

cp-logic

Umsetzung der Konzepte aus der Vorlesung
Diskrete Mathematik und Logik
in eine Toolbox für Studierende (Bachelorarbeit)

Adrianus Kleemans

22.05.2014

u^b

UNIVERSITÄT
BERN

Inhaltsverzeichnis

- 1 Einleitung
- 2 Aufbau
- 3 Algorithmen
- 4 Beispiele - Demo
- 5 Schluss

u^b

UNIVERSITÄT
BERN

GUI der Toolbox

```

>> B = NOT p3 AND p4 AND p5 AND (NOT p2 OR p3 OR p1) AND NOT (NOT p1 OR p0)
[1] B =  $\neg p_3 \wedge p_4 \wedge p_5 \wedge (\neg p_2 \vee p_3 \vee p_1) \wedge \neg (\neg p_1 \vee p_0)$ 

>> C = NOT p0 AND p1 AND NOT p2 AND (p1 OR NOT p2)
[2] C =  $\neg p_0 \wedge p_1 \wedge \neg p_2 \wedge (p_1 \vee \neg p_2)$ 

>> nnf(A)
[3]  $(p_0 \vee p_1) \wedge (p_2 \wedge ((\neg p_3 \wedge \neg p_4) \wedge p_5))$ 

>> cnf(B)
[4]  $((\neg p_3 \wedge p_4) \wedge p_5) \wedge ((\neg p_2 \vee p_3) \vee p_1) \wedge (p_1 \wedge \neg p_0)$ 

>> D = ((p0 IMPL p1) AND (p2 IMPL p1)) IMPL ((p0 OR p2) IMPL p1)
[5] D =  $((p_0 \rightarrow p_1) \wedge (p_2 \rightarrow p_1)) \rightarrow ((p_0 \vee p_2) \rightarrow p_1)$ 

>> pedantic(NOT NOT p0 AND NOT p1 AND NOT p3)
[6]  $(\neg \neg p_0 \wedge \neg p_1) \wedge \neg p_3$ 

>> dchains(NOT ((p0 AND p1 AND NOT p1 AND p2) OR NOT (p1 AND NOT p2 AND NOT p3)))
[7] Not a valid chain in PSC.
  
```

Go

Motivation

Ausgangspunkt: Skript **Diskrete Mathematik und Logik**
Im Skript werden viele Algorithmen beschrieben:

Definition 123 *The \rightarrow -reduction $\rho(A)$ of a formula A is inductively defined as follows:*

1. *If A is atomic, then $\rho(A) := A$.*
2. *If A is a formula $\neg B$, then $\rho(A) := \neg\rho(B)$.*
3. *If A is a formula $B \vee C$, then $\rho(A) := \rho(B) \vee \rho(C)$.*

Ideen:

- Lernprozess durch Ausprobieren
- Schnelles Feedback über Korrektheit

u^b

UNIVERSITÄT
BERN

Anforderungen an das Endprodukt

- Nähe zum Skript (gleiche Namen und Funktionen)
- Abbildung aller in der Vorlesung gewichteten Teile des Skripts
- Detaillierte Fehlerbehandlung, “fail safe”
- Graphische Darstellung (z.B. der Deduktionsketten)
- Einfache Eingabemöglichkeit für Benutzer
- Nachvollziehbarkeit (History, gespeicherte Formeln)
- Laden von Formeln aus Dateien

Anforderungen an die Technologie

Diverse Anforderungen, welche die Möglichkeiten einschränken:

- Unicode-Unterstützung (\vee , \wedge , \neg , \top , ...)
- Geeignete Datenstrukturen (verschachtelte Listen, assoziative Arrays, binäre Bäume)
- Unabhängigkeit vom Betriebssystem
- Lesbarer Code, offen für Erweiterungen

Verwendete Technologien

Einige verwendete Technologien:

- Programmiersprache: Python
- GUI: QT4 (`pyside`), `pygame`
- Versionsverwaltung: `git`
- Unit tests: Modul `unittest`
- Unterstützende Technologien, z.B. `pyinstaller` zur Paketierung

Methoden

Entwicklung mit einigen recht intuitiven Methoden:

- TDD (Test Driven Development): Unit Tests
- Orientierung der Tests an Use cases (direkt aus Skript und Übungen)
- Grundklasse: `Formula`, andere Klassen bauen darauf auf
- Interaktive Erarbeitung, Feedback der Betreuer

