Software Abstractions for Description Logic Systems

Michael Wessel

Institute for Software Systems
Hamburg University of Technology (TUHH)
Germany

Contents

- Motivation
- Description Logics
  - Syntax, Semantics, Satisfiability
- Software Abstractions
  - Substrate Data Model
  - MiDeLoRa Space and Provers
- Tableau Calculi
  - Mathematical Perspective
  - Software Perspective
- Why Lisp?
- Conclusion

Motivation for MIDELORA

- Statement: description logic (DL) systems are very complicated software artefacts
  - Intellectual complexity (tableau calculi, optimizations)
  - Software complexity
- Thesis: problem-specific software abstractions can reduce complexity and enhance comprehensibility → maintainabilty, . . .
- Flexibility / genericity w.r.t. various “dimensions” in the knowledge-representation design space
  - Support different DLs
  - Support different information representation media

⇒ Toolkit/framework with orthogonal building blocks

Research Questions & Answers

? What are reasonable building blocks for DL systems?

⇒ Standard DL notions like TBox, ABox (too coarse)

⇒ Idea: turn mathematical notions into software abstractions (e.g., tableau rules)

? Enable (implementation) reuse

- Implementation reuse is important here, due to the complexity
- Via inheritance (open-closed principle)
- Via configurable components (black-box reuse)

? How to organize the design & inheritance space in which these software abstractions reside?

⇒ MIDELORA space

Description Logics

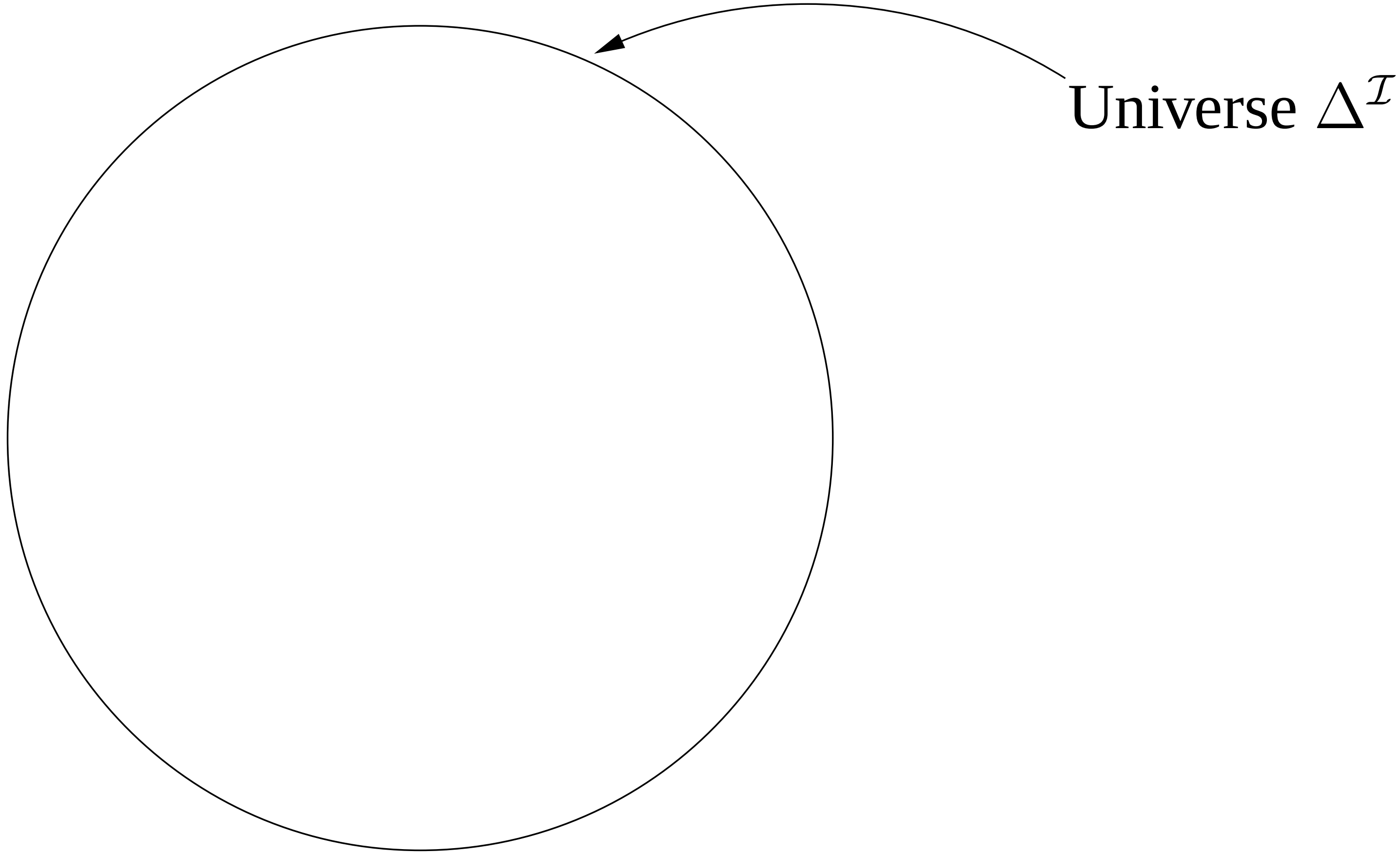
- Family of (decidable) logics, most are (strict) subsets of predicate logic in a variable-free syntax, or modal logics
- Central notions (1):
  - Concepts (classes): denote / represent sets of individuals in some UOD (interpretation domain $\Delta^{\mathcal{I}}$)
    - atomic concepts (concept names): $woman$

    $\Rightarrow$ Semantics: $woman^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
    - complex concepts (descriptions): $person \sqcap female$

    $\Rightarrow$ $(person \sqcap female)^{\mathcal{I}} = person^{\mathcal{I}} \cap female^{\mathcal{I}}$
  - Roles: denote binary relationships, has_child
  - Subsumption: $woman$ is more general than $mother$: $mother \dot{\sqsubseteq} woman$, $mother^{\mathcal{I}} \subseteq woman^{\mathcal{I}}$

