ESTRELLA - 027655

European project for
Standardized Transparent Representations
in order to Extend LegaL Accessibility
Specific Targeted Research or Innovation Project

Deliverable N°: 1.6

Specification of APIs for Interacting with and using Legal Knowledge Systems

Workpackage 1 – LKIF Specification

Report Version:	Revised FINAL
Report Preparation Date:	January 2008
Classification:	Public
Contract Start Date:	1 January 2006
Duration:	30 months
Project Coordinator:	Universiteit van Amsterdam (NL)
Lead Contractor Deliverable:	Fraunhofer FOKUS (DE)
Participating Contractors:	Universiteit van Amsterdam (NL), Universita di Bologna (IT), University of Liverpool (UK), Fraunhofer FOKUS (DE), RuleWise b.v. (NL), RuleBurst (EUROPE) Ltd. (UK), knowledgeTools International GmbH (DE), Budapesti Corvinus Egyetem (HU)

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 D1.6 and documents the first version of the Application Programmer Interfaces (APIs) of the ESTRELLA platform for Legal Knowledge Systems. This deliverable is the result of Task T1.7, defined in the Technical Annex on page 27/51 as follows:

> Defining a set of Application Programmer Interfaces (APIs) for interacting with and using LKIF legal knowledge systems. We will develop a standard set of functions for asserting facts and rules, asking queries, asking for explanations, etc. Together with LKIF, this should give application developers all they need to both build and use legal knowledge bases, in a vendor independent way.

The APIs presented in this report are implemented by the "reference implementation of an inference engine" (Deliverable D1.5) for the Legal Knowledge Interchange Format (LKIF, Deliverable D1.1). The API's documented in this report also serve as documentation for the reference inference engine.

Contents

Chapter 1

Introduction

This ESTRELLA report documents the first version of the Application Programmer Interfaces (APIs) for interacting with Legal Knowledge Systems. This report is Deliverable D1.6, which is the result of Task T1.7. This task is defined in the Technical Annex on page 27/51 as follows:

> Defining a set of Application Programmer Interfaces (APIs) for interacting with and using LKIF legal knowledge systems. We will develop a standard set of functions for asserting facts and rules, asking queries, asking for explanations, etc. Together with LKIF, this should give application developers all they need to both build and use legal knowledge bases, in a vendor independent way.

The APIs presented in this report are implemented by the "reference implementation of an inference engine" for the Legal Knowledge Interchange Format (LKIF). This inference engine, more specifically its source code, is Deliverable D1.5. This API report also serves as documentation for the inference engine.

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 Application Programmer Interfaces

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

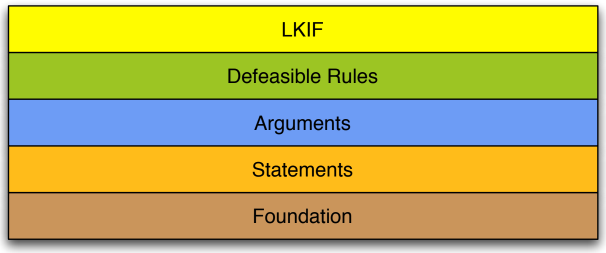

Figure 1.1: Module Layers

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 (*config*), managing possibly infinite sequences of data generated lazily, by need (*stream*), and for heuristically searching problem spaces *search*.

Statement. 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.

Argument. The argument layer provides modules for constructing, evaluating and visualizing argument graphs, also called 'inference graphs' (*argument*,

argument-diagram). It also provides modules (*argument-state*, *argument-search*) for applying argumentation schemes to search heuristically for argument graphs in which some goal statement is acceptable (i.e. provable or presumably true) or not acceptable. A *argument-builtins* module provides an argument generator for common goal statements about arithmetic, strings, lists, dates and so on.

Defeasible Rule. The defeasible rule layer provides a **rule** module for representing defeasible legal rules and generating arguments from sets of rules.

LKIF. The Legal Knowledge Interchange Format (LKIF) layer provides an *lkif* module for importing and exporting rules and arguments in XML, using LKIF's Compact Syntax.

Notice that not every layer of the Legal Knowledge Interchange Format (LKIF), has a corresponding layer in the APIs. The ontology layer of LKIF does not require a custom API, since the OWL Web Ontology Language [6] provides this layer and existing standard APIs, including SPARQL [19] and DIG [2] are available for processing OWL files. APIs for the case (precedent) layer of LKIF are under development and will be included in the final version of the ESTRELLA APIs, in Deliverable D4.4. APIs for the rules and inference graph layers of LKIF are however covered in this document, in Chapter 4 and Chapter 5, respectively.

1.2 The Functional Style of the APIs

The rest of this report presents the APIs in detail, with a chapter for each layer of the system architecture shown in Figure 1.1. The APIs have been specified in a functional programming style and implemented in a functional programming language, PLT Scheme [18]. This implementation constitutes the ESTRELLA Reference Inference Engine (Deliverable D1.5). The reference inference engine is intended to serve as a kind of executable specification. Its purpose is not to be an efficient, production quality implementation, but rather to provide a concise operational definition of the semantics of LKIF and illustrate in a clear way how LKIF may be used to support legal reasoning and argumentation.

Most commercial code these days is implemented in object-oriented programming languages, such as Java or C++, and specified using semi-formal and graphical methods for object-oriented design, such as the Unified Modeling Language [9]. Object-oriented programming is a species of imperative, procedural programming, rather than functional programming, which like logic programming is a kind of declarative programming. In declarative programming, the goal is to specify the meaning of the program, by writing the program as a set of mathematical functions or logical formulas. Since commercial inference engines for LKIF will likely be implemented in languages widely used by industry and such languages, with few exceptions, do not yet support functional programming, we face a dilemma in how best to define and present the ESTRELLA APIs. If we reconstruct the APIs in an object-oriented style, using for example UML, then they would no longer correspond to the running code of the reference inference engine, which is a functional program. On the other hand, if we present the APIs in a functional style, then companies interested in developing an inference engine for LKIF may themselves be confronted with the task of reconstructing the APIs in an object-oriented style. Each company may do this differently, possibly missing an opportunity to facilitate a market of interoperable components for legal knowledge-based systems.

There seems to be no way to resolve this dilemma in a way which meets all of our goals, at least not with a single specification of the APIs. We must make a choice. Because we think it is most important at this stage to have a clear, mathematically well-founded specification of the meaning of LKIF and the ESTRELLA platform and to provide documentation for the reference inference engine sufficient for pilot applications and experiments, we have opted to present the APIs in a functional style, documenting the executable specification which is the reference inference engine.

