

Formale Methoden und Werkzeuge

Vorlesung

Wintersemester 2024, 25

Johannes Waldmann, HTWK Leipzig

30. Januar 2026

Die Übung vor der ersten Vorlesung

Organisation

- **Folien:** `https://www.imn.htwk-leipzig.de/~waldmann/lehre.html`
- **Quelltexte, Diskussionen Hausaufgaben:**
`https://gitlab.dit.htwk-leipzig.de/johannes.waldmann/fmw-ws25`
- **Software:** `ghc`, `cabal`; `minisat`, `ersatz`; `z3`, `hasmt`; `Agda`
- **nachfolgend:** Beispiele für Hausaufgaben (Zweck: Wiederholung Prädikaten- und Aussagenlogik)

Hausaufgaben

WS 25: 1b, 5, 4a, 2.

1. Pierluigi Crescenzi, Viggo Kann *A compendium of NP optimization problems*

<https://web.archive.org/web/20200227195925/http://www.nada.kth.se/~viggo/problemelist/compendium.html>

Beispiele:

- (a) Minimum File Transfer Scheduling (node195).
- (b) Minimum Dynamic Storage Allocation (node163).

Erläutern Sie die Spezifikation an einem Beispiel. Geben Sie eine lösbare Instanz sowie dafür zwei Lösungen mit unterschiedlichem Maß an.

2. Aufgabe: formalisieren Sie *Math Magic* Februar 2007.

Was ist dabei für Springer und König einfacher als für Dame, Läufer, Turm?

(allgemein für Math Magic: offene Fragen <https://erich-friedman.github.io/mathmagic/unsolved.html>)

3. Aufgabe: formalisieren Sie Wolkenkratzer oder Towers (was ist der Unterschied?)

<https://www.janko.at/Raetsel/Wolkenkratzer>

<https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/towers.html>

- $B = \{0, 1, \dots, n - 1\}$, Unbekannte: $h_{x,y} \in B$ für $x, y \in B$
- Constraint für eine Zeile x : (entspr. für Spalte)

$\bigvee_{p \in \text{Permutationen}(0, \dots, n-1), p \text{ kompatibel mit Vorgaben}} \bigwedge_{y \in \{0, \dots, n-1\}} (h_{x,y} = p(y))$

Bsp: $n = 4$, Vorgabe links 2, rechts 1, kompatibel sind
[0, 2, 1, 3], [2, 0, 1, 3], [2, 1, 0, 3], [2, 1, 0, 3].

- diese Formel wird exponentiell groß (wg. Anzahl Permutationen),

Folge-Aufgabe: *geht das auch polynomiell?*

- Geben Sie eine Aufgabenstellung der Größe $w \times w$ an mit $4 \cdot w$ Vorgaben (d.h., *alle* Vorgaben), die mehr als eine Lösung hat.

... für $w = 4$, für größere w (einige, alle)

- (offene Frage?) Geben Sie eine eindeutig lösbare Instanz mit möglichst wenigen Vorgaben an. (Sind $w - 1$ Vorgaben für $w \times w$ immer möglich?)

4. Modellierung von Puzzles (aus Tatham-Collection)

- (a) geben Sie ein Modell an für *Pegs* (Solitaire). Hinweise:
- Zustand = Boolesche Matrix,
 - Schritt = Relation zwischen Matrizen,
 - Lösung = Schrittfolge (\Rightarrow Zustandsfolge). Wieviele Schritte?
- (b) Modell für *Unruly*. (keine Zustandsfolge, sondern direkt die Lösung. Welches sind die Unbekannten, welches sind ihre Beziehungen, untereinander und zur Vorgabe)
- (c) modellieren Sie *Untangle*

Vergleichen Sie mit den tatsächlichen Quelltexten

<https://git.tartarus.org/?p=simon/puzzles.git>

5. Constraint für monotone kompatible Bewertungsfunktion:
- lösen Sie mit Z3 (ist im Pool installiert, vgl. <https://www.imn.htwk-leipzig.de/~waldmann/etc/pool/>)
 - eine *kleinste* Lösung finden (Summe von P, Q, R, S möglichst klein) — dafür Assert(s) hinzufügen.
 - Abstieg der so gefundenen Bewertungsfunktion nachrechnen für $abab \rightarrow baab \rightarrow baba \rightarrow bbaa$
 - gibt diese Bewertungsfunktion die maximale Schrittzahl genau wieder? (nein)
 - Folge-Aufgabe: entspr. Constraint-System für Bewertungsfunktion für $ab \rightarrow bba$ aufstellen und lösen.

Einleitung

Formale Methoden und Werkzeuge ...

