Funktionsweise eines Resolutionsbeweisers

Vortrag im Rahmen der Vorlesung
Künstliche Intelligenz WS 2007/2008

Agenda

→ **Einführung**

Überblick über das System

Die Algorithmen im Detail

Optimierungsansätze

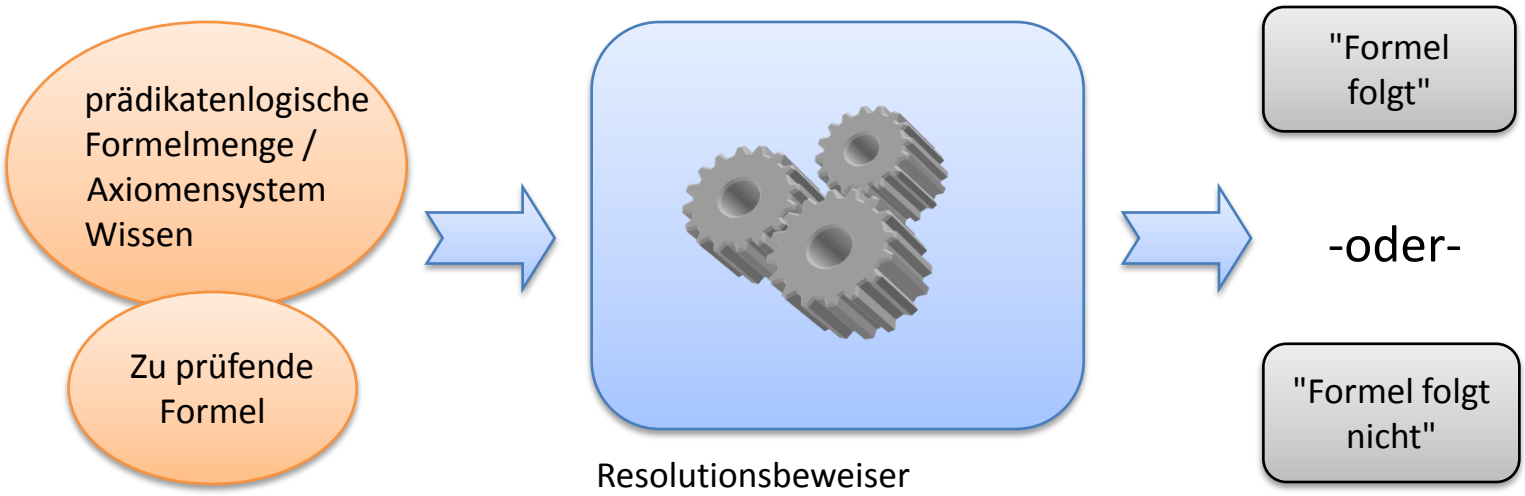
Zusammenfassung

Die Problemstellung: Eine Software zum automatischen Beweisen

Ziel

- Entwicklung eines Softwaresystems zur automatischen Beweisführung mit Hilfe des Resolutionskalküls

→ Überprüfung eines prädikatenlogischen Ausdrucks 1. Stufe mit einem Algorithmus, ausgehend von einer Wissensbasis



Anforderungen an einen automatischen Beweiser

Herausforderung

- Wie ermittelt man für eine gegebene Formelmenge Φ, ob daraus die Formel φ folgt ($\Phi \models \varphi$)?

Schwierigkeit

- Eine einfache Berechnung bzw. systematisches Überprüfen (wie z.B. in der Aussagenlogik) ist nicht möglich!
- Problem: Quantoren ($\forall x\, P(x)$ würde bedeuten, *alle* x auszuprobieren!)

Lösung

- Verwendung eines Beweiskalküls K ($\Phi \vdash_K \varphi$)
- Beweiskalkül = System formaler Schlussregeln (durch syntaktische Kriterien definiert)

→ Eine Herleitung ist eine reine Symbolmanipulation, es ist kein Verständnis der Bedeutung der Symbole notwendig!

Nutzung des Resolutionskalküls

Wünschenswerte Eigenschaften eines Kalküls

- Korrekt: Es können nur wahre Formeln hergeleitet werden
- Vollständig: *Jede* wahre Formel kann hergeleitet werden

Funktionsweise von automatischen Beweisern `prove`^{K}

- Aufzählung aller herleitbaren Formeln Φ^K, dann Prüfung, ob φ zu Φ^K gehört
- Das Problem, ob φ zu Φ^K gehört, ist semi-entscheidbar

Eigenschaften des Resolutionskalküls

- Beschränkung auf die Klauselsprache $\mathcal{L}$
- Aufgabe der Vollständigkeit (es werden nicht alle Formeln hergeleitet)
  - Keine Herleitung von allgemeingültigen Aussagen
  - Keine Herleitung von Spezialisierungen
  - Widerspruchsvollständigkeit: Wenn $\Phi \models \square$, dann $\Phi \vdash_K \square$
- Beweis durch: $\Phi \models \varphi$ genau dann, wenn $\mathcal{S}(\Phi \cup \{\neg\varphi\}) \vdash_R \square$

Durchführung eines Beispielbeweises

Unsere Wissensbasis

1. $\forall x$ (MasterStudent(x) $\longrightarrow$ Student(x))
2. $\forall x$ (Student(x) $\longrightarrow$ ArbeitetViel(x))
3. MasterStudent(Stefan)

Die Frage

- ArbeitetViel(Stefan) ?

Darstellung als Klauseln

1. $\neg$ MasterStudent(x) $\vee$ Student(x)
2. $\neg$ Student(y) $\vee$ ArbeitetViel(y)
3. MasterStudent(Stefan)
4. $\neg$ ArbeitetViel(Stefan)

Agenda

Einführung

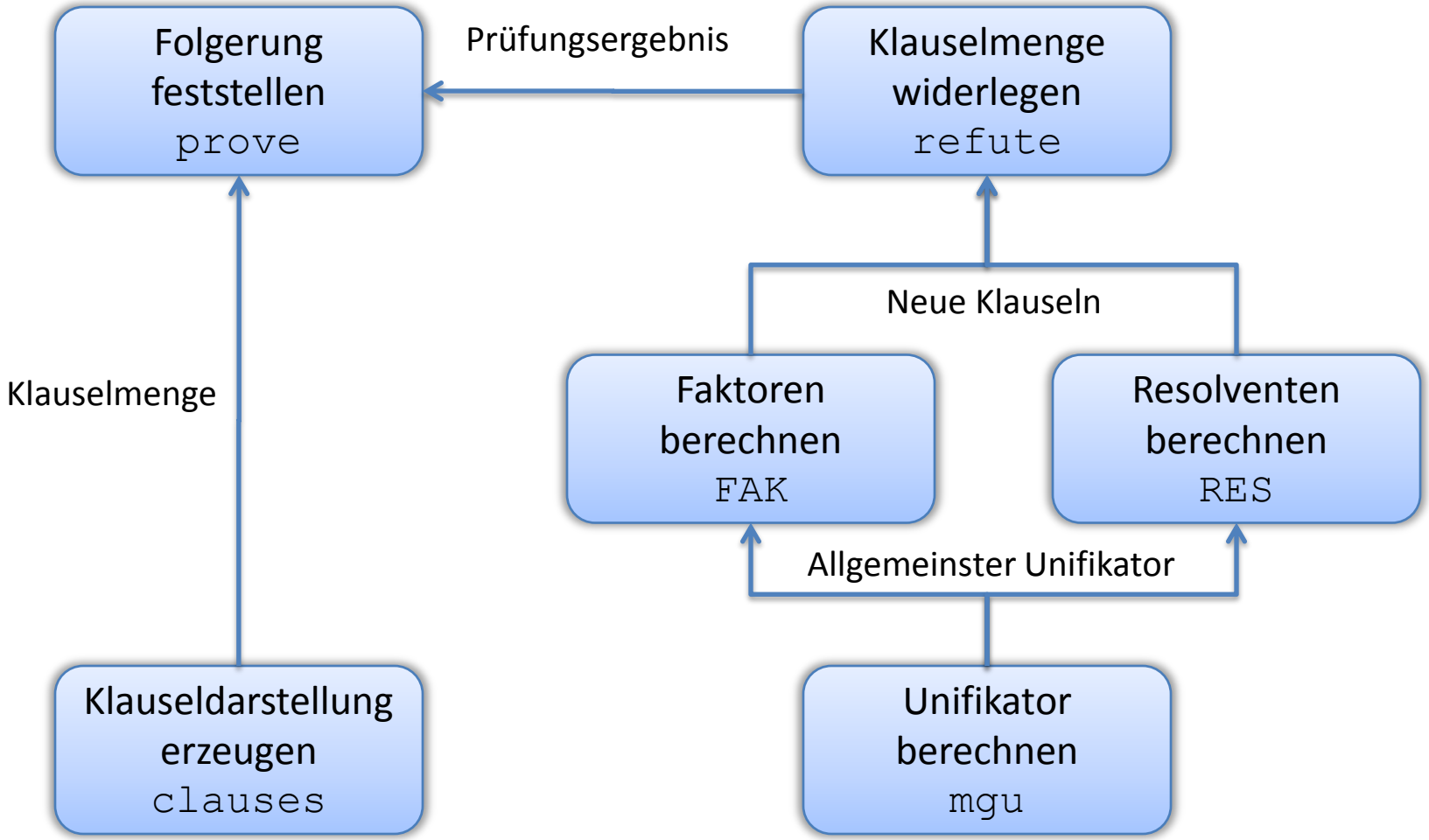
→ **Überblick über das System**

Die Algorithmen im Detail

Optimierungsansätze

Zusammenfassung

Der Resolutionsbeweiser besteht aus 6 einzelnen Algorithmen



Quelle: Dittmann, Daniel: Implementierung eines Resolutionsbeweisers, Seminar an der FH Wedel, SS 2005