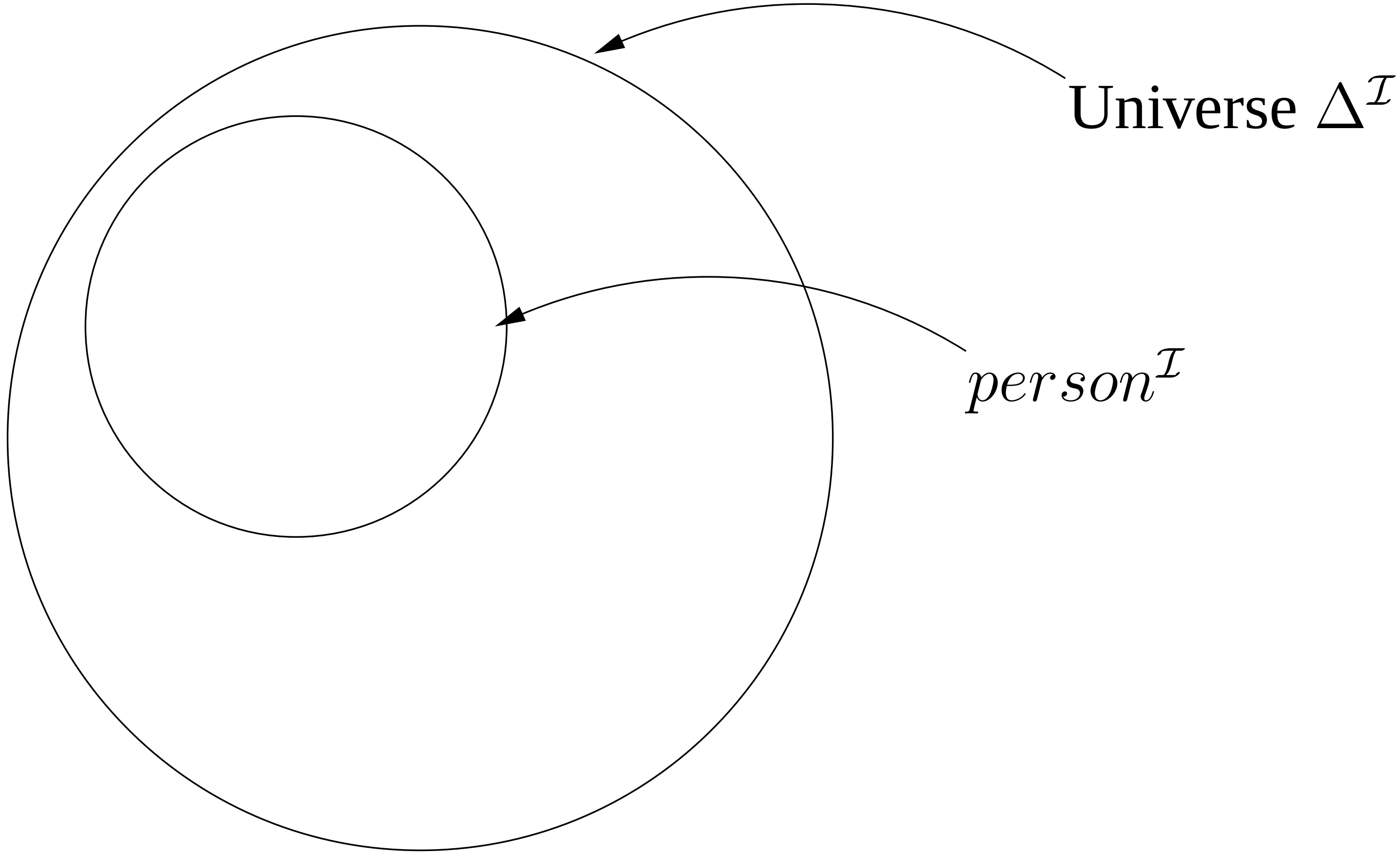
Description Logics

Illustration of an interpretation



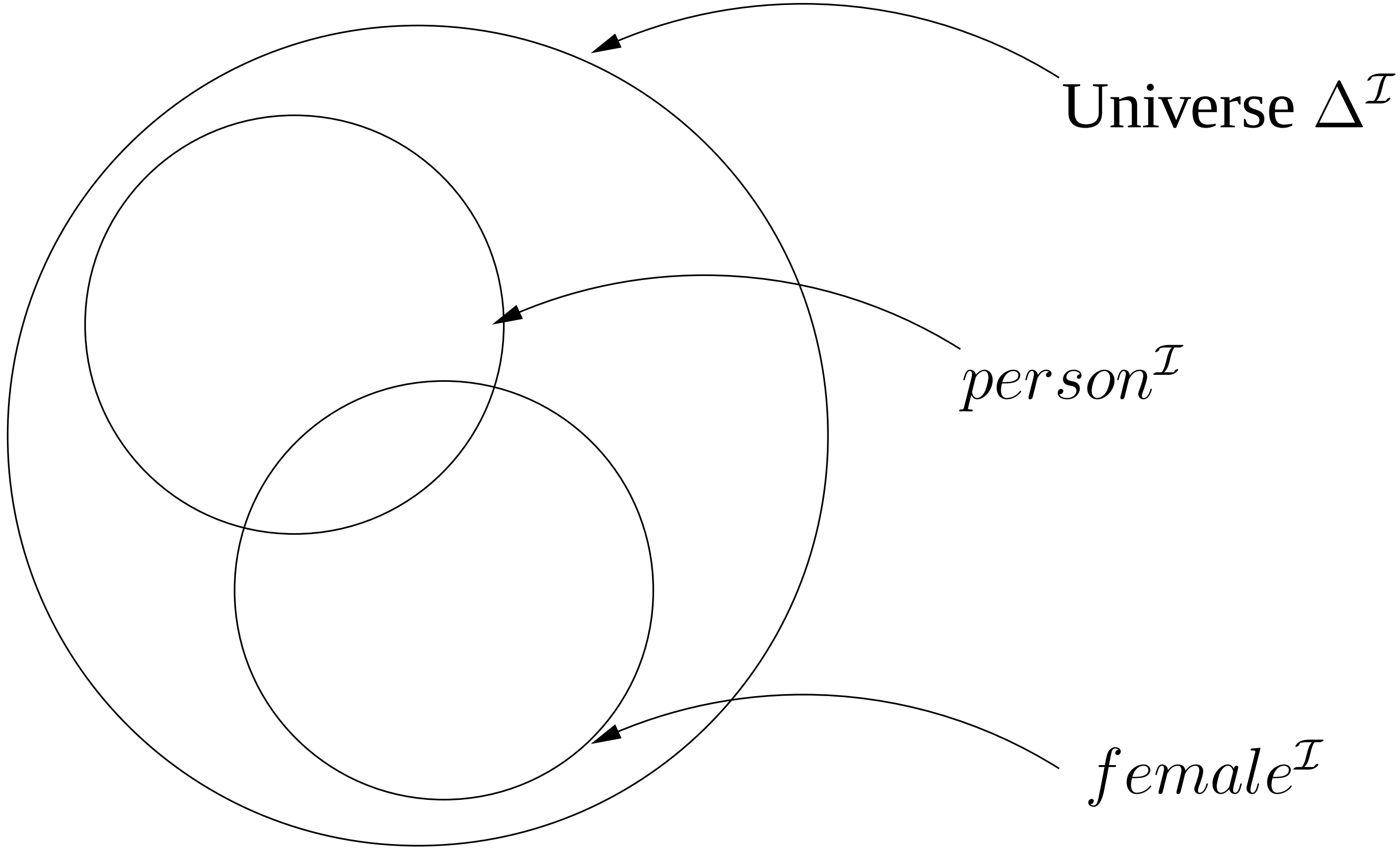
Description Logics

Illustration of an interpretation



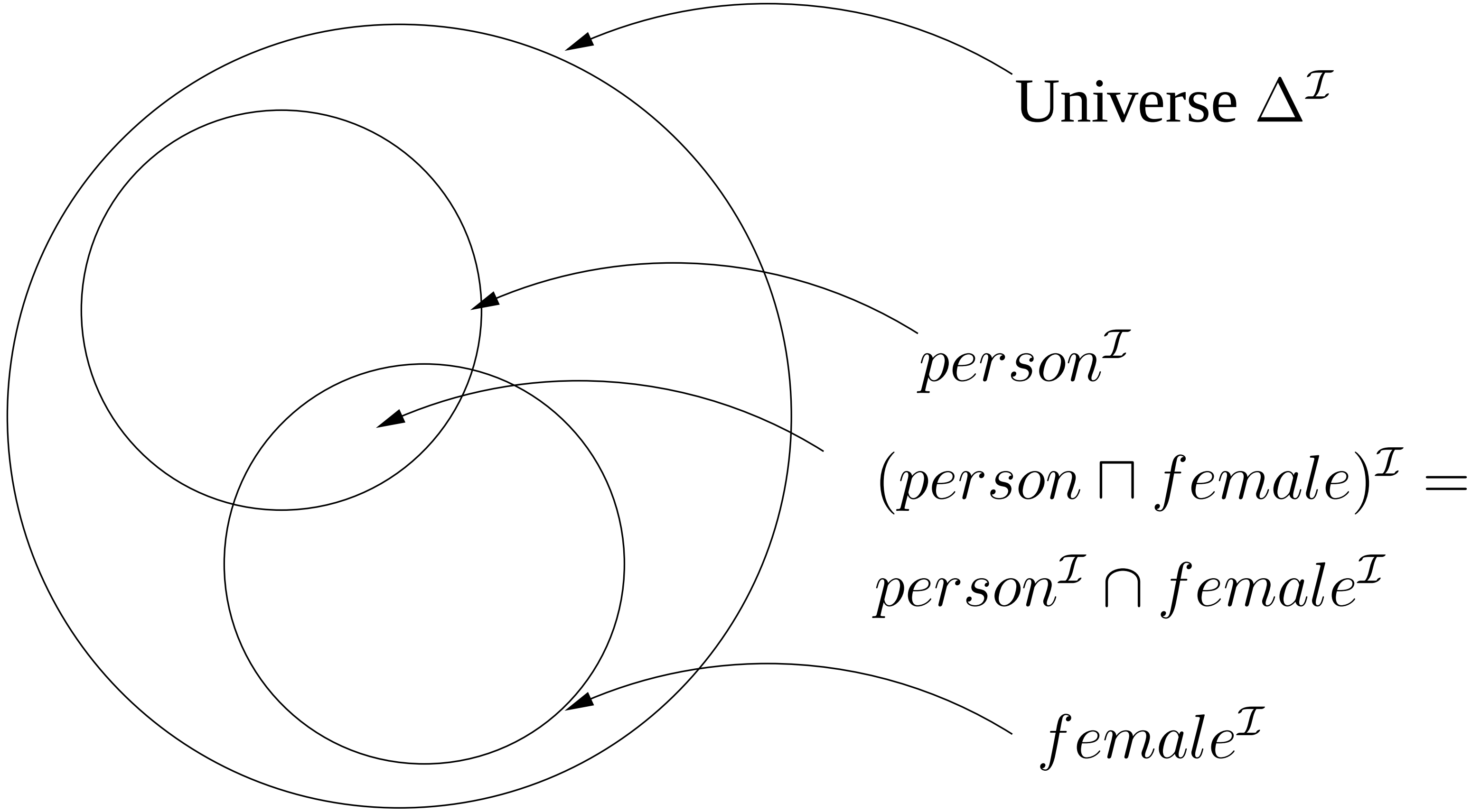
Description Logics

Illustration of an interpretation



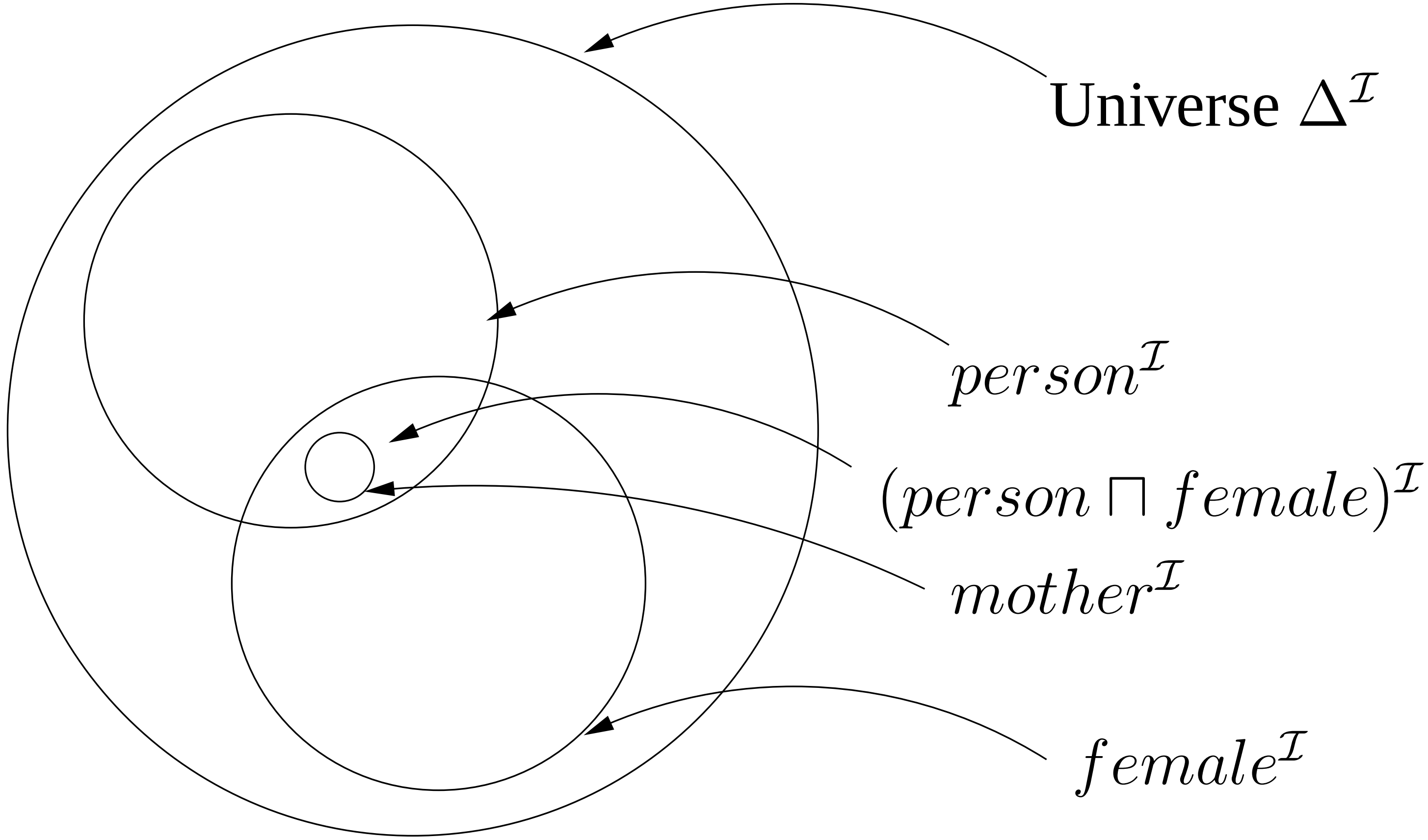
Description Logics

Illustration of an interpretation



Description Logics

Illustration of an interpretation



Description Logics (2)

- Central notions (2):
  - Concept Satisfiability: is there some interpretation $\mathcal{I}$ such that $C^{\mathcal{I}} \neq \emptyset$? $\mathcal{I}$ is called a model of C then.

  $\Rightarrow$ e.g., $C \sqcap \neg C$ unsatisfiable
  - Concept Subsumption: does $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ hold in all $\mathcal{I}$'s?

    $\Rightarrow$ D is more general than C, $C \dot{\sqsubseteq} D$, e.g. $person \sqcap female \dot{\sqsubseteq} person$
  - TBox (terminological box), "background knowledge"
    - set of axioms, $C \dot{\sqsubseteq} D$, $C \dot{\equiv} D$
    - reduce possible interpretations: $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, $C^{\mathcal{I}} = D^{\mathcal{I}}$
    - definitions: $\{woman \dot{\equiv} person \sqcap female\}$

    $\Rightarrow$ $woman^{\mathcal{I}} = person^{\mathcal{I}} \cap female^{\mathcal{I}}$
    - $woman \sqcap \neg female$ now unsatisfiable

Description Logics (3)