Since there is no standard metalanguage for documenting the interfaces of functional programs, or more specifically Scheme programs, we will use an ad hoc method, inspired by Standard ML [15] and the TypedScm [14] dialect of Scheme.[1] To illustrate this method, here is an example interface for a module, not part of the ESTRELLA APIs, for handling queues:

```
(interface queue
    (value empty-queue queue)
    (value enqeue (∀ (α) (↦ ((queue-of α) α) (queue-of α))))
    (value empty? (↦ (queue) boolean))
    (value dequeue (∀ (α) (↦ ((queue-of α))
                             (pair-of α (queue-of α)))))
    (value front (∀ (α) (↦ ((queue-of α)) α)))
)
```

[1]The Estrella reference engine, however, is implemented in PLT Scheme, not TypedScm. Our notation for type declarations is a metalanguage.

Chapter 2

Foundational Layer

The foundation layer consists of modules for configuring the system for a particular installation (*config*), managing possibly infinite sequences of data generated lazily, by need (*stream*), and for defining and heuristically searching problem spaces (*state*, *search*).

2.1 Config

The *config* module defines a number of constants which may need to be modified to configure the system for a particular installation. Currently there is only a single parameter, *dot-viewer*, for stating the file system path to an command for viewing dot diagrams.[1] The command must accept the file name of the dot file to be viewed as its first argument, and require no further arguments. If you are using GraphViz on an Mac OS X computer, and you've installed the GraphViz program in the Applications folder, *dot-viewer* should be defined as follows:

```
(define dot-viewer
        "/Applications/Graphviz/Graphviz.app/Contents/MacOS/Graphviz")
```

2.2 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 [1].

The *stream* module extends the stream library of the Scheme Request for Implementation (SRFI), Number 40 [3]. Only the extensions are documented here. See the documentation for SRFI 40 [3] for further information.

[1]For more information about the dot language and Graphviz, an open source program for visualizing graphs represented using the dot language, see http://www.graphviz.org.

```
(interface stream
    ;; SFRI 40 functions included implicitly

    (value stream-append
        (∀ (α)
            (↦ ((stream-of α) (stream-of α))
                (stream-of α))))

    (value stream-interleave
        (∀ (α)
            (↦ ((stream-of α) (stream-of α))
                (stream-of α)))

    (value stream-accumulate
        (∀ (α β)
            (↦ ((↦ (α β) β) β (stream-of α))
                β)))

    (value stream-flatten
        (∀ (α)
            (↦ ((stream-of (stream-of α)))
                (stream-of α))))

    (value stream-flatmap
        (∀ (α β)
            (↦ ((↦ (α) (stream-of β)) (stream-of α))
                (stream-of β))))

    (value stream->list
        (∀ (α)
            (↦ ((stream-of α))
                (list-of α))))
)
```

(*stream-append s1 s2*) is analogous to the *append* function for lists: it returns a stream consisting of all the elements of *s1* followed by all the elements of *s2*. Of course, if *s1* is infinite the elements of *s2* will never be reached.

(*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 (∀ (α β) (↦ (α) (*stream-of* β))), to each element of *s*.

(*stream->list s*) constructs a list from the members of the stream *s*. This is a partial function which is undefined if *s* is infinite.

2.3 Search

The *search* module provides methods for defining and heuristically searching spaces [21] 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.

```
(interface search
    (include stream)
    (type state)

    (value make-root (↦ (state) node))
    (value make-node (↦ (int any node state) node))

    (value node-depth (↦ (node) integer))
    (value node-label (↦ (node) any))
    (value node-parent (↦ (node) node))
    (value node-state (↦ (node) state))

    (value node? (↦ (any) boolean))
    (value root? (↦ (node) boolean))

    (type space (↦ (node) (stream-of node)))

    (value make-problem
        (↦ (node space (↦ state boolean))
            problem))

    (value problem-root (↦ (problem) node))
    (value problem-space (↦ (problem) space))
    (value problem-goal (↦ (problem) (↦ (state) boolean)))

    (value problem? (↦ (any) boolean))

    (value make-resource (↦ (integer) resource))
    (value resource-empty? (↦ (resource) boolean))
    (value resource-amount (↦ (resource) integer))
    (value use (↦ (resource) void))

    (type strategy (↦ (problem) (stream-of node)))
    (type resource-limited-strategy (↦ (resource) strategy))

    (value search (↦ (problem strategy) (stream-of node)))
    (value path (↦ (node) (stream-of node)))

    (value depth-first resource-limited-strategy))
    (value breadth-first resource-limited-strategy))
    (value iterative-deepening
        (↦ (integer integer) resource-limited-strategy))

    (value best-first
        (↦ ((↦ (node node) (set −1 0 1)))
            resource-limited-strategy))
)
```

(*make-root state*) and (*make-node depth label parent state*) construct nodes of the search space. (*make-root state*) handles the base case, for the root of the search tree. Again, nothing in the *search* module depends on the structure of the state. It can 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.

The *node-depth*, *node-label*, *node-parent* and *node-state* functions select the respective parts a *node*.

The (*node? x*) function is a predicate for checking whether some value is a node. The (*root? node*) function is a predicate which is true iff the *node* is the root node of a search space.

A search *space* is a function (*f parent*) of type (↦ (*node*) (*stream-of node*)) which, when applied to some *parent* node produces a stream of the children of *parent*. The depth of each of these children must be (+ (*node-depth parent*) 1). Notice that a node may have infinitely many successors nodes in the space, so long as they are enumerable.

(*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 *problem-root*, *problem-space* and *problem-goal* functions selects the respective parts of a problem. (*problem? x*) checks whether some value is a problem.

A search *strategy* is a function of type (↦ (*problem*) (*stream-of node*)). 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 true if the resource 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 implmented 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.

(*breath-first r*) constructs a limited breath-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. [16, 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-struct state (r1c1 r1c2 r1c3
                      r2c1 r2c2 r2c3
                      r3c1 r3c2 r3c3))
```

There are four moves in the game. A tile may be moved up, down, left or right to fill the empty space. We will label transitions in the space with the name of the move.