Agenda

Einführung

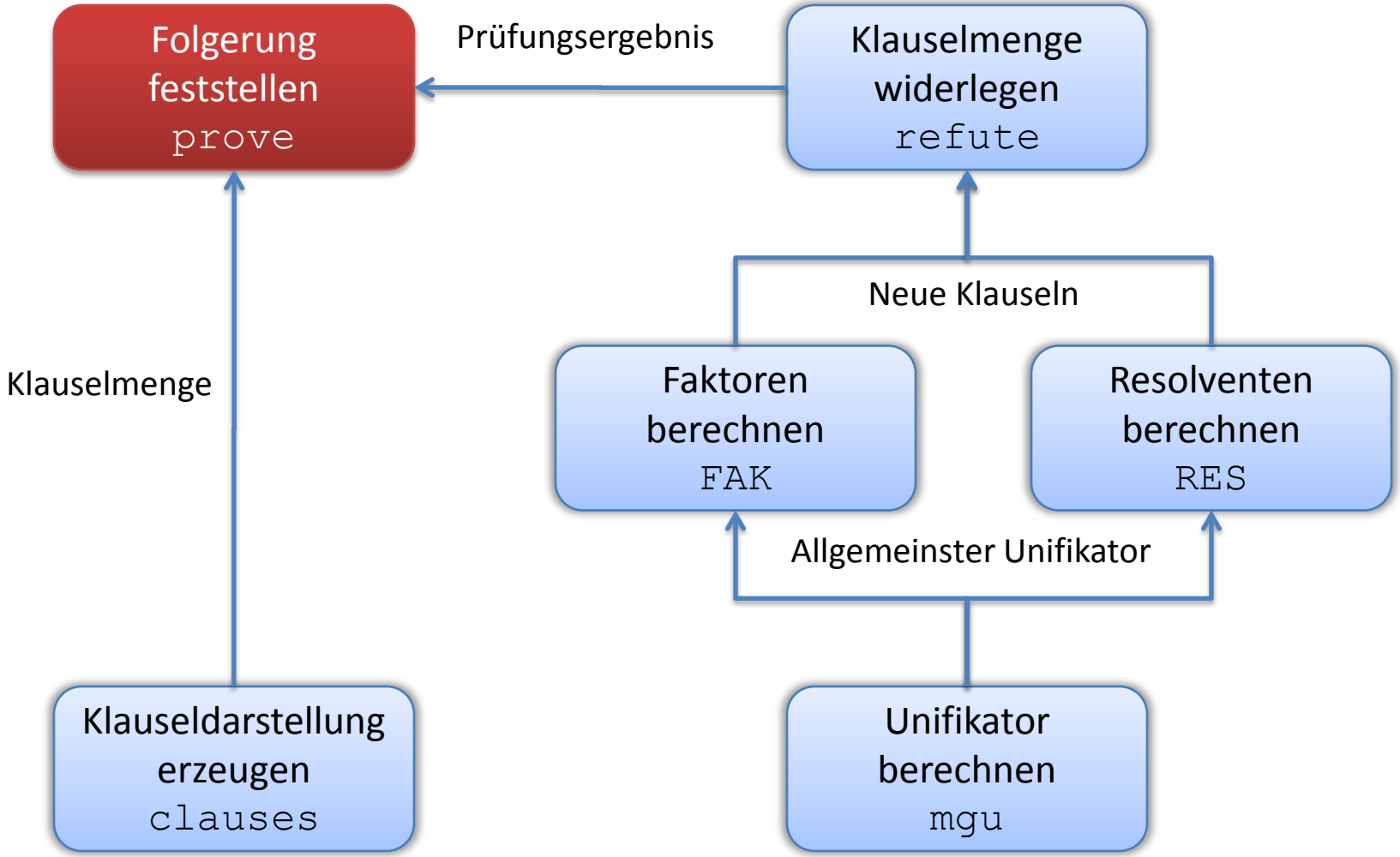
Überblick über das System

→ **Die Algorithmen im Detail**

Optimierungsansätze

Zusammenfassung

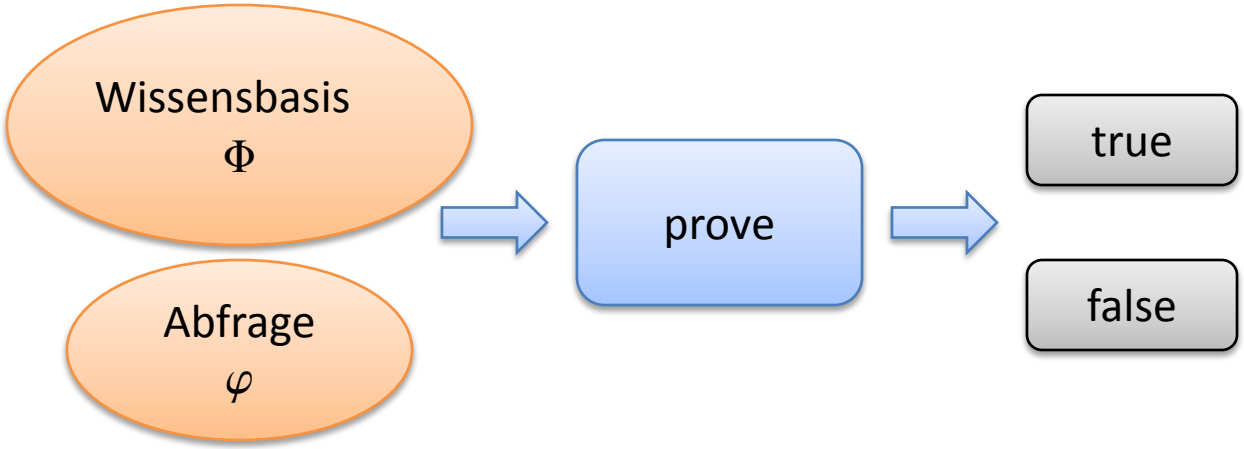
Der Beweisalgorithmus `prove` als Fassade des Systems



Der Beweisalgorithmus `prove` als Fassade des Systems

Anforderungen und Funktionalität

- Schnittstelle des Systems
- Darstellung der Inputgrößen in Prädikatenlogik 1. Stufe
- Nutzung des Resolutionskalküls ist nicht sichtbar



Der Beweisalgorithmus `prove` als Fassade des Systems

Funktionsweise

1. Umformung des Inputs (Formelmenge Φ, zu beweisende Formel φ) in Klauseldarstellung
2. Vereinigung der Klauselmenge $\mathcal{S}(\Phi)$ mit der Klauselmenge der negierten Formel $\mathcal{S}(\neg\varphi)$
3. Versuch der Widerlegung der gebildeten Klauselmenge

Pseudocode

function `prove` $(\Phi \in 2^{\mathcal{F}}, \varphi \in \mathcal{F})$ **:bool**

$\mathcal{S}_{\mathcal{A}} :=$ `clauses` (Φ)

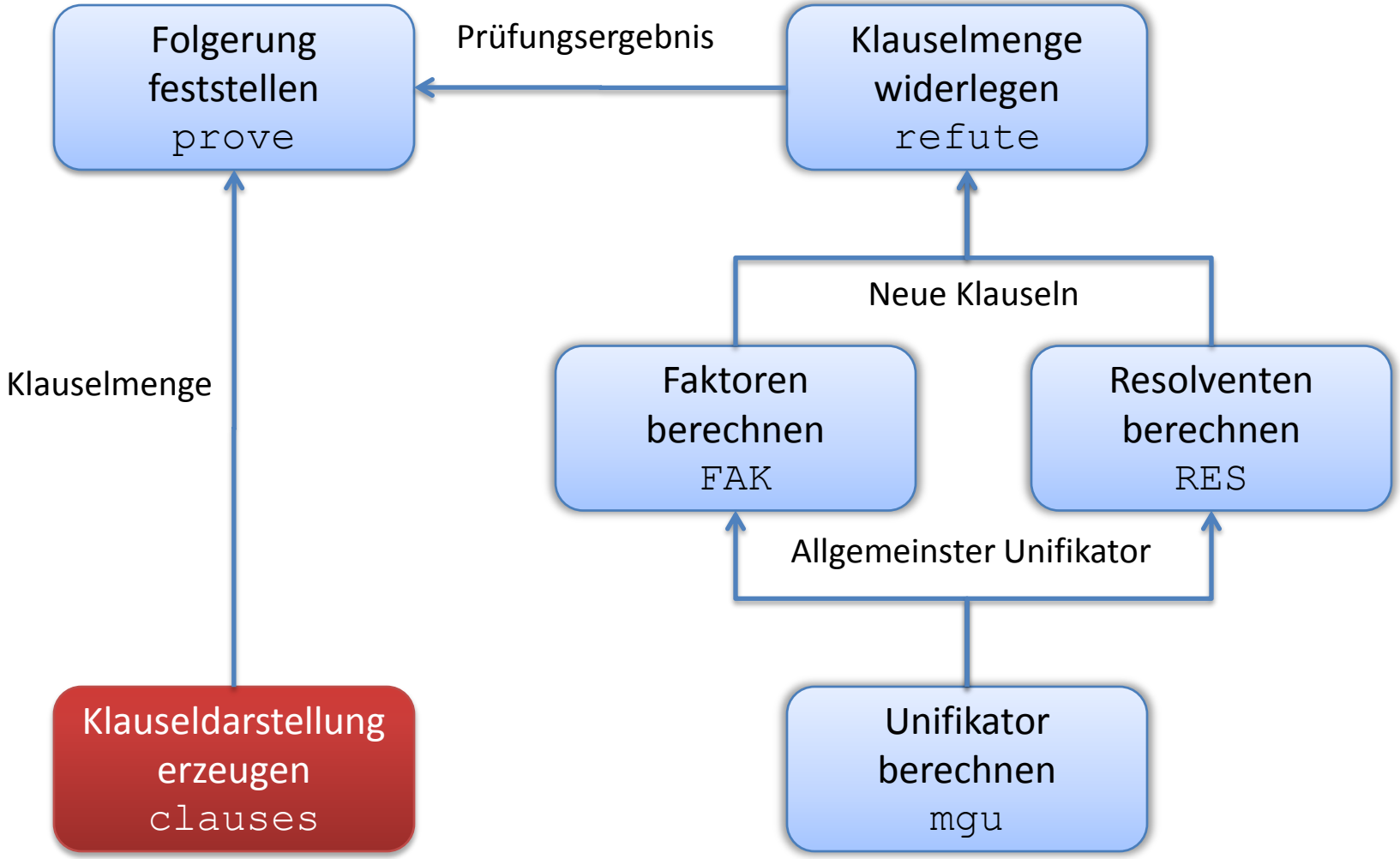
$\mathcal{S}_{\mathcal{T}} :=$ `clauses` $(\{\neg\varphi\})$