Test Driven Development

Zuerst die Spezifikation des Testfalls:

```
def test_invalidFormula_propositions(self):  
    try: f = Formula('p0 OR OR NOT p1')  
    except FormulaInvalidError: pass  
    else: self.fail("Connectives without propositions  
        are not allowed.")
```

Danach in der Formel-Klasse umsetzen, bis der Testfall erfolgreich durchläuft. \Rightarrow Auch später noch Nutzen der Tests als Regressionstests!

u^b

UNIVERSITÄT
BERN

Ein paar Kennzahlen

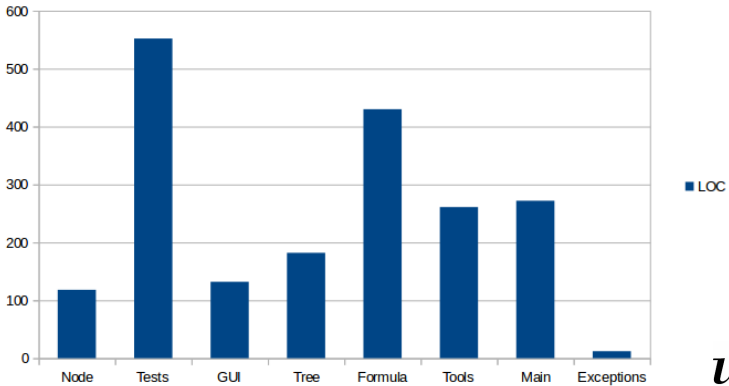
Ein paar Kennzahlen:

- 8 Klassen mit insgesamt ~2000 Zeilen Code
- 35 Seiten Bericht + zusätzliche Dokumentation im Code für ~70 Funktionen
- insgesamt ~120 Tests, ein Grossteil für die Formel-Klasse
- allermeisten Anforderungen wurden umgesetzt

u^b

UNIVERSITÄT
BERN

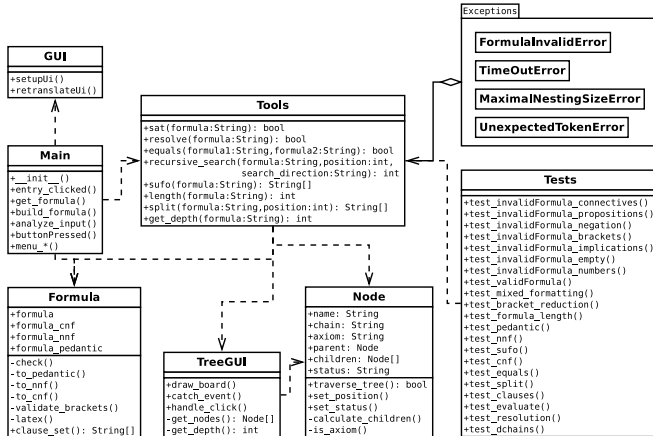
Klassen-Aufteilung



u^b

UNIVERSITÄT
BERN

Aufbau (UML) I



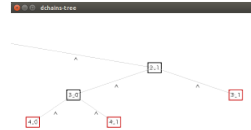
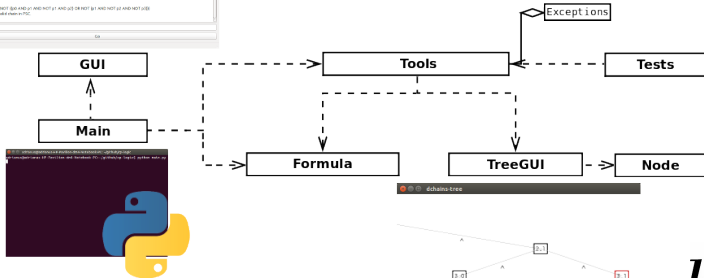
u^b