- ... zur Spezifikation, Analyse (Verifikation), Synthese von Hard- und Softwaresystemen.
 - für (geplantes) System S (Bsp: CPU, Programm) *formales Modell* M angeben (Bsp: logische Schaltung, endl. Automat)
 - gewünschte System-Eigenschaft E angeben (Bsp: Boolesche Formel, reguläre Sprache)
 - Analyse: mit Werkzeug(unterstützung) feststellen, ob das Modell die Eigenschaft hat (Bsp: $M \Rightarrow E$, $M \subseteq E$)
 - Synthese (*Constraint-Programmierung*): Werkzeug konstruiert aus Constraint E ein passendes Modell M

Die Rolle der Abstraktion

- Schritt von System S zu Modell M ist *Abstraktion* (es werden Details ignoriert), das ist
 - teils *nützlich* (verbessert die Übersicht, Einsicht),
 - teils *notwendig* (nur wenn $M \subseteq E$ entscheidbar oder M aus E berechenbar, kann überhaupt ein Werkzeug zur Analyse bzw. Synthese implementiert werden)
- in den einfachen Beispielen (besonders am Anfang der VL) beginnen wir die Betrachtung bereits bei M (damit wir sehen, welche Theorien angewendet werden sollen)
- das Abstrahieren muß aber auch geübt werden, es ist aber eine anwendungsspezifische Kunst

Industrielle Anwendungen von FMW

- Verifikation von Schaltkreisen
(*bevor* man diese tatsächlich produziert)
 $F = \text{S-Implementierung}(x) \neq \text{S-Spezifikation}(x)$
wenn F unerfüllbar ($\neg \exists x.F$), dann ist Impl. korrekt
- Verifikation von Software durch *model checking*:
Programmzustände abstrahieren durch
Zustandsprädikate, Programmabläufe durch endliche
Automaten.
z. B. Thomas Ball et al. 2004: *Static Driver Verifier*
<https://www.microsoft.com/en-us/research/project/slam/publications/>
benutzt Constraint-Solver Z3 (Nikolaj Björner et al.,
2007–) <https://github.com/Z3Prover/z3/wiki>

Industrielle Anwendungen von FMW

- automatische Analyse des Ressourcenverbrauchs von Programmen
 - Termination (jede Rechnung hält)
 - Komplexität (... nach $O(n^2)$ Schritten)
- mittels *Bewertungen* von Programmzuständen:
 - $W : \text{Zustandsmenge} \rightarrow \mathbb{N}$
 - wenn $z_1 \rightarrow z_2$, dann $W(z_1) > W(z_2)$.
- Parameter der Bewertung werden durch Constraint-System beschrieben.
- vgl. Carsten Fuhs: *Automated Termination Analysis...*, Intl. School on Rewriting, 2022 <https://viam.science.tsu.ge/clas2022/isr/termination.html>

Anwendung: Polynom-Interpretationen

- Berechnungsmodell: Wortersetzung (\approx Turingmaschine)

- Programm: $ab \rightarrow ba$ (\approx Bubble-Sort)

Beispiel-Rechnung: $abab \rightarrow baab \rightarrow baba \rightarrow bbaa$

- Bewertung W durch *Interpretation*: lineare Funktionen

$$f_a(x) = Px + Q, f_b(x) = Rx + S \text{ mit } P, Q, R, S \in \mathbb{N}$$

$$W(abab) = f_a(f_b(f_a(f_b(0))))), \dots$$

- Interp. ist monoton: $x > y \Rightarrow f_a(x) > f_a(y) \wedge f_b(x) > f_b(y)$

- Interp. ist kompatibel mit Programm: $f_a(f_b(x)) > f_b(f_a(x))$

- resultierendes Constraint-System für P, Q, R, S ,

- Lösung mittels Z3

Constraint-Programmierung—Beispiel

```
(set-logic QF_NIA) (set-option :produce-models true)
(declare-fun P () Int) (declare-fun Q () Int)
(declare-fun R () Int) (declare-fun S () Int)
(assert (and (< 0 P) (<= 0 Q) (< 0 R) (<= 0 S)))
(assert (> (+ (* P S) Q) (+ (* R Q) S)))
(check-sat) (get-value (P Q R S))
```

- *Constraint-System*: eine prädikatenlogische Formel F
 $0 < P \wedge \dots \wedge P \cdot S + Q > R \times Q + S$
- *Lösung*: Interpretation (Var.-Belegung), für die F wahr ist
- CP ist eine Form der *deklarativen* Programmierung.
- *Vorteil*: Benutzung von allgemeinen Suchverfahren (bereichs-, aber nicht anwendungsspezifisch).