$r :=$ `refute` $(\mathcal{S}_{\mathcal{A}} \cup \mathcal{S}_{\mathcal{T}})$

return (r)

end

Umformung von Formeln in Klauseln durch `clauses`



Umformung von Formeln in Klauseln durch `clauses`

Anforderungen und Funktionalität

- Umformung prädikatenlogischer Formeln in Klauseln
- Grundlage für Nutzbarkeit des Resolutionskalküls durch `prove`
- Keine äquivalente Darstellung möglich
- Widerspruchsvollständigkeit bleibt erhalten

Umformung von Formeln in Klauseln durch `clauses`

Funktionsweise

1. Umformung in prenexe Normalform mit Matrix in konjunktiver Normalform durch logische Umformungen:

   z.B. de Morgan: $\neg[\varphi_1 \wedge \varphi_2] \sim [\neg\varphi_1 \vee \neg\varphi_2]$

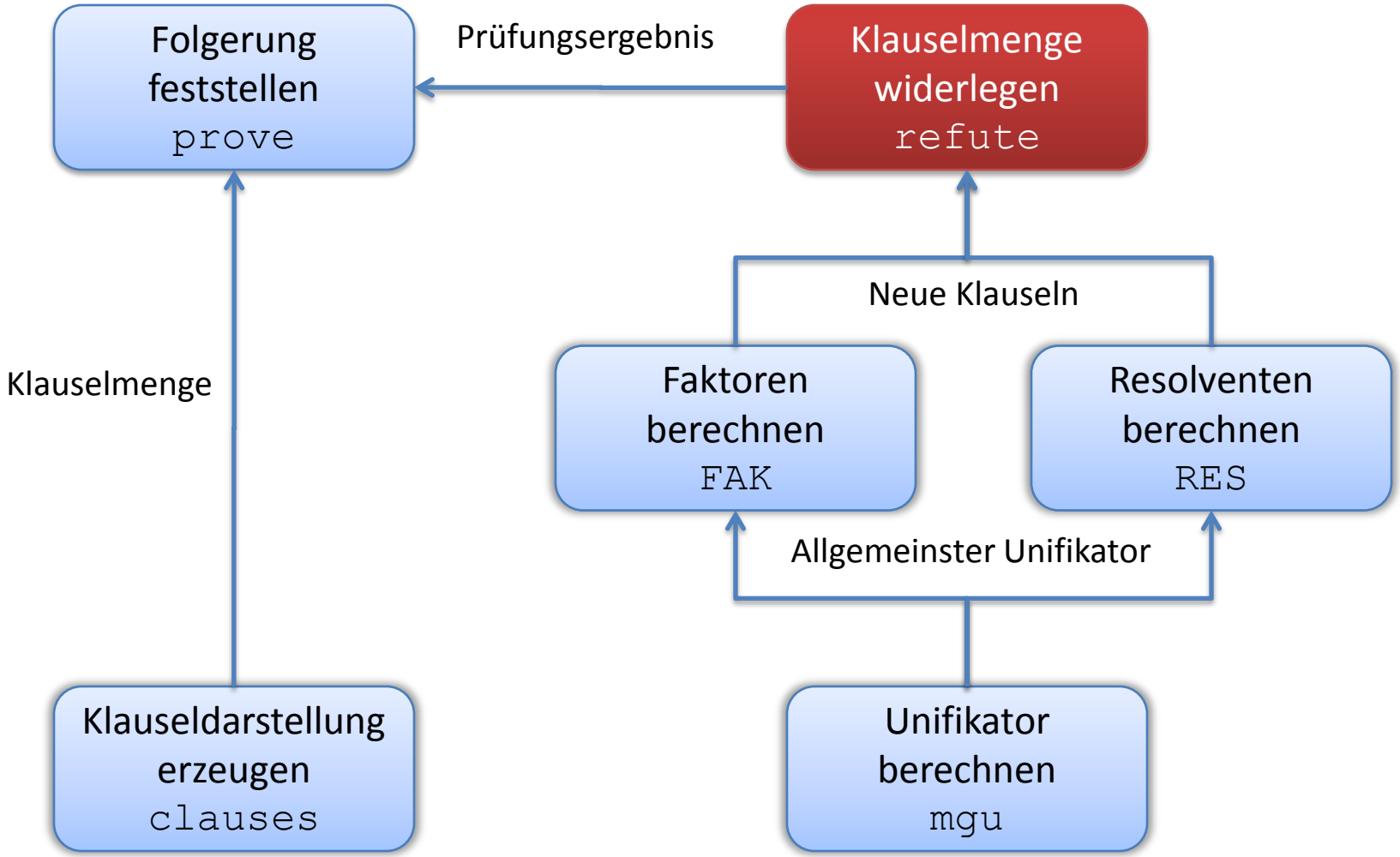
2. Eliminierung von Existenzquantoren durch Skolemisierung

   z.B. $\forall x \exists y[P(x\ y)]$ wird zu $\forall x[P(x\ f(x))]$

3. Umformung in Klauselschreibweise (Mengendarstellung)

   z.B. $\forall x,y[P(x) \vee Q(x)]$ wird zu $\{P(x), Q(x)\}$

Der Widerlegungsalgorithmus `refute`



Der Widerlegungsalgorithmus `refute`

Anforderungen und Funktionalität

- `prove` als Überbau und Schnittstelle (koordinierende Instanz)
- `refute` als Systemkern (austauschbar)
- Implementation des Resolutionskalküls
- Beweis durch Widerlegung anstelle einer direkten Herleitung

Der Widerlegungsalgorithmus `refute`

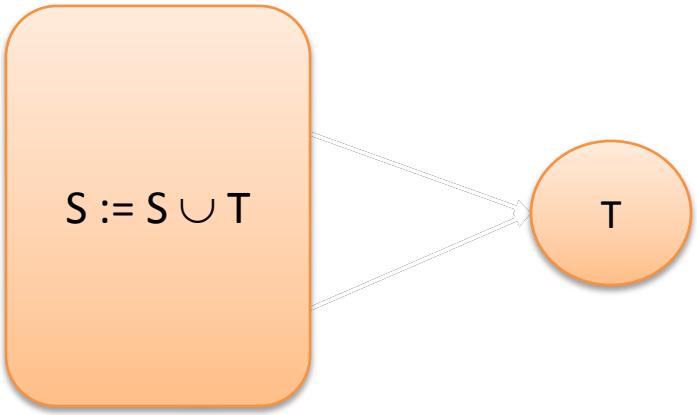
Funktionsweise

- Datenstrukturen:
  - Akkumulator S für sämtliche Klauseln der bisherigen Herleitung
  - Behälter T für Klauseln der letzten Generation
- Vorgehen:
  - Zuletzt erzeugte Klauseln (T) mit bisherigen Klauseln resolvieren
  - S um T ergänzen, neu resolvierte Klauseln werden neues T
  - Stopp, wenn leere Klausel hergeleitet wurde oder keine Resolutionsmöglichkeiten mehr bestehen
  - Entspricht einem Breitensuchverfahren