```
(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 PLT Scheme's pattern matching library, the goal predicate for this state can be defined as follows:

```
(define (goal? s)
  (match s
    (($ state
        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)
         (($ state 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)))
         (($ state 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)))
         (($ state 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)))
         (($ state 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)))
         (($ state 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)))
         (($ state 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)
         (($ state 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)))
         (($ state 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)))
         (($ state 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)))
         (($ state 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)))
         (($ state 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)))
         (($ state 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)
         (($ state 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)))
         (($ state 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)))
         (($ state 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)))
         (($ state 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)))
         (($ state 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)))
         (($ state 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)
      (($ state 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)))
      (($ state 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)))
      (($ state 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)))
      (($ state 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)))
      (($ state 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)))
      (($ state 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) ())))
   (apply 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 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

```
(interface statement
    (type statement datum)
    (value statement-equal? (↦ (statement statement) boolean))
    (value statement-compare (↦ (statement statement) (set −1 0 1)))
    (value statement-positive? (↦ (statement) boolean))
    (value statement-negative? (↦ (statement) boolean))
    (value statement-complement (↦ (statement) statement))
    (value statement-atom (↦ (statement) statement))
    (value statement->sxml (↦ (statement) datum))
)
```

A *statement* is a Scheme *datum*, i.e. symbolic expression, obeying the following grammar:

```
<statement> ::= <atom> | (not <atom>)
<atom> ::= <symbol> | (<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. In logic, such statements are also known as 'literals'.

Here are some example statements:

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

(*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. It is defined as a synonym for Scheme's *equal?* function:

(**define** *statement-equal? equal?*)

(*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 is not negated. Conversely, (*statement-negative? s1*) is true iff s1 *is* negated.

(*statement-atom s1*) returns the atom of the statement *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.

3.2 Unify

The *unify* module provides a purely functional implementation of Robinson's unification algorithm [20], based on the Dybvig's Scheme implementation in [7, Section 9.10].

```
(interface unify
    (value variable? (↦ (any) boolean))
    (type substitution (↦ (datum) (datum))
    (value identity substitution)
    (value unify* (↦ (datum datum function function boolean) any))
    (value unify (↦ datum datum) (union substitution (set #f)))
)
```

(*varible? x*) is true iff *x* is a symbol beginning with a question mark character. Example variables: *?y*, *?agreement*.

A *substition* environment is represented as a function of type (↦ (*datum*) *datum*). To retrieve the value of a variable, *v*, in a substitution environment, just apply the function representing the substitution environment, *f*, to the variable: (*f v*).

The *identity* function is a substitution environment which maps every datum to itself. It can be viewed as the empty substitution environment.

(*unify∗ u v s ks kf occurs-check*) is the general unification algorithm. It tries to unify *u* and *v* in the substitution environment *s*. If successful, the function *ks* is applied to the resulting substitution environment. If not successful, the function *kf* is applied to error message (string). The *occurs-check* flag determines whether unification will fail if a variable is contained in the term with which it is to be unified. In the semantics of first-order logic, unification should fail in this case. However, since the check is expensive and some programs can assure, by construction, that the variable will not occur in the term, this 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. Here is an example:

```
> (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 argument graphs, also called 'inference graphs' (*argument*, *argument-map*). It also provides modules (*argument-state*, *argument-search*) for applying argumentation schemes to search heuristically for argument graphs in which some goal statement is acceptable (i.e. provable or presumably true) or not acceptable. A *argument-builtins* module provides an argument generator for common goal statements about arithmetic, strings, lists, dates and so on.

4.1 Argument

The *argument* module implements the Carneades model of argument graphs [12]. Argument graphs, also called *inference graphs* [17] are directed, acyclic, bipartite graphs, consisting of statement nodes and argument nodes. 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 both premises and conclusions is a literal, i.e. either a positive atomic statement (atom) or a negated atomic statement. However, the statements of statement nodes in argument graphs are all positive literals. There are two kinds of arguments, pro arguments and con arguments. An argument pro a negative statement, (*not s*), is equivalent to an argument con *s*. A negated premise is modeled in argument graphs 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 transformations of negated premises and conclusions enable statement nodes in argument graphs to be restricted to positive literals without loss of meaning.

Because the argument interface is rather long, it will be presented incrementally in the following sections. Here is an outline of the whole interface, referencing the sections included.

(**interface** *argument*
  (**include** *statement*)
  ; Section 4.1.1 Premise
  ; Section 4.1.2 Argument
  ; Section 4.1.3 Argument Context
  ; Section 4.1.4 Argument Graph
  ; Section 4.1.5 Argument Evaluation

)

4.1.1 Premise

(**value** *premise?* (↦ (*any*) *boolean*))
(**value** *make-ordinary-premise*
  (↦ (*statement boolean* (*union symbol* (*set* #f))) *premise*))
(**value** *ordinary-premise?* (↦ (*any*) *boolean*)
(**value** *make-exception*
  (↦ (*statement boolean* (*union symbol* (*set* #f))) *premise*))
(**value** *exception?* (↦ (*any*) *boolean*))
(**value** *make-assumption*
  (↦ (*statement boolean* (*union symbol* (*set* #f))) *premise*))
(**value** *assumption?* (↦ (*any*) *boolean*))
(**value** *premise-atom* (↦ (*premise*) *statement*))
(**value** *premise-polarity* (↦ (*premise*) *boolean*)
(**value** *premise-role* (↦ (*premise*) *symbol*))
(**value** *premise-statement* (↦ (*premise*) *statement*))
(**value** *pr* (↦ (*statement* . *boolean*) *premise*))
(**value** *am* (↦ (*statement* . *boolean*) *premise*))
(**value** *ex* (↦ (*statement* . *boolean*) *premise*))
(**value** *premise=?* (↦ (*premise premise*) *boolean*))
(**value** *negative-premise?* (↦ (*premise*) *boolean*))
(**value** *positive-premise?* (↦ (*premise*) *boolean*))

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*), (*make-exception atom polarity role*) and (*make-assumption atom polarity role*). In these functions, *atom* is an atomic statement and *polarity* is boolean value stating whether the premise is positive or negative. If *polarity* is true, i.e. #t, then the premise is positive, otherwise it is negative. The *role* h symbol can be used to name the role of the premise in the argument, such as 'major or 'minor. The role is optional; To provide no role, use #f.

(*premise? x*), (*ordinary-premise? x*), (*exception? x*) and (*assumption? x*) are predicates for testing whether *x* is a value of the respective type.

(*premise-atom x*), (*premise-polarity x*), (*premise-role x*) are selector functions for accessing the parts of a premise.

(*premise-statement x*) constructs a positive or negative statment, i.e. a literal, from the premise, depending on its polarity. If the premise is negative, i.e. its polarity is #f, 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))

(*pr s p*), (*am s p*), and (*ex s p*) functions are convenient abbreviations, when no role is needed, for (*make-ordinary-premise s p* #f), (*make-assumption s p* #f) and (*make-exception s p* #f). The polarity can be omitted for positive premises, as in e.g. (*pr s*). That is, premises constructed using these functions are positive by default.

(*premise=? p1 p2*) is #t 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.

(*negative-premise? p1*) is #t if (*premise-parity p1*) is #f. Conversely, (*positive-premise? p2*) is #t if (*premise-parity p2*) is #t.

4.1.2 Argument