Aufbau (UML) II

```

1) 1) Notation Classical Propositional Logic
2) 2) Example: "Book"

--> B = NOT p1 AND p2 AND p3 AND (NOT p4 OR p5 OR p6 AND NOT p7 OR p8)
10) B = NOT (p1 OR p2 OR p3 OR p4 OR p5 OR p6)
--> C = NOT (p1 AND p2 AND NOT p3 AND NOT p4 OR NOT p5)
20) C = NOT p1 OR NOT p2 OR p3 OR NOT p4 OR NOT p5
--> NOT(B)
30) (p1 OR p2) OR (p3 OR p4 OR p5 OR p6)
--> NOT(C)
40) ((p1 OR p2 OR p3 OR p4 OR p5 OR p6) OR (p1 OR p2 OR p3))
50) D = (p1 AND p2 AND p3 AND p4 AND p5 OR p6) AND (p7 AND p8)
60) D = (p1 AND p2) OR (p1 AND p3) OR (p1 AND p4) OR (p1 AND p5) OR (p1 AND p6)
--> AND(NOT NOT p1 AND NOT p2 AND NOT p3)
70) C OR NOT p1 OR NOT p2 OR NOT p3
--> AND(NOT (p1 AND p2 AND NOT p3) OR NOT p4 OR NOT p5 OR NOT p6)
71) Not a valid classical logic
  
```



Klassen

- **Main** - Einstiegspunkt
- **GUI** - Python/QT4-Benutzeroberfläche (Fenster mit Menü)
- **Formula** - Formel (mit Normalformen, Sufos, etc.)
- **Tools** - Weiterführende Funktionen (Erfüllbarkeit, Klauselmengen, lineare Suche, Tiefensuche etc.)
- **TreeGUI** - Deduktionsketten-Baum (Darstellung)
- **Node** - Deduktionsketten-Knoten (rekursiv berechnet)
- **Exceptions** - eigene Fehlerbehandlung
- **Tests** - Funktionale Tests, Regressionstests

u^b

UNIVERSITÄT
BERN

Normalformen

Basisformel $A = p_1 \vee p_2 \rightarrow \top$.

- *pedantic()*: Hierarchische Struktur: \neg vor $\vee \wedge$ vor \rightarrow

$$\textit{pedantic}(A) = (p_1 \vee p_2) \rightarrow \top$$

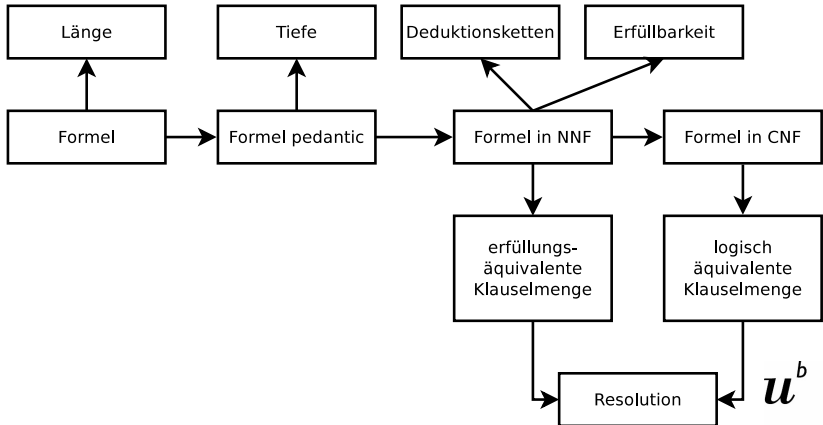
- *nnf()*: Implikationen, Negationen nur vor atomaren Prop.

$$\textit{nnf}(A) = (\neg p_1 \wedge \neg p_2) \vee \top$$

- *cnf()*: Konstanten ersetzen, $p_1 \vee (p_2 \wedge p_3) \rightarrow (p_1 \vee p_2) \wedge (p_1 \vee p_3)$

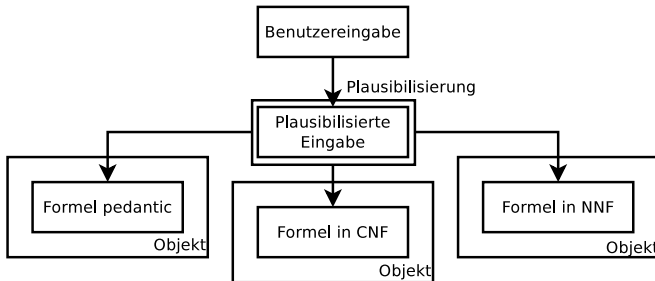
$$\textit{cnf}(A) = (\neg p_1 \vee (p_0 \vee \neg p_0)) \wedge (\neg p_2 \vee (p_0 \vee \neg p_0)) \mathbf{u}^b$$