Der Widerlegungsalgorithmus `refute`

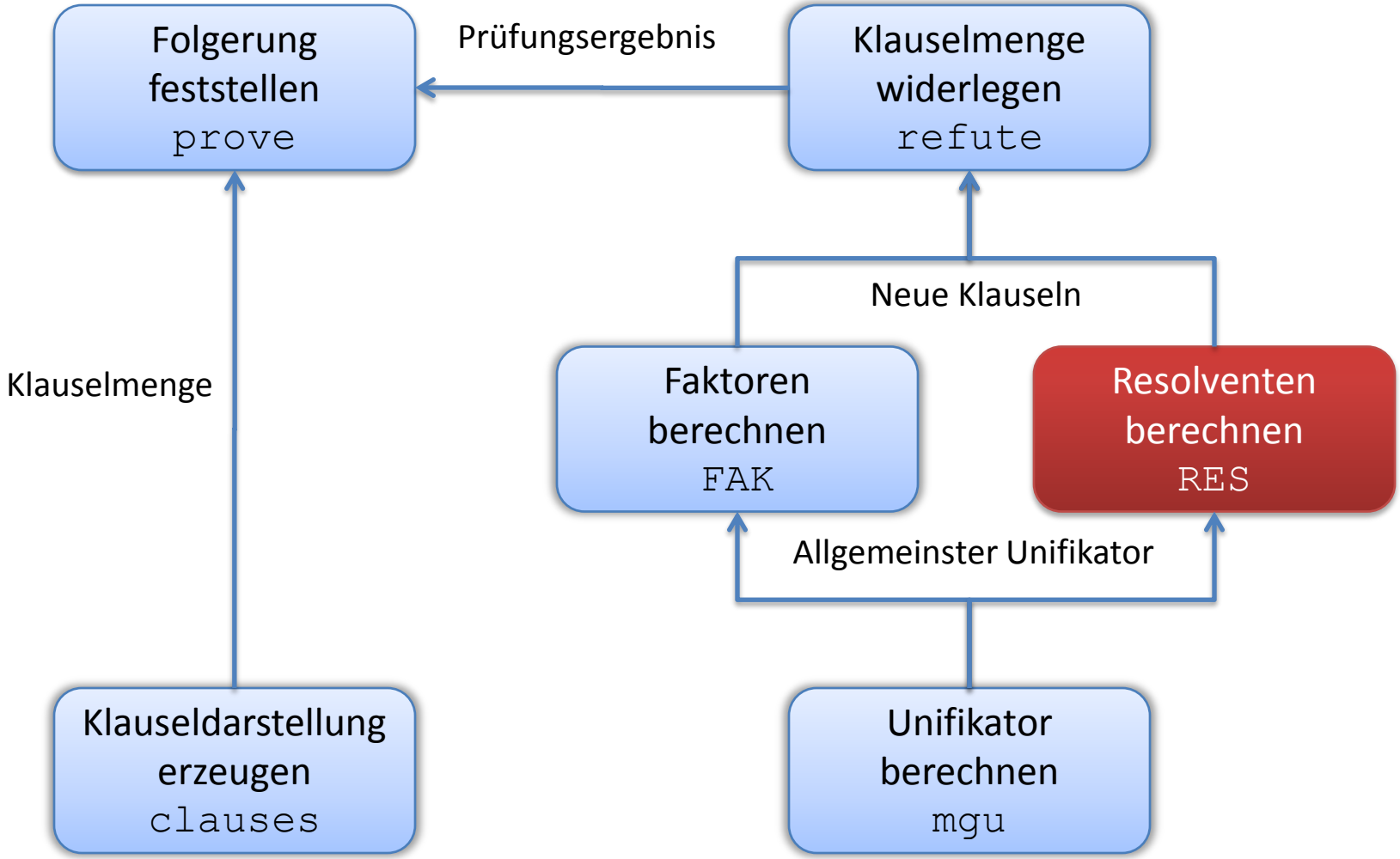
Pseudocode*

```
function refute(S ∈ 2^L) :bool
    T := S
    while T ≠ ∅ and □ ∉ T do
        T := RES(S, T)
        if □ ∉ T then
            S := S ∪ T
        fi
    done
    return not(T = ∅)
end
```



** Nicht widerlegungsvollständig*

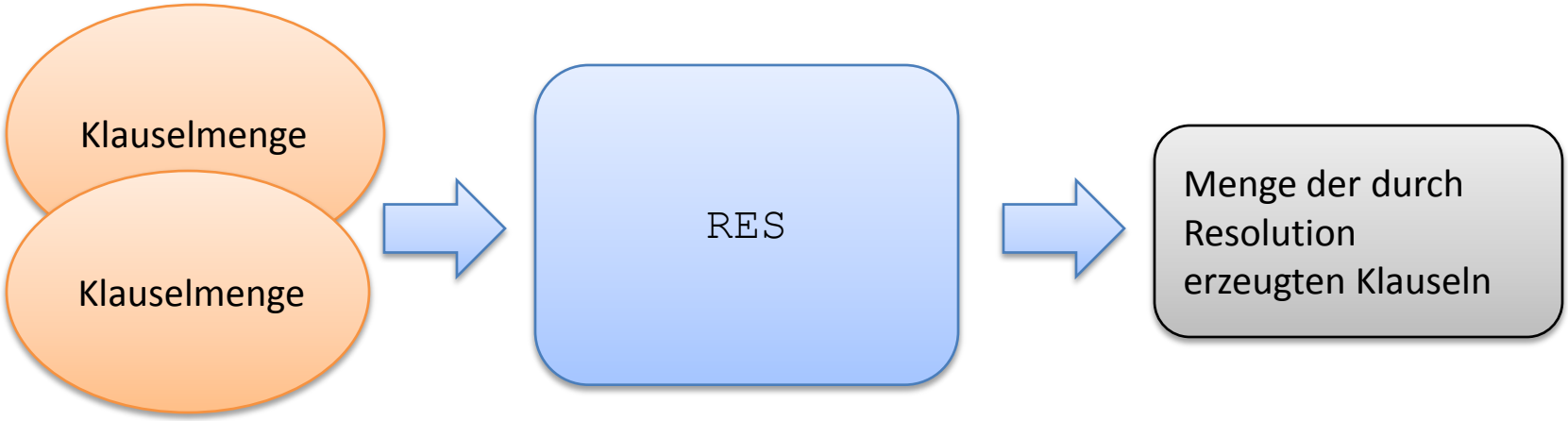
Berechnung von Resolventen durch RES



Berechnung von Resolventen durch RES

Anforderungen und Funktionalität

- Berechnung aller Resolventen, die zwischen den Literalen zweier Klauselmengen möglich sind
- Stopp, falls □ hergeleitet



Berechnung von Resolventen durch RES

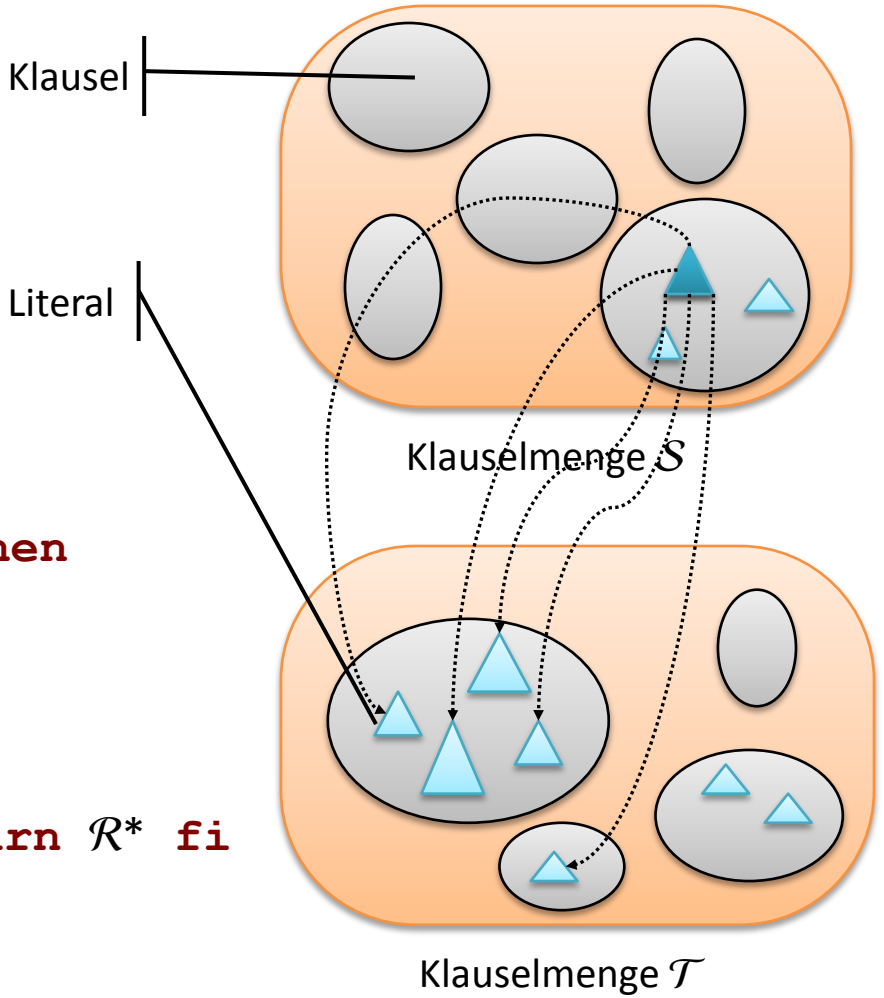
Funktionsweise

- Jedes Literal einer Klauselmenge mit jedem Literal der anderen Klauselmenge auf Resolvierbarkeit testen (paarweiser Test)
- Voraussetzungen für Resolvierbarkeit
  - Literale komplementär
  - Jeweils zugehörige Atome unifizierbar
- Gegebenenfalls Resolventenbildung

