schematics Cookbook, `http://schemecookbook.org/view/Cookbook/FunctionalHeap`.

2.5 Stream

A stream in the sense meant here is data structure representing a possibly infinite sequence of data objects. The elements of the stream are computed as they are needed, using delayed or 'lazy' evaluation [Abelson and Sussman, 1985].

The `stream` module is a modest extension of the stream library of the Scheme Request for Implementation (SRFI), Number 41, by Philip L. Bewig.[4] Only the extensions are documented here.

`(stream-interleave s1 s2)` combines elements from two streams in a way which assures that elements of `s2` are reached even if `s1` is infinite. For example, if `s1` is `(1 2 3)` and `s2` is `(4 5 6)`, then `(stream-interleave s1 s2)` is `(1 4 2 5 3 6)`.

`(stream-accumulate f i s)` combines the elements of the stream `s` by applying the function `f` to each element of `s` and the result of accumulating, recursively, the rest of `s`, i.e. `(stream-accumulate f i (stream-cdr s)`, if `s` is not empty, or the initial value, `i`, if `s` is empty.

`(stream-flatten s)` combines a stream of streams, `s`, into a single stream consisting of all the elements of these streams. The streams are combined using `stream-interleave` to assure that elements of all the streams in `s` are reached, even when one or more of these streams are infinite.

`(stream-flatmap f s)` flattens the stream of streams produced by applying the function `f`, of type `(forall (alpha beta) (-> (alpha) (stream-of beta)))`, to each element of `s`.

2.6 Search

The `search` module provides methods for defining and heuristically searching spaces [Russell and Norvig, 2003] to find solutions to problems. The module does not depend on the structure of the states in the problem space. Thus `state` is declared to be an abstract type in the interface of the `search` module.

`(make-root state)` constructs a node of the search space. `make-root` handles the base case, for the root of the search tree. Nothing in the `search` module depends on the structure of the state. It can be a value of any type.

`(make-node depth label parent state)` constructs a successor node of some `parent` node. The `depth` of the successor node should be `(+ (node-depth parent) 1)`. The `label` can be any value providing information about the transition, i.e. link, between the nodes in the space. The `state` value is responsible for representing the application-specific content of the node.

`(node-depth node)` returns the depth of the node as an integer.

[4] `http://srfi.schemers.org/srfi-41/srfi-41.html`

`(node-label node)` returns the label of the node.

`(node-parent node)` returns the parent node of the node.

`(node-state node)` returns the state of the node.

`(node? x)` is a predicate for checking whether some value is a node.

`(root? node)` is a predicate which is true iff the `node` is the root node of a search space.

`(make-problem root space goal)` defines a search problem, where the task is to try to find a node in the `space`, starting from the `root` node, which satisfies the `goal` predicate. The `space` must be a function which maps a `parent` node to a stream of successor nodes. The depth of each of these successor nodes is 1 greater than the depth of the `parent` node. Notice that a node may have infinitely many successors nodes in the space, so long as they are enumerable.

`(problem-root problem)` returns the root node of a problem.

`(problem-space problem)` returns the space of a problem.

`(problem-goal goal)` returns the goal of a problem.

`(problem? x)` checks whether some value, `x`, is a problem.

A search `strategy` is a function which maps a problem to a stream of nodes. Given a problem, a strategy has the job of actually searching the space of the problem in some way to find solutions, i.e. nodes which satisfy the goal predicate of the problem. Let `s` be a strategy and `p` be a problem. The stream of nodes returned by `(s p)` consists of just the solutions to the problem. Since the values of a stream can be computed as needed, accessing the tail of the stream of solutions, i.e. `(stream-cdr (s p))`, can cause the strategy `s` to backtrack to search for further solutions.

Resources can be used to implement search strategies which terminate after some resource limit has been exhausted. Typically, expansion of a node in the search space expends a unit of the resource.

`(make-resource i)` constructs a resource with `i` units.

`(resource-empty? r)` is a predicate which is true if all the available units of the resource have been exhausted.

`(resource-amount r)` returns the amount of units of the resource remaining.

`(use r)` decrements the number of remaining units of the resource by one. Notice that resources are implemented using a mutable data structure. That is, contrary to the general architecture of the system, resources are implemented in an imperative style, rather than functionally.

`(search problem strategy)` just applies the `strategy` to the `problem`, i.e. is defined to be the same as `(strategy problem)`. The only purpose of the `search` function is to help document places in programs where searches are taking place.

`(path node)` returns a list of the labels of the transitions from the given node back to the root node of the space.

`(depth-first r)` constructs a limited depth-first search strategy, where `r` is a resource limiting not the maximum depth of the search tree, but rather the maximum number of nodes which may be expanded when searching the tree. If you want to set a depth-limit, use the `iterative-deepening` strategy.

`(breadth-first r)` constructs a limited breadth-first search strategy, given a resource `r`. Here too the limit is on the number of nodes which may be expanded.

`(iterative-deepening init step)` is a resource-limited hybrid search strategy, combining features of depth-first and breadth-first search. To construct a search strategy, the resulting resource-limited strategy needs to be applied to a resource, as in this example:

```
((iterative-deepening 10 10) (make-resource 2000))
```

The iterative-deepening strategy uses depth-first search to a maximum depth equal to `init`. When this maximum depth is reached, it backtracks and continues the search, depth-first. After the entire search tree above the `init` depth is searched, assuming this part of the tree is finite, the maximum depth is incremented by `step` amount and the search process is repeated, from the beginning, using this greater depth limit. This strategy has the obvious disadvantage of redundantly searching the top part of the search space on each iteration. On the positive side, it can, like breadth-first search, find solutions which would be missed by purely depth-first search, when there are cycles in the search space. And, also like breadth-first search, it tends to find easier solutions, i.e. the ones requiring a fewer number of steps, first. (However to maximize this property, `init` and `step` should be relatively small.) Finally, iterative-deepening shares with depth-first search the advantage of requiring in general less memory than breadth-first search. As in the depth-first and breadth-first stratagies, the `limit` specifies the maximum number of nodes which may be expanded during search, not the maximum depth of search.

Example: Eight Puzzle To illustrate the `search` module, let's implement the 8-puzzle, which is often used in Artificial Intelligence textbooks to teach heuristic search, e.g. [Nilsson, 1982, pp. 18-29].

We begin by representing the state of the 8-puzzle. It consists of a 3 by 3 matrix with 8 tiles, numbered 1 to 8. One of the cells in the matrix is empty. We use the number 0 to represent this empty cell.

```
(define (make-state r1c1 r1c2 r1c3
                    r2c1 r2c2 r2c3
                    r3c1 r3c2 r3c3)
  (list r1c1 r1c2 r1c3
        r2c1 r2c2 r2c3
        r3c1 r3c2 r3c3))

(define (state-r1c1 s) (list-ref s 0))
(define (state-r1c2 s) (list-ref s 1))
(define (state-r1c3 s) (list-ref s 2))
(define (state-r2c1 s) (list-ref s 3))
(define (state-r2c2 s) (list-ref s 4))
(define (state-r2c3 s) (list-ref s 5))
(define (state-r3c1 s) (list-ref s 6))
(define (state-r3c2 s) (list-ref s 7))
(define (state-r3c3 s) (list-ref s 8))
```

There are four moves in the game. A tile may be moved up, down, left or right to fill the empty space. Equivalently, one can view the empty space as being moved. Our model takes this second perspective. We will label transitions in the space with the name of the move of the empty space, as follows:

```
(define (move? x) (member x '(up down right left)))
```

The goal is to find a state in which the tiles have a particular, chosen order. Let's search for ways to constuct this state:

```
1 2 3
8 0 4
7 6 5
```

Using Alex Shinn's pattern matching library, which has been ported to R6RS Scheme, the goal predicate for this state can be defined as follows:

```
(define (goal? s)
  (match s
    ((1 2 3
      8 0 4
      7 6 5) #t)
    (_ #f)))
```

The moves still need to defined. This job is done by the `move` function, of type `(-> (move node) (or node #f))`, defined next. If the preconditions of a move, `m`, are satisfied by a state `s`, then (move m s) is the state resulting from making the move. If the preconditions are not satisfied, then `(move m s)` is false, denoted `#f`.

We will use pattern matching to define the four moves, `up`, `down`, `left` and `right`, as follows:

```
(define (move m p)
  (let ((depth (+ (node-depth p) 1)))
    (case m
      ((up)
       (match (node-state p)
         ((a b c 0 d e f g h)
          (make-node depth 'up p (make-state 0 b c a d e f g h)))
         ((a b c d 0 e f g h)
          (make-node depth 'up p (make-state a 0 c d b e f g h)))
         ((a b c d e 0 f g h)
          (make-node depth 'up p (make-state a b 0 d e c f g h)))
         ((a b c d e f 0 g h)
          (make-node depth 'up p (make-state a b c 0 e f d g h)))
         ((a b c d e f g 0 h)
          (make-node depth 'up p (make-state a b c d 0 f g e h)))
         ((a b c d e f g h 0)
          (make-node depth 'up p (make-state a b c d e 0 g h f)))
         (_ #f)))
      ((down)
       (match (node-state p)
         ((0 b c d e f g h i)
          (make-node depth 'down p (make-state d b c 0 e f g h i)))
         ((a 0 c d e f g h i)
          (make-node depth 'down p (make-state a e c d 0 f g h i)))
         ((a b 0 d e f g h i)
          (make-node depth 'down p (make-state a b f d e 0 g h i)))
         ((a b c 0 e f g h i)
          (make-node depth 'down p (make-state a b c g e f 0 h i)))
         ((a b c d 0 f g h i)
          (make-node depth 'down p (make-state a b c d h f g 0 i)))
         ((a b c d e 0 g h i)
          (make-node depth 'down p (make-state a b c d e i g h 0)))
         (_ #f)))
      ((right)
       (match (node-state p)
         ((0 b c d e f g h i)
          (make-node depth 'right p (make-state b 0 c d e f g h i)))
         ((a b c 0 e f g h i)
          (make-node depth 'right p (make-state a b c e 0 f g h i)))
         ((a b c d e f 0 h i)
          (make-node depth 'right p (make-state a b c d e f h 0 i)))
         ((a 0 c d e f g h i)
          (make-node depth 'right p (make-state a c 0 d e f g h i)))
         ((a b c d 0 f g h i)
          (make-node depth 'right p (make-state a b c d f 0 g h i)))
         ((a b c d e f g 0 i)
          (make-node depth 'right p (make-state a b c d e f g i 0)))
         (_ #f)))
      ((left)
       (match (node-state p)
         ((a 0 c d e f g h i)
          (make-node depth 'left p (make-state 0 a c d e f g h i)))
         ((a b c d 0 f g h i)
          (make-node depth 'left p (make-state a b c 0 d f g h i)))
         ((a b c d e f g 0 i)
          (make-node depth 'left p (make-state a b c d e f 0 g i)))
         ((a b 0 d e f g h i)
          (make-node depth 'left p (make-state a 0 b d e f g h i)))
         ((a b c d e 0 g h i)
          (make-node depth 'left p (make-state a b c d 0 e g h i)))
         ((a b c d e f g h 0)
          (make-node depth 'left p (make-state a b c d e f g 0 h)))
         (_ #f)))
      (else #f))))
```

Notice that each of the moves labels the successor node with the name of the move. We will make use of this to produce a plan for transforming the initial state into the goal state, using the `path` function.

