Künstliche Intelligenz

Sebastian Iwanowski
FH Wedel

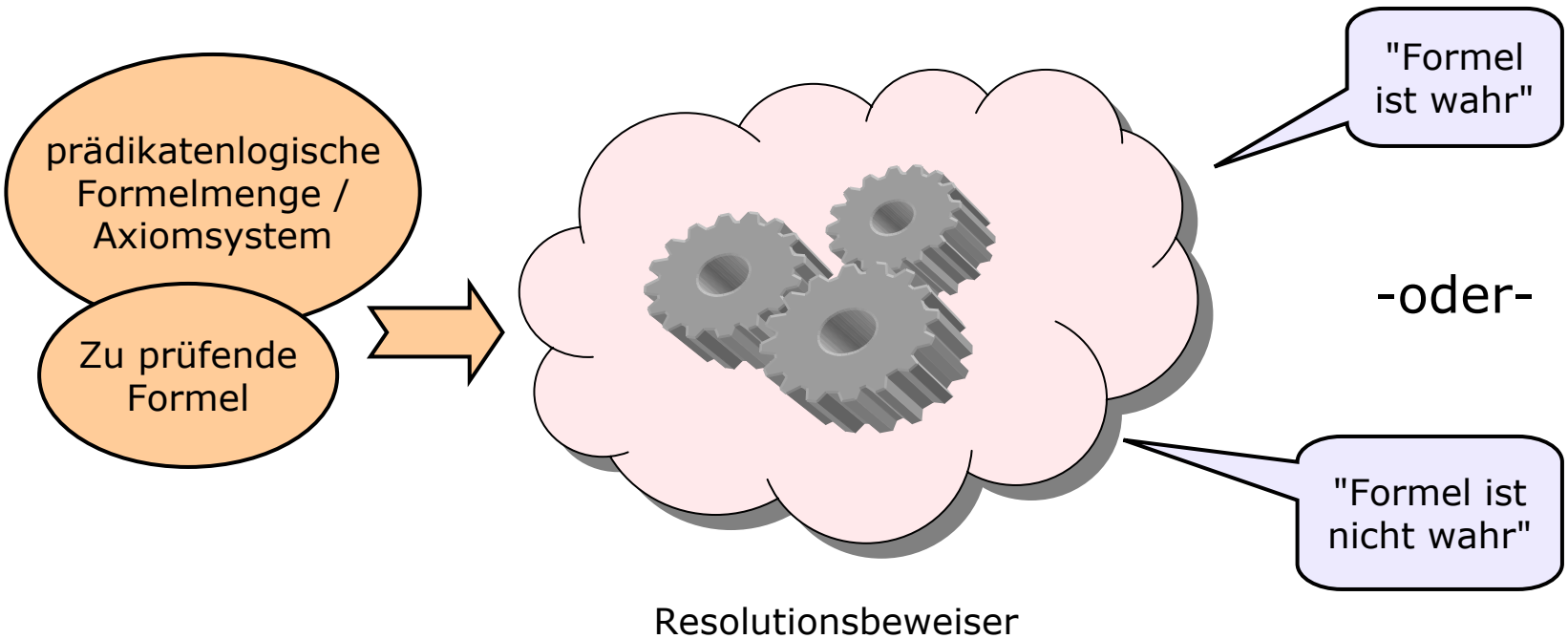
Kap. 2:
KI-Logik

2.3: Funktionsweise eines Resolutionsbeweisers

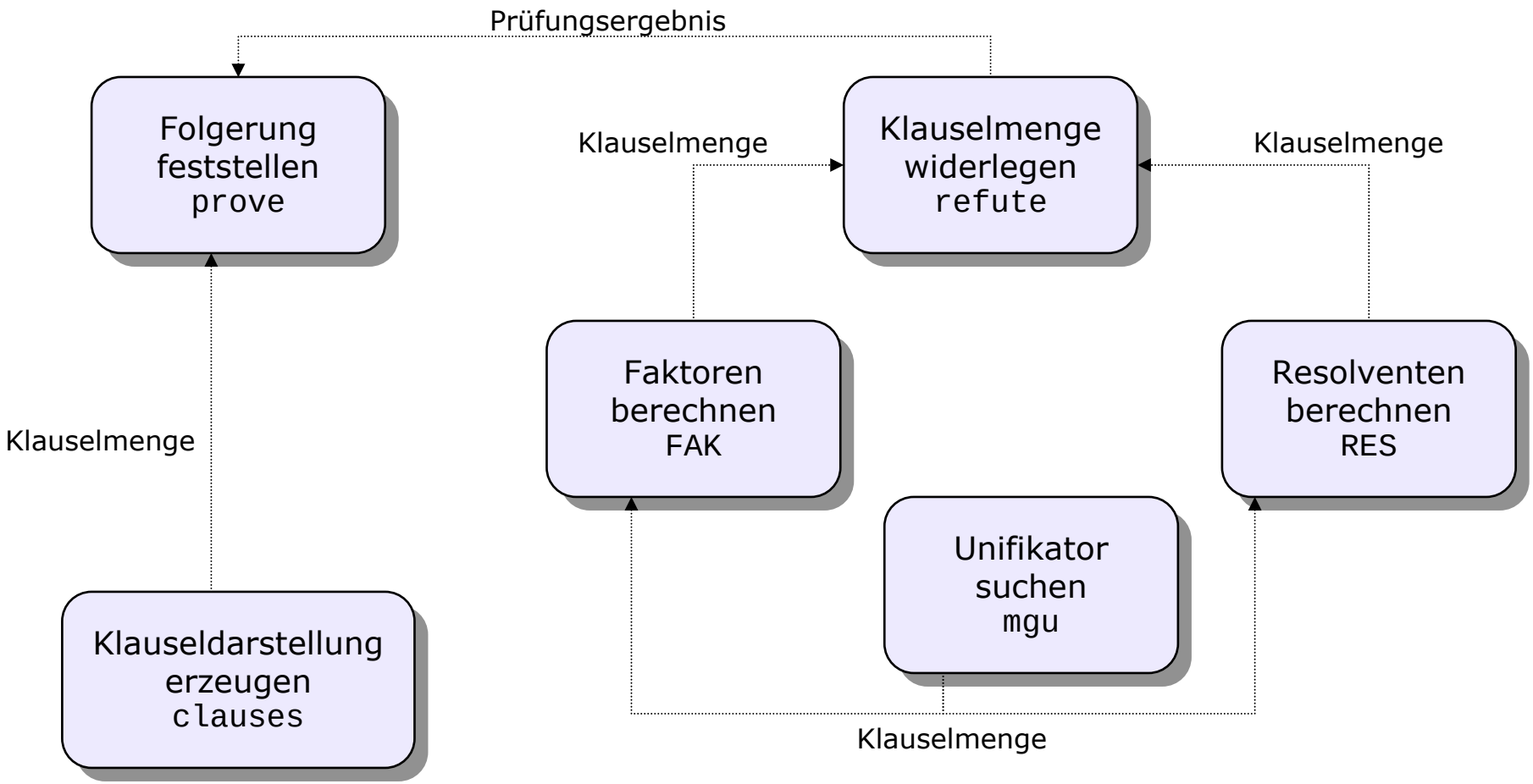
Die Folien dieser Vorlesung beruhen im Wesentlichen auf Folien des Seminarvortrags von Daniel Dittmann, gehalten im SS 2005.