Constraints in der Unterhaltungsmathematik

- Nikoli (1980–, „the first puzzle magazine in Japan.“)
<https://www.nikoli.co.jp/en/puzzles/>
- Erich Friedman: *Math Magic* 1998–2020
<https://erich-friedman.github.io/mathmagic/>
- Simon Tatham’s Portable Puzzle Collection <https://www.chiark.greenend.org.uk/~sgtatham/puzzles/>
- Angela und Otto Janko: <http://www.janko.at/Raetsel/> ,
- Donald Knuth: *A Potpourri of Puzzles*, 2022,
TAOCP, Band 4, Pre-Faszikel 9b,
<https://cs.stanford.edu/~knuth/taocp.html>,
<https://cs.stanford.edu/~knuth/fasc9b.ps.gz>

Wettbewerbe für Constraint-Solver

- für aussagenlogische Formeln:

`http://www.satcompetition.org/`

(SAT = satisfiability)

- für prädikatenlogische Formeln

`https://smt-comp.github.io/`

(SMT = satisfiability modulo theories)

Theorien: \mathbb{Z} mit \leq , Plus, Mal; \mathbb{R} mit \leq , Plus; ...

- Termination und Komplexität

`https://www.termination-portal.org/wiki/`

`Termination_Competition`

Gliederung der Vorlesung

- Aussagenlogik
 - zur Modellierung (SAT-Kodierung) von Systemen mit endlichem Zustandsraum, begrenzter Schritt-Zahl (bounded model checking)
 - Werkzeuge: SAT-Kompiler (Tseitin-Transformation), SAT-Solver (Propagation, Backtracking, Lernen)
- eingeschränkte Prädikatenlogik
 - Unbekannte aus unendlichen Bereichen (Zahlen)
 - Werkzeuge: SMT-Solver (SAT modulo Theory)
- allgemeine Beweis-Systeme
 - zur Modellierung beliebiger (unbeschränkter) Systeme
 - Methode: Typisierung (Curry-Howard-Isomorphie, program = proof), Werkzeuge: interaktive Beweiser

Organisatorisches

- jede Woche 1 Vorlesung + 1 Übung
- Hausaufgaben (*Haus* bedeutet: zuhause bearbeiten, in der Übung diskutieren)
 - Aufgaben im Skript
 - Aufgaben in `autotool`
- Prüfungszulassung: regelmäßiges und erfolgreiches Bearbeiten von Hausaufgaben
- Klausur (2 h, keine Hilfsmittel).
- Quelltexte, Planung und Diskussion der Übungsaufgaben
`https://gitlab.dit.htwk-leipzig.de/johannes.waldmann/fmw-ws24` (Projekt-Mitgliedschaft beantragen, Zugang wird dann auf Mitglieder eingeschränkt)

Literatur

- Cerone, A., Roggenbach, M., Schlingloff, B.-H., Schneider, G., Shaikh, S.A.: *Teaching formal methods for software engineering - ten principles*, Informatica Didactica, Nr. 9, (2015). <https://www.informaticadidactica.de/uploads/Artikel/Schlinghoff2015/Schlinghoff2015.pdf>
- Uwe Schöning, Jacob Toran: *Das Erfüllbarkeitsproblem SAT*, Lehmanns (2012)
- Christel Baier, Joost Katoen: *Principles of Model Checking*. MIT Press, Cambridge (2008)
- Samuel Mimram: *Program = Proof*, <https://www.lix.polytechnique.fr/Labo/Samuel.Mimram/teaching/INF551/course.pdf>

Hausaufgaben

WS 25: 1b, 5, 4a, 2.

1. Pierluigi Crescenzi, Viggo Kann *A compendium of NP optimization problems*

<https://web.archive.org/web/20200227195925/http://www.nada.kth.se/~viggo/problemelist/compendium.html>

Beispiele:

- (a) Minimum File Transfer Scheduling (node195).
- (b) Minimum Dynamic Storage Allocation (node163).

Erläutern Sie die Spezifikation an einem Beispiel. Geben Sie eine lösbare Instanz sowie dafür zwei Lösungen mit unterschiedlichem Maß an.

2. Aufgabe: formalisieren Sie *Math Magic* Februar 2007.

Was ist dabei für Springer und König einfacher als für Dame, Läufer, Turm?

(allgemein für Math Magic: offene Fragen <https://erich-friedman.github.io/mathmagic/unsolved.html>)

3. Aufgabe: formalisieren Sie Wolkenkratzer oder Towers (was ist der Unterschied?)

<https://www.janko.at/Raetsel/Wolkenkratzer>

<https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/towers.html>

- $B = \{0, 1, \dots, n - 1\}$, Unbekannte: $h_{x,y} \in B$ für $x, y \in B$
- Constraint für eine Zeile x : (entspr. für Spalte)

$\bigvee_{p \in \text{Permutationen}(0, \dots, n-1), p \text{ kompatibel mit Vorgaben}} \bigwedge_{y \in \{0, \dots, n-1\}} (h_{x,y} = p(y))$

Bsp: $n = 4$, Vorgabe links 2, rechts 1, kompatibel sind
[0, 2, 1, 3], [2, 0, 1, 3], [2, 1, 0, 3], [2, 1, 0, 3].