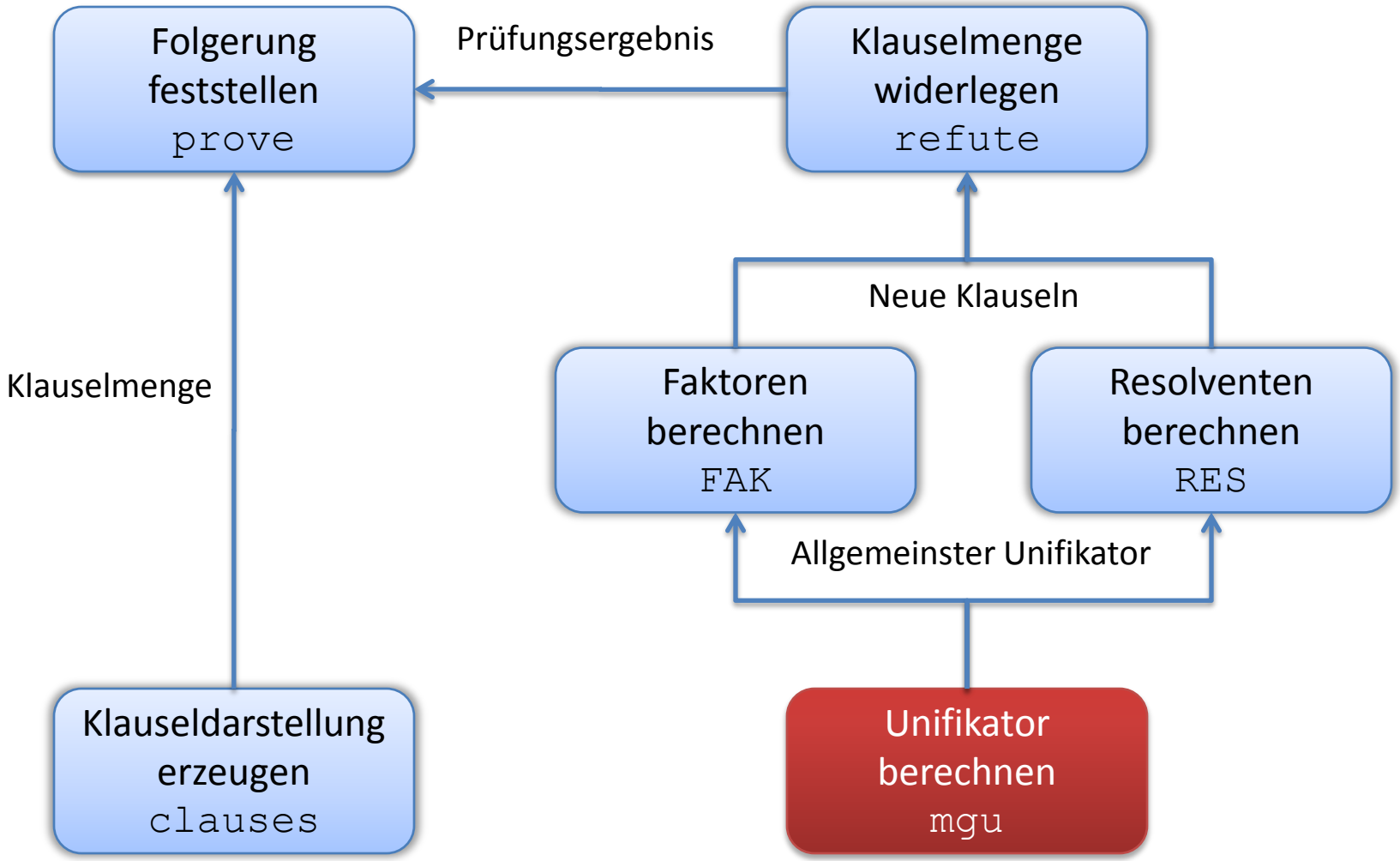
Berechnung von Resolventen durch RES

Pseudocode

```
function RES (S, T ∈ 2^L) : 2^L
  R* := ∅
  T' := T
  for all C ∈ S do
    T' := T \ {C}
    for all L ∈ C do
      for all D ∈ T' do
        for all K ∈ D do
          if L komplementär K then
            σ := mgu (|L|, |K|)
            if σ ≠ failed then
              R := σ(C−L) ∪ σ(D−K)
              R* := R* ∪ {R}
              if R = □ then return R* fi
  fi fi done done done done
  return R*
end
```



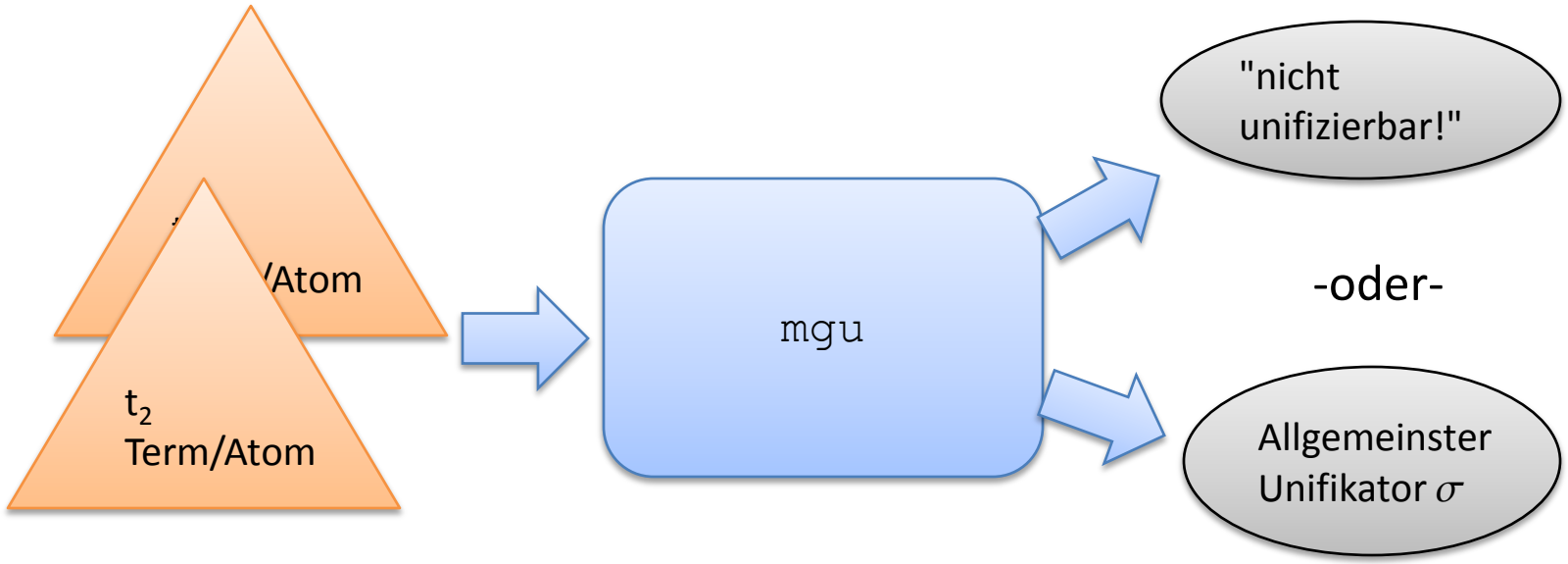
Berechnung eines allgemeinsten Unifikators durch `mgu`



Berechnung eines allgemeinsten Unifikators durch `mgu`

Zwei Aufgaben

- Test auf Unifizierbarkeit
- Ggf. Berechnung eines allgemeinsten Unifikators



Berechnung eines allgemeinsten Unifikators durch `mgu`

Gründe für Nicht-Unifizierbarkeit

1. Occur Failure: $t_1=x \;\; t_2=..f(...x...)..$
   "egal wie x ersetzt wird, Funktionssymbol bleibt immer erhalten"

   z.B. $t_1=x \;\; t_2=f(x)$

2. Clash Failure: $t_1=g(q_1 \ldots q_n) \;\; t_2=h(r_1 \ldots r_m)$ mit $g \neq h$
   "aus g wird nie h, egal wie ersetzt wird"

   z.B. $t_1=f(x) \;\; t_2=g(x)$

Berechnung eines allgemeinsten Unifikators durch `mgu`

Berechnung des Unifikators

▪ Terme gleich	Fertig (ϵ als Unifikator)
▪ t_1 oder t_2 Variable • Enthalten in zweitem Term • Ersetzen durch zweiten Term	Fertig (z.B. $t_1=x$, $t_2=f(y)$)
▪ Geschachtelte Ausdrücke • Verschieden • Gleich	$t_1 = G(q_1 \dots q_n)$, $t_2 = H(r_1 \dots r_m)$ *Clash Failure* sequentiell Unifikatoren für Parameter suchen und Lösung aus diesen zusammensetzen (s. Bsp.)