Ziel des Resolutionsbeweisers

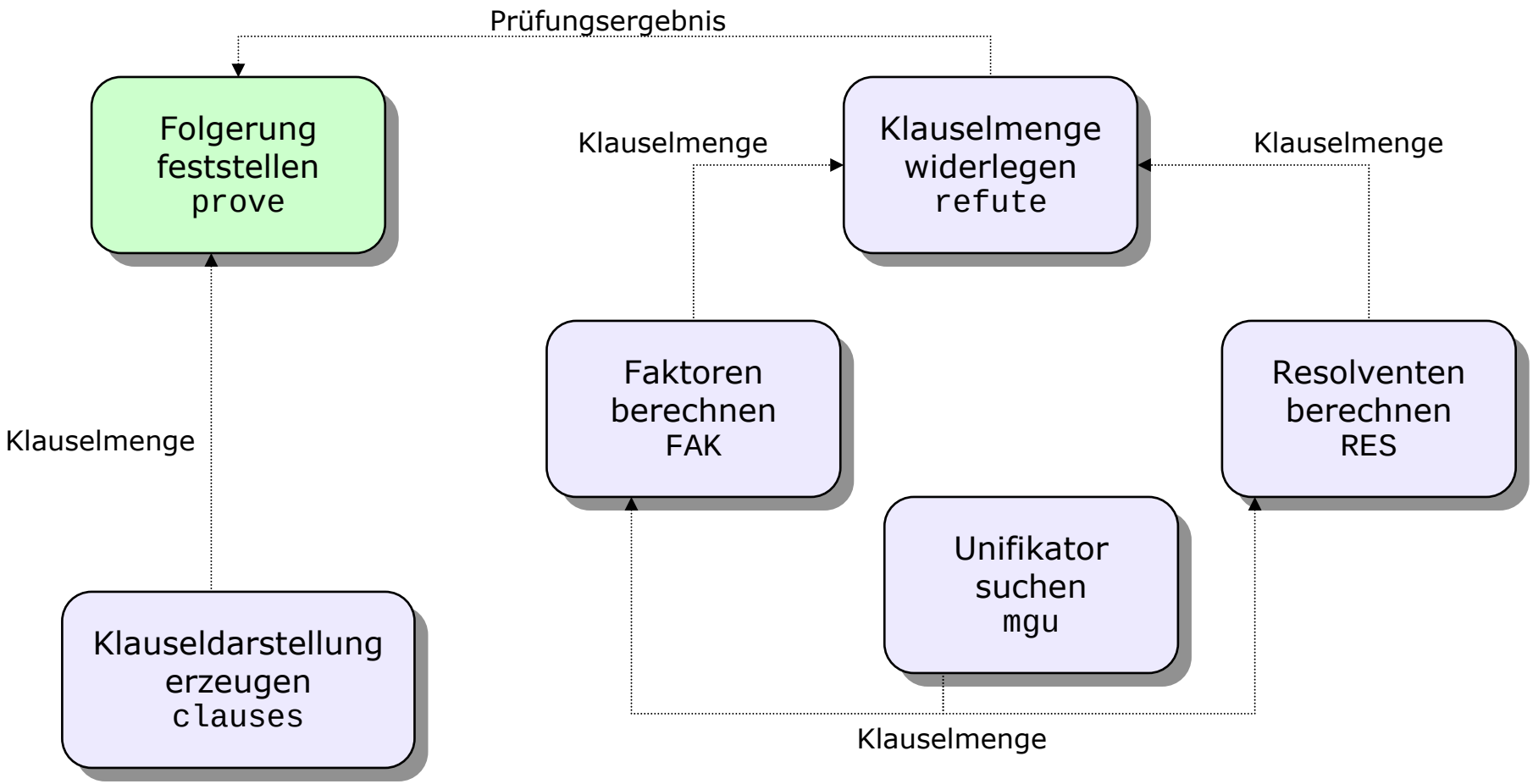
- Entwicklung eines Softwaresystems zur automatischen Beweisführung mit Hilfe des Resolutionskalküls