- diese Formel wird exponentiell groß (wg. Anzahl Permutationen),

Folge-Aufgabe: *geht das auch polynomiell?*

- Geben Sie eine Aufgabenstellung der Größe $w \times w$ an mit $4 \cdot w$ Vorgaben (d.h., *alle* Vorgaben), die mehr als eine Lösung hat.

... für $w = 4$, für größere w (einige, alle)

- (offene Frage?) Geben Sie eine eindeutig lösbare Instanz mit möglichst wenigen Vorgaben an. (Sind $w - 1$ Vorgaben für $w \times w$ immer möglich?)

4. Modellierung von Puzzles (aus Tatham-Collection)

- (a) geben Sie ein Modell an für *Pegs* (Solitaire). Hinweise:
- Zustand = Boolesche Matrix,
 - Schritt = Relation zwischen Matrizen,
 - Lösung = Schrittfolge (\Rightarrow Zustandsfolge). Wieviele Schritte?
- (b) Modell für *Unruly*. (keine Zustandsfolge, sondern direkt die Lösung. Welches sind die Unbekannten, welches sind ihre Beziehungen, untereinander und zur Vorgabe)
- (c) modellieren Sie *Untangle*

Vergleichen Sie mit den tatsächlichen Quelltexten

<https://git.tartarus.org/?p=simon/puzzles.git>

5. Constraint für monotone kompatible Bewertungsfunktion:
- lösen Sie mit Z3 (ist im Pool installiert, vgl. <https://www.imn.htwk-leipzig.de/~waldmann/etc/pool/>)
 - eine *kleinste* Lösung finden (Summe von P, Q, R, S möglichst klein) — dafür Assert(s) hinzufügen.
 - Abstieg der so gefundenen Bewertungsfunktion nachrechnen für $abab \rightarrow baab \rightarrow baba \rightarrow bbaa$
 - gibt diese Bewertungsfunktion die maximale Schrittzahl genau wieder? (nein)
 - Folge-Aufgabe: entspr. Constraint-System für Bewertungsfunktion für $ab \rightarrow bba$ aufstellen und lösen.

Erfüllbarkeit aussagenlogischer Formeln (SAT)

Aussagenlogik: Syntax

aussagenlogische Formel:

- elementar: Variable v_1, \dots
- zusammengesetzt: durch Operatoren
 - einstellig: Negation
 - zweistellig: Implikation, Äquivalenz, Antivalenz,
 - mehrstellig möglich: Konjunktion, Disjunktion,

damit auch Quantifikation über endlichen Bereich E
($\forall x \in E : F$) ist (endliche!) Konjunktion $\bigwedge_{x \in E} F$

Aussagenlogik: Semantik

- Wertebereich $\mathbb{B} = \{0, 1\}$, Halbring $(\mathbb{B}, \vee, \wedge, 0, 1)$
Übung: weitere Halbringe mit 2 Elementen?
- *Belegung* ist Abbildung $b : V \rightarrow \mathbb{B}$
- *Wert* einer Formel F unter Belegung b : $\text{val}(F, b)$
- wenn $\text{val}(F, b) = 1$, dann ist b ein *Modell* von F ,
Schreibweise: $b \models F$
- *Modellmenge* $\text{Mod}(F) = \{b \mid b \models F\}$
- F *erfüllbar*, wenn $\text{Mod}(F) \neq \emptyset$
- *Modellmenge einer Formelmenge*:
 $\text{Mod}(M) = \{b \mid \forall F \in M : b \models F\}$

Formulierung von SAT-Problemen mit Ersatz

- Autoren: Edward Kmett et al.,

<https://hackage.haskell.org/package/ersatz>,

- `import Prelude hiding ((&&), (||), not)`

```
import Ersatz
```

```
main = do
```

```
  ans <- solveWith minisat $ do
```

```
    p <- exists @Bit ; q <- exists @Bit
```

```
    assert $ (p || not q) && (not p || q)
```

```
    return [p,q]
```

```
  case ans of (Satisfied, Just res) -> print res
```

- Unbekannte erzeugen (`exists`), Formel konstruieren (`&&`, ...), assertieren, lösen, Antwort benutzen