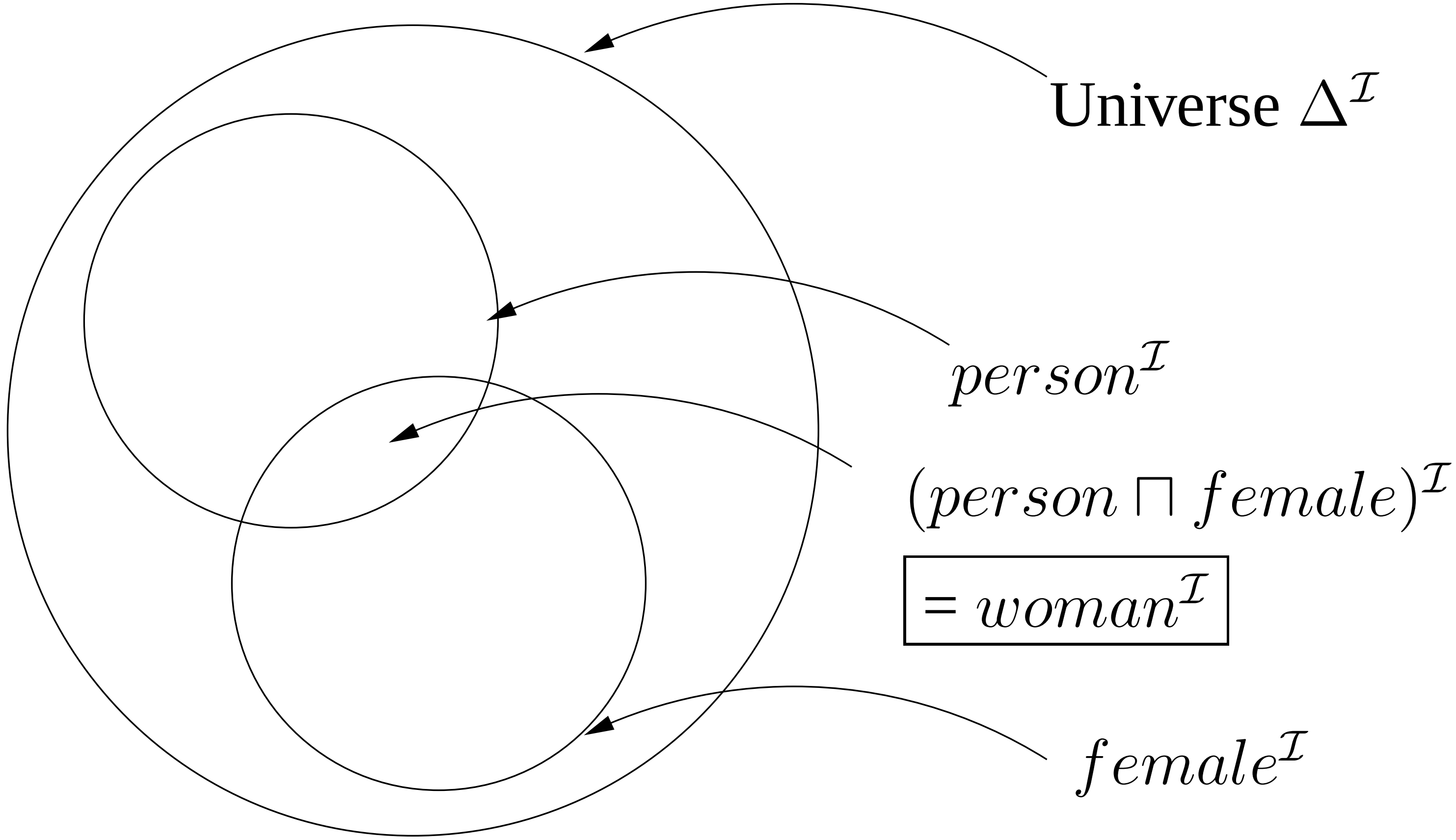
- Central notions (3):
  - ABox (assertional Box), individuals and relationships
    - individuals = constants, e.g., $betty$

    $\Rightarrow$ $betty^{\mathcal{I}} = \boxed{\text{the real betty}}$
    - set of assertions, $betty : woman$, $(betty, charles) : has\_child$ (concept and role assertions)
    - constrain / reduce possible interpretations: $betty^{\mathcal{I}} \in woman^{\mathcal{I}}$, $(betty^{\mathcal{I}}, charles^{\mathcal{I}}) \in has\_child^{\mathcal{I}}$
    - ABox = node- and edge-labeled graph
    - ABox satisfiability ("Database consistent?")

    $\Rightarrow$ $\{betty : woman, betty : \neg female\}$ is unsatisfiable

Description Logics (3)

Effect of TBox axiom $woman \doteq person \sqcap female$



Description Logics (4)

- Central notions (4):
  - not only boolean operators offered, but also quantifiers (over role fillers = "slot fillers")

⇒ existential: $mother \doteq woman \sqcap \boxed{\exists has\_child.person}$

⇒
```
(equivalent mother
      (and woman (some has-child person)))
```

⇒ FOPL: $\forall x.(mother(x) \leftrightarrow woman(x) \wedge \exists y.(has\_child(x,y) \wedge person(y)))$

⇒ universal: $mother\_without\_daughters \doteq mother \sqcap \boxed{\forall has\_child.male}$

⇒ FOPL: $\forall x.(mother\_without\_daughters(x) \leftrightarrow mother(x) \wedge \forall y.(has\_child(x,y) \rightarrow person(y)))$

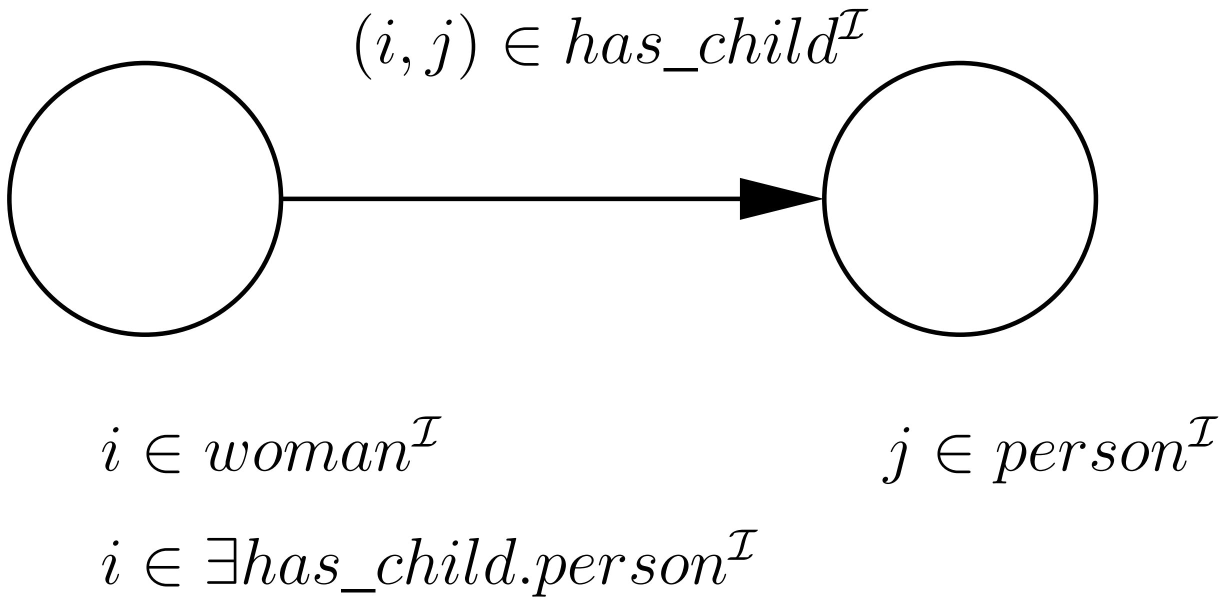
Description Logics (5)

- Core inference problem: (concept, TBox, ABox) satisfiability
- Decidable with Tableau calculi
  - attempt to construct a model (satisfying interpretation), witnessing satisfiability
  - if unsuccessful, unsatisfiable
- Tableau = finite representation of a model
- Very similar to an ABox (node- and edge-labeled graph)
- Input ABox augmented with assertions added by the calculus
- Illustration of models

Description Logics (5)

Concept model of $mother$ w.r.t. TBox

$\{mother \doteq woman \sqcap \exists has\_child.person\}$:

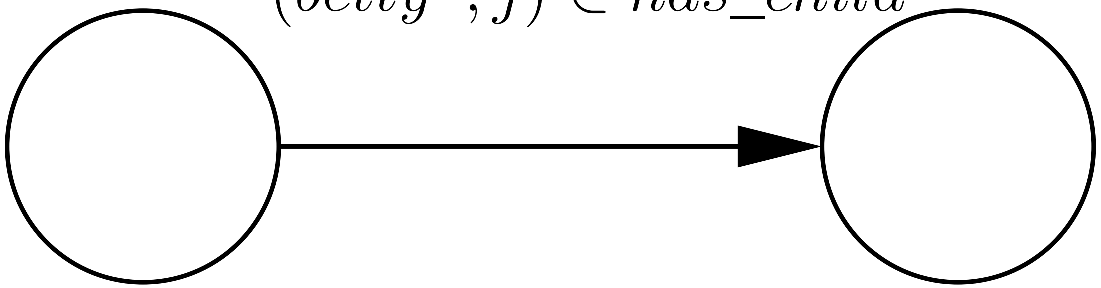


Description Logics (5)