```
(value make-argument
    (↦ (symbol
         (set 'pro 'con)
         atom
         (list-of premise)
         (union symbol (one-of #f)))
        argument))

(value argument? (↦ (any) boolean))
(value argument-id (↦ (argument) symbol))
(value argument-direction (↦ (argument) (set 'pro 'con)))
(value argument-conclusion (↦ (argument) statement))
(value argument-premises (↦ (argument) (list-of premise)))
(value argument-scheme (↦ (argument) symbol))
(value pro (↦ (symbol statement (list-of premise)) argument)
(value con (↦ (symbol statement (list-of premise)) argument))

(syntax define-argument)
(value argument->datum (↦ (argument) datum))
(value argument->sxml (↦ (argument) datum))
(value add-premise (↦ (premise argument) 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 *scheme* is either a symbol naming the argumentation scheme applied to produce this argument, or #f if you do not want to provide this information.

(*pro id conclusion premises*) and (*con id conclusion premises*) are convenient, simpler constructor functions for pro and con arguments, respectively, when the name of the argumentation scheme is not needed.

(*argument? x*) is predicate for testing whether some value is an argument. *argument-id*, *argument-direction*, *argument-conclusion*, *argument-premises* and *argument-scheme* are selector functions for assessing the respective parts of an argument.

(*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)))
#f))
```

(*argument->datum a1*) and (*argument->sxml a1*) represents the argument *a1* as a symbolic expression, in formats suitable for displaying the argument for humans.[1] 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* having *p* as an additional premise. Enthymemes are arguments with implicit, unstated premises [23, p. 178]. *add-premise* provides a way to incrementally add premises to an argument as they become revealed.

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 prior relation on arguments.

Statement Status

```
(type status (set 'stated 'questioned 'accepted 'rejected))
(value status? (↦ (symbol) boolean))
```

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

Proof Standard

```
(type proof-standard
    (set 'se    ; scintilla of the evidence
         'ba    ; best argument
         'dv)) ; dialectical validatity

(value proof-standard? (↦ (symbol) boolean))
(value complementary-proof-standard
    (↦ (proof-standard) proof-standard))
```

[1] Well, at least programmers.

(*proof-standard? x*) is a predicate for checking whether a symbol names one of the declared proof-standards: 'se (scintilla of the evidence), 'ba (best argument), or 'dv (dialectical validity).

(*complementary-proof-standard ps*) is the proof standard which results by reversing the roles of pro and con arguments in the proof standard *ps*. For example, 'ba denotes the best argument standard, which is satisified iff a statement is supported by at least one defensible pro argument and no con argument is defensible. Thus (*complementary-proof-standard* 'ba) denotes the proof standard which is satisfied iff a statement is support by at least one defensible con argument and no pro argument is defensible.

Argument Context

(**value** *default-context context*)
(**value** *context?* (↦ (*any*) *boolean*))

(**value** *status* (↦ (*context statement*) *status*))
(**value** *proof-standard* (↦ (*context statement*) *proof-standard*))
(**value** *prior* (↦ (*context argument argument*) *boolean*))

(**value** *question* (↦ (*context statement* ...) *context*))
(**value** *accept* (↦ (*context statement* ...) *context*))
(**value** *reject* (↦ (*context statement* ...) *context*))
(**value** *assign-standard* (↦ (*context proof-standard statement* ...) *context*))

(**value** *stated?* (↦ (*context statement*) *boolean*))
(**value** *questioned?* (↦ (*context statement*) *boolean*))
(**value** *accepted?* (↦ (*context statement*) *boolean*))
(**value** *rejected?* (↦ (*context statement*) *boolean*))
(**value** *decided?* (↦ (*context statement*) *boolean*))
(**value** *issue?* (↦ (*context statement*) *boolean*))

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.

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

(*status c s*) returns the status of statement *s* in context *c*. Similarly, (*proof-standard c s*) returns the proof standard of *s* in *c*.

As discussed previously, (*prior c a1 a2*) tests whether argument *a1* has priority over argument *a2* in context *c*.

question, *accept*, *reject* and *assign-standard* modify a context, non-destructively. (*question c s*) returns a context in which the status of *s* in *c* has been changed to 'questioned. Similarly, (*accept c s*) and (*reject c s*) return contexts in which the status of *c* in *c* has been changed to 'accepted and 'rejected, respectively. (*assign-standard c ps s*) returns a context in which the proof standard assigned to *s* in *c* is changed to *ps*. All of these functions take any number 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*. Similarly, (*questioned? c s*), (*accepted? c s*) and (*rejected? c s*) are true iff *s* is 'questioned, 'accepted or 'rejected, respectively, in *c*. (*decided? c s*) is true just if s is 'accepted or 'rejected in c. Finally, (*issue? c s*) is true just when s is not decided in c.

4.1.4 Argument Graph

Argument graphs, also called *inference graphs* [17] are directed, acyclic, bipartite graphs, consisting of statement nodes and argument nodes. The functions documented in this subsection provides methods for constructing argument graphs and retrieving parts of argument graphs.

(**value** *empty-argument-graph argument-graph*)
(**value** *argument-graph?* (↦ (*any*) *boolean*))
(**value** *assert*∗ (↦ (*argument-graph argument boolean*) *argument-graph*))
(**value** *assert* (↦ (*argument-graph argument* ...) *argument-graph*))
(**value** *update* (↦ (*argument-graph argument* ...) *argument-graph*))
(**value** *arguments* (↦ (*argument-graph*) (*list-of argument*)))
(**value** *statements* (↦ (*argument-graph*) (*list-of statement*))
(**value** *questions* (↦ (*argument-graph context*) (*list-of statement*)))
(**value** *facts* (↦ (*argument-graph context*) (*list-of statement*)))
(**value** *accepted-statements*
    (↦ (*argument-graph context*) (*list-of statement*)))
(**value** *rejected-statements*
    (↦ (*argument-graph context*) (*list-of statement*)))
(**value** *stated-statements*
    (↦ (*argument-graph context*) (*list-of statement*)))
(**value** *issues*
    (↦ (*argument-graph context*) (*list-of statement*)))
(**value** *relevant?* (↦ (*argument-graph statement statement*) *boolean*))
(**value** *statements* (↦ (*argument-graph*) (*list-of statement*)))
(**value** *relevant-statements*
    (↦ (*argument-graph statement*) (*list-of statement*)))
(**value** *schemes-applied*
    (↦ (*argument-graph statement*) (*list-of symbol*)))

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, i.e. #t, 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 ag*) and (*statements ag*) return a list of the arguments and statements, respectively, in an argument graph. (*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 ag c*), (*rejected-statements ag c*), and (*stated-statements ag c*), return a list of the accepted, rejected and stated statements, respectively, of the argument graph *ag* in 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 in *arg*. 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.

(**value** *satisfies?*
  (↦ (*argument-graph context statement proof-standard*)
    boolean))

(**value** *acceptable?*
  (↦ (*argument-graph context statement*)
    boolean))

(**value** *holds?*
  (↦ (*argument-graph context premise*)
    boolean))

(**value** *defensible?*
  (↦ (*argument-graph context argument*)
    boolean))

(*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 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 [12] for further details.

Finally, (*defensible? ag c arg*) is true just the argument *arg* of argument graph *ag* is defensible in context *c*. This is the case only if all of the premises of *arg* hold in *c*.

4.2 Argument Diagram

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