Recall that a problem `space` is represented by a function of type `(-> (node) (stream-of node))` The 8-puzzle space is represented by the following function, named `moves`:

```
(define (moves n)
  (define (f m)
    (let ((r (move m n)))
      (if r (list r) '())))
  (list->stream (apply append (map f '(up down right left)))))
```

The `moves` function simply constructs a stream from the result of applying each of the four moves, `up`, `down`, `left` and `right` to the given node, `n`, ignoring any move whose precondition is not satisfied.

We are ready to create 8-puzzle problems and search for solutions. Let's start from this state:

```
    2 8 3
    1 6 4
    7 0 5
```

This problem can be defined as follows:

```
(define pr1
  (make-problem (make-root (make-state 2 8 3 1 6 4 7 0 5))
                moves
                goal?))
```

Now let's use breadth-first search, with a limit of 1000 node expansions, to try to find one or more solutions:

```
(define s1 (search pr1 (breadth-first (make-resource 1000))))
```

We can test whether any solutions have been found with `(not (stream-null? s1))`. If the stream of solutions is not null, we can retrieve the path from the initial state to the goal state of the first solution, `(stream-car s1)`, as follows: `(path (stream-car s1))`. In this example, a solution has been found, and its path is *(up up left down right)*.

Finally, let's search the same problem space again, this time using iterative deepening, with an initial depth of a, a step of 10 and, as before, a maximum of 1000 node expansions:

```
(define s2 (search pr1 ((iterative-deepening 10 10) (make-resource 200))))
```

This iterative-deepening strategy also finds a solution, but not an optimal one: `(path (stream-car s2))` yields *(up up down up down up down up left down right)*.

Chapter 3

Statement Layer

The statement layer provides a module for comparing and decomposing statements (`statement`), abstracting away syntactic details which are irrelevant for the higher layers, and a module implementing a unification algorithm (`unify`), needed for implementing inference engines for logics with variables ranging over compound terms, such as first-order logic and LKIF Rules.

3.1 Statement

We use the term 'statement' to mean propositions which can be represented as postive or negative atomic formulas of first-order logic. In logic, such statements are usually called 'literals'.

Statements are represented here as Scheme symbolic expressions obeying the following grammar:

```
<statement> ::= <atom> | (not <atom>)
<atom> ::= <symbol> | <string> | (<symbol> <datum>*)
```

An `atom` is an atomic statement, i.e. a simple proposition without any logical operators, such as negation (`not`), conjunction (`and`) or disjunction (`or`). A `datum` is any Scheme symbolic expression, including symbols, booleans, strings, numbers and lists. A `statement` is an atom or negated atom, i.e. a literal of first-order logic.

Atomic formulas of propositional logic can be represented as symbols or strings.

Here are some example statements:

```
liable
"The security interest is perfected"
(initiates event1 (possesses ?p ?o))
(holds (perfected ?s ?c) ?p)
(applies UCC-306-1 (proceeds ?s ?p))
```

The `statement` module provides the following functions.

`(statement-equal? s1 s2)` tests whether two statements are equal. It defines an equivalence relation, i.e. a relation which is symmetric, reflexive, and transitive. Note: No statement represented as a symbol equals a statement represented as a string. For example, the statement `liable` does not equal the statement `"liable"`.

`(statement-compare s1 s2)` defines a total ordering on statements. If `s1` is less than `s2`, then `(statement-compare s1 s2)` is *-1*. If `s1` is equal to `s2`, then `(statement-compare s1 s2)` is *0*. Finally, if `s1` is greater than `s2`, then `(statement-compare s1 s2)` is *1*

`(statement-positive? s1)` is true iff `s1` represents a postive literal.

`(statement-negative? s1)` is true iff s1 *is* represents a negative literal.

`(statement-complement s1)` returns the logical complement of the statement `s1`. If `s1` is a positive literal, then `(complement s1)` is a negative literal and vice versa.

`(statement-atom s1)` If `s1` is a positive statement, then `(statement-atom s1)` equals `s1`. If `s1` is a negative statement, `(not s2)`, then `statement-atom` strips the negation operator from `s2`. That is, `(statement-atom '(not s2))` is *s2*.

`(summarize s1)` converts the statement `s1` to a string representation for use in reports and diagrams. If `s1` is a string, then `(summarize s1)` equals `s1`. Other statements are converted to strings using the `write` procedure from the R6RS simple i/o library.

3.2 Unify

The `unify` module provides a purely functional implementation of Robinson's unification algorithm [Robinson, 1965], based on the Dybvig's Scheme implementation in [Dybvig, 2003, Section 9.10].

Substitution environments are represented as functions which map *variables* to *terms*. To retrieve the value of a variable, `v` in a substitution environment `e`, just apply `e` to `v`: `(e v)`.

Terms are represented as Scheme symbolic expressions. Every symbolic expression can represent a term. Variables are represented as symbols beginning with a question mark character. Here are two examples: `?y`, `?agreement`.

`(varible? x)` is true iff `x` is a symbol representing a variable, i.e. a symbol whose first character is a question mark.

`identity` is a subsitutions environment which maps every variable to itself. It can be viewed as the empty substitution environment.

`(unify* u v s ks kf occurs-check)` attempts to unify `u` and `v` with a continuation-passing style that returns a substitution to the success argument `ks` or an error message to the failure argument `kf`. The occurs check is performed only if `occurs-check` is not `#f`. In the semantics of first-order logic, unification should fail if a variable is contained in the term with which it is to be unified. However, since the check is expensive and some programs can assure, by construction, that the variable will not occur in the term, the `occurs-check` flag allows the occurs check to be turned-off, improving performance.

`(unify u v)` provides a simplified interface to the `unify*` function, where the initial substitution is the `identity` function, the success function returns the resulting substitution environment, the failure function returns `#f`, and the occurs check is turned off.

`(genvar)` generates a unique, fresh variable.

`(rename-variables table term)` Returns a term constructed by systematically renaming the variables in `term`, using fresh variables, keeping track of the replacements in `table`, which is a R6RS Scheme hash table from the `(rnrs hashtables)` standard library. Warning: `rename-variables` has side-effects, since it can modifies the hashtable.

`(ground? term)` tests whether `term` does not contain any variables.

Here's a example of the `unify` function at work:

```
> (define s1 (unify '(foo ?x b) '(foo a ?y)))
> (s1 '?x)
'a
> (s1 '?y)
'b
> (unify '(foo ?x a) '(foo a b))
#f
```

Chapter 4

Argument Layer

The argument layer provides modules for constructing, evaluating and visualizing proofs as argument graphs (`argument`, `argument-diagram`). It also provides a module (`argument-search`) for applying argumentation schemes to search heuristically for argument graphs in which some goal statement is acceptable (i.e. presumably true) or not acceptable. The `argument-builtins` module constructs arguments about some standard predicates.

4.1 Argument

The `argument` module implements the Carneades model of argument graphs [Gordon et al., 2007]. It may be viewed as an instantiation or refinement of the conceptual model of the Argument Interchange Format devleoped in the European ASPIC project [Carlos Ches et al., 2006].[1] Argument graphs, also called *inference graphs* [Pollock, 1992] are directed, acyclic, bipartite graphs, consisting of statement nodes and argument nodes which represent defeasible, presumptive proofs. An argument consists of a set of premises and a conclusion. There are three kinds of premises: ordinary premises, exceptions and assumptions. In argument graphs, premises are modeled as links (arcs) from statements nodes to argument nodes. The type of the premise is modeled by labeling the link from the statement node to the argument node. The conclusion of an argument is a statement and is modeled in argument graphs by a link from the argument to the statement of its conclusion. The statement of premises and conclusions is a positive statement, i.e. representing an atomic proposition. There are two kinds of arguments, pro arguments and con arguments. An argument pro a negative statement, `(not s)`, is represented as an argument con `s`. A negated premise is modeled as a property of the premise, i.e. as a property of the link from the (positive) atom of the premise to the argument. These conventions enable statement nodes in argument graphs to be restricted to positive literals without loss of expressivity.

Because the argument module is rather long, it will be presented incrementally in the following sections: Premise (Section 4.1.1), Argument (Section 4.1.2), Argument Context (Section 4.1.3), Argument Graph (Section 4.1.4) and Argument Evaluation (Section 4.1.5).

[1] ASPIC (IST-002307) was an Integrated Project of the European Unions 6th Framework.

4.1.1 Premise

There are three kinds of premise: ordinary premises, exceptions and assumptions. Accordingly there are three constructor functions for premises:

`(make-orinary-premise atom polarity role)` constructs an ordinary premise.

`(make-exception atom polarity role)` constructs an exceptional premise, and

`(make-assumption atom polarity role)` constructs an assumption.

In these constructor functions, `atom` is a positive statement and `polarity` is a boolean value stating whether the premise is positive or negative. If `polarity` is true then the premise is positive, otherwise it is negative. The `role` is a string which can be used to name the role of the premise in the argument, such as `"major"` or `"minor"`. To provide no role, use the empty string, `""`.

`premise?`, `ordinary-premise?`, `exception?` and `assumption?` are predicates for testing the type of the premise:

`(premise? x)` tests whether some object `x` is a premise.

`(ordinary-premise? x)` tests whether the object `x` is an ordinary premise.

`(exception? x)` tests whether the object `x` is an exceptional premise, and

`(assumption? x)` tests whether `x` is an assumption.

`premise-atom`, `premise-polarity` and `premise-role` are selector functions for accessing the parts of a premise:

`(premise-atom x)` returns the statement of the atom `x`.

`(premise-polarity x)` returns the polarity of the atom `x`. If `(premise-polarity x)` is true, then `x` is a positive premise. Othewise it is a negative premise.

`(premise-role x)` returns string role of the premise as a string.

`(premise-statement x)` constructs a positive or negative statement, from the premise, `x`, depending on the polarity of the premise. If the polarity of the premise is not true, then the resulting statement is a negative literal. For example:

```
> (premise-statement (make-assumption '(agreement-with-minor ?x)
                                      #f
                                      'minor)))
(not (agreement-with-minor ?x))}
```

The `pr`, `am` and `ex` functions are convenient alternatives, when no role is needed, for constructing premises. The statement passed to these functions may be positive or negative. A negative or positive premise will be constructed, as appropriate. The atom of the resulting premise is always a positive statement.