Berechnung eines allgemeinsten Unifikators durch mgu

Vollständiger Algorithmus (1)

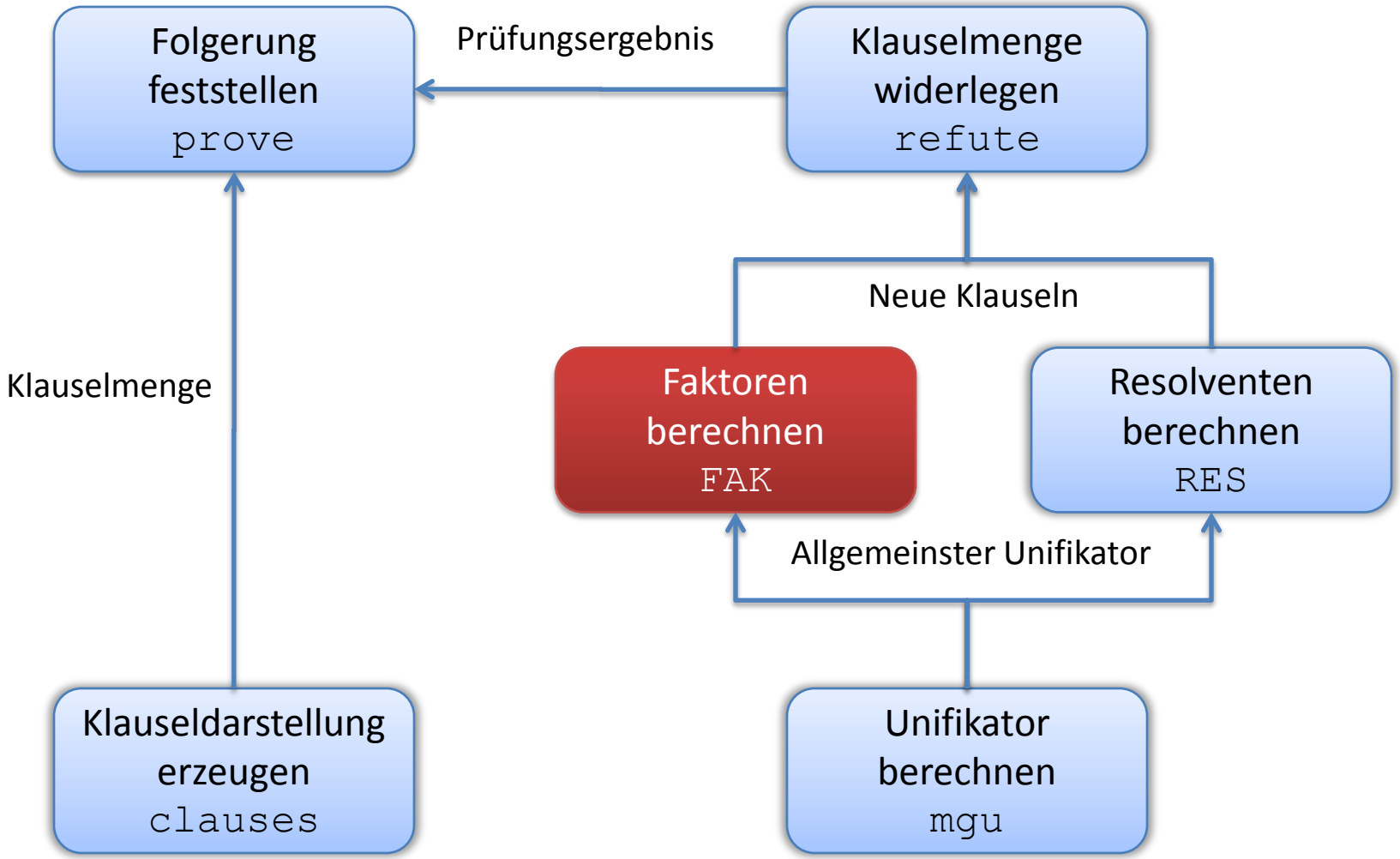
```
function mgu(t1, t2 ∈ T ∪ AT) : SUB ∪ {failed}
  if t1 = t2 then return {} fi          // 1. Fall
  if t1 ist Variable then               // 2. Fall
    if t1 ist in Prädikat von t2 then
      return failed                     // "occur failure"
    else
      return {t1 / t2}
    fi
  fi
  if t2 ist Variable then return mgu(t2, t1) fi
  ...
```

Berechnung eines allgemeinsten Unifikators durch `mgu`

Vollständiger Algorithmus (2)

```
...
 assume t1 = G(q1 ... qn), t2 = H(r1 ... rm)      // 3. Fall
  if G ≠ H then return failed fi   // "clash failure"
  assume t1 = f(q1 ... qn), t2 = f(r1 ... rn), n > 0
  σ := ε
for i := 1 to n do
      θ := mgu (σ(qi), σ(ri))
      assume σ = {x1/s1, ... , xn/sn}
 σ := θ ∪ {x1/θ(s1), ... , xn/θ(sn)}
  done
return σ
end
```

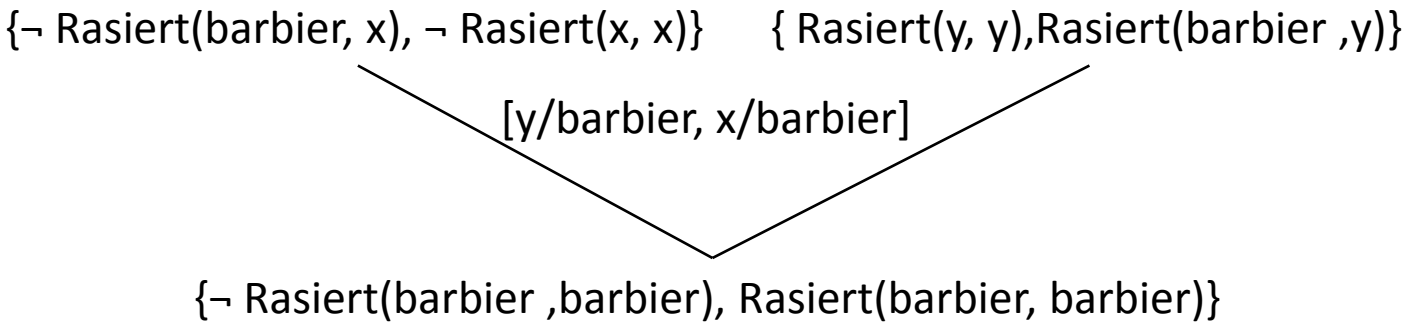
Berechnung von Faktoren durch `FAK`



Berechnung von Faktoren durch `FAK`

Motivation des FAK-Algorithmus

- „Der Barbier rasiert eine Person genau dann, wenn sie sich nicht selbst rasiert." (Russellsche Antinomie)
- $\forall$ x [Rasiert(barbier, x) $\Leftrightarrow$ ¬Rasiert(x , x)]
- Enthält Widerspruch, der sich aber per Resolution nicht herleiten lässt!



Quelle: http://www.informatik.uni-ulm.de/ki/Edu/Vorlesungen/GdKI/WS0203/r72.pdf

Berechnung von Faktoren durch FAK

Der neue Widerlegungsalgorithmus refute

```
function refute(S ∈ 2^L) :bool
   T := FAK(S)
   S := T
   while T ≠ ∅ and □ ∉ T do
        T := RES(S, T)
        if □ ∉ T then
                T := FAK(T)
                S := S ∪ T
        fi
   done
   return not(T = ∅)
end
```

Berechnung von Faktoren durch `FAK`

Anforderungen und Funktionalität

- Berechnung aller möglichen Faktoren einer Klauselmenge
- Input S Teil des Outputs (auch Faktorisierung über ϵ möglich)

Berechnung von Faktoren durch FAK

Idee

- Teilalgorithmus `fak` für eine Klausel
- Iteration über alle Elemente der Input-Menge

Pseudocode

```
function FAK(S ∈ 2^L) : 2^L
   F := ∅
   for all C ∈ S do
      F := F ∪ fak(C)
   done
   return F
end
```

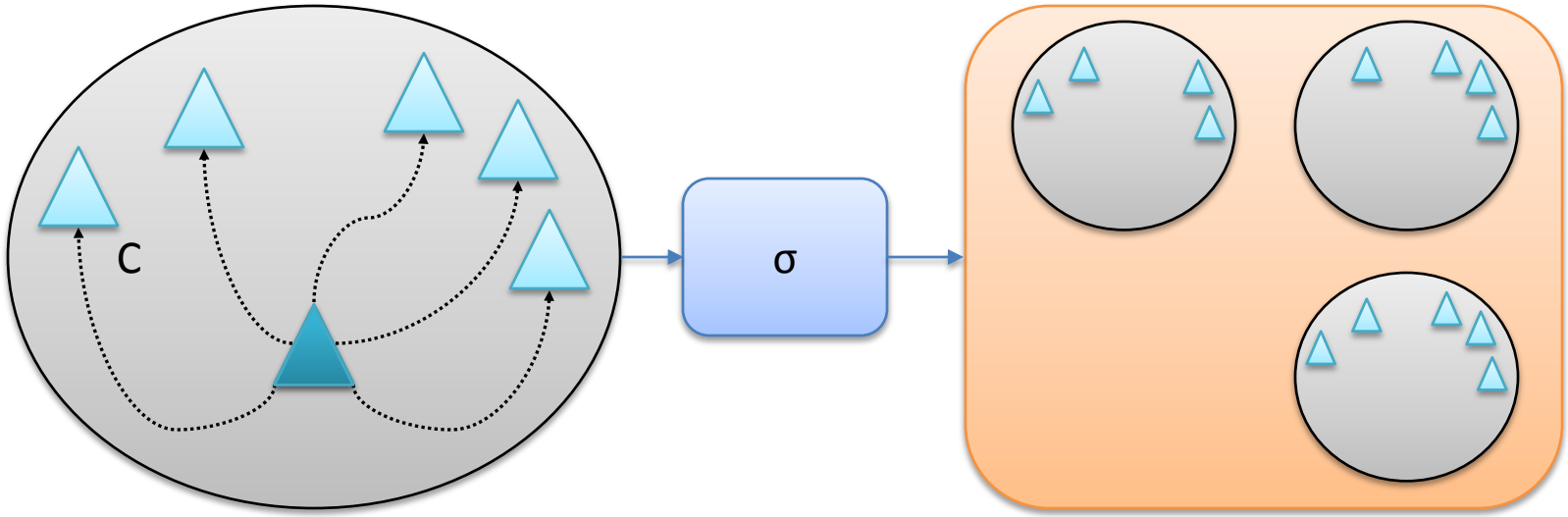
S = {¬ Rasiert(barbier,x), ¬ Rasiert(x,x)},
{ Rasiert(y, y),Rasiert(barbier, y)}