ABox model of $\{betty : mother\}$ w.r.t. TBox

$\{mother \doteq woman \sqcap \exists has\_child.person\}$

$(betty^{\mathcal{I}}, j) \in has\_child^{\mathcal{I}}$



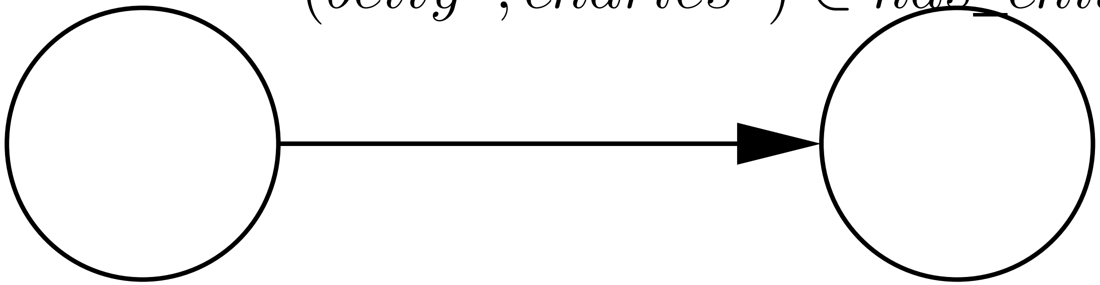
$betty^{\mathcal{I}} \in woman^{\mathcal{I}}$ $\quad j \in person^{\mathcal{I}}$

$betty^{\mathcal{I}} \in \exists has\_child.person^{\mathcal{I}}$

Description Logics (5)

ABox model of $\{betty : mother, (betty, charles) : has\_child\}$
w.r.t. TBox $\{mother \doteq woman \sqcap \exists has\_child.person\}$

$(betty^{\mathcal{I}}, charles^{\mathcal{I}}) \in has\_child^{\mathcal{I}}$

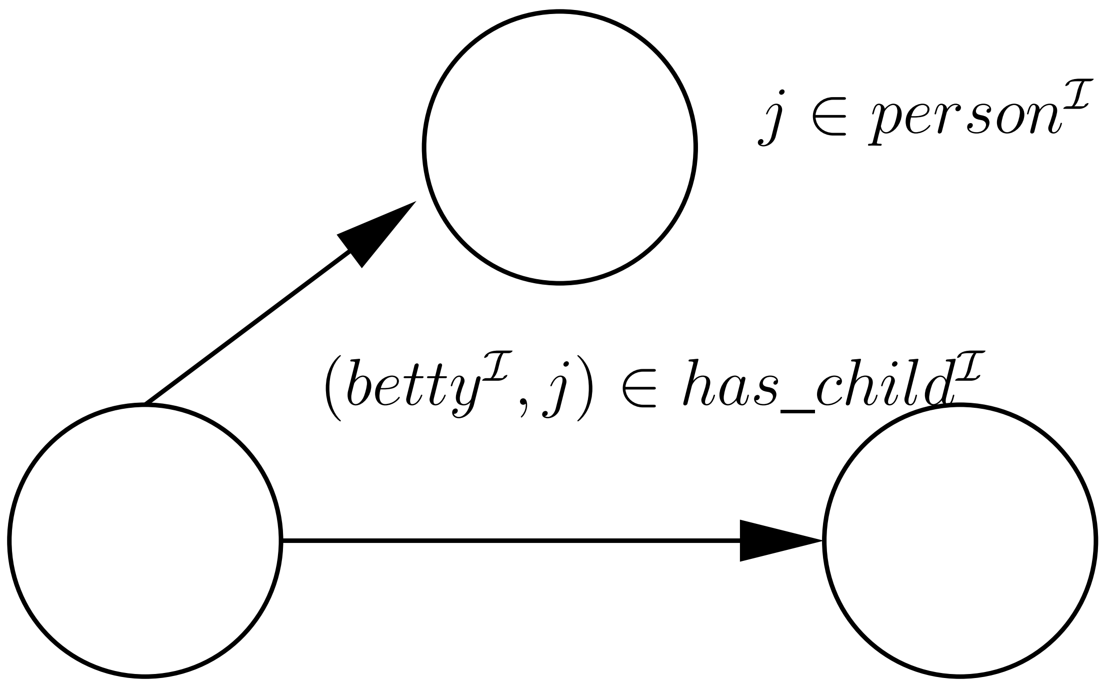


$betty^{\mathcal{I}} \in woman^{\mathcal{I}}$ $charles^{\mathcal{I}} \in person^{\mathcal{I}}$

$betty^{\mathcal{I}} \in \exists has\_child.person^{\mathcal{I}}$

Description Logics (5)

ABox model of $\{betty : mother, (betty, charles) : has\_child\}$
w.r.t. TBox $\{mother \doteq woman \sqcap \exists has\_child.person\}$ (2)



$betty^{\mathcal{I}} \in woman^{\mathcal{I}}$

$betty^{\mathcal{I}} \in \exists has\_child.person^{\mathcal{I}}$

The DL Family

- A DL is a logic

  $\Rightarrow$ formal language (set of well-formed expressions)

  $\Rightarrow$ with model-theoretic semantics ($\Rightarrow$ reasoning)

- $\mathcal{ALC}$: concept constructors $\{\sqcap, \sqcup, \exists R.C, \forall R.C\}$
- $\mathcal{ALCI}$: $\mathcal{ALC}$ plus so-called inverse roles (R^{-1})
- Subset relationship between DLs: $\mathcal{ALC} \subseteq \mathcal{ALCI}$
- An $\mathcal{ALCI}$ prover can of course be used for $\mathcal{ALCI}$
- Often: $\mathcal{DL} \subseteq \mathcal{DL}' \Rightarrow$ more expressive $\Rightarrow$ higher (computational) complexity
- Optimizations sometimes only for "smaller" DLs known (e.g., model merging for DLs without inverse roles)

Motivation Continued - Optimizations

- Optimizations strictly necessary (many practically relevant DLs are at least EXPTIME-complete)
- Applicable optimizations have to be detected and applied automatically by the DL prover

⇒ Complicates implementation quite a bit (`(if ...)`)

- MIDELORA approach: instead of defining one prover for a very expressive DL, define many small and concise provers for DLs:

⊕ Optimizations can be “pinpointed” and localized

⊕ Non-comparable (w.r.t. $\subseteq$) branches in the DL family can be implemented (e.g., non-standard DLs)

? Many small provers instead of one big prover ⇒ easier?