`(pr s)` constructs an ordinary premise from the statement `s`.

`(am s)` constructs an assumption from the statement `s`.

`(ex s)` constructs an exception from the statement `s`.

`(premise=? p1 p2)` is true if `p1` and `p2` are equal premises. Two premises, `p1` and `p2`, are equal if `(statement-compare (premise-atom p1) (premise-atom p2))` is *0* and `p1` and `p1` have equal parity and roles.

`(positive-premise? p2)` checks if the parity of the premise `p` is true.

`(negative-premise? p1)` checks if the parity of the premise `p` is not true.

4.1.2 Argument

Arguments link a set of premises to a conclusion. There are two kinds of arguments, pro and con, depending on whether the premises support or oppose the conclusion of the argument.

`(make-argument id direction conclusion premises scheme)` constructs an argument. The `id` is a symbol naming the argument. It is the user's responsibility to assure this name is unique in the application context. The direction is a symbol, either `pro` or `'con`. The `conclusion` is a statement and `premises` is a list of premises. The `scheme` is a string naming the argumentation scheme applied to produce this argument, or the empty string `""` if the scheme is not provided.

`(argument? x)` is predicate for testing whether some value is an argument.

`(argument-id argument)` returns the symbol which is the id of the argument.

`(argument-direction argument)` returns the symbol `'pro` or `'con`.

`(argument-conclusion argument)` returns the statement which is the conclusion of argument.

`(argument-premises argument)` returns a list of the premises of the argument.

`(argument-scheme argument)` returns a string naming the scheme applied to construct the argument, or the empty string, `""`, if the scheme has not been provided.

The `pro` and `con` functions, not to be confused with the `'pro` and `'con` symbols representing the direction of an argument, provide a simpler interface for constructing argument when a scheme is not needed:

`(pro id conclusion premises)` returns a pro argument with the given id, conclusion and list of premises.

`(con id conclusion premises)` returns a con argument with the given id, conclusion and list of premises.

`define-argument` is a convenient syntactic abstraction (macro) for constructing an argument and assigning it to a Scheme variable with the same name as its id.

For example:

```
(define-argument a1
    (pro tweety-flies)
    (am birds-fly)
    (am tweety-isa-bird)
    (ex tweety-is-abnormal)))
```

has the same meaning as

```
(define a1
    (make-argument 'a1
       'pro '(flies Tweety)
        (list (am 'birds-fly)
              (am '(bird Tweety))
              (ex '(abnormal Tweety)))
        ""))
```

`(argument->datum a1)` represents the argument `a1` as a symbolic expression, in a format suitable for displaying the argument to humans.[2]

`(datum->argument expression)` constructs an argument from a symbolic expression of the form produced by `argument->datum`.

For example, the above argument about Tweety flying could be displayed like this:

```
> (argument->datum a1)

(argument
    (@ (id a1) (direction pro) (scheme #f))
    (assumption birds-fly)
    (assumption (bird Tweety))
    (exception (abnormal Tweety))
    (conclusion (flies Tweety)))
```

`(add-premise p a)` constructs an argument from the argument `a` by adding `p` as an additional premise. Enthymemes are arguments with implicit, unstated premises [Walton, 2006, p. 178]. `add-premise` provides a way to incrementally add premises to an argument as they become revealed.

[2]Well, at least programmers.

4.1.3 Argument Context

Arguments are exchanged in dialogues regulated by procedural rules, called 'protocols'. The participants of the dialogue take turn making various kinds of speech acts, such as asking questions, putting forward arguments, raising issues or making decisions. If argumentation is viewed as a kind of game, metaphorically, then such speech acts are moves in the game. After each move, the state of the game changes. When evaluating arguments, the state of the dialogue in which the arguments have been put forward needs to be taken into consideration.

The state of dialogue, insofar as it is relevant for evaluating arguments, is modeled by an argument `context` recording the dialogical `status` of each statement used in the dialog (`'stated`, `'questioned`, `'accepted`, or `'rejected`), the `proof-standard` currently assigned to each statement, and a priority relation on arguments.

`(status? x)` checks whether the value `x` is a symbol denoting the status of statement in a dialogue: `'stated`, `'questioned`, `'accepted`, or `'rejected`.

`(proof-standard? x)` checks whether the value `x` is a symbol denoting a proof standard: `'se` (scintilla of evidence), `'ba` (best argument) or `'dv` (dialectical validity). See [Gordon et al., 2007] for definitions of these proof standards.

`(make-context status standard compare)` returns a context constructed from `status`, a table mapping statements to their dialectical status, `standard`, a function mapping statements to their proof standard, and `compare`, a function for partially ordering arguments by their strength. `(compare a1 a2)` is `1`, if `a1` is strictly stronger than `a2`, `0`, if `a1` and `a2` are equally strong, and `-1`, if `a2` is strictly stonger than `a1`.

`(context? x)` tests whether `x` is a context.

`(context-status c)` returns the status table of the context `c`.

`(context-standard c)` returns the function mapping statements to their proof standard in the context `c`.

`(context-compare c)` returns the function of the context `c` which partially orders arguments by their strength.

`default-context` is an argument context which assigns the `dv` standard, dialectical validity, to all statements and considers all arguments to have the same priority. That is, for all arguments `a1` and `a2`, `(prior default-context a1 a2)` is *#f*.

`(status c s)` returns the status of statement `s` in context `c`.

`(proof-standard c s)` returns the proof standard of `s` in `c`.

`(prior c a1 a2)` tests whether argument `a1` has priority over argument `a2` in context `c`.

`(state c l)` returns the context resulting from assigning each statement in a list of statements, `l`, in the context `c`, the status `'stated`.

`(question c l)` returns the context resulting from assigning each statement in a list of statements, `l`, in the context `c`, the status `'questioned`.

`(accept c l)` returns the context resulting from assigning each statement in a list of statements, `l`, in the context `c`, the status `'accepted`.

`(reject c l)` returns the context resulting from assigning each statement in a list of statements, `l`, in the context `c`, the status `'rejected`.

`(assign-standard c ps s)` returns a context in which the proof standard assigned to `s` in `c` is changed to `ps`.

The `state`, `question`, `accept` and `reject` functions each process a list of statements, so that the status or proof-standard of several statements can be changed together. For example, several statements can be accepted as in this example: `(accept c s1 s2 s3)`. And the dialectical validity proof standard can be assigned to these statements using `(assign-standard c 'dv s1 s2 s3)`.

The following functions are for checking the status of some statement in a context, as special purpose alternatives to the `status` function.

`(stated? c s)` is true iff `s` is `'stated` in `c`.

`(questioned? c s)` is true iff `s` is `'questioned` in `c`.

`(accepted? c s)` is true iff `s` is `'accepted` in `c`.

`(rejected? c s)` is true iff `'rejected` is in `c`.

`(decided? c s)` is true just if `s` is `'accepted` or `'rejected` in `c`.

`(issue? c s)` is true just when `s` is not decided in `c`.

4.1.4 Argument Graph

Argument graphs, also called *inference graphs* [Pollock, 1992] are directed, acyclic, bipartite graphs, consisting of statement nodes and argument nodes representing proofs. The functions documented in this subsection provides methods for constructing argument graphs and retrieving parts of argument graphs.

`empty-argument-graph` is an argument graph with neither argument nor statement nodes.

`(argument-graph? x)` checks whether a value is an argument graph.

`assert*`, `assert` and `update` extend argument graphs with further arguments:

`(assert* ag arg replace)` constructs an argument graph by extending `ag` with the argument `arg`. If `replace` is true, then an argument in `ag` with the same id as the id of `arg` will be replaced by `arg` in the resulting argument graph. If `replace` is false, the argument will not be replaced and an error will be raised. Thus `assert*` is a partial function which is not defined if an existing argument is not to be replaced.

`(assert ag arg1 ... argn)` extends `ag` with `arg1` to `argn`, using `(assert ag argi #f)`. That is, no argument with the same id will be replaced. If any argument in `arg1` to `argn` has the same id as any other argument in this sequence, or an argument in `ag`, an error will be raised.

`(update ag arg1 ... argn)` is similar to `assert`, but replaces any argument with the same id. The arguments in `arg1` to `argn` are added from left to right, so if any argument `argi` has the same id as an argument `argj`, where `i` is less than `j`, then `argi` will be replaced by `argj` in the resulting argument graph.

`arguments` and `statements` return a list of the arguments and statements, respectivey, of an argument graph:

`(arguments ag)` returns a list of the arguments of the argument graph `ag`.

`(statements ag)` returns a list of the statements of the argument graph `ag`.

`(questions ag)` returns a list of all the questioned statement in `ag`.

`(facts ag c)` returns a list of all the statements in `ag` which have been accepted or rejected in context `c`. Recall that all statements in an argument graph are atomic. If a statement `atom` in an argument graph `ag` has been rejected in context `c`, then `(not atom)` will be a member of `(facts ag c)`.

`accepted-statements`, `rejected-statements`, `stated-statements` and `issues` return a list of the accepted, rejected, stated and undecided statements, respectively, of an argument graph:

`(accepted-statements ag c)` returns a list of the accepted statements of an argument `ag` in the context `c`.

`(rejected-statements ag c)` returns a list of the rejected statements of an argument `ag` in the context `c`.

`(stated-statements ag c)` returns a list of the stated statements of an argument `ag` in the context `c`.

`(issues ag c)` returns a list of all of the undecided statements of `ag` in context `c`.

`(relevant-statements ag goal)` returns a list of all of the statements in argument graph `ag` which are relevant for assessing the acceptability of some `goal` statement. A statement `s` is relevant for proving `goal` if the acceptability `goal` depends on the acceptability of `s`. A statement `s1` depends on a statement `s2` is `s1` equals `s2`, `s2` is the statement of premise of an argument pro or con `s1` or, recursively, depends on the statement of a premise of `s1`. Notice that `goal` is, according to these definitions, relevant for proving itself and will thus be included in the resulting list.

`(schemes-applied ag s)` returns a list of the names (symbols) of all the schemes which have been applied in arguments pro or con statement `s` in argument graph `ag`.

4.1.5 Argument Evaluation

By argument evaluation, we mean determing whether a statement in an argument graph satisfies its proof standard in some context.

`(satisfies? ag c s ps)` is true iff the statement `s` of argument graph `ag` satisfies the proof-standard `ps` in the context `c`. This depends of course on the selected proof standard. The `'se` standard, scintilla of the evidence, is satisfied only the statement `s` is supported by at least one defensible pro argument in `ag`. The `'dv` standard, dialectical validity, is satisfied only if `s` is supported by at least one defensible pro argument in `ag`, like scinctilla, but requires in addition that `ag` contain no defensible argument argument con `s`. Finally, the `'ba` standard (best argument) is satisfied only if `ag` contains a defensible argument pro `s` which has priority over every defensible argument in `ag` con `s`.