Modularisierung und Datenflüsse im Resolutionsbeweiser



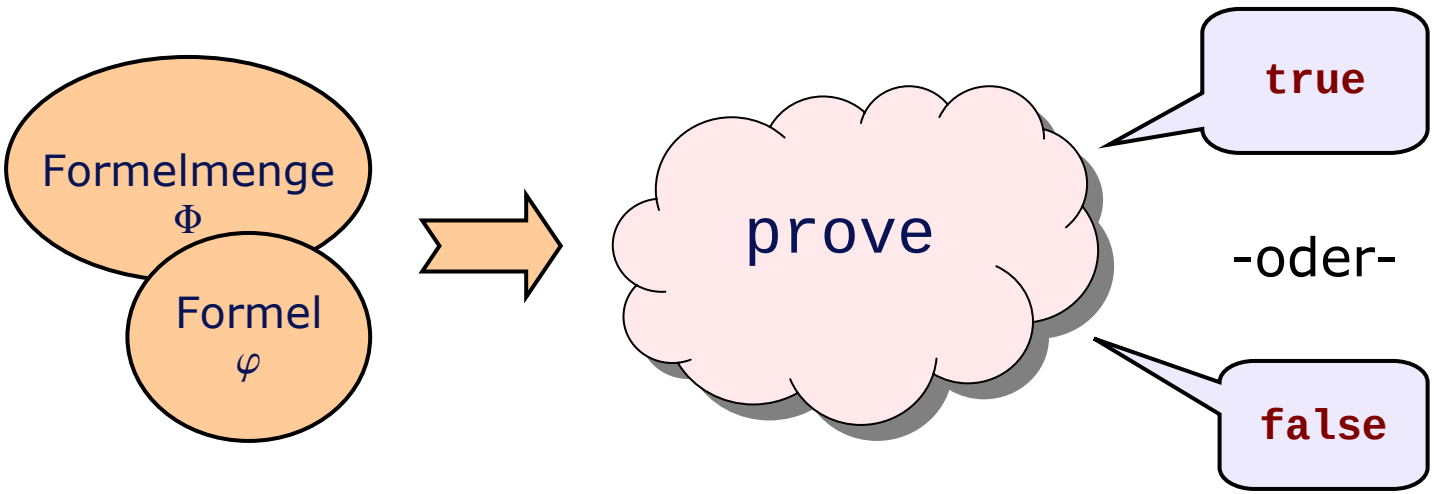
Modularisierung und Datenflüsse im Resolutionsbeweiser



prove: Anforderungen und Funktionalität

- Schnittstelle des Systems

- Darstellung der Inputgrößen in Prädikatenlogik 1. Stufe (Nutzung des Resolutionskalküls transparent)



- Aufgabe:
  Suche die Belegungen für die freien Variablen, unter denen die Folgerung true ist.

prove: Funktionsweise

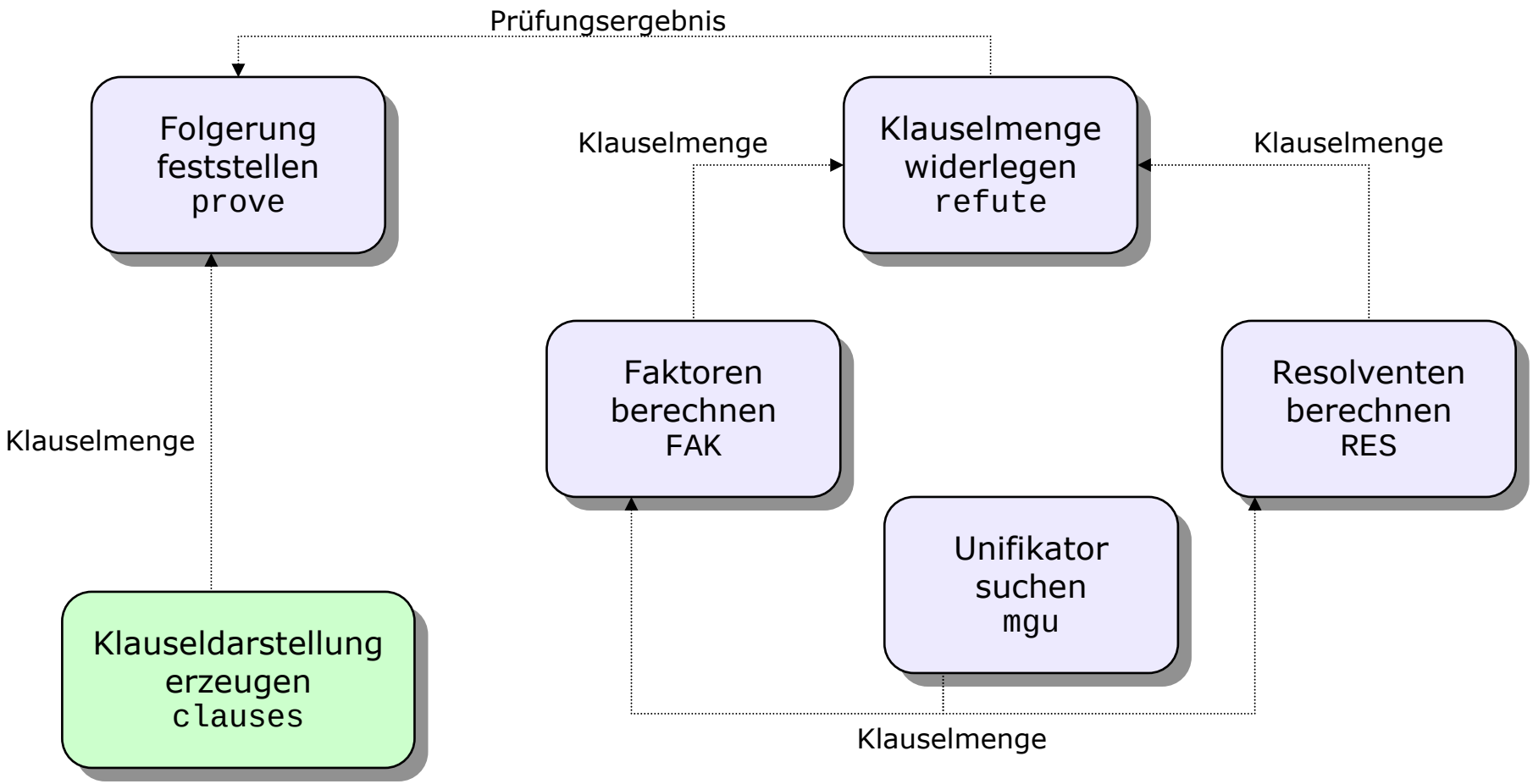
1. Umformung der Inputs in Klauseldarstellung

2. Negation der zu überprüfenden Formel mit Axiomsystem vereinigen

3. Versuch, neu gebildete Klauselmenge zu widerlegen

```
function prove(Φ ∈ 2^F, φ ∈ F):bool
   S_A := clauses(Φ)
   S_T := clauses({¬φ})
   r := refute(S_A ∪ S_T)
   return(r)
end
```

Modularisierung und Datenflüsse im Resolutionsbeweiser



clauses: Anforderungen und Funktionalität

- Umformung prädikatenlogischer Formeln in Klauseln
- Grundlage für Nutzbarkeit des Resolutionskalküls durch `prove`
- Äquivalente Darstellung nicht möglich, daher Beschränkung auf (Un)Erfüllbarkeit

φ φ^*

Satz:

φ ist erfüllbar $\Leftrightarrow$ φ^* ist erfüllbar
φ ist unerfüllbar $\Leftrightarrow$ φ^* ist unerfüllbar

clauses: Funktionsweise

Normierung beliebiger Formeln durch Quantorenelimierung

Definition: Eine Formel φ ist in Pränexer Normalform (PNF):
$\varphi = \Delta\, x_1\, \Delta\, x_2 \ldots \Delta\, x_k\, \psi$ mit Δ ist Quantor und ψ ist quantorenfreie Formel.