- zu Implementierung vgl. <https://www.imn.>

htwk-leipzig.de/~waldmann/etc/untutorial/ersatz/

Benutzung von SAT-Solvern

- Eingabeformat: SAT-Problem in CNF:
 - Variable = positive natürliche Zahl
 - Literal = ganze Zahl ($\neq 0$, mit Vorzeichen)
 - Klausel = Zeile, abgeschlossen durch 0.
 - Formel = Header `p cnf <#Variablen> <#Klauseln>`,
danach Klauseln
- Beispiel: die (konj. Normal-)Formel $(v_1 \vee \neg v_2) \wedge (\neg v_1 \vee v_2)$:

```
p cnf 2 2
1 -2 0
-1 2 0
```

- Löser (Bsp.): `minisat input.cnf output.text`
Niklas Eén, Niklas Sörensson: *An Extensible SAT Solver*,
2003, <https://minisat.se/>,

SAT-Modellierung für das N -Damen-Problem

stelle möglichst viele Damen auf $N \times N$ -Schachbrett,
die sich nicht gegenseitig bedrohen.

- Unbekannte: $q_{x,y}$ für $(x, y) \in P = \{1, \dots, N\}^2$
mit Bedeutung: $q_{x,y} \iff$ Position (x, y) ist belegt
- Constraint:
$$\bigwedge_{a,b \in F, a \text{ bedroht } b} \neg q_a \vee \neg q_b.$$
- „möglichst viele“ läßt sich hier vereinfachen zu:
„in jeder Zeile genau eine“.
vereinfachen zu: „. . . wenigstens eine.“

wir schreiben Programm,

N -Damen mit Ersatz

- Hilfsfunktionen für Boolesche Matrizen mit

```
import qualified Ersatz.Relation as R
```

- `queens n = do`

```
  out <- solveWith (cryptominisat5Path "kissat") $ do
```

```
    b <- R.relation ((1,1), (n,n))
```

```
    assert $ flip all (rows b) $ \ r -> or r
```

```
    assert $ flip all (R.indices b) $ \ p ->
```

```
      flip all (R.indices b) $ \ q ->
```

```
        encode (reaches p q) ==> not (b R.! p && b R.! q)
```

```
    return b
```

```
  case out of
```

```
    (Satisfied, Just b) -> do putStrLn $ R.table b
```

- (vollst. Quelltext: siehe Repo)

Zusammenfassung Ersatz (bisher)

- innerhalb von `solveWith` steht Befehlsfolge, Befehl ändert Zustand (Variablenzähler, Klausel-Ausgabe)
 - `exists @Bit :: MonadSAT s m => m Bit, R.relation _`
konstruiert neue `Variable(n)`
 - `assert :: Bit -> m ()`
übersetzt Formel in CNF, gibt Klauseln aus
- der Typ `Bit` beschreibt aussagenlogische Formeln (genauer: Schaltkreise), d.h., *symbolische* Repräsentation von (unbekannten) Wahrheitswerten
Typ `Bool` beschreibt konkrete (bekannte) Wahrheitswerte
booleschen Operatoren (`import Ersatz`) sind polymorph
 - `not :: Boolean b => b -> b,`
 - `instance Boolean Bool, instance Boolean Bit`

Modellierung durch SAT: Ramsey

gesucht ist Kanten-2-Färbung des K_5 ohne einfarbigen K_3 .

- Aussagenvariablen $f_{i,j}$ = Kante (i, j) ist rot (sonst blau).
- Constraints:

$$\forall p : \forall q : \forall r : (p < q \wedge q < r) \Rightarrow ((f_{p,q} \vee f_{q,r} \vee f_{p,r}) \wedge \dots)$$

das ist ein Beispiel für ein Ramsey-Problem

(F. P. Ramsey, 1903–1930)

<http://www-groups.dcs.st-and.ac.uk/~history/Biographies/Ramsey.html>

diese sind schwer, z. B. ist bis heute unbekannt: gibt es eine Kanten-2-Färbung des K_{43} ohne einfarbigen K_5 ?

<http://www1.combinatorics.org/Surveys/ds1/sur.pdf>

SAT-Modell für Peg-Solitaire

- Spielzug: Stein überspringen und entfernen,
Aufgabe: existiert Zugfolge von Initial (gegeben)
zu Final (genau ein Stein übrig)
- Beispiel: Initial ist volles Rechteck minus ein Stein
- (Wdhlg.) aussagenlog. Modell als Folge von Relationen
 $[B_0, B_1, \dots, B_n]$, für die gilt $\forall k : \text{step}(B_k, B_{k+1})$.
- eine Realisierung: $\text{move}(S, m, d, T)$ mit: S, T Brett,
 m Position (des übersprungenen Steins), d Richtung
und $\text{step}(S, T) = \bigvee_{m,d} \text{move}(S, m, d, T)$
- Finalität von B_n ist einfach: das Zählen muß nicht
SAT-kodiert werden—warum?