`(acceptable? ag c s)` is true just if `s` satisfies its proof standard in context `c`, given the arguments in the argument graph `ag`.

`(holds? ag c p)` is true if the premise `p` of argument graph `ag` holds in the context `c`. This depends on the type of premise. An ordinary premise holds if it is a positive premise and its statement has been accepted in context `c` or if is a negative premise and its statement has been rejected in `c`. If the statement of an ordinary premise is undecided, then the premise holds only if the statement is acceptable. An assumption is like an ordinary premise in most respects. It holds under the same conditions unless its statement has not been questioned or decided, i.e. it is merely `'stated`. Whereas a stated ordinary premise holds only if its statement is acceptable, a stated assumption holds whether or not its statement is acceptable. Finally, a negative exception holds if its statement has been accepted and a positive exception holds if its statement has been rejected. If the statement of an exception is undecided, the exception holds only if the statement is not acceptable. See [Gordon et al., 2007] for further details.

`(all-premises-hold? ag c arg)` is true if all of the premises of `arg` hold in `c`. Such arguments were called ‘defensible’ in [Gordon et al., 2007].

4.2 Argument Diagram

The `argument-diagram` module provides functions for visualizing argument graphs.

`(diagram* ag c subs statement->string p)` constructs a string in the DOT language [Gansner and North, 2000, Koutsofios and North, 1993] representing a diagram of the argument graph `ag` in the context `c` and writes it out to the port `p`. Any variables are replaced with their values in the substitution environment `subs`. The `statement->string` function is responsible for translating statements into the form to be displayed in boxes in the diagram.

`(view* ag c subs statement->string)` uses `diagram*` to produce a diagram of the argument graph `ag` and displays the resulting DOT file using the `viewer` specified in the `config` module. See Section 2.1 for further information about the `Config` module.

`(diagram ag c)` provides a simplified interface to the `diagram*` function, using the `identity` substitution environment and the standard Scheme `write` procedure, from the `(rnrs io simple)` library to convert statements to strings. The diagram is written to the `current-output-port`.

`(view ag c)` provides a simplified interface to the `view*` function, using the `identity` substitution environment and the standard Scheme `write` procedure, from the `(rnrs io simple)` library to convert statements to strings.

We will illustrate the use of the `argument-diagram` module with the following argument-graph, represented in the Legal Knowledge Interchange Format:

```
<?xml version="1.0" encoding="UTF-8"?>
<lkif>
  <argument-graphs>
    <argument-graph>
      <statements>
        <statement id="contract" value="unknown" assumption="true">
          <s>There is a contract</s>
        </statement>
        <statement id="writing" value="unknown" assumption="true">
          <s>The agreement is in writing.</s>
        </statement>
        <statement id="real-estate" value="unknown" assumption="true">
          <s>The agreement is for the sale of real estate.</s>
        </statement>
        <statement id="agreement" value="unknown" assumption="true">
          <s>There is an agreement.</s>
        </statement>
        <statement id="minor" value="unknown" assumption="true">
          <s>One of the parties is a minor.</s>
        </statement>
  <statement id="email" value="unknown" assumption="true">
    <s>The agreement was by email.</s>
  </statement>
  <statement id="deed" value="true" assumption="true">
    <s assumable="true">There is a deed.</s>
  </statement>
</statements>

<arguments>
  <argument id="a1" direction="pro">
    <conclusion statement="contract"/>
    <premises>
      <premise statement="agreement"/>
      <premise exception="true" statement="minor"/>
    </premises>
  </argument>

  <argument id="a2" direction="con">
    <conclusion statement="contract"/>
    <premises>
      <premise polarity="negative" statement="writing"/>
      <premise statement="real-estate"/>
    </premises>
  </argument>

  <argument id="a3" direction="con">
    <conclusion statement="writing"/>
    <premises>
      <premise statement="email"/>
    </premises>
  </argument>

  <argument id="a4" direction="pro">
    <conclusion statement="agreement"/>
    <premises>
      <premise statement="deed"/>
    </premises>
  </argument>

  <argument id="a5" direction="pro">
    <conclusion statement="real-estate"/>
    <premises>
      <premise statement="deed"/>
    </premises>
  </argument>
</arguments>
```

```
    </argument-graph>
  </argument-graphs>
</lkif>
```

Suppose this XML is stored in a file named "contract.xml". The following R6RS Scheme program uses the `argument-diagram` module, together with the LKIF module of Section 6, to generate and view a diagram of this example argument graph:

```
#!r6rs
(import (rnrs)
        (carneades lkif2)
        (carneades argument-diagram))
(define import-data (lkif-import "contract.xml"))
(define stages (lkif-data-stages import-data))
(define st1 (car stages))
(view (stage-argument-graph st1) (stage-context st1))
```

This program causes the diagram shown in Figure 4.1 to be displayed.

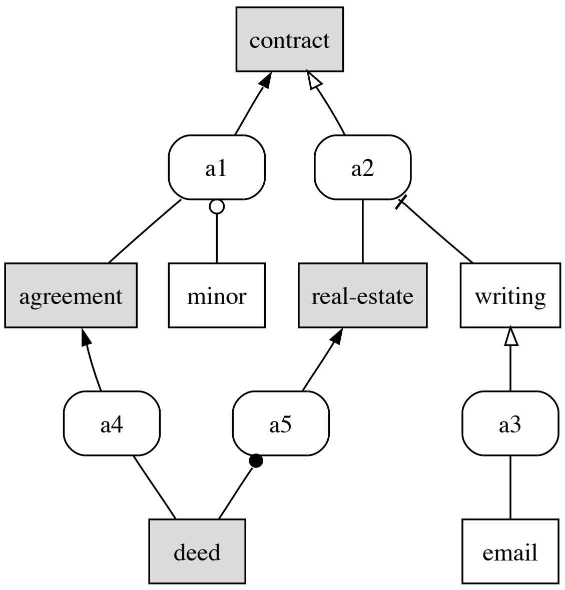

Figure 4.1: Example Argument Diagram

4.3 Argument Search

The `argument-search` module applies argument generators to search for argument graphs in which some goal statement is either acceptable or not acceptable, depending on the viewpoint.

A `state` here is a data structure for representing the states in a search space of argument graphs.

A `viewpoint` is either `'pro` or scm'con some goal statement. The opposing viewpoint of `'pro` is `'con` and vice versa.

`(opposing-viewpoint viewpoint)` equals `'con` if `'pro` and, vice versa, equals `'pro` if `viewpoint` equals `'con`.

`(make-state topic viewpoint pro-goals con-goals context substitutions arguments)` constructs a state. `topic` is the main statement at issue. `viewpoint` is the viewpoint, `'pro` or `'con` used to construct this state. The primary goal of the `'pro` viewpoint is to try to construct or find argument graphs in which the `topic` is acceptable. The primary goal of the `'con` viewpoint, on the other hand, is to try to construct argument graphs in which `topic` is not acceptable. The pro and con goals of the state, `pro-goals` and `con-goals`, are lists of suggested statements to try to prove, as subgoals, in order to advance the pro and con viewpoints, respectively. These lists of goals provide heuristic information useful for focusing and controlling the search for arguments and counterarguments. The `context` of a state is the argument context, as defined in the argument module (Section 4.1.3), to use when evaluating whether some goal statement is acceptable or not in the argument graph. `substitutions` is the substitution environment (Section 3.2) to use when unifying goals with statement patterns in methods for generating arguments instantiating various argumentation schemes, such as arguments from rules. Finally, `arguments` is the state's argument graph.

`(initial-state goal c)` constructs the initial or root state in the search space for the given `goal` statement in the `c` context. In the initial state, the viewpoint is `'pro` if the `goal` statement is a positive literal and `'con` if the statement is a negative literal. The topic of the state is the atom of goal statement; that is, if the goal statement is negative, the negation operator is stripped from the goal to form the topic. The list of pro goals of the initial state consists of just the topic statement. The list of con goals is initially empty. The initial substitution environment is `identity` and the initial argument graph is the `empty-argument-graph`.

`(state? x)` is a predicate for checking whether some value is a state in the search space of argument graphs.

`state-topic`, `state-viewpoint`, `state-pro-goals`, `state-con-goals`, `state-context`, `state-substitutions`, and `state-arguments` are functions for selecting the parts of an argument state:

`(state-topic s)` returns the statement which is the main topic of state `s`.

`(state-viewpoint s)` returns the viewpoint of the state `s`, either `'pro` or `'con`.

`(state-pro-goals s)` returns a list of the pro goals of the state `s`, where each goal is a statement.

`(state-con-goals s)` returns a list of the con goals of the state `s`, where each goal is a statement.

`(state-context s)` returns the context of the state `s`.

`(state-substitutions s)` returns the variable substitutions of the state `s`.

`(state-arguments s)` returns the argument graph of the state `s`.

An *argument-generator* is a function mapping a `state` in the search space of argument graphs to a stream of *responses*, where each *response* is a record consisting of a substitution environment and an argument graph. This definition of argument generator is intentionally quite abstract; it is independent of the way knowledge or evidence used to construct arguments is represented and of the argumentation schemes instantiated by the arguments so constructed.

`(make-response subs arg)` contructs a response from a substitution environment, `subs`, and an argument, `arg`.

`(response? x)` is a predicate which tests whether an object, `x`, is a response.

`(response-argument response)` returns the argument of the response.

`(response-substitutions response` returns the substitution environment of the response.

Generators are used to construct a search space for arguments. Recall from Section 2.6 that a search *strategy* is a function which maps a problem to a stream of nodes. Strategies provided by the scmsearch module include depth-first search, breadth-first search and iterative deepening. Other search strategies can be defined by users in a similar way. Such strategies are defined in a way which is independent of the problem space to be searched.

`(find-arguments strategy resource state generators)` uses `(strategy resource)` to search the space defined by `(make-root state)` and the transitions available using the argument generators. A goal state in this space contains an argument graph whose `topic` statement is acceptable in this argument graph, given the context of the state, if the viewpoint of the state is `'pro`, or not acceptable, if the viewpoint of the state is `'con`. `find-arguments` returns a stream of goal states. Each goal state represents an alternative argument graph pro or con the topic, depending on the viewpoint of the state. If no such argument graph can be found, the resulting stream will be empty. Since the elements of a stream are computed on demand, accessing successive elements of the stream causes the search procedure to backtrack to try to find alternative arguments.

`(find-best-arguments strategy resource max-turns state generators)` returns a stream of the best arguments which can be found, both pro and con the topic statement, given at most `max-turns` turns, where the viewpoint is alterated between pro and con on each turn. An argument graph of some state `s1` on turn `i` is considered best if no counterarguments can be found on turn `(+ i 1)` using the available `generators` and the remaining resources.