Definition: Sei $\varphi = \forall\, x_1\, \forall\, x_2 \ldots \forall\, x_k\, \exists\, y\, \psi$ eine prädikatenlogische Formel.
Dann ist $\varphi^* = \forall\, x_1\, \forall\, x_2 \ldots \forall\, x_k\, \psi\, [y/f(x_1,\ldots x_k)]$ die Formel, in der jedes Vorkommen von y durch $f(x_1,\ldots x_k)$ ersetzt wurde.
φ^* heißt Skolemisierung von φ und $f(x_1,\ldots x_k)$ die zugehörige Skolemfunktion.

Umwandlung von beliebigen prädikatenlogischen Formeln in quantorenfreie Darstellung:

1. Umformung in PNF

   z.B. $\forall x\, P(x,y) \wedge \exists y: Q(y)$ wird zu $\forall x\, \exists z\, P(x,y) \wedge Q(z)$

2. Eliminierung von Existenzquantoren durch Skolemisierung

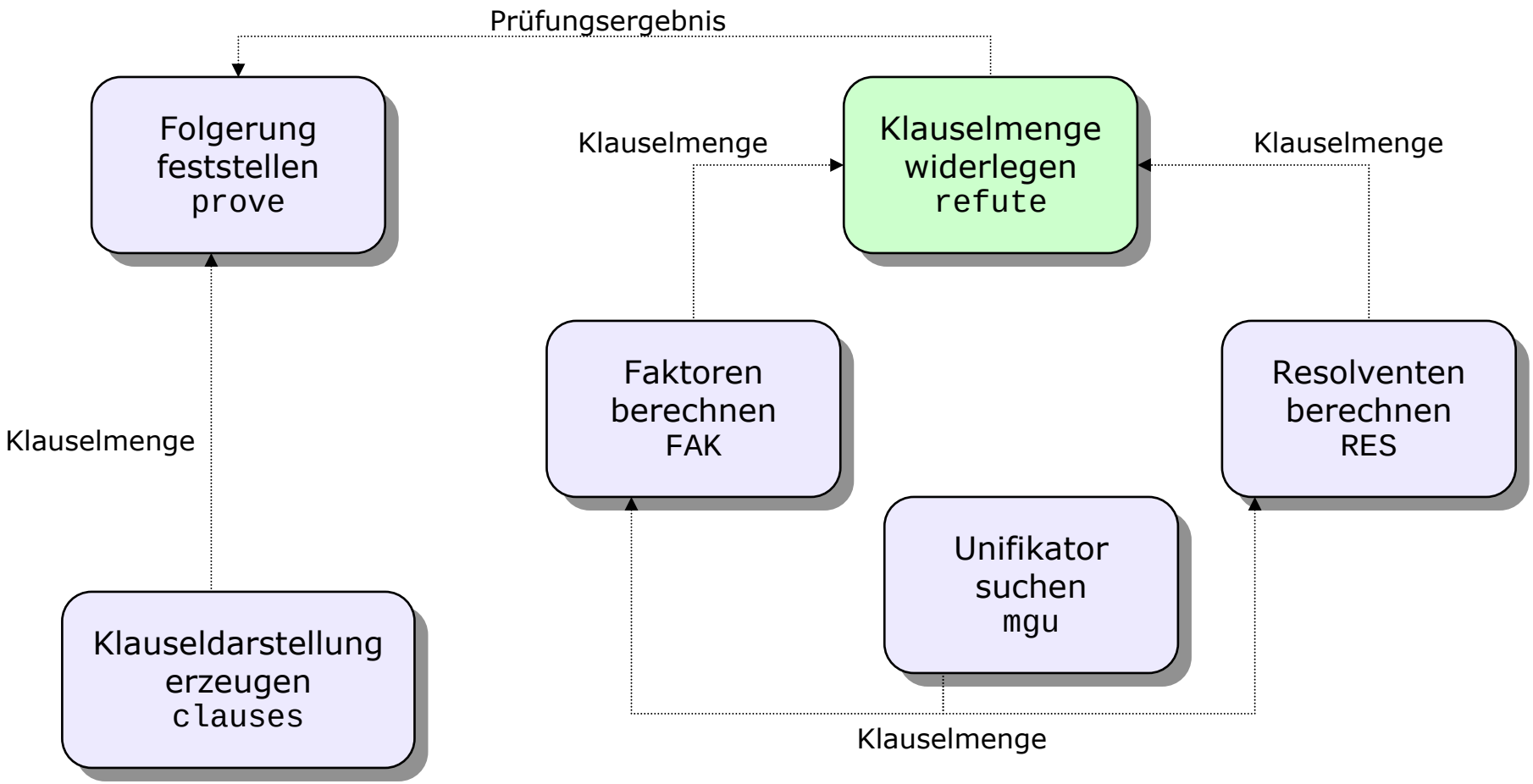
   z.B. $\forall x \exists y: P(x,y)$ wird zu $\forall x\, P(x,f(x))$

3. Ersetzung von Allquantoren durch Existenzquantoren mit verallgemeinerten deMorgan

   z.B. $\forall x,y: P(x) \vee Q(x)$ wird zu $\neg\, \exists\, x,y: \neg\, P(x) \wedge \neg\, Q(x)$