Hausaufgaben

WS 25: 2,3,4 (5, 7, 8 siehe Repo)

1. unterschiedliche Halbringe auf zwei Elementen?
2. für die Formel $S(b, h)$ (abhängig von Parametern $b, h \in \mathbb{N}$)

Variablen: $v_{x,y}$ für $1 \leq x \leq b, 1 \leq y \leq h$

Constraints:

- für jedes x gilt: wenigstens einer von $v_{x,1}, v_{x,2}, \dots, v_{x,h}$ ist wahr
- und für jedes y gilt: höchstens einer von $v_{1,y}, v_{2,y}, \dots, v_{b,y}$ ist wahr

- (a) unter welcher Bedingung an b, h ist $S(b, h)$ erfüllbar?
Für den erfüllbaren Fall: geben Sie ein Modell an.

Für den nicht erfüllbaren Fall: einen Beweis.

- (b) Erzeugen Sie (eine konjunktive Normalform für) $S(b, h)$ durch ein Programm (Sprache/Bibliothek beliebig) (b, h von der Kommandozeile, Ausgabe nach stdout)
- (c) Lösen Sie $S(b, h)$ durch minisat (kissat, Z3, ...), vergleichen Sie die Laufzeiten (auch im nicht erfüllbaren Fall).

3. Für $S(b, h)$ (vorige Aufgabe): Formel-Konstruktion, Löser-Aufruf mit Ersatz.

- v vom Typ `Relation Int Int` (vgl. N Damen)
- „für jedes x “: verwenden Sie `rows`
- „für jedes y “: schreiben und verwenden Sie entsprechende Fkt. `columns`
- „höchstens einer“: verwenden Sie „keine zwei“.

4. Für $a, b \geq 2$: die Ramsey-Zahl $R(a, b)$ ist die kleinste Zahl n , für die gilt: jede rot-blau-Kantenfärbung eines K_n enthält einen roten K_a oder einen blauen K_b .

(Der Satz von Ramsey ist, daß es für jedes a, b tatsächlich solche n gibt.)

(a) Beweisen Sie:

i. $R(a, b) = R(b, a)$

ii. $R(2, b) = b$

iii. $R(a + 1, b + 1) \leq R(a, b + 1) + R(a + 1, b)$

(das liefert einen Beweis des Satzes von Ramsey)

iv. wenn dabei beide Summanden rechts gerade Zahlen sind, dann $R(a + 1, b + 1) < \dots$

(b) Bestimmen Sie damit obere Schranken für

$R(3, 3), R(3, 4), R(4, 4)$ und vergleichen Sie mit den

unteren Schranken durch SAT-Kodierung.

5. SAT-Kodierung für $R(a, b)$ mit Ersatz:

- `main = ramsey 3 3 5`

```
ramsey a b n = do
  out <- solveWith (cryptominisat5Path "kissat") $
    r <- R.symmetric_relation ((1,1), (n,n))
    assert $ flip all (subs a [1 .. n]) $ \ c ->
      flip any (subs 2 c) $ \ [x,y] ->
        r R.! (x,y)
    assert $ flip all (subs b [1 .. n]) $ \ c ->
      flip any (subs 2 c) $ \ [x,y] ->
        not $ r R.! (x,y)
  return r
...
```

- **Hilfsfunktion: verteilten Teilfolgen gegebener Länge:**

Beispiel: `subs 3 [1, 2, 3, 4, 5]`

`= [[1, 2, 3], [1, 2, 4], [1, 2, 5], ..., [3, 4, 5]]`

(nicht notwendig in dieser Reihenfolge)

`subs :: Int -> [a] -> [[a]]`

`subs 0 xs = [[]] ; subs k [] = []`

`subs k (x:xs) = map _ _ <> subs k xs`

mit `subs a [1 .. n]` zur Auswahl des K_a sowie

`subs 2 c` zur Auswahl der Kanten.

- Diskutieren Sie Existenz (obere Schranke) und SAT-Kodierung für dreifarbigen Ramsey:

$R(a, b, c) :=$ das kleinste n mit: jeder Kanten-3-Färbung des K_n enthält einen roten K_a oder einen grünen K_b oder einen blauen K_c .

Ergänzen und beweisen: $R(a, b, c) \leq R(a, R(b, c))$,
anwenden für $R(3, 3, 3)$.

6. Modellierung als aussagenlogisches Constraint:

- Rösselsprung (= Hamiltonkreis)

- Norinori

<https://nikoli.com/en/puzzles/norinori/>

- ABCEndView (oder ähnlich)

<https://www.janko.at/Raetsel/AbcEndView/>

Vorgehen bei Modellierung:

- welches sind die Unbekannten, was ist deren Bedeutung?

(Wie rekonstruiert man eine Lösung aus der Belegung, die der Solver liefert?)

- welches sind die Constraints?