Examples showing how to use the `argument-search` module will have to wait until Section 5, where one way to generate arguments is presented, using defeasible rules. Without arguments generators, there is no space to search.

Chapter 5

Inference Layer

The inference layer provides modules for constructing arguments from ontologies (Section 5.1), defeasible inference rules (Section 5.2), precedent cases (Section 5.4, and testimonial evidence (Section 5.5). Also provided in the inference layer is a module (Section 5.3) which enables rulebases to embed Scheme expressions in defeasible inference rules and have them evaluated, using Scheme's `eval` procedure, when searching for arguments. This module also provides methods for constructing arguments about standard LKIF predicates.

5.1 Ontologies

In computer science, an ontology is a representation of concepts and relations among concepts, typically expressed in some decidable subset of first-order logic, such as description logic [Baader et al., 2003]. Such ontologies play an important role in integrating systems, by providing a formal mechanism for sharing terminology, and also in the context of the Semantic Web [Berners-Lee et al., 2001] for providing machine-processable meta-data about web resources and services. There is a World Wide Web standard for modeling ontologies, based on description logic, called the Web Ontology Langauge (OWL) [McGuinness and van Harmelen,]. LKIF, the Legal Knowledge Interchange Format [ESTRELLA Project, 2008b], provides a way to import OWL ontologies.

In this section, we present a module of the reference inference engine which provides one way to construct arguments from ontologies. KRSS [Patel-Schneider and Swartout, 1994] is the format for ontologies supported directly by the reference inference engine. KRSS was chosen, rather than OWL, because KRSS represents ontologies using Lisp symbolic expressions, which is a native Scheme data type and thus very convenient to use in Scheme programs. OWL ontologies can be converted automatically to KRSS, for example by exporting the OWL ontology to KRSS using the 4.x version of the Protege ontology editor [for Biomedical Informatics Research, 2007]. A future version of this module may provide procedures for importing OWL files.

The Description Logic Programming (DLP) dialect of description logic [Grosof et al., 2003] is supported by this module. DLP is semantically within the Horn clause subset of first-order logic, which enables DLP programs to be translated into rules which can be used with the rule module of the reference inference engine

(Section 5.2) to construct arguments. Thus, the ontology module can be viewed as a compiler from ontologies to rules.

The advantage of DLP is that it allows reasoning with ontologies and defeasible rules to be integrated in a simple, transparent way which produces argument graphs suitable for use in explanations and justifications of decisions. A disadvantage of this approach is that the DLP dialect of description logic is somewhat less expressive than the dialect of description logic dialect supported by OWL. Thus some of the axioms in an OWL ontology will be ignored when using this DLP approach, causing some possible arguments to be missed.

There is an alternative approach for using ontologies to construct arguments, but this alternative approach is applicable only in scenarios where all of material facts of the case have already been determined, or can be assumed to have been determined, before defeasible rules are applied to construct arguments about the legal consequences of the facts. Such fact or data-driven scenarios are not uncommon in public administration, for example when processing forms, but they depend on strong simplifying assumptions about the nature of legal reasoning. In general the tasks of finding the facts of a case and assessing their legal consequences are intertwined, requiring some shifting from one task to the other, in an iterative fashion, trying to construct a coherent theory of both the facts and the law.

Let us call this alternative approach the *preprocessing* approach, and the approach supported by the ontology model of the reference inference engine the *direct* approach, since the description logic axioms are interpreted directly by the reference inference engine in this approach.

The steps of the preprocessing approach are as follows:

1. Let $\mathcal{I}$ be a set of constants denoting individuals of the case and let $\mathcal{F}$ be a set of ground, atomic formulas, called the *material facts*, about these individuals. Let us assume that all these atomic formulas are represented in the description logic subset of OWL (OWL-DL).

2. Let $\mathcal{O}$ be an ontology of the concepts of the applicable legal domain, represented in OWL-DL. Using a state-of-the-art description logic theorem prover, such as Pellet [Sirin et al., 2006], compute all of the classes in $\mathcal{O}$ to which each individual in $\mathcal{I}$ belongs. (The reasoning service provided by description logic theorem provers which enables the classification is called *realization.*)

3. Construct an extended set of material facts, $\mathcal{I}'$, consisting of all of the formulas of $\mathcal{I}$ plus a formula $C(i)$ for each individual $i\mathcal{I}$ and class C determined by the previous step, using the ontology reasoner, to be a class to which the individual i belongs.

4. Construct an LKIF theory in which each material fact in $\mathcal{I}'$ is represented as an axiom in first-order logic, using the standard translation from description logic to first-order logic [Grosof et al., 2003].

5. Construct an initial argument context, C_1, (Section 4.1.3) by accepting every axiom in the theory.

6. Construct a generator, G, for arguments from the rules of the theory (Section 5.2).

7. Given a goal statement, P, use the argument generator G to search for argument graphs, starting with the context C, to find graphs in which P is acceptable or not acceptable, depending on the your standpoint and interests.

The preprocessing approach is the one used by the Harness methodology for case assessment [ESTRELLA Project, 2008a]. It is important to notice that this methodology is supported by the ESTRELLA reference inference engine, used together with any OWL reasoner which provides the realization service.

For the sake of comparison, here are the steps of the direct approach supported by the ontology module of the reference inference engine:

1. Let $\mathcal{O}$ be an ontology of the concepts of the applicable legal domain, represented in OWL. Serialize this ontology using the KRSS syntax [Patel-Schneider and Swartout, 1994]. This can be done using the 4.x version of the Protege ontology editor [for Biomedical Informatics Research, 2007] [1].

2. Using the LKIF module (Section 6), import the LKIF file containing the theory, i.e. the axioms and defeasible rules, to be used to construct arguments.

3. Construct a generator G_1 for arguments from the ontology $\mathcal{O}$, using the ontology module described in this section, and a generator for arguments from the rules in the theory, using the rules module (Section 5.2) and use both of these generators to construct for arguments.

4. Construct an initial argument context, C_1, (Section 4.1.3) by accepting every axiom in the theory.

5. Construct a generator, G_2, for arguments from the rules of the theory (Section 5.2).

6. Given a goal statement, P, use both of the argument generators, G_1 and G_2, to search for argument graphs, starting with the context C, to find graphs in which P is acceptable or not acceptable, depending on the your standpoint and interests.

Notice that the last three steps of both approaches is almost the same, except that in the direct approach arguments are generated from both the ontology and the rules of the theory, dynamically, whereas in the preprocessing approach all the propositions which can be deduced from the ontology have been statically assumed as axioms of the theory and only rules are used to construct arguments.

Now, finally, here are the procedures provided by the ontology module, called `dlp`, because of it support for Description Logic Programming.

`(ontology krss-axiom ...)` is a macro which compiles a sequence of ontology axioms, represented in KRSS, into a rulebase (Section 5.2).

[1] `http://protege.stanford.edu/`

`(generate-arguments-from-ontology o1)` constructs a procedure for generating arguments from the ontology `o1`, where `o1` is a rulebase constructed using the `ontology` macro described above.

Here's a small example Scheme program, about family relations, illustrating how ontologies can be represented in KRSS and used to construct arguments:

```
#!r6rs
(import (rnrs)
        (carneades dlp)
        (carneades shell)
        (carneades argument-builtins)
        (carneades lib srfi lightweight-testing))

(ontology kb1
  (related Caroline Tom parent)
  (related Caroline Ines parent)
  (related Dustin Tom parent)
  (related Dustin Ines parent)
  (related Tom Gloria parent)
  (related Ines Hildegard parent)
  (instance Tom Male)
  (instance Tom Parent)
  (instance Ines Female)
  (instance Ines Parent)

  (define-primitive-role parent ancestor)
  (define-concept Father (and Male Parent))
  (define-concept Mother (and Female Parent))
  (define-primitive-role (transitive-closure ancestor) ancestor))

(define (engine max-nodes max-turns)
  (make-engine max-nodes max-turns
               (list (generate-arguments-from-ontology kb1 '()))))

(define e1 (engine 100 10))

(check (all-acceptable? '(parent ?x ?y) e1) => #t)
(check (all-acceptable? '(ancestor ?x ?y) e1) => #t)
(check (failure? '(parent Hildegard Tom) e1) => #t)
(check (all-acceptable? '(applies ?r (parent ?x ?y)) e1) => #t)
(check (all-acceptable? '(Father Tom) e1) => #t)

(check-report)
```

5.2 Defeasible Inference Rules

The `rule` module implements generators for arguments from defeasible inference rules [Gordon, 2007, Gordon, 2008b, Gordon, 2008a]. Rulebases represented in Legal Knowledge Interchange Format can be imported using the LKIF module (Section 6).

The rule langauge supported by the reference inference engine, i.e. this module, is not as expressive as the LKIF rule language. The restrictions are as follows:

- The head of a rule may consists only of a sequence of literals, not arbitrary first-order logic formulas.
- The `all` and `exists` quantifiers are not supported.

Despite these limitations, the rule language supported by this module is still very expressive, certainly compared to most logic programming langauges based on Horn clause logic:

- Multiple conclusions are allowed in the head of rules.
- Both positive and negative literals are allowed in the head of rules.

- With the exception of quantiers, the rule bodies may contain a sequence of arbitrary formulas of first-order logic.

These extensions to Horn-clause logic make it much easier to model the rules of legislation *isomorphically*, i.e. in a way which preserves their structure [Bench-Capon and Coenen, 1992].

Just as Lisp symbolic expressions (s-expressions) are used to represent ontologies in the ontology module (Section 5.1), so too are symbolic expression used to represent rules in this module. The s-expression syntax for rules follows.

```
<statement> =
    <atom>
  | (not <atom>)

<rule> =
    (<symbol> (if <body> <head>))
  | (<symbol> <statement> ...)

<head> =
    <statement>
  | (and <statement> ...)

<condition> =
<statement>
  | (unless <statement>)
  | (assuming <statement>)
```

```
<clause> =
<condition>
  | (and <condition> ...)

<body> =
    <clause>
  | (or <clause> ...)
```

The `symbol` in these forms is for the id of the rule being defined. The head of a rule consists of one or more statements. Similarly, the body of a rule consists of one or more conditions. Here are a couple of example rules, using this syntax:

```
(rule r1
    (if (parent ?x ?y)
        (ancestor ?x ?y)))

(rule r2
  (if (and (ancestor ?x ?z)
           (ancestor ?z ?y))
      (ancestor ?x ?y)))
```

The second rule form is for conditionless rules. These are something like the 'facts' of Prolog and Horn clause logic, except they are defeasible and several can be included together in a single 'clause'. Here is an example:

```
(rule facts
    (parent Caroline Tom)
    (parent Caroline Ines)
    (parent Dustin Tom)
    (parent Dustin Ines)
    (parent Tom Gloria)
    (parent Ines Hildegard))
```

The following procedures are provided by the `rule` module.