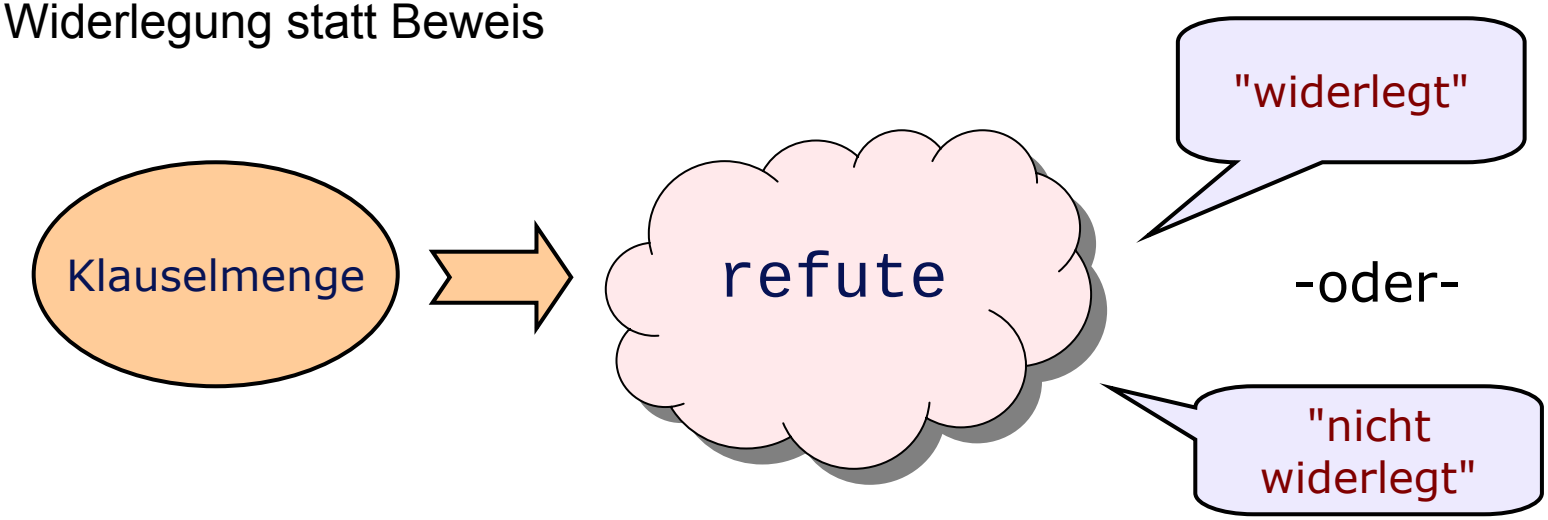
4. Nochmalige Eliminierung der neuen Existenzquantoren durch Skolemisierung

Modularisierung und Datenflüsse im Resolutionsbeweiser



refute: Anforderungen und Funktionalität

- `prove` als Überbau und Schnittstelle (koordinierende Instanz)
- `refute` als Systemkern (austauschbar)
- Implementation des Resolutionskalküls
- Widerlegung statt Beweis



- Aufgabe:
  Suche die Belegungen für die Variablen, unter denen die Klauselmenge widerlegt ist.

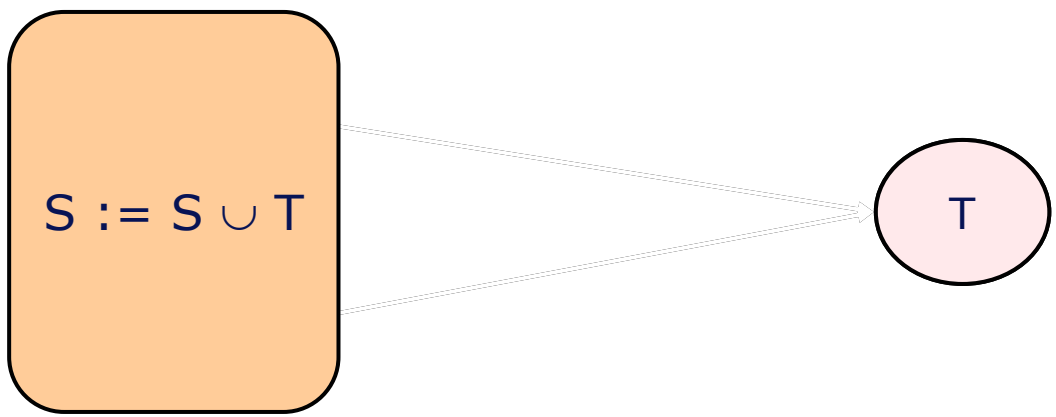
refute: Funktionsweise

Datenstrukturen:

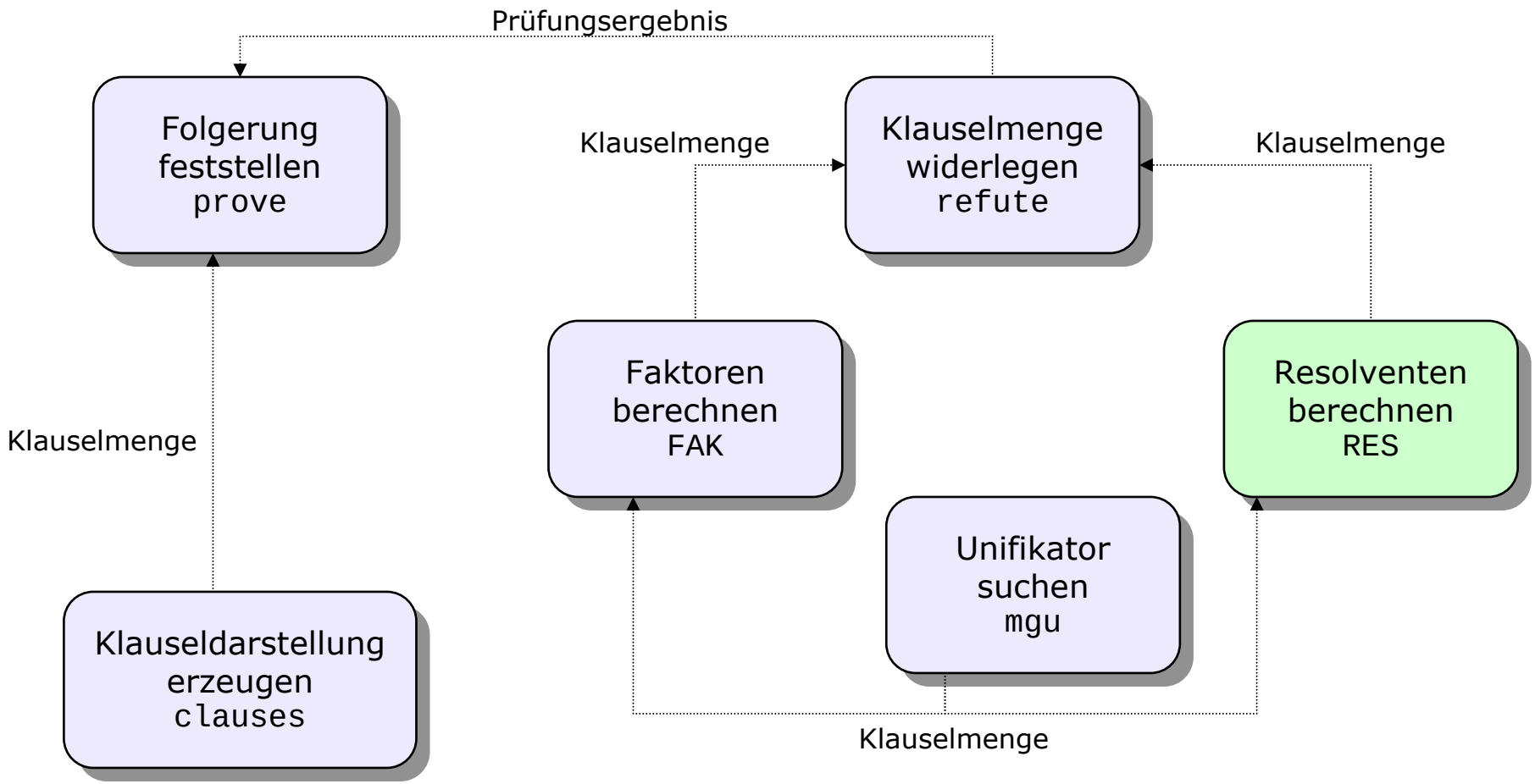
- Akkumulator S für sämtliche Klauseln der bisherigen Herleitung
- Behälter T für Klauseln der letzten Generation

Vorgehen:

- Zuletzt erzeugte Klauseln (T) mit bisherigen Klauseln resolvieren
- S um T ergänzen, neu resolvierte Klauseln werden neues T
- Stopp wenn leere Klausel hergeleitet oder keine Resolutionsmöglichkeiten mehr

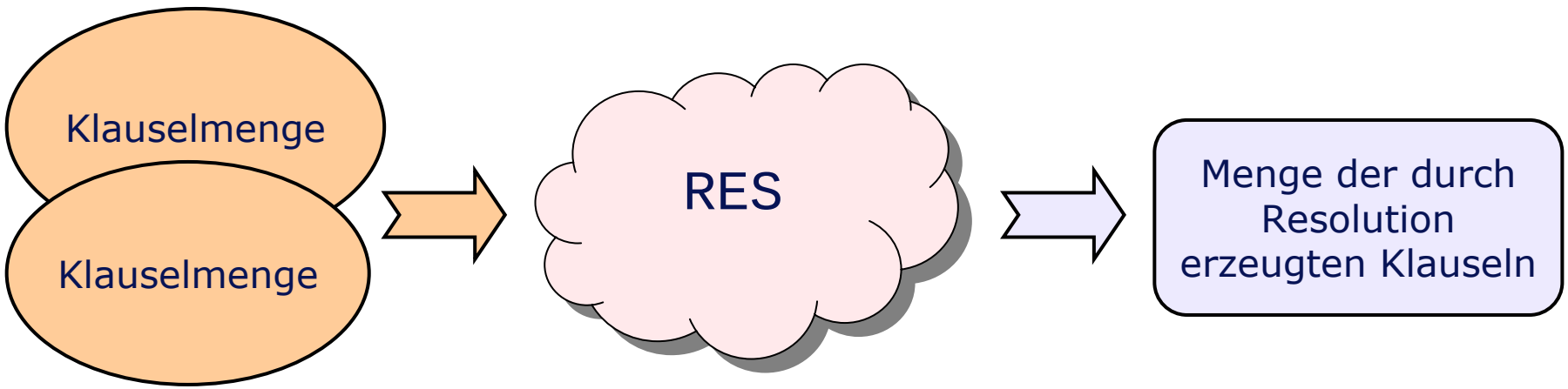


Modularisierung und Datenflüsse im Resolutionsbeweiser



RES: Anforderungen und Funktionalität

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



RES: Resolution in der Prädikatenlogik

Definition:

Seien C_1, und C_2 Klauseln, L_1 Literal der Klausel C_1, L_2 Literal der Klausel C_2 und σ eine Variablensubstitution.

Nach Durchführung von σ seien die Literale L_1 und L_2 komplementär

Dann ist **$Res(C_1, L_1, C_2, L_2, \sigma)$** die Resolvente nach Durchführung der Ersetzung σ