```
(interface argument-diagram
   (include unify argument-graph)

   (value diagram*
      (↦ (argument-graph context substitution boolean port)
         void))

   (value view*
      (↦ (argument-graph context substitution boolean)
         void))

   (value diagram (↦ (argument-graph context) void))
   (value view (↦ (argument-graph context) void))
)
```

(*diagram∗ ag c subs verbose p*) generates a diagram for argument graph *ag* in the context *c*, replacing any variables with their values in the substitution environment *subs*, and writes it out to the port *p*. The *verbose* flag provides basic control over the amount of detail displayed in the diagram. If *verbose* is false, some details are suppressed in order to produce more compact diagrams. The diagrams are represented using the DOT language [10, 13] of the GraphViz [8] system.

(*view∗ ag c subs verbose*) uses *diagram∗* to produce a diagram of the argument graph *ag*, but opens and displays the resulting DOT file using the GraphViz application, rather than writing the diagram to a file. The location of the GraphViz application, or some other application for displaying DOT files, must be provided by customizing the *Config* module. See Section 2.1 for further information about the *Config* module.

The (*diagram ag c*) and (*view ag c*) provide simplified interfaces to the *diagram∗* and *view∗* functions, using the *identity* substitution environment and producing verbose diagrams. *diagram* writes the diagram to the *current-output-port*.

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>

   <s id="contract" src="There is a contract."/>
   <s id="writing" src="The agreement is in writing."/>
   <s id="real-estate" src="The agreement is for the sale of real estate."/>
   <s id="agreement" src="There is an agreement."/>
   <s id="minor" src="One of the parties is a minor."/>
```

```
    <s id="email" src="The agreement was by email."/>
    <s id="deed" src="There is a deed."/>

    <argument-graph>
        <argument id="a1" direction="pro" >
            <premise statement="agreement"/>
            <exception statement="minor"/>
            <conclusion statement="contract"/>
        </argument>

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

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

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

        <argument id="a5" direction="pro" >
            <assumption statement="deed"/>
            <conclusion statement="real-estate"/>
        </argument>
    </argument-graph>
</lkif>
```

Suppose this XML is stored in a file named ”contract.xml”. The following PLT 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:

```
(define imports (lkif:import "contract.xml"))
(define ag1 (car (list:filter argument-graph? imports)))
(define c1 (accept default-context 'deed))
(define c2 (reject c1 'email))
(view ag1 c2)
```

This code causes the following diagram to be displayed, using GraphViz:

4.3 Argument State

The *argument-state* module provides a data structure for representing the states in a search space of argument graphs. These states will be used to define search problems in which the goal is to find states having an argument graph in which some goal statement is acceptable or not, depending on whether the viewpoint is pro or con the goal.

```
(interface argument-state
```

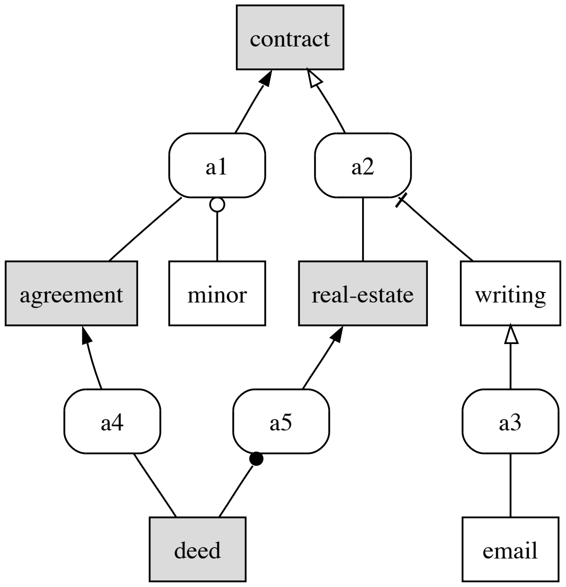

Figure 4.1: Example Argument Diagram

```
(include unify argument)

 (type viewpoint (set 'pro 'con))
 (value opposing-viewpoint (↦ (viewpoint) viewpoint))

 (value make-state
     (↦ (statement           ; topic
          viewpoint           ; pro or con
          (list-of statement)  ; pro goals
          (list-of statement)  ; con goals
          context
          substitutions
          argument-graph)
         state)

 (value state? (↦ (any) boolean))
 (value state-topic (↦ (state) statement))
 (value state-viewpont (↦ (state) viewpoint))
 (value state-pro-goals (↦ (state) (list-of statement)))
 (value state-con-goals (↦ (state) (list-of statement)))
 (value state-context (↦ (state) context))
 (value state-substitutions (↦ (state) substitution))
 (value state-arguments (↦ (state) argument-graph))

 (value initial-state (↦ (statement context) state))
)
```

A *viewpoint* is either 'pro or scm'con some goal statement. The opposing viewpoint of 'pro is 'con and vice versa: (*opposing-viewpoint* 'pro) equals 'con and (*opposing-viewpoint* 'con) equals 'pro.

(*make-state topic viewpoint pro-goals con-goals context substitutions arguments*) constructs a state. Here, *topic* is the main statement at issue. 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. *viewpoint* is the viewpoint, 'pro or 'con used to construct this state. 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 focussing 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 state's argument graph.

(*state? x*) is a predicate for checking whether some value is an argument state. (*state-topic s*), (*state-viewpoint s*), (*state-pro-goals s*), (*state-con-goals s*), (*state-context s*), (*state-substitutions s*), and (*state-arguments s*) are selector functions for accessing the various fields of an argument state.

(*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*.