MiDeLoRa Design Rationales

- Only easier, if provers are very concise
- Common components have to be reused by different provers (tableau rules) ⇒ move software complexity in reusable tableau rules, provers focus on intellectual complexity
- Define dedicated prover for $\mathcal{DL}$ only for good reasons, otherwise inherit a prover for $\mathcal{DL} \subseteq \mathcal{DL}'$ (if possible)

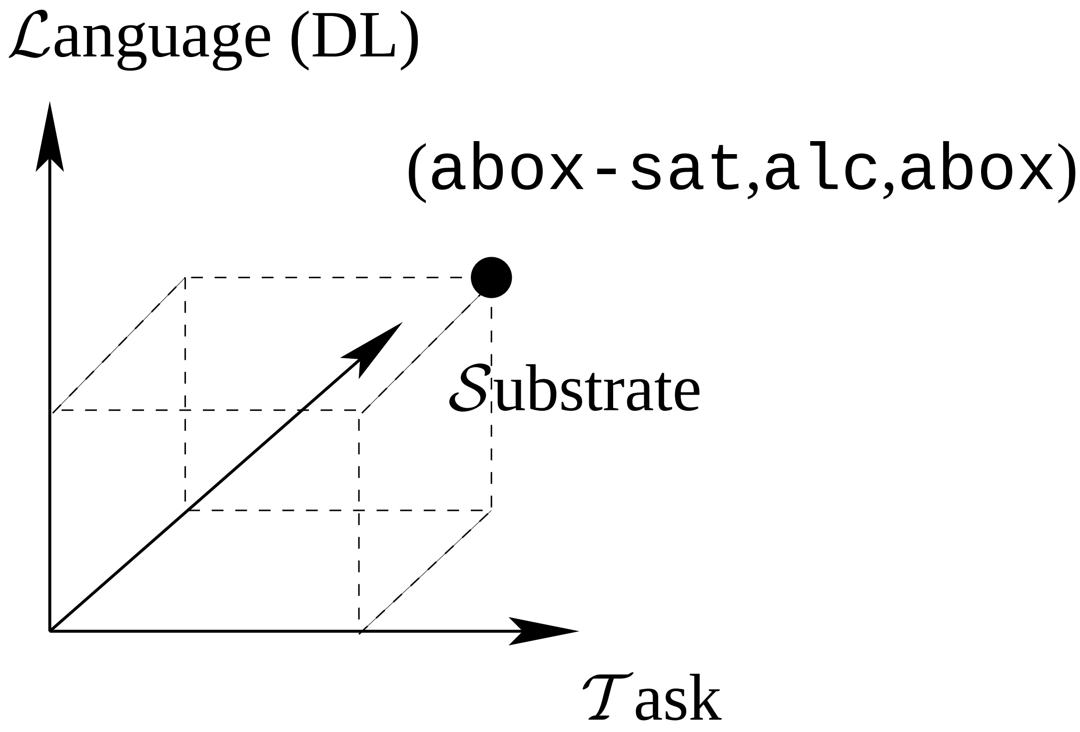
  ⇒ define CLOS language classes for DLs (e.g., `ALC`, `ALCI`), use method dispatch for prover selection

  - a good reason: $\mathcal{DL}$ allows for certain optimization (⇒ complicates implementation), but $\mathcal{DL}'$ doesn't
- If possible, provers do not commit to a concrete ABox representation (substrate protocol)

Substrate Data Model

- Node- and edge-labeled graph $S = (V, E, L_V, L_E, \mathcal{L}_\mathcal{V}, \mathcal{L}_\mathcal{E})$
- Variable description languages $\mathcal{L}_\mathcal{V}$, $\mathcal{L}_\mathcal{E}$, e.g. $\mathcal{L}_\mathcal{V} =_{def} \mathcal{ALC}$, $\mathcal{L}_\mathcal{E} =_{def} \mathcal{N}_\mathcal{R}$ for $\mathcal{ALC}$ ABox ($\Rightarrow$ flexibility)
- Abstract CLOS classes `substrate`, `node`, `edge`, `node-description`, `edge-description` etc. (but “template methods”)
- Substrate protocol (data abstraction)
  - `create-node`, `create-edge`
  - `get-nodes`, `get-edges`
  - `loop-over-nodes`, `loop-over-edges`
  - (indexed) access: `get-matching-nodes <descr.>`, `loop-over-matching-nodes`, ...

MIDELORA Space



- Prover: ternary multi-method

  `(defprover (abox-sat alc abox) ...)`

- CLOS classes for $\mathcal{DL}$s and $\mathcal{S}$ubstrates (ABoxes)
- Symbol dispatch for $\mathcal{T}$ axis

⇒ Provers can cover "planes" (not spaces)

MIDELORA Space (2)

- Language classes: $\mathcal{ALC} \subseteq \mathcal{ALCI}$, $\mathcal{ALC} \subseteq \mathcal{ALC}_{\mathcal{R}^+}$
  1. `(defclass alc (alci) ...)` (co-variant)

  $\Rightarrow$ $\mathcal{ALCI}$ prover is sufficient, dedicated $\mathcal{ALC}$ prover can be defined if reasonable, standard dispatch will work

  2. `(defclass alci (alc) ...)` (contra-variant)

  $\Rightarrow$ $\mathcal{ALC}$ prover incomplete for $\mathcal{ALCI}$, thus, both provers are needed if standard dispatch shall work (bad)

- Represent characteristic properties as mixin classes
  - co-variant properties: `(defclass alc (alci admits-model-merging-p) ...)`
  - contra-variant properties: `(defclass alcr+ (alc needs-blocking-p) ...)`

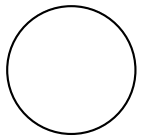
MIDELORA Space (3)

- Decisions
  - Most properties are in fact contra-variant
  - ⇒ Arrange language classes in a contra-variant way ...
  - ... and define non-standard dispatch for $\mathcal{L}$-argument (downcast $\mathcal{L}$ argument until prover found)
- Alternative idea (thanks to a reviewer): negative properties
  - however, positive properties needed for dispatch
  - solution: assume properties to be true by default
  - specialized behavior on the absence of information??
- $\mathcal{S}$-axis: co-variant standard CLOS dispatch
- $\mathcal{T}$-axis: reuse via delegation, not inheritance (problem reduction, e.g. individual_instance? $\Rightarrow \neg$ abox_sat?)

Tableau Calculi

Tableau Expansion of $C \sqcap (\exists R.D \sqcup \exists R.E) \sqcap \forall R.\neg D$

1. Create initial node:

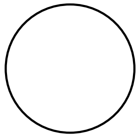


$$C \sqcap (\exists R.D \sqcup \exists R.E) \sqcap \forall R.\neg D$$

Tableau Calculi

Tableau Expansion of $C \sqcap \exists R.(D \sqcup E) \sqcap \forall R.\neg D$

2. Break up conjunction ($\sqcap$-rule)



$C \sqcap (\exists R.D \sqcup \exists R.E) \sqcap \forall R.\neg D$

$\boxed{C}$

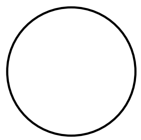
$\boxed{(\exists R.D) \sqcup (\exists R.E)}$

$\boxed{\forall R.\neg D}$

Tableau Calculi

Tableau Expansion of $C \sqcap \exists R.(D \sqcup E) \sqcap \forall R.\neg D$

3. Expand disjunction $(\exists R.D) \sqcup (\exists R.E)$ (⊔-rule)



$C \sqcap (\exists R.D \sqcup \exists R.E) \sqcap \forall R.\neg D$

C

$(\exists R.D) \sqcup (\exists R.E)$

$\forall R.\neg D$

$\boxed{\exists R.D}$

Tableau Calculi

Tableau Expansion of $C \sqcap \exists R.(D \sqcup E) \sqcap \forall R.\neg D$

4. Expand existential restriction $\exists R.D$ ($\exists$-rule)

$C \sqcap (\exists R.D \sqcup \exists R.E) \sqcap \forall R.\neg D$ $\qquad D$

C

$(\exists R.D) \sqcup (\exists R.E)$

$\forall R.\neg D$

$\exists R.D$

Tableau Calculi

Tableau Expansion of $C \sqcap \exists R.(D \sqcup E) \sqcap \forall R.\neg D$

5. Apply universal restriction $\forall R.\neg D$ ($\forall$-rule) $\Rightarrow$ ↯

$C \sqcap (\exists R.D \sqcup \exists R.E) \sqcap \forall R.\neg D$ $\quad$ D, $\boxed{\neg D}$ ↯

C

$(\exists R.D) \sqcup (\exists R.E)$

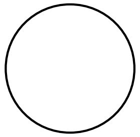
$\forall R.\neg D$

$\exists R.D$

Tableau Calculi

Tableau Expansion of $C \sqcap \exists R.(D \sqcup E) \sqcap \forall R.\neg D$

6. Backtracking, reconsider disjunction $(\exists R.D) \sqcup (\exists R.E)$



$C \sqcap (\exists R.D \sqcup \exists R.E) \sqcap \forall R.\neg D$

C

$(\exists R.D) \sqcup (\exists R.E)$

$\forall R.\neg D$

$\boxed{\exists R.E}$

Tableau Calculi

Tableau Expansion of $C \sqcap \exists R.(D \sqcup E) \sqcap \forall R.\neg D$

7. Expand existential restriction $\exists R.E$ ($\exists$-rule)

$C \sqcap (\exists R.D \sqcup \exists R.E) \sqcap \forall R.\neg D$ $\quad \boxed{E}$