F = {¬ Rasiert(barbier, x), ¬ Rasiert(x, x)},
{ Rasiert(y, y), Rasiert(barbier, y)},
{¬ Rasiert(barbier, barbier)},
{Rasiert(barbier, barbier)}

Berechnung von Faktoren durch `FAK`

Teilalgorithmus fak

- Innerhalb einer Klausel wird jedes Literal mit jedem auf Unifizierbarkeit überprüft
- Anwendung des gefunden Unifikators auf gesamte Klausel (=Faktorisierung)
- Sammlung der Teilergebnisse



Berechnung von Faktoren durch FAK

Teilalgorithmus fak – Pseudocode

```
function fak(C ∈ L) : 2^L
  S := ∅
  D := C
  for all L ∈ C do
    D := D − L
    for all K ∈ D do
      if not L komplementär K then
        σ := mgu(|L|, |K|)
        if σ ≠ failed then
          S := S ∪ fak(σ(C) − L))
        fi
      fi
    done
  done
  return S ∪ {C}
end
```

C = {¬ Rasiert(barbier, x), ¬ Rasiert(x, x)}

{¬ Rasiert(barbier, x), ¬ Rasiert(x, x)},
{¬ Rasiert(barbier, barbier)}

Agenda

Einführung

Überblick über das System

Die Algorithmen im Detail

→ **Optimierungsansätze**

Zusammenfassung

Probleme in der bisherigen Realisierung

- Komplexitätsklasse, wenn der Algorithmus terminiert: NPV

- Grundsystem zu ineffizient für Praxis, z.B.
  - exponentieller Klauselzuwachs in refute
  - Aufwändige (O(n*m)) Resolventenbildung in res

- Optimierungsmöglichkeiten insbesondere für
  - `prove`
  - `refute`
  - `RES`

Splitting – Verbesserung von `prove`

Grundgedanke

- Für $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_n$: φ folgt aus Φ gdw. alle φ_i aus Φ folgen

→ Prüfung für alle φ_i , wobei tendenziell jeweils weniger Klauseln erzeugt werden

function `prove` $(\Phi \in 2^{\mathcal{F}}, \varphi \in \mathcal{F})$ **:bool**

$\mathcal{S}_{\mathcal{A}} :=$ `clauses` (Φ)

assume $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_n$

$i := 0$

repeat

$i := i + 1$

$\mathcal{S}_{\mathcal{T}} :=$ `clauses` $(\{\neg \varphi_i\})$

$r :=$ `refute` $(\mathcal{S}_{\mathcal{A}} \cup \mathcal{S}_{\mathcal{T}})$

until not r **or** $i = k$

return r

end

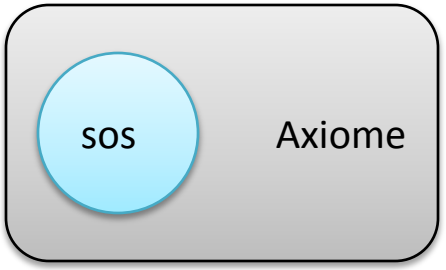
Löschregeln und Ableitungsstrategien - `refute`

- Reduktion der durch refute erzeugten Klauseln durch Verbotskriterien zur Herleitung neuer Klauseln
- Löschregeln: unabhängig von Entstehungsgeschichte der Klausel

  Bsp.: **Tautologieklauseln** ($\{A, \neg A\} \subset C$)
- Ableitungsstrategien: abhängig von Entstehungsgeschichte der Klausel

  Bsp.: **set-of-support**

  Prämisse: Axiomensystem erfüllbar
  Zwischen Axiomen keinen Widerspruch suchen



Zu widerlegende Klauselmenge

Optimierungsmöglichkeiten für RES

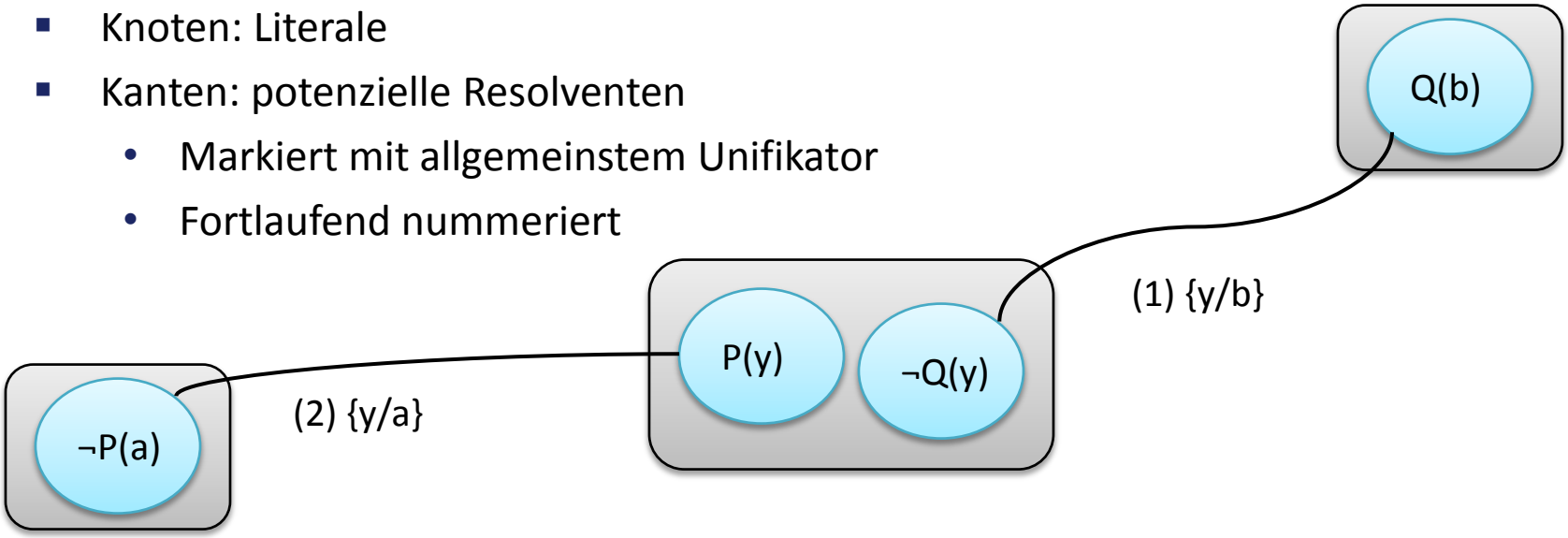
Grundgedanke

- Aufwand von RES=O(n*m) *"jeder mit jedem"*
- Häufig nur Bruchteil der getesteten Verbindungen resolvierbar
- Problem: Ineffizienz
- Idee: nur bestimmte Verbindungen testen
- Realisierung: **Klauselgraphen**

Aufbau von Klauselgraphen

Aufbau

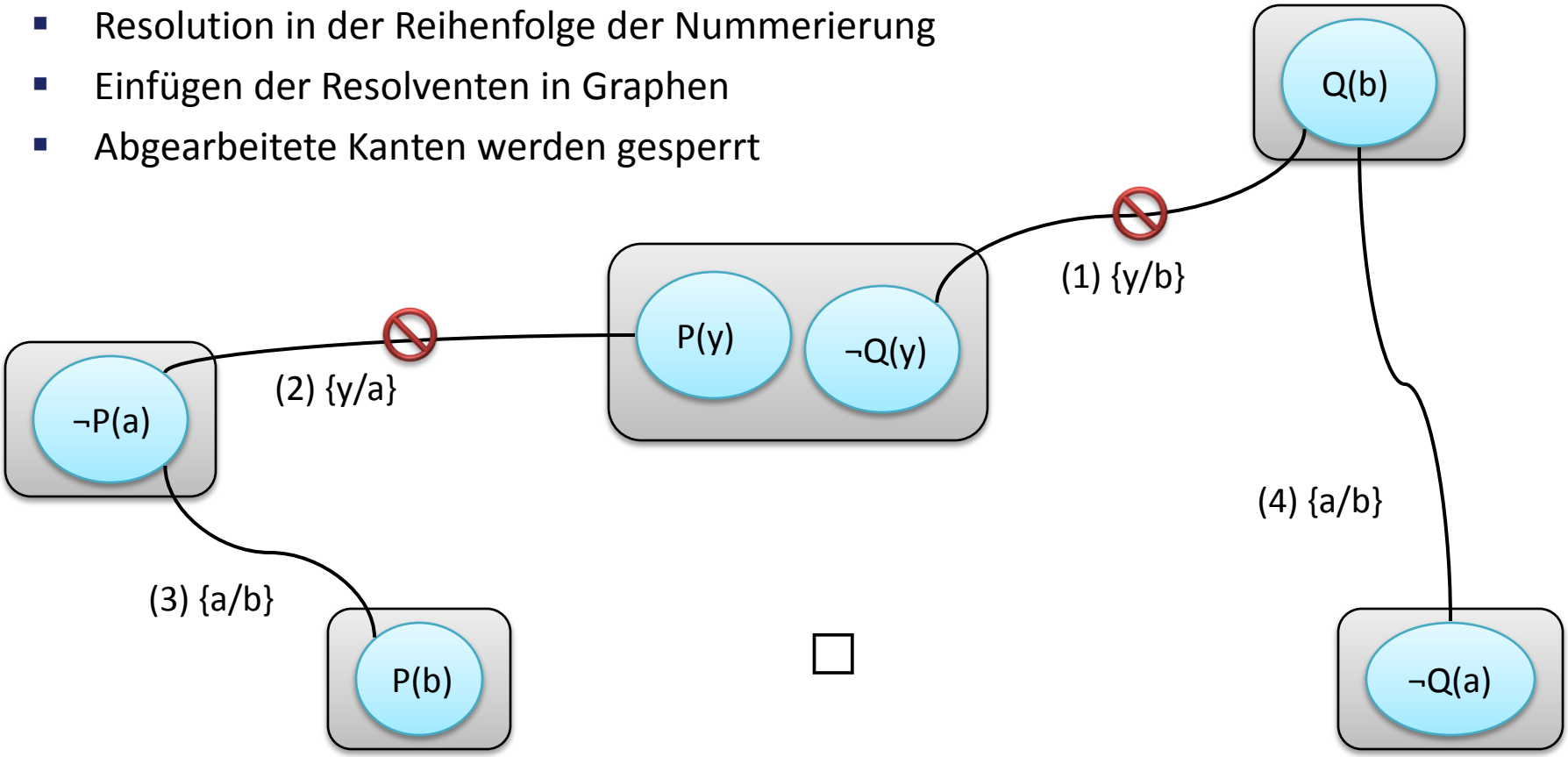
- Knoten: Literale
- Kanten: potenzielle Resolventen
  - Markiert mit allgemeinstem Unifikator
  - Fortlaufend nummeriert