(wie stellt man sie in CNF dar? — falls nötig)

7. Unruly (S. Tatham Puzzles)

```
u1 = M.fromList -- eine Aufgabe (8 x 8)
  $ map (,False) [(1,7), (3,2), (5,1), (5,3), (5,4), (5,7),
  <> map (,True) [(1,4), (6,2), (6,3), (7,5), (8,4), (8,7)]
unruly u = do
  let bnd = ((1,1), (8,8))
  out <- solveWith (cryptominisat5Path "kissat") $ do
    b <- R.relation bnd
    assert $ flip all (M.toList u) $ \ (k,v) ->
      b R.! k === encode v
    assert $ flip all (rows b <> columns b) $ \ rc ->
      balanced rc && mixed 3 rc
  ...
```

wie kann man feststellen, daß es genau eine Lösung gibt? (Solver nochmals aufrufen für modifizierte Formel.)

8. Peg (S. Tatham Puzzles)

```
peg b = do
  let bnd = ((1,1), (b,b))
      full = A.genArray bnd $ \ i -> True
      start :: A.Array (Int,Int) Bool
      start = full A.// [((1,2), False)]
  out <- solveWith (cryptominisat5Path "kissat") $ do
    boards <- replicateM (b*b - 1) $ R.relation bnd
    assert $ R.equals (head boards) (encode start)
    assert $ all step $ zip boards $ drop 1 boards
    return boards
  ...
type Board = R.Relation Int Int

step :: (Board, Board) -> Bit
move :: Board -> (Int,Int) -> (Int,Int) -> Board ->
```

Anwendg.: Bounded Model Checking

Begriff, Motivation

- *model checking*: feststellen, ob
 - ein *Modell* eines realen Hard- oder Softwaresystems (z.B. Zustandsübergangssystem f. nebenläufiges Programm)
 - eine *Spezifikation* erfüllt (z.B. gegenseitiger Ausschluß, Liveness, Fairness)
- *symbolic model checking*:
symbolische Repräsentation von Zustandsfolgen
im Unterschied zu tatsächlicher Ausführung (Simulation)
- *bounded*: für Folgen beschränkter Länge

Literatur, Software

- Armin Biere et al.: *Symbolic Model Checking without BDDs*, TACAS 1999, <http://fmv.jku.at/bmc/>
Software damals: Übersetzung nach SAT, später: SMT (QB_BV), Solver: <http://fmv.jku.at/boolector/>
- Daniel Kroening und Ofer Strichman: *Decision Procedures, an algorithmic point of view*, Springer, 2008.
<http://www.decision-procedures.org/>
Software: <http://www.cprover.org/cbmc/>
- Nikolaj Bjørner et al.: *Program Verification as Satisfiability Modulo Theories*, SMT-Workshop 2012,
<http://smt2012.loria.fr/>
Softw.: <https://github.com/Z3Prover/z3/wiki>

BMC für Mutual Exclusion-Protokolle

System mit zwei (gleichartigen) Prozessen A, B :

A0: maybe goto A1

A1: if 1 goto A1 else goto A2

A2: 1 := 1; goto A3

A3: [critical;] goto A4

A4: 1 := 0; goto A0

B0: maybe goto B1

B1: if 1 goto B1 else goto B2

B2: 1 := 1; goto B3

B3: [critical;] goto B4

B4: 1 := 0; goto B0

Schließen sich A3 und B3 gegenseitig aus? (Nein.)

(nach: Donald E. Knuth: TAOCP, Vol. 4 Fasz. 6, S. 20ff)

Modell: Zustandsübergangssystem

Zustände:

- jeder Zustand besteht aus:
 - Inhalte der Speicherstellen (hier: $l \in \{0, 1\}$)
 - Programmzähler (PC) jedes Prozesses (hier: $A \in \{0 \dots 4\}, B \in \{0 \dots 4\}$)
- Initialzustand: $I = \{l = 0, A = 0, B = 0\}$
- Menge der Fehlerzustände: $F = \{A = 3, B = 3\}$

Übergangsrelation (nichtdeterministisch): für $P \in \{A, B\}$:

- P führt eine Aktion aus (schreibt Speicher, ändert PC)

Aussagenlog. Formel für $I \rightarrow^{\leq k} F$ angeben,
deren Erfüllbarkeit durch SAT- oder SMT-Solver bestimmen

One-Hot-Kodierung

- $p \in \{0 \dots 4\}$ symbolisch repräsentieren durch Folge $[p_0, \dots, p_4]$ von symbolischen Wahrheitswerten
... von denen *genau einer* wahr ist
Bsp: $p = 3$ kodiert durch $[0, 0, 0, 1, 0]$.
- das ist die *one hot*-Kodierung (eine Stelle ist *hot* = stromführend)
- eine Realisierung ist $(\bigvee_i p_i) \wedge \bigwedge_{i < j} (\neg p_i \vee \neg p_j)$
- es gibt andere Kodierungen für endliche Bereiche (z.B.: binär: benötigt weniger Unbekannte, aber evtl. größere Formeln)

Übung BMC

- **Software:** `https://git.imn.htwk-leipzig.de/waldmann/boumchak`
- überprüfe 1. gegenseitigen Ausschluß, 2. deadlock, 3. livelock (starvation) für weitere Systeme, z.B.