Übersicht Abhängigkeiten Normalformen



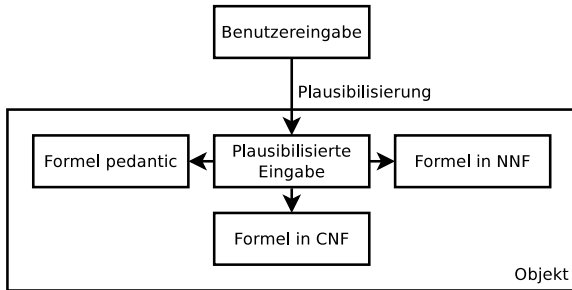
Modellierung Formel-Klasse I

2 Ideen: Entweder die einzelnen Normalformen als einzelne Objekte abbilden...



Modellierung Formel-Klasse II

...oder Formel mit Normalformen als einzelnes Objekt abbilden.



⇒ Beide Ansätze Vor- und Nachteile.

In meinem Konzept zweiter Vorschlag sinnvoller (Metavar.)

Funktionsübersicht

Die wichtigsten Funktionen:

- `nnf()`, `cnf()`, `pedantic()` - Normalformen
- `length()` - Länge
- `sufo()` - Subformulas
- `sat()` - Test auf Erfüllbarkeit
- `clause_set()` - Klauselmengen
- `evaluate()` - Evaluiert Formeln auf Wahrheitswert
- `resolution()` - Resolution
- `dchains()` - Deduktionsketten

u^b

UNIVERSITÄT
BERN

Fehlermeldungen

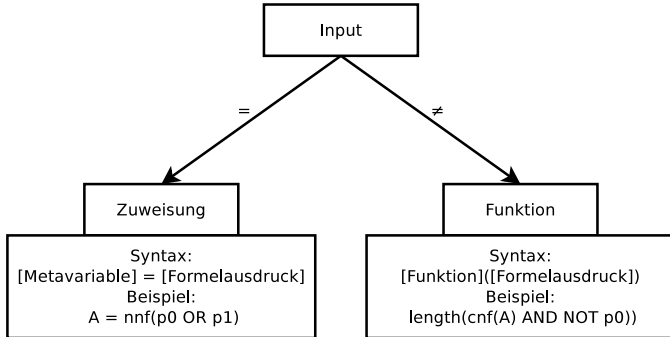
Katalog von Fehlermeldungen zur Hilfe des Benutzers:

- Ungültige Zeichen: p_0a
- Proposition ohne Index: p
- Nicht verknüpfte Propositionen: p_0p_1
- Alleinstehende Indizes oder Negation: $p_0 \vee 2$ oder $p_0 \vee \neg$
- Ungültige Konjunktion: $p_0 \wedge \vee p_1$
- Gemischte \wedge und \vee auf selber Ebene: $p_0 \wedge p_1 \vee p_2$
- Ungültige Implikation: $\rightarrow p_0$ oder $p_0 \rightarrow p_1 \rightarrow p_2$
- Ungültige Klammersetzung: $((p_0 \wedge p_1) \vee p_2$
- ...

u^b

Benutzereingabe

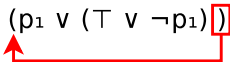
Verarbeitung der Eingabe durch Benutzer:



u^b

Suche in Formeln

Als Beispiel für die Auflösung von Negationen (NNF) müssen zusammengehörige Klammernpaare identifiziert werden.

$$p_0 \wedge (p_1 \vee (\neg \vee \neg p_1)) \rightarrow p_2$$


⇒ Einfachste Möglichkeit, die zugehörige Klammer zu finden?

Konzept Formeltiefe

Einführung des Konzepts der **Tiefe**. Zerlegung einer Formel

$$p_0 \wedge (\neg p_1 \vee (p_2 \wedge \neg p_3) \vee (p_1 \wedge (\top \vee p_3)))$$

in verschiedene Ebenen:

Ebene	
0	$p_0 \wedge ($
1	$\neg p_1 \vee ($
2	$p_2 \wedge \neg p_3$
3	$\top \vee p_3$

Satisfiability

Pseudocode:

- ❶ Wandle Formel in NNF um
- ❷ Sammle Anzahl n atomare Propositionen
- ❸ Für alle 2^n Valuationen, tue folgendes:
 - ❶ Setze Valuation an Stelle der Propositionen ein
 - ❷ Evaluiere den Wahrheitswert und reduziere diesen auf `true` oder `false`
 - ❸ Falls `true`, beende und gib diese Valuation aus.