`(make-rule id strict head body)` constructs a rule with the given `id`, `head` and `body`. If the rule is `strict`, then it is not subject to the 'critical questions' usually applicable to rules. They cannot be excluded by other rules, defeated by rules of higher priority or determined to be invalid.

`(rule? x)` checks whether some value `x` is a rule. `rule-id`, `rule-strict`, `rule-head`, and `rule-body` are functions for accesssing the parts of a rule.

`(rule ...)` is a macro providing convenient forms for defining rules, using grammar above. Rules constructed with

`(rule* ...)` is macro is a variations of the `rule` macro, for constructing *strict* rules, i.e. rules which are not subject to any critical questions. (Notice that this interpretation of 'strict' is weaker than the meaning of strict rules in Defeasible Logic [Nute, 1994].)

`(rulebase r1 ... rn)` constructs a rulebase from a sequences of rule expressions, `r1` to `rn`, where each expression is an application of the `rule` or `rule*` macro.

`empty-rulebase` is, as its name suggests, a rulebase with an empty set of rules.

`(add-rules rb l)` constructs a rulebase by extending the rulebase `rb`, nondestructively, with the rules in the list `l`. Values in `l` which are not rules are ignored.

To illustrate, let's use the `rulebase` function to package the rules above in a `family` rulebase :

```
(define family
    (rulebase
        (rule r1
            (if (parent ?x ?y)
                (ancestor ?x ?y)))

        (rule r2
          (if (and (ancestor ?x ?z)
                   (ancestor ?z ?y))
              (ancestor ?x ?y)))

        (rule facts
            (parent Caroline Tom)
            (parent Caroline Ines)
            (parent Dustin Tom)
            (parent Dustin Ines)
            (parent Tom Gloria)
            (parent Ines Hildegard))))
```

`(rulebase-rules rb)` returns a list of the rules in the rulebase `rb`. The rulebase itself is not implemented as a list, but rather uses a hash table to efficiently retrieve relevant rules about particular predicates.

`(generate-arguments-from-rules rb questions)` constructs a procedure for generating arguments from rules, where `rb` is a rulebase and `questions` is a list of the 'critical questions' [Walton, 2006] to use when constructing arguments against the applicability of rules. `questions` must be a list of symbols, all of which are members of `'(excluded priority valid)`, which name following critical questions:

excluded. Is some other rule applicable which excludes the applicability of this rule?

priority. Is some some conflicting rule applicable which has priority over this rule?

valid. Is this rule (still) valid? Has it, for example, been repealed?

`(rule->datum r)` translates the rule `r` into a symbolic expression, using the same syntax for rules implemented by the `rule` and `rule*` macros described previously. (A symbolic expression is called a 'datum' in Scheme.)

`(rulebase->datum rb)` translates an entire rulebase into a symbolic expression.

For example, using the `family` rulebase defined above:

```
> (rulebase->datum family))

((rule* r1 (if (parent ?x ?y) (ancestor ?x ?y)))
 (rule* r2 (if (and (ancestor ?x ?z) (ancestor ?z ?y)) (ancestor ?x ?y)))
 (rule* facts
  (parent Caroline Tom)
  (parent Caroline Ines)
  (parent Dustin Tom)
  (parent Dustin Ines)
  (parent Tom Gloria)
  (parent Ines Hildegard)))
```

To illustrate how the output format of `rulebase->datum` is the same as the format used by `rule` and `rule*` to input rules, let's construct another rulebase with copies of the rules in the `family` rulebase:

```
(define family-copy
    (apply rulebase
        (map eval (rulebase->datum family))))
```

Finally, to complete this chapter on rules, below is a sample Scheme program showing how to use a rulebase to construct arguments:

```
#!r6rs
(import (rnrs base)
        (carneades shell)
        (carneades argument-builtins)
        (carneades rule)
        (carneades lib srfi lightweight-testing))

(define null '())

(define rb1
  (rulebase

   (rule r1
         (if (and (movable ?c)
                  (unless (money ?c)))
             (goods ?c)))

   (rule r2 (if (coins ?x) (money ?x)))


   (rule* facts
          (movable item1)
          (coins item1))

   )) ; end of rule base

(define (engine max-nodes max-turns critical-questions)
  (make-engine max-nodes max-turns
               (list (generate-arguments-from-rules
                         rb1
                         critical-questions))))

; pro argument. coins are goods since they are movable:
(check (all-acceptable? '(goods item1) (engine 20 1 null)) => #t)

; con argument: coins are not goods since they are money:
(check (all-acceptable? '(goods item1) (engine 20 2 null))=> #f)
(check-report)
```

5.3 Argument Builtins

A `argument-builtins` provides rulebases with an API to the Scheme programming language, allowing Scheme expressions to be embedded in defeasible inference rules (Section 5.2) and evaluated when searching for arguments. The `argument-builtins` module also defines rules for a number of builtin-predicates, such as the `(priority rule1 rule2)` predicate for reasoning about rule priorities.

This module exports only one value, the `builtins` argument generator:

> Currently, `argument-builtins` constructs arguments about two standard predicates, `eval` and `prior`:evaluates the Scheme expression `expr` in the `(environment '(rnrs))` environment. Before the expression is evaluated, any logical variables in the expression are first substituted by their values in the substitution environment of the state of the search space. The result of the evaluation is unified with term `term`, which may be a logical variable. The goal succeeds only if the evaluation of the expression does not raise a Scheme exception and the result of the evaluation is unifiable with `term`. Otherwise the goal fails. defines the standard `prior` predicate for resolving conflicts among conflicting rules. It is implemented using the following defeasible inference rules:

```
 (eval term expr)(priority ?r1 ?r2) (rule* priority1
      (if (and (applies ?r2 ?p1)
               (prior ?r2 ?r1))
          (priority ?r2 ?r1 (not ?p1))))

    (rule* priority2
      (if (and (applies ?r2 (not ?p1))
               (prior ?r2 ?r1))
```

```
        (priority ?r2 ?r1 ?p1)))
```

The `argument-builtins` module uses the `rule` module (Section 5.2) to generate arguments from these rules. The `prior` predicate used by these rules is to be defined by the rules of the particular legal domain. The `applies` predicate used by these rules is built into the inference engine of the `rule` module, but unlike the `priority` predicate is not defined using rules.

5.4 Precedent Cases

There are various forms of case-based reasoning. The simplest forms are variations of the scheme for argument from analogy, which use some similarity measure to compare cases. More complex schemes compare theories constructed from a set of cases, and order competing theories by their coherence. The `case` module presented in this section implements Wyner and Bench-Capon's reconstruction of the CATO [Aleven, 1997] model of analogical case-based reasoning as a set of argumentation schemes [Wyner and Bench-Capon, 2007]. CATO, in turn, is a refinement of Ashley's work on HYPO [Ashley, 1990].

The challenge when modeling reasoning by analogy is to operationalize the concept of similarity. In CATO, a casebase is about a particular *issue*, such as, in a case-base about family law, whether providing support to a family member would cause undue hardship. A case is modeled as a set of propositional *factors*, arranged in a *factor hierarchy*. Each factor favors one side of the issue. Factors in favor of the proposition at issue are called *plaintiff factors*; factors against the proposition at issue are called *defendant factors*. Two cases are considered similar if they have factors in common. If two conflicting precedents are similar to the current case, the argument from the more *on-point* case is stronger. Let cc be the current case. Let $F(c)$ be the set of factors of the current case, $F(p_1)$ be the factors of a precedent case, p_1, and $F(p_2)$ be the factors of another precedent case, p_2. Then p_1 is more one point than p_2 if and only if:

$$F(p_1) \cap F(c) \supset F(p_2) \cap F(c)$$

Arguments are constructed by comparing the set of factors of the current case with the factors of precedent cases. Each precedent case is modeled as a set of factors together with the decision of the case regarding the issue of the casebase, undue hardship in our example. Wyner and Bench-Capon defined seven *partitions* of the set of factors of a precedent case compared to the current case. Each partition maps a case to a set of factors. For example, 0irst partition is the intersection of the plaintiff factors in precedent case and the current case. Similarly, the second partition is the intersection of the defendant factors of precedent case and the current case.

Wyner and Bench-Capon defined six case-based argumentation schemes using these partitions. The three example schemes below are based on Wyner and Bench-Capon's, but reflect more closely their implementation in Carneades.

Factor Comparison Scheme

Let cc be the current case and q be the proposition at issue in the casebase.

Premise. The factors of the current case favor party p, denoted $\mathsf{factors\text{-}favor}(p)$.

Conclusion. q, if $p = \mathsf{plaintiff}$, otherwise $\mathsf{complement}(q)$.

Preference from Precedent Scheme

Let cc be the current case, pc_1 be some precedent case and p be a party.

P Factors Premise. P_1 is $\mathsf{partition}_1(pc_1)$.

D Factors Premise. P_2 is $\mathsf{partition}_2(pc_1)$.

Outcome Premise. $\mathsf{decision}(pc_1) = \mathsf{plaintiff}$.

Counterexample Exception. There exists a precedent case, pc_2, such that $\mathsf{is\text{-}counterexample}(pc_2, pc_1)$.

Conclusion. $\mathsf{factors\text{-}favor}(p)$

Counterexample Scheme

Let pc_1 and pc_2 be precedent cases.

Premise. $\mathsf{decision}(pc_1) = p_1$

Premise. $\mathsf{decision}(pc_2) = \mathsf{other\text{-}party}(p_1)$

Premise. $\mathsf{more\text{-}on\text{-}point}(pc_2, pc_1)$.

Conclusion. $\mathsf{is\text{-}counterexample}(pc_2, pc_1)$

Here are the procedures exported by the `case` module:

`(other-party party)` returns `'defendant` if `party` equals `'plaintiff` and vice versa.

`(make-factor statement favors parent)` constructs a factor from the proposition `statement`, where `favors` equals the party favored by the faactor, `'plaintiff` or `'defendant`, and `parent` is the parent factor of this factor or the boolean value false, `#f`, if this factor has no parent.

`(factor? x)` tests whether the value `x` is a factor.

`(factor-statement f)` returns the statement of the factor `f`.

`(factor-favors f)` returns the party favored by the factor `f`.

`(factor-parent f)` returns the parent factor of the factor `f`.

`(make-factors l)` maps a list of factors, `l`, to a set of factors containing just the elements of this list.

`(make-case name winner factors)` constructs a case with the given `name`, a string, where `winner` is the party, `'plaintiff` or `'defendant`, in whose favor the case was decided, or `'undecided` if the case has not been decided, and `factors` is a *list* (not set) of the factors of the case.

`(case? x)` tests whether the value `x` is a case.