4.4 Argument Search

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

```
(interface argument-search
    (include argument-state search stream)

    (type argument-generator (↦ (state) (stream-of state)))

    (value find-arguments
        (↦ (strategy resource state (list-of generator))
            (stream-of state)))

    (value find-best-arguments
        (↦ (strategy resource max state (list-of argument-generator))
            (stream-of state)))
)
```

An *argument-generator* is a function of type (↦ (*state*) (*stream-of state*)), where *state* is an argument state, as described in Section 4.3. 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.

Generators are used to construct a problem space. Given a list of argument generators, a list of transitions of type (↦ (*node*) (*stream-of node*) can be constructed.

Given an initial state, *s*, the root node of the problem space is (*make-root s*), where *make-root* is function provided by the Search module of Section 2.3. The problem space is induced by applying the transitions to this root node and, recursively, to the nodes reachable by these transitions.

Recall also from Section 2.3 that a search *strategy* is a function of type (↦ (*problem*) (*stream-of node*)). Strategies provided by the Search module of Section 2.3 include resource limited 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 the 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, again 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.

Finally, (*find-best-arguments strategy resource max 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, where the viewpoing 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.

4.5 Argument Builtins

The *argument-builtins* module will provide an argument generator for common goal statements about arithmetic, strings, lists, dates and so on.

```
(interface argument-builtins
    (include argument-state stream)
    (type generator (↦ (state) (stream-of state))
    (value builtins generator)
)
```

This module only provides one value, the *builtins* argument generator.

Currently, the *argument-builtins* doesn't provide much, just definitions of two priority rules, using the LKIF rules:

```
(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 presented in Chapter 5 to generate arguments from these rules. The set of builtin rules will be extended in future versions of these APIs.

Not all builtins will be implemented using rules. Many, such as those for string processing and math, will be implemented using handlers written directly in the implementation language, PLT Scheme. The set of builtins to be provided by the ESTRELLA platform is currently under discussion in the project.

Chapter 5

Defeasible Rule Layer

The defeasible rule layer provides a **rule** module for representing defeasible legal rules and generating arguments from sets of rules.

```
(interface rule
    (include statement argument-search)

    (type condition datum)

    (value make-rule
        (↦ (symbol boolean (list-of statement) (list-of condition))
            rule))

    (value rule? (↦ (any) boolean))
    (value rule-strict (↦ (rule) boolean))
    (value rule-id (↦ (rule) symbol))
    (value rule-head (↦ (rule) (list-of statement)))
    (value rule-body (↦ (rule) (list-of condition)))

    (syntax rule)
    (syntax rule*)

    (value empty-rulebase rulebase)
    (value rulebase (↦ (list-of rule) rulebase))
    (value rulebase? (↦ (any) boolean)
    (value add-rules (↦ (rulebase (list-of any)) rulebase))

    (value rulebase-rules (↦ (rulebase) (list-of rule)))

    (type rule-critical-questions (set 'excluded 'priority 'valid))
    (value generate-arguments-from-rules
        (↦ (rulebase (list-of rule-critical-questions))
            argument-generator))

    (value rule->datum (↦ (rule) datum)
    (value rulebase->datum (↦ (rulebase) datum)
)
```

A *condition* is a symbolic expression of the following form:

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

(*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.

The **rule** macro provides convenient forms for defining rules, using the following grammar:

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

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

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

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 **rule∗** macro is just like the **rule** macro, except that the rules defined are strict.

A *rulebase* models a set of rules. *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. (*rulebase r1 ... rn*) constructs a rulebase containing the rules *r1* to *rn*.

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.

A generator for arguments from rules is produced with (*generate-arguments-from-rules rb questions*), where *rb* is a rulebase and *questions* is a list of the 'critical questions' [23] 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*) and (*rulebase->datum rb*) translates rules and rulebases, respectively, into symbolic expressions, using the same syntax for rules implemented by the **rule** and **rule∗** macros described previously. For example, using the *family* rulebase:

```
> (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, we show how to use a rulebase to answer queries by searching a problem space:

```
> (define s1 (initial-state '(ancestor Caroline Gloria)
                         default-context))
> (define g1 (generate-arguments-from-rules family null))
> (define r (make-resource 50))
> (define results (find-arguments depth-first r s1 (list g1)))
> (define s2 (stream-car results))
> (view* (state-arguments s2)
         (state-context s2)
         (state-substitutions s2) #t)
```

The final command above, (*view** . . .), causes the diagram shown in Figure 5.1 to be displayed, using GraphView.

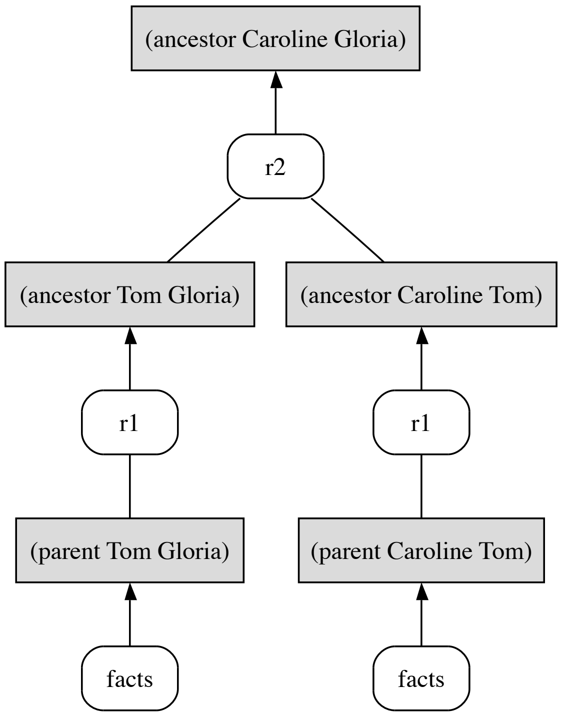

Figure 5.1: The First Argument Graph Found

Chapter 6

LKIF Layer

The Legal Knowledge Interchange Format (LKIF) layer provides an *lkif* module for importing and exporting rules and arguments in XML, using LKIF's Compact Syntax.

The initial version of LKIF is defined in Deliverable D1.1 [4]. LKIF's Compact Syntax, used here, is more recent and will be defined as part of the 'Refined Specification of the Legal Knowledge Interchange Format (LKIF)' in Deliverable 4.1.

```
(interface lkif
    (include statement argument rule)
    (type lkif-value (union statement argument-graph rule))
    (value import (↦ (path) (list-of lkif-value)))
    (value export (↦ ((list-of lkif-value) . path) void))
)
```

By an LKIF value, we mean a *statement*, *argument-graph* or **rule**. (Other LKIF values, e.g. for legal cases, are planned for the next version of this report, to be delivered as D4.4.)