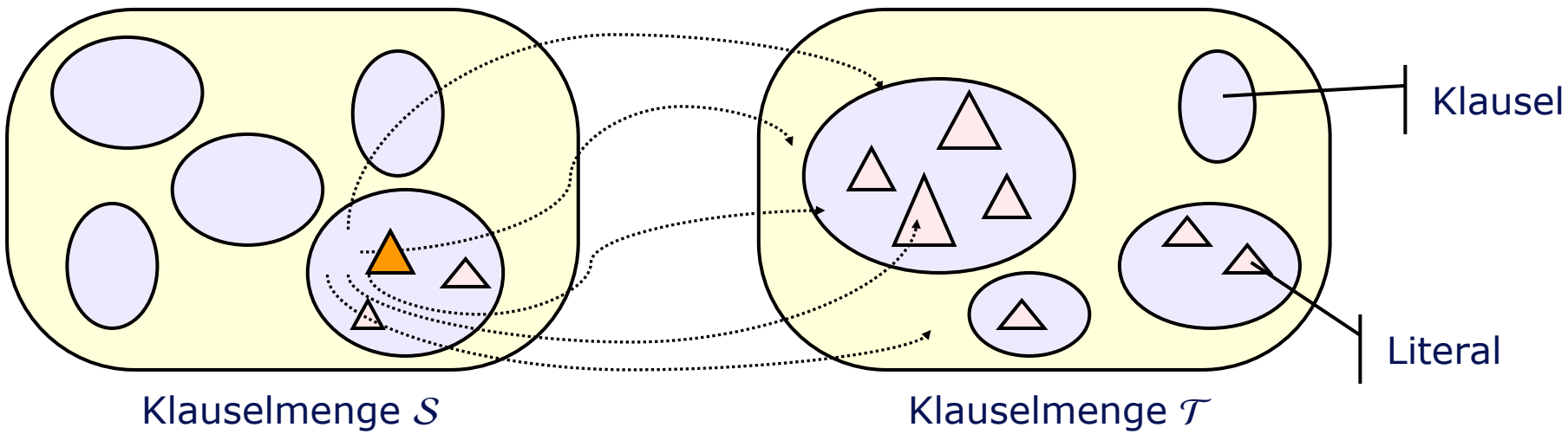
Bsp.: $C_1=\{P(x), \neg Q(f(y))\}$ $C_2=\{Q(z), R(g(z))\}$

$L_1= \neg Q(f(y))$ $L_2=Q(z)$ $\sigma=[z/f(y)]$

$Res(C_1, L_1, C_2, L_2, \sigma) = \{P(x), R(g(f(y)))\}$

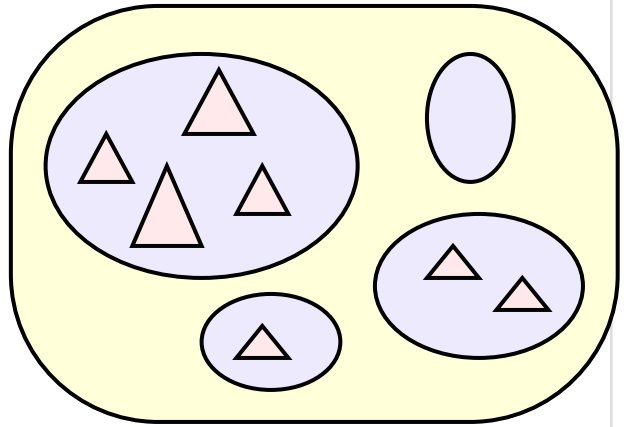
RES: Grundsätzliche Funktionsweise

- Jede Klausel der einen wird mit jeder Klausel der anderen Menge auf Resolvierbarkeit getestet.
- Jedes Literal der einen wird mit jedem Literal der anderen Klausel auf Resolvierbarkeit getestet.
- Ggf. Resolventenbildung

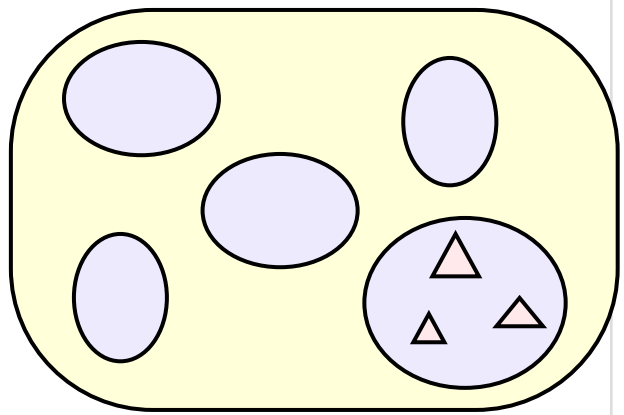


RES: Implementierung im Detail

```
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
```