C

$(\exists R.D) \sqcup (\exists R.E)$

$\forall R.\neg D$

$\exists R.E$

Tableau Calculi

Tableau Expansion of $C \sqcap \exists R.(D \sqcup E) \sqcap \forall R.\neg D$

8. Apply universal restriction $\forall R.\neg D$ ($\forall$-rule) $\Rightarrow$ done

$C \sqcap (\exists R.D \sqcup \exists R.E) \sqcap \forall R.\neg D$ $\quad$ E, $\boxed{\neg D}$

C

$(\exists R.D) \sqcup (\exists R.E)$

$\forall R.\neg D$

$\exists R.E$

Tableau Rules for $\mathcal{ALC}$

$\sqcap$-Regel :

if 1. $x : C_1 \sqcap C_2 \in \mathcal{A}$

2. $\{x : C_1, x : C_2\} \not\subseteq \mathcal{A}$

then

$\mathcal{A}' := \mathcal{A} \cup \{x : C_1, x : C_2\}$

$\sqcup$-Regel :

if 1. $x : C_1 \sqcup C_2 \in \mathcal{A}$

2. $\{x : C_1, x : C_2\} \cap \mathcal{A} = \emptyset$

then $\mathcal{A}' := \mathcal{A} \cup \{x : C_1\}$

$\mathcal{A}_1 := \mathcal{A} \cup \{x : C_2\}$

$\exists$-Regel :

if 1. $x : \exists R.C_1 \in \mathcal{A}$

2. es gibt kein y, sodass

$\{y : C_1, (x, y) : R\} \subseteq \mathcal{A}$

then

$\mathcal{A}' := \mathcal{A} \cup \{y : C_1, (x, y) : R\}$

$\forall$-Regel :

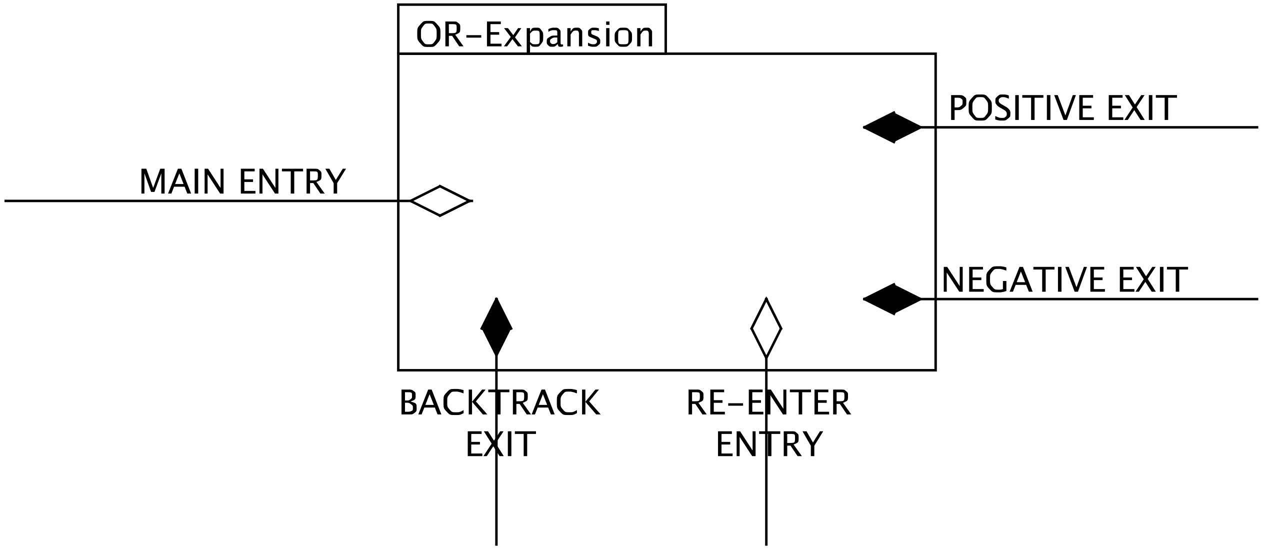
if 1. $\{x : \forall R.C_1, (x, y) : R\} \subseteq \mathcal{A}$

2. $y : C_1 \notin \mathcal{A}$

then $\mathcal{A}' := \mathcal{A} \cup \{y : C_1\}$

- non-determinism: $\sqcup$-rule $\Rightarrow$ search needed
- if the rules can be applied in such a way that a complete and clash-free tableau is produced $\Rightarrow$ ABox satisfiable

5-Port Model for Rules



- `MAIN ENTRY`: new rule incarnation
- `POSITIVE EXIT`: rule was applied
- `NEGATIVE EXIT`: rule was not applied
- `BACKTRACK EXIT`: return control to parent incarnation
- `RE-ENTER ENTRY`: get control back from parent incarnation

Simple $\mathcal{ALC}$ Prover in Lisp (1)

```
(defun alc-sat (concept)
  (labels ((alc-sat1 (expanded unexpanded)
             (labels ((get-negated-concept (concept)
                        (nnf `(not ,concept)))
                      (select-concept-if-present (type)
                        (find-if #'(lambda (concept)
                                     (and (consp concept)
                                          (eq (first concept) type)))
                                 unexpanded))
                      (select-atom-if-present ()
                        (find-if #'(lambda (concept)
                                     (or (symbolp concept)
                                         (and (consp concept)
                                              (eq (first concept) 'not)
                                              (symbolp (second concept)))))
                                 unexpanded))
                      (clash (concept)
                        (let ((negated-concept (get-negated-concept concept)))
                          (find negated-concept expanded :test #'equal)))
                      (register-as-expanded (concept)
                        (unless (clash concept)
                          (alc-sat1 (cons concept expanded)
                                    (remove concept unexpanded :test #'equal)))))
```

Simple $\mathcal{ALC}$ Prover in Lisp (2)

```
(let ((atom (select-atom-if-present)))
  (if atom
      (register-as-expanded atom)
    ;; else
    (let ((and-concept (select-concept-if-present 'and)))
      (if and-concept
          (progn
            (dolist (conjunct (rest and-concept))
              (when (clash conjunct)
                (return-from alc-sat1 nil))
              (push conjunct unexpanded))
            (register-as-expanded and-concept))
        ;; else
        (let ((or-concept (select-concept-if-present 'or)))
          (if or-concept
              (let ((unexpanded-old unexpanded))
                (some #'(lambda (arg)
                          (unless (clash arg)
                            (setf unexpanded
                              (cons arg unexpanded-old))
                            (register-as-expanded or-concept)))
                      (rest or-concept)))
            ;; else
```

Simple $\mathcal{ALC}$ Prover in Lisp (3)

```
;; else
(let ((some-concept (select-concept-if-present 'some)))
  (if some-concept
      (let* ((qualification (third some-concept))
             (role (second some-concept))
             (initial-label
              (cons
                qualification
               (mapcar #'third
                   (remove-if-not
                     #'(lambda (concept)
                          (and (consp concept)
                               (eq (first concept) 'all)
                               (eq (second concept) role)))
                        unexpanded)))))
        (and (alc-sat1 nil initial-label)
             (register-as-expanded some-concept)))
    ;; else
    t)))))))))))
```

. . . concise, but too simple

- Satisfiability of concepts in NNF only (without TBox)
- No ABox representation (of course), but . . .
- . . . implicit tableau representation (stack)
- Stack frame = tableau state = state in search space = rule incarnation
- No tableau / ABox data abstraction (and lists don't scale): suppose hash tables were used for set representation? ⇒ generic substrate data model
- No optimizations, many `if`'s would have to be included
- But backtracking for free! (unboxed data structures)