(*import path*) constructs a list of LKIF values by parsing and translating the XML file of the given *path* name. The XML file must be a well-formed XML document and valid according to the grammar rules of LKIF's Commpact Syntax. The grammar is available in both Relax NG [5] and as an XML Schema Definition [22].

(*export list path*) translates the LKIF objects in *list* into XML, using LKIF's Compact Syntax and writes the XML to the file named by the *path*. The *path* can be omitted. If no *path* is provided, the XML is written to Scheme's (*current-output-port*), i.e. the current default output file.

Let's illustrate these import and export functions with the following example family law rulebase, in LKIF's Compact Syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<lkif>

    <rule id="facts" strict="yes" >
        <s>parent Dustin Tom</s>
        <s>mother Tom Gloria</s>
        <s>needy Gloria</s>
        <not><s>capacity-to-provide-support Dustin</s></not>
    </rule>

    <rule id="r1" strict="yes" >
    <body><s>ancestor ?x ?y</s></body>
    <head><s>relative ?x ?y</s></head>
</rule>

<rule id="r4" strict="yes" >
    <body><s>parent ?x ?y</s></body>
    <head><s>ancestor ?x ?y</s></head>
</rule>

<rule id="r2" strict="yes" >
    <body><s>ancestor ?x ?z</s>
          <s>ancestor ?z ?y</s></body>
    <head><s>ancestor ?x ?y</s></head>
</rule>

<rule id="r3" strict="yes" >
    <body><s>ancestor ?x ?y</s></body>
    <head><s>descendent ?y ?x</s></head>
</rule>

<rule id="r5" strict="yes" >
     <body><s>mother ?x ?y</s></body>
     <head><s>parent ?x ?y</s></head>
</rule>

<rule id="r6" strict="yes" >
    <body><s>father ?x ?y</s></body>
    <head><s>parent ?x ?y</s></head>
</rule>

<rule id="r7" strict="yes" >
    <body><s>sibling ?x ?y</s></body>
    <head><s>relative ?x ?y</s></head>
</rule>

<rule id="r8" strict="yes" >
    <body><s>brother ?x ?y</s></body>
    <head><s>sibling ?x ?y</s></head>
</rule>

<rule id="r9" strict="yes" >
    <body><s>sister ?x ?y</s></body>
    <head><s>sibling ?x ?y</s></head>
</rule>

<rule id="r10" strict="yes" >
    <body><s>parent ?x ?z</s>
          <s>parent ?z ?y</s></body>
    <head><s>grandparent ?x ?y</s></head>
</rule>

<rule id="s1601-BGB" >
    <body><s>direct-lineage ?x ?y</s></body>
    <head><s>obligated-to-support ?x ?y</s></head>
    </rule>

    <rule id="s1589a-BGB" >
        <body><s>ancestor ?x ?y</s></body>
        <head><s>direct-lineage ?x ?y</s></head>
    </rule>

    <rule id="s1589b-BGB" >
        <body><s>descendent ?x ?y</s></body>
        <head><s>direct-lineage ?x ?y</s></head>
    </rule>

    <rule id="s1741-BGB" >
        <body><s>adopted-by ?x ?y</s></body>
        <head><s>ancestor ?x ?y</s></head>
    </rule>

    <rule id="s1590-BGB" >
        <body><s>relative-of-spouse ?x ?y</s></body>
        <head><not><s>obligated-to-support ?x ?y</s></not></head>
    </rule>

    <rule id="s1602a-BGB" >
        <body><not><s>needy ?x</s></not></body>
        <head><not><s>obligated-to-support ?y ?x</s></not></head>
    </rule>

    <rule id="s1602b-BGB" >
        <body><not><s>able-to-support-himself ?x</s></not></body>
        <head><s>needy ?x</s></head>
    </rule>

    <rule id="s1603-BGB" >
        <body><not><s>capacity-to-provide-support ?x</s></not></body>
        <head><not><s>obligated-to-support ?x ?y</s></not></head>
    </rule>

    <rule id="s1611a-BGB" >
        <body><s>neediness-caused-by-own-immoral-behavior ?x</s></body>
        <head><s>excluded s1601-BGB <s>obligated-to-support ?y ?x</s></s></head>
    </rule>

    <rule id="s91-BSHG" >
        <body><s>undue-hardship ?x <s>obligated-to-support ?x ?y</s></s></body>
        <head><s>excluded s1601-BGB <s>obligated-to-support ?x ?y</s></s></head>
    </rule>
</lkif>
```

Let's suppose this XML is stored in a file named "family-support.xml". This file can be imported to create a rulebase as follows:

```
(define family-support
   (add-rules empty-rulebase
          (import "family-support.xml")))
```

Notice that the rulebase included some facts about a particular case, near the top, to simplify the example. Using the *family-support* rulebase, we can pose a query, about whether or Dustin is obligated to support his grandmother, Gloria, as follows:

```
> (define s1 (initial-state '(obligated-to-support Dustin Gloria)
                              default-context))
> (define g1 (generate-arguments-from-rules family-support null))
> (define r (make-resource 50))
> (define results (find-arguments depth-first r s1 (list g1 builtins)))
> (define s2 (stream-car results))
> (view* (state-arguments s2)
         (state-context s2)
         (state-substitutions s2) #t)
```

The (*view∗* ...) command above displays the diagram shown in Figure 6.1, using GraphView.

Conversely, the *family-support* rulebase can be exported to XML as follows. Here we just print the XML to the current output port, by omitting the pathname.

```
> (display (export (rulebase-rules family-support)))