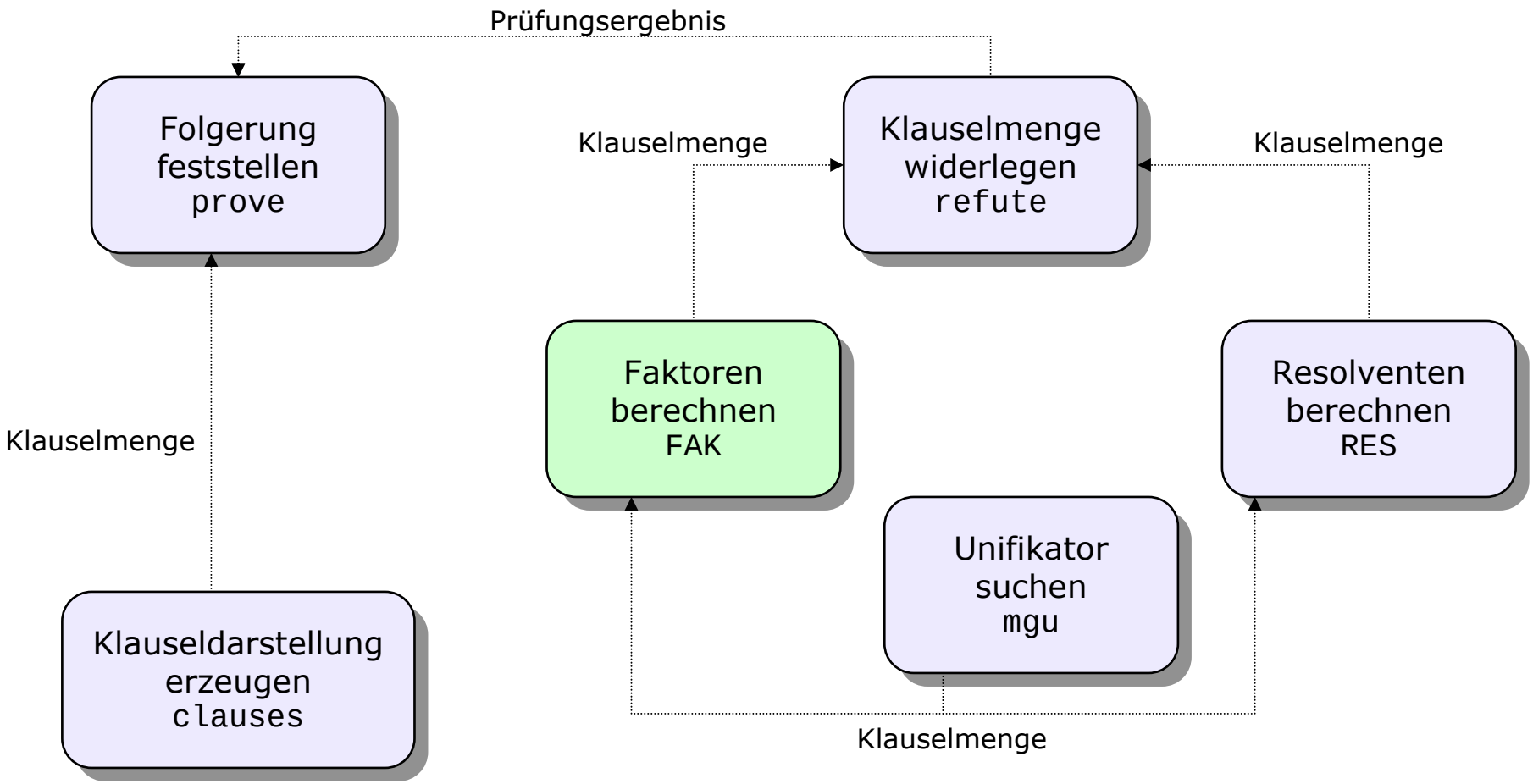


Klauselmenge $\mathcal{T}$



Klauselmenge $\mathcal{S}$

Modularisierung und Datenflüsse im Resolutionsbeweiser



FAK: Motivation

Russellsche Antinomie:

„Der Barbier rasiert eine Person genau dann, wenn sie sich nicht selbst rasiert."

$\forall$ x [Rasiert(barbier, x) $\Leftrightarrow$ ¬Rasiert(x , x)]

Enthält Widerspruch, der sich aber durch RES alleine nicht herleiten lässt:

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

[y/barbier, x/barbier]

RES leitet her: {¬ Rasiert(barbier ,barbier), Rasiert(barbier, barbier)}

Wir wollen herleiten: {¬ Rasiert(barbier ,barbier)} {Rasiert(barbier, barbier)}

entnommen aus:
Studentenvorlesung von Björn Peemöller und Stefan Roggensack, gehalten im WS 2007/2008

FAK: Faktoren in der Prädikatenlogik

Definition:

Sei C eine Klausel, und L eine Menge von Literalen dieser Klausel C.

σ sei der mgu von L (kann leer sein).

Dann ist **Fak(C, L)= σ(C)** der Faktor von C bezüglich L.

Bsp.: C ={P(x), P(f(y)), Q(z)}

L= {P(x), P(f(y))}

σ=[x/f(y)]

Fak(C, L) = {P(f(y)), Q(z)}

FAK: Anforderungen und Funktionalität

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

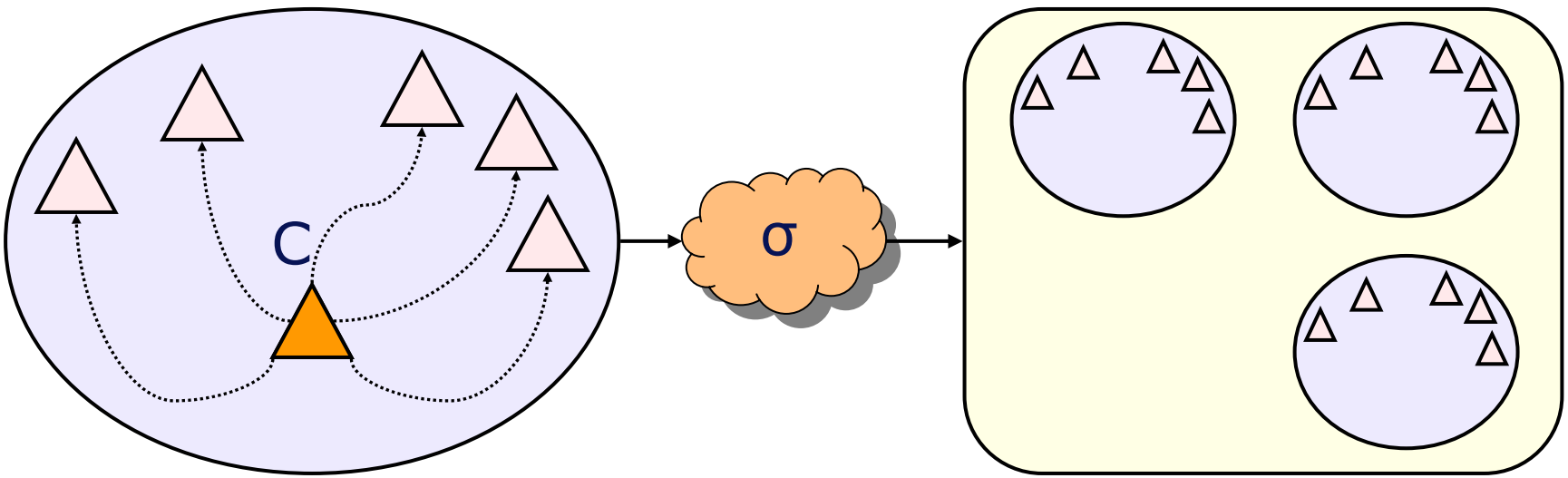
FAK: Grundsätzliche Funktionsweise

- Faktoren haben nur eine Elternklausel
- Teilalgorithmus `fak` für eine Klausel
- Iteration über alle Elemente der Input-Menge

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

FAK: Funktionsweise von Teilalgorithmus fak

- Jedes Literal mit jedem auf Unifizierbarkeit überprüfen
- Anwendung des gefunden Unifikators auf gesamte Klausel (=Faktorisierung)
- Sammlung der Teilergebnisse



FAK: Detailimplementierung von Teilalgorithmus fak

```
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))
        fi
      fi
    done
  done
  return S ∪ {C}
end
```

Zusammenfassung und Ausblick

Resolutionsbeweiser:

Implementierung eines Systems aus sechs Algorithmen

- `mgu`
- `RES`
- `FAK`
- `clauses`
- `refute`
- `prove`

Teilalgorithmen in beispielhaften Grundversionen

Ausblick:

Für praktische Implementierungen Optimierungen nötig

Ableitungsstrategien – sehr viele Möglichkeiten