`(case-name c)` returns the string naming the case `c`.

`(case-pfactors c)` returns the set (not list) of the plaintiff factors of the case `c`.

`(case-dfactors c)` returns the set (not list) of the defendant factors of the case `c`.

`(case-factors c)` returns the set of all the factors of the case `c`.

`(case-winner c)` returns the party, `'plaintiff` or `'defendant`, in whose favor the case `c` was decided, or `'undecided` if the case has not been decided.

`(case-statements c)` returns a list of all the statements of the factors of the case `c`.

`(make-casebase issue factors cases)` constructs a casebase about the statement `issue` from the given lists of `factors` and `cases`.

`(casebase? x)` tests whether the value `x` is a casebase.

`(casebase-issue cb)` returns the statement which is the issue of the casebase `cb`.

`(casebase-factors cb)` returns the set (not list) of the factors of the casebase `cb`.

`(casebase-cases cb)` returns the set (not list) of the cases of the casebase `cb`.

`(list-cases cb)` returns a list of the cases in the casebase `cb`.

`(get-case cb name)` returns the case of the given `name`, a string, in the casebase `cb`, or the boolean value false, `#f`, if `cb` does not contain a case with this name.

`(partition1 cc pc)` returns intersection of the plaintiff factors of the current case, `cc`, and the plaintiff factors of the precedent case, `pc`.

`(partition2 cc pc)` returns intersection of the defendant factors of the current case, `cc`, and the plaintiff factors of the precedent case, `pc`.

`(partition3 cc pc)` returns the set difference of the plaintiff factors of the current case, `cc`, and the plaintiff factors of the precedent case, `pc`.

`(partition4 cc pc)` returns set difference of the defendant factors of the precedent case, `pc`, and the defendant factors of the current case, `pc`.

`(partition5 cc pc)` returns the set difference of the defendant factors of the current case `cc` and the defendant factors of the precedent case `pc`.

`(partition6 cc pc)` returns the set difference of the plaintiff factors of the precedent case `pc` and the plaintiff factors of the current case `cc`

`(partition7 cb cc pc)` returns the factors of the casebase `cb` which are not factors in either the current case `cc` or the precedent case `pc`.

`(more-on-point pc1 pc2 cc)` first checks whether the precedent case `pc1` is more on point than the precedent case `pc2`, compared to the current case `cc`. If `pc1` is more on point than `pc2` then the set of factors of `pc1` which are not in `pc2` are returned. If `pc1` is not more on point than `pc2`, then the empty set is returned.

`(as-on-point pc1 pc2 cc)` is similar to `more-on-point`, but is satisfied if the factors `pc1` has in common with `cc` is a superset, not necessarily a proper superset, of the factors `pc2` has in common with `cc`. If `pc1` is as on point than `pc2` then the set of factors of `pc1` which are not in `pc2` are returned. If `pc1` is not as on point than `pc2`, then the empty set is returned.

`(common-parent? f1 f2)` returns true, `#t`, if and only if the factors `f1` and `f2` have the same parent factor.

`(decided-for-other-party? pc1 pc2)` tests whether the winner of the precedent case `pc1` is the other party than the one which won the precedent case `pc2`. If `'plaintiff` was the winner of `pc1`, then `pc2` was decided for the other party only if it was won by `'defendant` and vice versa. If either case is undecided, the boolean value false, `#f`, is returned.

`(current-case state cb)` returns the set of factors of the current case. These are the factors of the casebase, `cb`, which have been accepted in the argument context (Section 4.1.3) of the given `state` of the argument search space (Section 4.3).

`(generate-arguments-from-cases cb)` returns a procedure for generating arguments using the case-base `cb`. Like all argument generators, the procedure maps a statement at issue and a state of the argument search space to a stream of reponses, where each is a record consisting of a substitution environment and an argument graph. See Section 4.3 for further information about argument generators and responses.

5.5 Testimonial Evidence

In court proceedings, a basic source of evidence about the facts of the case is witness testimony. Similarly, in administrative procedures of public agencies, such as procedures for determing tax obligations or rights to social benefits, citizens provide information about the facts, typically by completing forms. In both cases, the conclusions one may draw are only presumptively true. Witnesses do not always tell the truth or can be mistaken. Tax declarations are audited for good reasons. Thus the conventions of an agency, court or other organziation for drawing inferences from claims and testimony can be viewed as argumentation schemes.

Walton considers argument from testimonial evidence to be a specialization of the following scheme for argument from position to know [Walton, 2002, p. 46]:

Argument from Position to Know

Major Premise. Source a is in a position to know about things in a certain subject domain s containing proposition p.

Minor Premise. a asserts that p is true (false).

Trustworthiness Exception. a is not trustworthy, reliable or honest.

Conclusion. p is true (false).

One way to implement a computational model of witness testimony is to use a database to store answers to questions. In the `evidence` module of the reference inference engine, conceptually uses a database schema with the following four tables:

1. A *witness* table storing information about persons, such as their name and contact information.
2. A *question* table storing the information needed for asking questions of the form: Is it the case *object* is a/the *predicate* of *subject*? For example: Is it that case that Gertrude is the mother of John? Readers familiar with the Resource Description Framework (RDF) will recognize such statements as *triples* [Manola and Miller, 2004]. Triples can represent both binary and unary relations and thus are sufficient for representing all description logic assertions. Unary relations can be modeled using an 'is a' predicate, for exmaple "Joe is a person". The question table records the type of the object of each predicate (e.g. symbol, number, string, Boolean) and the cardinality of the predicate (one or many), along with a text to be used as a template for asking questions in natural language.
3. A *answer* table stores the answers to questions. For predicates with a cardinality greater than 1, it is also noted whether all values of the object of the predicate have been provided by the witness, or only some. If a witness asserts there are no further values, then this can be used by an argumentation scheme to construct arguments against claims of other values, as will be discussed in more detail below.
4. Finally, a *form* table stores a set of forms, where each form is a sequence of questions. This enables dialogues to be structured more coherently, by asking related questions at the same time. For example, when asking for a person's name, one could also ask for essential contact information.

This database is used to construct arguments for propositions at issue by first matching the proposition at issue to the questions in the questions table. If a question can be found, we then check whether the question for this issue has already been asked. The question has been asked if there is an entry in the answer table for this question. If the witness was not able to provide any answers, the set of values will be empty. If the question has not been asked, or the witness when first answering the question indicated he knew further answers, the question is asked and the answers are both stored in the answers table and used to construct the arguments returned by the argumentation scheme. If the question has been previously asked and the witness had indicated that he had provided all the answers he was able to provide, arguments are constructed from these answers and returned, without modifying the answers table. While this is admittedly a very operational and procedural description of the process of constructing arguments using this scheme, rather than a declarative definition of a mathematical function, it should be remembered that argumentation schemes, in their role as argument generators, are *methods* for constructing arguments. While some of these methods can be defined functionally others are more naturally defined in procedural terms.

If the statement at issue is `(p s o)`, the witness has testified that he provided all the values of the `p` property of `s`, and `o` is not one of those values, than an argument is constructed *con* the statement `(p so)` from this testimony. Such a con argument is reminiscent of *negation as failure* (NAF) in logic programming, but is different in a few ways. First, it is restricted in scope to triples with particular predicates and subjects, whereas NAF applies to all atomic propositions. Second, such arguments are supported by the testimony of a witness who expressly stated that no further values exist, whereas NAF is based on the closed-world assumption, that all relevant facts are known and in the database. In our approach, the closed-world assumption is not made. Finally, con arguments constructed in this way can be rebutted or undercut by other arguments, also by testimony of other witnesses. There is nothing comparable in NAF.

This model of a scheme for argument from witness testimony provides functionality similar to the way rule-based systems ask users for information when there are no rules for deriving some needed fact. But our model is more general as it can handle possibly conflicting testimony about the same issue from more than one witness.

The procedures provided by the `evidence` module are:

`(make-witness name)` returns a witness object with the given `name`, a string. In this verion of the module, the name of the witness is the only information about the witness which can be provided. Other properties of witnesses may be added in future versions.

`(witness-name w)` returns the name, a string, of the witness `w`.

`(make-question predicate type cardinality text)` constructs a question object for the symbol `predicate`. The `type` is a symbol specifing the acceptable type of answers to the question, one of `’symbol`, `’number`, `’boolean`, or `’string`. The `cardinality` is a symbol, either `’one` or `’many`, specifying whether the relation denoted by the predicate has exactly one member, or may have more than one member. In future versions of this module, other cardinalities may be supported. Finally, the `text` is a string providing a natural language version of the question, for use in asking the question to human users. Optionally, the string may include a slot or field for the subject of the `(predicate subject object)` triple, represented using the `" a"` directive of the Intermediate Format Strings standard (SRFI 48). For example, the text for asking for someone’s age could be `"What is  a’s age?"`.

`(question-predicate q)` returns the predicate symbol of the question `q`.

`(question-type q)` returns the type symbol of the question `q`.

`(question-cardinality q)` returns the cardinality symbol, `’one` or `’many` of the question `q`.

`(question-text q)` returns the text string of the question `q`.

`(answer? x)` tests whether an object `x` is an answer to some question. (The constructor function for answers is not exported from the module.)

`(answer-question a)` returns the questions responded to by the answer `a`.

`(answer-values a)` returns a list of the values which have been provided has answers to the question `(answer-question a)`.

`(answer-closed? a)` tests whether the witness has testified that all values answering the question `(answer-question a)` have been provided.

`(make-form questions help)` constructs a form object from a list of questions. The `help` parameter is an symbolic expression, in SXML format[2], representing an HTML file for providing information for helping users to understand the questions.

`(form-questions f)` returns a list of the questions of the form `f`.

`(form-help f)` returns the help text, a string, of the form `f`.

`(make-testimony witness forms)` constructs an object for storing the testimony of the given `witness`. Questions will be asked using the provided list of forms.

`(testimony-witness t)` returns the witness of the testimony `t`.

`(testimony-forms t)` returns the list of the forms used to ask questions during the testimony `t`.

`(testimony-statements t)` returns a list of statements (Section 3.1), i.e. a list of atomic propositions, representing all of answers to all of the questions asked the witness during testimony `t`.

`(get-form t p)` returns the form to be used to ask questions about the predicate `p`, a symbol, during the testimony `t`, or false, `#f`, if there is no form assigned to `p`.

`(get-question t p)` returns the question assigned the predicate symbol `p` in the testimony `t`, or false, `#f`, if no question has been assigend to `p`.

`(get-answer t q s)` returns the answer(s) to the question `q` in the testimony `t` about the subject `s`, a symbol.

`(already-asked? t q s)` checks whether the question `q` has already been asked about the subject `s`, a symbol, in the testimony `t`.