Verwendung von Klauselgraphen

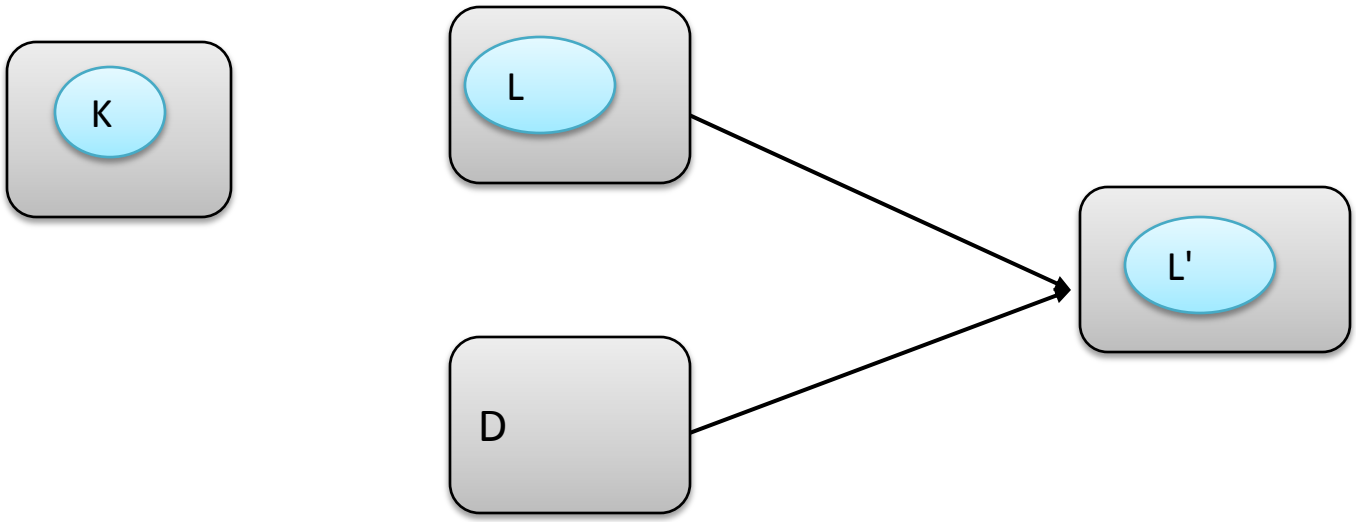
Verfahren

- Resolution in der Reihenfolge der Nummerierung
- Einfügen der Resolventen in Graphen
- Abgearbeitete Kanten werden gesperrt



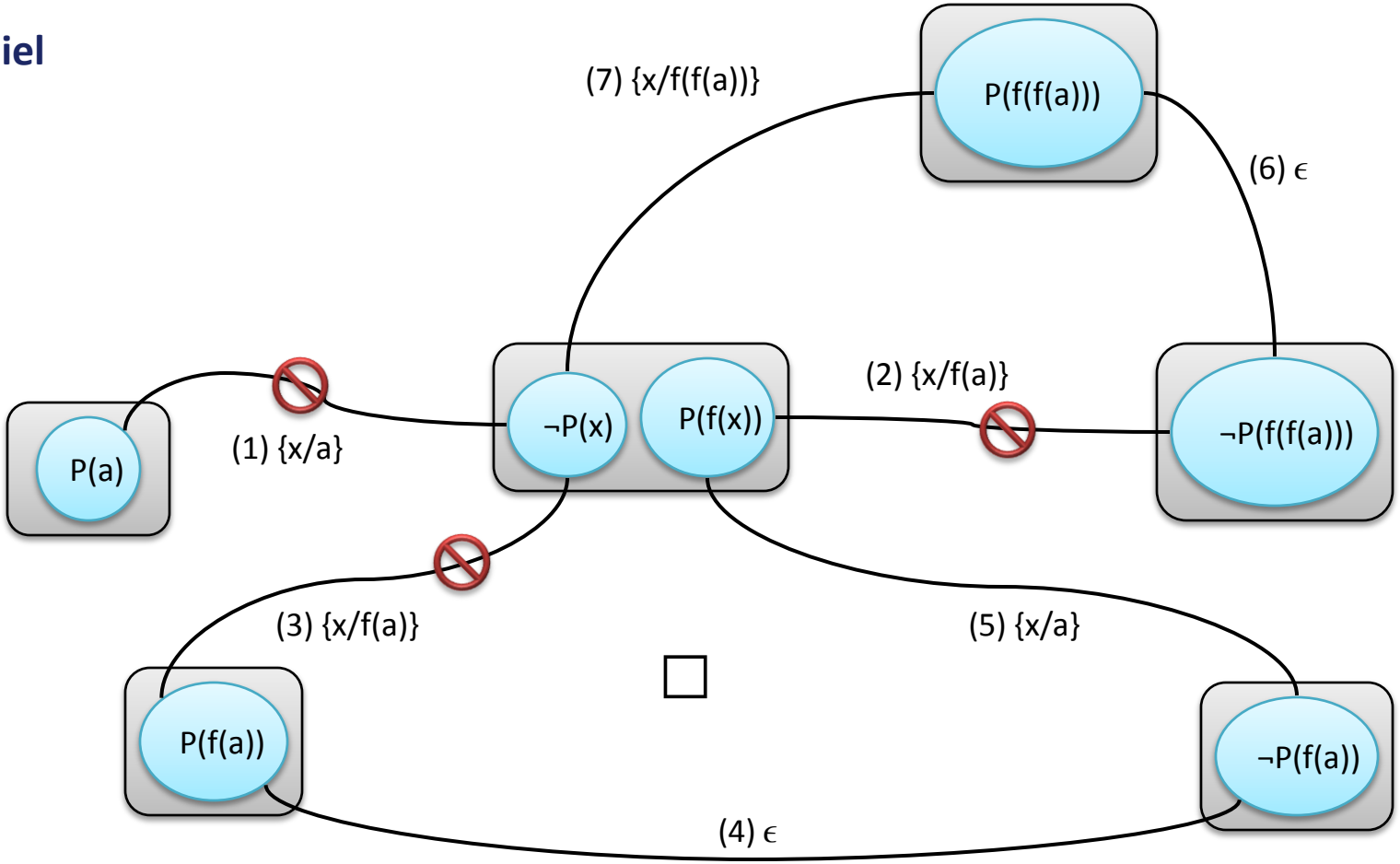
Optimierung durch Klauselgraphen

- Einschränkung des Suchraums auf
  1. Partner der Elternliterale
  2. Literale der Elternklauseln



Optimierung durch Klauselgraphen

Beispiel



Agenda

Einführung

Überblick über das System

Die Algorithmen im Detail

Optimierungsansätze

→ **Zusammenfassung**

Zusammenfassung

Beispielimplementierung

- Implementierung eines Systems aus sechs Algorithmen
- Verwendung des Resolutionskalküls mit
  - Resolution und Faktorisierung als Schlussregeln
  - Unifikation

Ausblick

- Weitere Optimierung möglich, z.B. durch
  - Klauselgraphen
  - Splitting
  - Lösch-/Ableitungsstrategien
- Aber:
  - Problem bleibt semi-entscheidbar und exponentiell
  - Optimierungen für Spezialfälle schwer/unmöglich

Quellen

- Dittmann, Daniel: Implementierung eines Resolutionsbeweisers, Seminararbeit an der FH Wedel, SS 2005
- Walther, Christoph: Automatisches Beweisen, in: Görz, Günter et al. (Hrsg.): Handbuch der künstlichen Intelligenz, 4., korrigierte Auflage, München, Wien: Oldenbourg, 2003
- Russel, Stuart; Norvig, Peter: Künstliche Intelligenz - Ein moderner Ansatz, 2. Auflage, München : Addison-Wesley, 2004
- Chiou, Charles: First-Order Logic Resolution Theorem Prover In Haskell, http://www.cs.yale.edu/homes/cc392/report.html, 2001 – Abruf am 19. November 2007
- Von Henke, Prof. Dr. F. W.: Einführung in die Künstliche Intelligenz - Vorlesung an der Universität Ulm, WS 2002/2003, http://www.informatik.uni-ulm.de/ki/Edu/Vorlesungen/GdKI/WS0203/r72.pdf - Abruf am 09. Dezember 2007

Fragen?

Vielen Dank für die Aufmerksamkeit.