⇒ Cannot survive complex input

abox_sat in MIDELORA for $\mathcal{ALC}$

```
(defprover ((abox-sat alc abox))
  (:init
   (perform (initial-abox-saturation)
     (:body
      (start-main))))
  (:main
      (perform (deterministic-expansion)
        (:body
         (if clashes
             (handle-clashes)
           (perform (or-expansion)
             (:positive
              (if clashes
                  (handle-clashes)
                (restart-main)))
             (:negative
                 (perform (some-expansion)
                   (:positive
                    (if clashes
                        (handle-clashes)
                      (restart-main)))
                   (:negative
                    (success)))))))))
  (:success
   (completion-found)))
```

- Focus on intellectual complexity, not software complexity
- ABox representation data abstraction
- Optimizations = additional rule applications

Prover `:main` in the 5-Port-Model

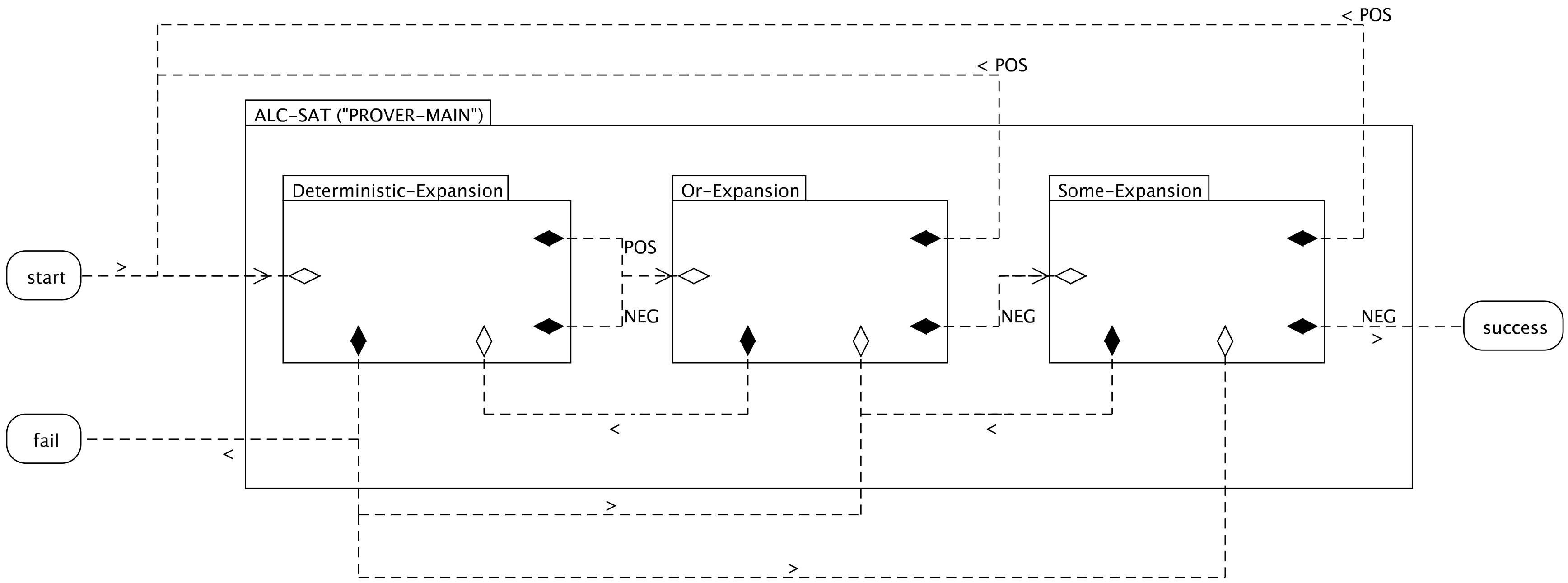


Tableau Rule Definition

```
(defrule some-expansion (dl-with-somes abox)
  (multiple-value-bind (some-concept node)
      (select-some-concept abox *strategy* language)
    (cond ((not node)
           +insert-negative-code+ )
          (t
           (let ((role (role some-concept))
                 (new-node nil))
             (register-as-expanded some-concept :node node)
             (setf new-node
                   (create-anonymous-node abox
                    :depends-on (list (list node some-concept))))
             (relate node new-node role
                     :old-p nil
                     :depends-on (list (list node some-concept)))
             (perform (compute-new-some-successor-label
                       :new-node new-node
                       :node node :role role
                       :concept some-concept))
             +insert-positive-code+ )))))
```

- Reusable components, often parameterizable (not shown here)
- ABox representation data abstraction
- Focus on software complexity, optimizations = clever programming

Data Abstraction and Backtracking

- Conceptually, an ABox substrate can be a simple list (simple $\mathcal{ALC}$ prover)

⇒ Backtracking easy if list is modified via `push, cons`; simply keep a pointer

- However, most substrate implementations will be boxed (ABox = CLOS object graph, or RDF triple store, . . .)
- Backtracking?
  - histories of command objects (“log file”)
  - compensation operations (`undo` method)

⇒ Memory intensive, lightweight objects (list structures)

- Rules are responsible to revert / “roll back” the tableau (not the prover)

Why Lisp? (1)

- Problem- / domain-specific macros
  - `defprover`
  - `defrule`
  - enforce thinking in a conceptual model
- Multiple inheritance
  - to organize reuse in the MiDeLoRa space
  - mixin arbitrary properties in language classes (`alc`) (ok, possible with interfaces too), ...
  - ...but also rules defined for mixin classes (e.g., `some-expansion` for `dl-with-somes`)
  - multiple substrate superclasses, e.g. `spatial-abox` (`spatial-substrate, abox`)

Why Lisp? (2)

- Multi-methods
  - mostly used at macro expansion time during expansion of `(perform <rule>)` ("prover compile time"): `get-rule-body-code` (fixes ABox class and DL), but also generic function calls can be coded
  - `defprover`: 3 ternary multi-methods `prover-init, -main, -succes`
  - often used: `entails-p` (relation specializations with binary methods)

Why Lisp? (3)

- Method combinations
  - often, sound but incomplete predicates are used as guards, e.g., for `entails-p (subsumes-p)`
  - if guard test returns `t` (resp. `nil`), return `t`, otherwise invoke “true” expensive test

  ⇒ `:around / call-next-method` idiom or `:and` method combination type
  - contra-variant dispatch possible in CLOS
- Other (standard) arguments
  - symbolic computation
  - automatic memory management
  - fast and mature implementations, . . .

Conclusion

- Performance tested so far seems to be OK, comparable to state-of-the-art reasoners of $\approx$ 2003 (but hasn't been tested extensively, unlike RACERPRO)
- MIDELORA: 2002 - 2005
- Focus on flexibility and genericity rather than utmost performance (research prototype)

$\Rightarrow$ Deliberately traded such aspects for some CPU cycles

$\Rightarrow$ Hope: enhanced software quality and maintainability through better comprehensibility

- High memory footprint, histories can become very long
- Not an "end user" framework
- Affinity with "Software Product Families"?

History: Lisp and DLs

- KL-ONE, Brachman/Schmolze, 1975-1985 (Interlisp)
- LOOM, Bates/Brill/MacGregor, 1987-?
- CLASSIC, Borgida/McGuinness/Patel-Schneider, 1989-1992
- KRIS, Baader/Hollunder/Hanscke, ca. 1991-1994
- original FACT, Horrocks, 1997-today (successors)
- RACER, Haarslev/Möller, 1999-2004
- RACERPRO, Haarslev/Möller/Wessel, 2004-today
- “standard” KRSS syntax, 1993:

  `(and woman (some has-child person) (all has-child male))`
- See chapter “Description Logic Systems” in DL Handbook by Möller and Haarslev ;-)

Thanks!

Work supported by

TUHH Technische Universität Hamburg-Harburg · TONES · boemie (Bootstrapping Ontology Evolution with Multimedia Information Extraction) · Racer Systems

λ