`(generate-arguments-from-testimony t)` returns a procedure for generating arguments from the testimony `t`. Like all argument generators, the procedure maps a statement at issue and a state of the argument search space to a stream of reponses, where each is a record consisting of a substitution environment and an argument graph. See Section 4.3 for further information about argument generators and responses.

[2] `http://ssax.sourceforge.net/`

Here is an example showing how to use this module to build set up a set of forms for asking questions and use them to generate arguments.

```
(define witness1 (make-witness "Respondent"))

(define form1
  (make-form
   ; questions
   (list (make-question 'mother 'symbol 'one "Who is ~a's mother?")
         (make-question 'father 'symbol 'one "Who is ~a's father?"))
   ; help text, in SXML format
   '()))

(define form2
  (make-form
   ; questions
   (list (make-question 'needy 'boolean 'one "Is ~a needy?"))
   ; help text
   '()))

(define form3
  (make-form
   ; questions
   (list (make-question
          'capacity-to-provide-support
          'boolean
          'one
          "Does ~a have the capacity to provide support?"))
   ; help text
   '()))

(define form4
  (make-form
   (list
    (make-question
     (factor-statement f1)
     'boolean
     'one
     "Has much support already been provided?")
    (make-question
     (factor-statement f2)
     'boolean
     'one
     "Would support be provided for only a short period of time?")
    (make-question
     (factor-statement f3)
     'boolean
```

```
      'one
      "Is it true that the respondent never had a close
       relationship with the beneficiary?")
     (make-question
      (factor-statement f4)
      'boolean
      'one
      "Would requiring respondent to provide support cause
      irreparable harm to his family?")
     (make-question
      (factor-statement f5)
      'boolean
      'one
      "Is it true that the respondent has never provided
       care to the beneficiary?")
     )
    ; help text
    '()))

(define testimony (make-testimony witness1 (list form1 form2 form3 form4)))

(define argument-generators
  (list (generate-arguments-from-testimony testimony) ; ask the user first
  ; other argument generators ommitted from the exmaple
 )
```

Chapter 6

LKIF Layer

The Legal Knowledge Interchange Format (LKIF) layer provides an `lkif2` module for importing and exporting rules and arguments in the final version of LKIF in the ESTRELLA project, defined in Deliverable D4.1, is supported by the `lkif2` module.[1]

An *lkif-data* value is a record containing Scheme representatiosn of the data imported from an LKIF file.

`(make-lkif-data sources context rulebase stages)` constructs an *lkif-data* object. `sources` is a list of *source* objects, as defined below; `context` is a *context* (Section 4.1.3) constructed from the axioms of the theory of defined in the LKIF file; *rulebase* is a rulebase (Section 5.2) constructed from the inference rules defined in the theory of the LKIF file; and `stages` is a list of *stages*, as defined below, where each stage combines an argument graph (Section 4.1.4) with a context.

`(lkif-data? x)` tests whether `x` is an lkif-data object.

`(lkif-data-sources lkif-data)` returns the list of sources of `lkif-data`.

`(lkif-data-context lkif-data)` returns the context of `lkif-data`.

`(lkif-data-rulebase lkif-data)` returns the rulebase of `lkif-data`.

`(lkif-data-stages lkif-data)` returns a list of the states of `lkif-data`.

An LKIF `source` is a record associating a Scheme object with a URI.

`(make-source element uri)` constructs a source object associating the `element` with the `uri`, a string which should be a syntactically valid URI.

`(source? object)` tests whether `object` is an LKIF source.

`(source-element src)` returns the element of the LKIF source, `src`.

`(source-uri sr)` returns the URI of the LKIF source, `src`.

An argumentation *stage* is an object encapsulating an argument graph (Section 4.1.4) and an argument context (Section 4.1.3) .

`(make-stage ag c)` constructs a stage from the argument graph `ag` and the context `c`.

`(stage? x)` tests whether `x` is an argument stage.

[1]An `lkif` module, supporting the previous version of LKIF, as defined in Deliverable D1.1, is also available.

`(stage-argument-graph s)` returns the argument graph of the stage `s`.

`(stage-context s)` returns the context of the stage `s`.

Now we are ready to describe the import and export functions for LKIF:

`(lkif-import path)` opens the LKIF file with the name `path`, a string, translates the file into an lkif-data object and, if the translation is successful, returns this object. If the file does not exist or is not a valid LKIF file, the result is unspecified.

`(lkif-export lkif-data port)` translate the information in the `lkif-data` object into a valid LKIF file and writes it to the given output port.

Let's illustrate these import and export functions with the following example. Suppose "family-law.xml" is an LKIF file. It can be imported and then exported to the current output port using the following simple program:

```
#!r6rs

(import (rnrs)
        (carneades lkif2)
        (rnrs io simple))

(define family-support (lkif-import "family-support.xml"))
(lkif-export family-support (standard-output-port))
```

Chapter 7

Acknowledgements

The work reported in this report was conducted as part of the European ESTRELLA project (IST-2004-027655). I would like the acknowledge the helpful feedback provided by members of ESTRELLA project, in particular during the development of the pilot applications which used the reference inference engine. In alphabetical order, I would like to thank Trevor Bench-Capon, Marcello di Bello, Kasper van den Berg, Matthew Bickham, Alexander Boer, Joost Breuker, Giuseppe Contissa, Tom van Engers, Andras Forhecz, Paul Hancock, Szymon Klarman, Rinke Hoekstra, Ulrike Liss, Andras Millinghofer, Monica Palmirani, Samir Sekkat, Reka Vas, Radboud Winkels, and Adam Wyner. I would also like to thank my colleagues at Fraunhofer FOKUS who have worked with me on the development of the inference engine, in particular Stefan Ballnat and Jonas Pattberg.

Bibliography

[Abelson and Sussman, 1985] Abelson, H. and Sussman, G. J. (1985). *Structure and Interpretation of Computer Programs.* MIT Press.

[Aleven, 1997] Aleven, V. (1997). *Teaching Case-Based Argumentation Through a Model and Examples.* Ph.d., University of Pittsburgh.

[Ashley, 1990] Ashley, K. D. (1990). *Modeling Legal Argument: Reasoning with Cases and Hypotheticals.* Artificial Intelligence and Legal Reasoning Series. MIT Press, Bradford Books.

[Baader et al., 2003] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P., editors (2003). *The Description Logic Handbook – Theory, Implementation and Applications.* Cambridge University Press.

[Bench-Capon and Coenen, 1992] Bench-Capon, T. and Coenen, F. P. (1992). Isomorphism and legal knowledge based systems. *Artificial Intelligence and Law*, 1(1):65–86.

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):34–43.

[Carlos Ches et al., 2006] Carlos Ches n., McGinnis, J., Modgil, S., Rahwan, I., Reed, C., Simari, G., South, M., Vreeswijk, G., and Willmott, S. (2006). Towards an argument interchange format. *Knowledge Engineering Review*, 21(4):293–316.

[Dybvig, 2003] Dybvig, R. K. (2003). *The Scheme Programming Language.* MIT Press, third edition edition.

[ESTRELLA Project, 2008a] ESTRELLA Project (2008a). The harness methodology for case assessment. Deliverable 4.6, European Commission.

[ESTRELLA Project, 2008b] ESTRELLA Project (2008b). The legal knowledge interchange format (LKIF). Deliverable 4.3, European Commission.

[for Biomedical Informatics Research, 2007] for Biomedical Informatics Research, S. C. (2007). The protege ontology editor and knowledge acquisition system. `http://protege.stanford.edu/`.

[Gansner and North, 2000] Gansner, E. and North, S. (2000). An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(11):1203–1233.

[Gordon, 2007] Gordon, T. F. (2007). Constructing arguments with a computational model of an argumentation scheme for legal rules. In *Proceedings of the Eleventh International Conference on Artificial Intelligence and Law*, pages 117–121.

[Gordon, 2008a] Gordon, T. F. (2008a). Hybrid reasoning with argumentation schemes. In *Proceedings of the 8th Workshop on Computational Models of Natural Argument (CMNA 08)*, pages 16–25, Patras, Greece. The 18th European Conference on Artificial Intelligence (ECAI 2008).

[Gordon, 2008b] Gordon, T. F. (2008b). LKIF rules: The rule language of the Legal Knowledge Interchange Format. In Casanovas, P., Casellas, N., Rubino, R., and Sartor, G., editors, *Computational Models of the Law*. Springer Verlag.

[Gordon et al., 2007] Gordon, T. F., Prakken, H., and Walton, D. (2007). The Carneades model of argument and burden of proof. *Artificial Intelligence*, 171(10-11):875–896.

[Grosof et al., 2003] Grosof, B. N., Horrocks, I., Volz, R., and Decker, S. (2003). Description logic programs: Combining logic programs with description logics. In *Proceedings of the Twelth International World Wide Web Conference (WWW 2003)*, pages 48–57, Budapest, Hungary. ACM.

[Koutsofios and North, 1993] Koutsofios, E. and North, S. (1993). Drawing graphs with DOT. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey.

[Manola and Miller, 2004] Manola, F. and Miller, E. (2004). RDF primer.

[McGuinness and van Harmelen,] McGuinness, D. L. and van Harmelen, F. OWL Web Ontology Language overview. `http://www.w3.org/TR/owl-features/`.

[Nilsson, 1982] Nilsson, N. J. (1982). *Principles of Artificial Intelligence*. Springer-Verlag, Berlin.

[Nute, 1994] Nute, D. (1994). Defeasible logic. In Gabbay, D., Hogger, C., and Robinson, J., editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 253–395. Clarendon Press, Oxford.

[Patel-Schneider and Swartout, 1994] Patel-Schneider, P. and Swartout, B. (1994). Description logic knowledge representation system specification from the KRSS group of the ARPA knowledge sharing effort. Report, AT&T Bell Laboratories, Murray Hill, New Jersey.

[Pollock, 1992] Pollock, J. L. (1992). How to reason defeasibly. *Artificial Intelligence*, 57:1–42.

[Project,] Project, T. P. S. PLT Scheme. `http://www.plt-scheme.org/`.

[Robinson, 1965] Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the American Association of Computing Machinery*, 12:23–41.

[Russell and Norvig, 2003] Russell, S. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Prentice Hall, second edition.

[Sirin et al., 2006] Sirin, E., Parsia, B., Grau, B., Kalyanpur, A., and Katz, Y. (2006). Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics (submitted)*.

[Walton, 2002] Walton, D. (2002). *Legal argumentation and evidence*. Pennsylvania State University Press, University Park, PA.

[Walton, 2006] Walton, D. (2006). *Fundamentals of Critical Argumentation.* Cambridge University Press, Cambridge, UK.

[Wyner and Bench-Capon, 2007] Wyner, A. and Bench-Capon, T. (2007). Argument schemes for legal case-based reasoning. In *JURIX 2007: The Twentieth Annual Conference on Legal Knowledge and Information Systems.*