Auflösung durch `evaluate()`, z.B.:

$$\begin{aligned} & \perp \vee (\perp \wedge \top) \\ & \perp \vee \perp \\ & \perp \end{aligned}$$

u^b

UNIVERSITÄT
BERN

Klammern setzen

Formeln für Meta-Variablen können nicht 1:1 eingesetzt werden:

$$>> A = p_0 \wedge \neg p_1$$

$$>> \neg A$$

$$[0] \neg p_0 \wedge \neg p_1$$

Zusätzliches Einfügen von Klammern entschärft dies:

$$>> \neg A$$

$$[0] \neg(p_0 \wedge \neg p_1)$$

Problem: Aufruf interner Funktionen verändert Formel!

$$(p_0 \wedge \neg p_1), ((p_0 \wedge \neg p_1)), (((p_0 \wedge \neg p_1))), \dots$$

u^b

UNIVERSITÄT
 BERN

Klammern reduzieren

Es wird ein Verfahren benötigt, um Klammern zu reduzieren.
 \Rightarrow Festhalten der Ebenenübergänge. Wird eine Ebene auf beiden Seiten “übersprungen”, ist der Übergang unnötig.

$$p_0 \vee \underbrace{(}_{0-1} \underbrace{(}_{1-2} p_1 \wedge \neg p_2 \underbrace{)}_{2-1} \underbrace{)}_{1-0}$$

Findet dazwischen ein Übergang statt, werden Klammern benötigt:

$$p_0 \vee \underbrace{(}_{0-1} \underbrace{(}_{1-2} p_1 \wedge \neg p_2 \underbrace{)}_{2-1} \vee \underbrace{(}_{1-2} p_0 \vee p_1 \underbrace{)}_{2-1} \underbrace{)}_{1-0}$$

u^b

Demo

Einige Beispiele:

- Eingabe von Hand und per Datei - `sample_file.formula`
 - Formeln
 - Metavariablen
 - Verschachtelungen
 - Normalformen
- Satisfiability
- Deduktionsketten - `dchains.formula`

u^b

UNIVERSITÄT
BERN

Herausforderungen I

Vorsicht vor “lazy evaluation” (rekursive Knotenberechnung):

```
def traverse():  
    ...  
    return ( child1.traverse() and child2.traverse() )
```

⇒ Baum wird nicht immer vollständig gezeichnet!

```
def traverse():  
    ...  
    a = child1.traverse()  
    b = child2.traverse()  
    return ( a and b )
```

Mit Zwischenspeichern werden Zwischenschritte ausgeführt.

Herausforderungen II

Komplexe Ausdrücke sollten evaluiert werden können:

>> $\text{length}(\text{cnf}(A \text{ AND } p_0 \wedge \neg p_3 \text{ AND NOT } B))$

- Auflösung von Metavariablen
- Verknüpfung von Metavariablen, in Kombination mit Formeln
- gemischter ASCII- und Unicode-Input
- Pipeline von Funktionsaufrufen

⇒ Plausibilisierungen auf jeder Stufe

u^b

UNIVERSITÄT
BERN

Herausforderungen III

Weitere Herausforderungen:

- “Saubere” und stabile Formel-Klasse als Basis
- Eigene Entscheidungen ergänzend zum Skript (z.B. pedantic-Form)
- Unicode-Unterstützung Linux/Windows
- Skalierbares & scrollbares Fenster für dchains

u^b

UNIVERSITÄT
BERN

Fazit

Fazit:

- Getestete, stabile Applikation mit graphischer Oberfläche
- Sehr viel gelernt (Logik, Zusammenspiel Technologie, GUI)
- Aufwand schwer abzuschätzen (z.B. Formel-Klasse)
- Umgehen mit Anforderungen, Änderungen, Prioritäten

u^b

UNIVERSITÄT
BERN

Fragen?

Fragen?

u^b

UNIVERSITÄT
BERN