<lkif>
  <rule id="facts" >
    <s>parent Dustin Tom </s>
    <s>mother Tom Gloria </s>
    <s>needy Gloria </s>
    <s>not <s>capacity-to-provide-support Dustin </s></s>
  </rule>
  <rule id="r1" >
    <head>
      <s>relative ?x ?y </s>
    </head>
    <body>
      <s>ancestor ?x ?y </s>
    </body>
  </rule>
...
  <rule id="s91-BSHG" >
    <head>
      <s>excluded s1601-BGB <s>obligated-to-support ?x ?y </s></s>
    </head>
    <body>
      <s>undue-hardship ?x <s>obligated-to-support ?x ?y </s></s>
    </body>
  </rule>
</lkif>
```

The output has been truncated to save space. Notice that the rules of the rulesbase must be exported, not the rulebase itself. The *export* function can translate rules, argument-graphs and statements into XML, but not rulebases directly.

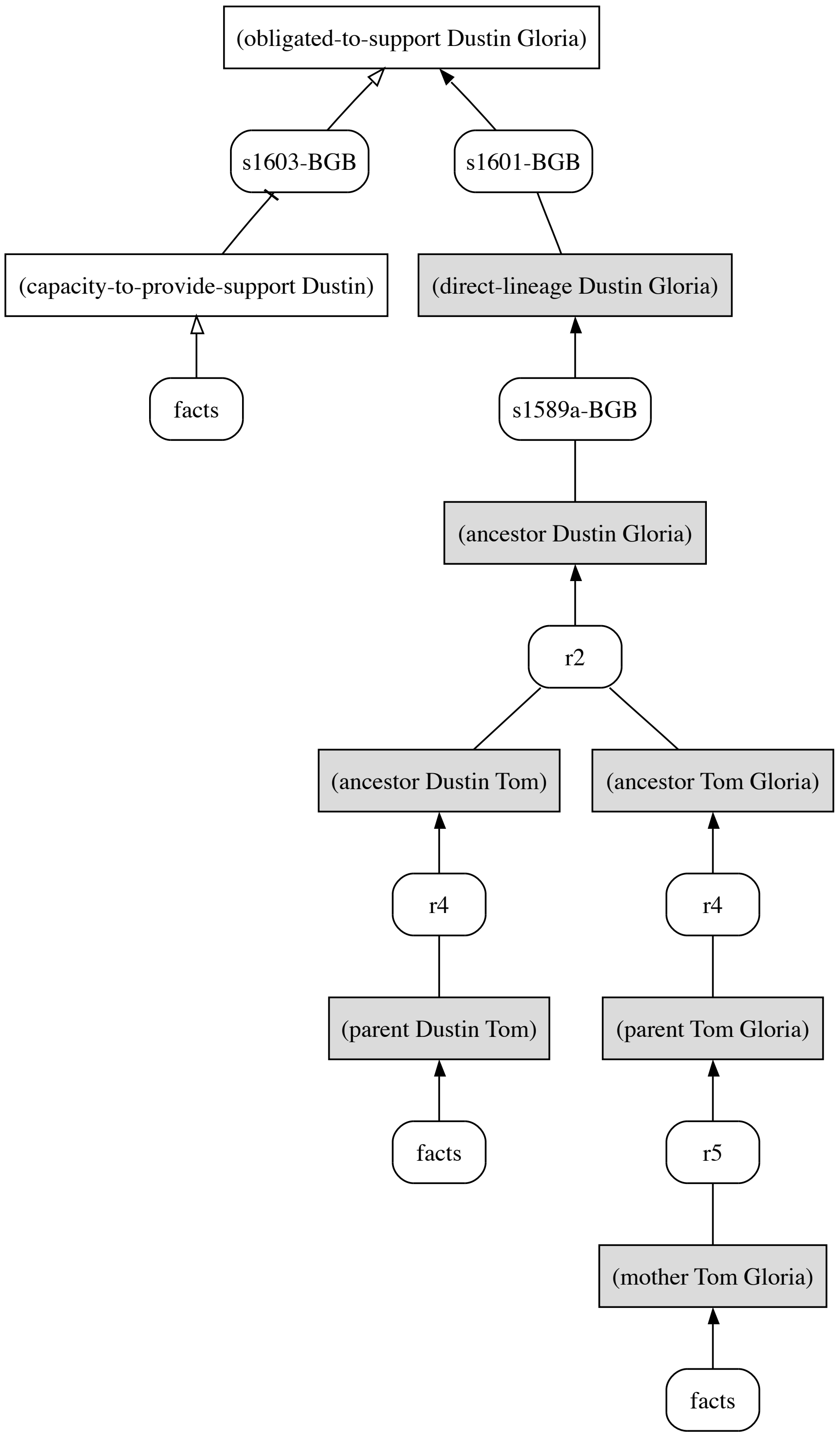

Figure 6.1: Argument Graph About a Support Obligation Issue

Bibliography

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[2] S. Bechhofer. The DIG Description Logic interface: DIG 1.1. Technical report, D1 Implementation Group, University of Manchester, 2003.

[3] Philip L. Bewig. SRFI 40: A library for streams. http://srfi.schemers.org/srfi-40/srfi-40.html.

[4] Alexander Boer, Thomas F. Gordon, Kasper von den Berg, András Förhécz, and Réka Vas. Specification of the Legal Knowledge Interchange Format — Deliverable 1.1. Technical report, ESTRELLA Project, 2007.

[5] James Clark. Relax ng. http://relaxng.org/, September 2003.

[6] Stanford University) Deborah L. McGuiness Deborah L. McGuinness (Knowledge Systems Laboratory and Frank van Harmelen. OWL web ontology language overview. http://www.w3.org/TR/owl-features/.

[7] R. Kent Dybvig. *The Scheme Programming Language*. MIT Press, third edition edition, 2003.

[8] John Ellson, Emden Gansner, Lefteris Koutsofios, Stepen C. North, and Gordon Woodhull. Graphviz — open source graph drawing tools. In *Proceedings of the 9th International Symposium on Graph Drawing (GD 2001)*, pages 483–484, Vienna, September 2001.

[9] Martin Fowler and Kendall Scott. *UML Distilled — A Brief Guide to the Standard Object Modeling Language*. Addison Wesley Longman, Inc., 2nd edition, 2000.

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

[11] Thomas F. Gordon. *The Pleadings Game; An Artificial Intelligence Model of Procedural Justice*. Kluwer, Dordrecht, 1995. Book version of 1993 Ph.D. Thesis; University of Darmstadt.

[12] Thomas F. Gordon, Henry Prakken, and Douglas Walton. The Carneades model of argument and burden of proof. *Artificial Intelligence*, 171(10-11):875–896, 2007.

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

[14] Gary T. Leavens, Curtis Clifton, and Brian Dorn. A type notation for scheme. Technical Report TR 05-18a, Department of Computer Science, Iowa State University, January 2006.

[15] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML, (Revised)*. MIT Press, 1997.

[16] Nils J. Nilsson. *Principles of Artificial Intelligence*. Springer-Verlag, Berlin, 1982.

[17] John L. Pollock. How to reason defeasibly. *Artificial Intelligence*, 57:1–42, 1992.

[18] The PLT Scheme Project. PLT Scheme. `http://www.plt-scheme.org/`.

[19] Eric Prudhommeaux and Andy Seaborne. SPARQL query language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`.

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

[21] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Prentice Hall, second edition, 2003.

[22] World Wide Web Consortium (W3C). Xml schema. `http://www.w3.org/XML/Schema`, May 2001.

[23] Douglas Walton. *Fundamentals of Critical Argumentation*. Cambridge University Press, Cambridge, UK, 2006.