E. W. Dijkstra, 1965:

`https://www.cs.utexas.edu/~EWD/transcriptions/EWD01xx/EWD123.html#2.1`

G. L. Peterson, *Myths About the Mutual Exclusion Problem*, Information Processing Letters 12(3) 1981, 115–116

Anzahl-Constraints

Definition, Motivation

- $\text{Count}_{\leq k}(x_1, \dots, x_n) := (\sum_i x_i) \leq k.$
- AMO (at most one): = $\text{Count}_{\leq 1}$, entspr. ALO, EXO
- Schubfach-Constraint (als Testfall, erfüllbar gdw. $B \leq H$)
 $\bigwedge_{1 \leq i \leq H} \text{AMO}(x_{ij} | 1 \leq j \leq B) \wedge \bigwedge_{1 \leq j \leq B} \text{ALO}(x_{ij} | 1 \leq i \leq H)$
- Schubfach für $B = H$: dann ist x Permutationsmatrix, repräsentiert Bijektion von $\{1, \dots, B\}$ auf sich
- Anwend.: Rösselsprung, Hamiltonkreis in $G = (V, E)$
Pfad p in G als Bijektion von Indexmenge $\{1, \dots, |V|\}$ in Knotenmenge V mit $\bigwedge_i (p(i), p(i+1)) \in E.$

SAT-Kodierung eines Rösselsprungs

```
let n = height * width; places = [0 .. n-1]

let decode p = divMod p width
    edge p q =
        let (px,py) = decode p; (qx,qy) = decode q
            in 5 == (px-qx)^2 + (py-qy)^2
    rotate (x:xs) = xs <> [x]

a <- replicateM n $ replicateM n $ exists @Bit
assert $ all exactly_one a
assert $ all exactly_one $ transpose a
assert $ flip all (zip a $ rotate a) $ \ (e, f) ->
    flip all places $ \ p -> e!!p ==>
        flip any (filter (edge p) places) (\ q -> f!!q)
```

SAT-Kodierungen von AMO (I) (quad, lin)

- quadratisch: $\text{AMO}(x_1, \dots, x_n) = \bigwedge \{\overline{x_i} \vee \overline{x_j} \mid 1 \leq i < j \leq n\}$

$\binom{n}{2}$ Klauseln, keine zusätzlichen Variablen

- linear: mit Kodierung $\text{enc} : x \mapsto (x_e, x_z) = (x \geq 1, x \geq 2)$:

$0 \mapsto (0, 0), 1 \mapsto (1, 0), 2, 3, \dots \mapsto (1, 1)$

Addition: $(x_e, x_z) \dagger_{\text{enc}} (y_e, y_z) = \dots$

so daß $\text{enc}(x + y) = \text{enc}(x) \dagger_{\text{enc}} \text{enc}(y)$

$\text{AMO}(x_1, \dots, x_n) = \text{let } (s_e, s_z) = \sum_{\text{enc}} \text{enc}(x_i) \text{ in } \dots$

SAT-Kodierungen von AMO (II) - log

- $\text{AMO}(x) = \exists h : (x_i \Rightarrow (i = h))$

h binär repräsentiert mit $\log n$ Bits.

- Bsp. $\text{AMO}(x_0, \dots, x_3) = \exists h_1, h_0 : \begin{array}{l} (\bar{x}_0 \vee \bar{h}_1) \wedge (\bar{x}_0 \vee \bar{h}_0) \\ \wedge (\bar{x}_1 \vee \bar{h}_1) \wedge (\bar{x}_1 \vee h_0) \\ \wedge (\bar{x}_2 \vee h_1) \wedge (\bar{x}_2 \vee \bar{h}_0) \\ \wedge (\bar{x}_3 \vee h_1) \wedge (\bar{x}_3 \vee h_0) \end{array}$

- $n \log n$ Klauseln, $\log n$ zusätzliche Variablen
- die Hilfsvariablen h_0, h_1 sind keine Funktionen der Eingangsvariablen. (wenn alle x_i falsch, dann h_i beliebig)
- Ü: man kann eine Skolem-Funktion trotzdem einfach angeben

SAT-Kodierungen von AMO (III) - sqrt

- für $\text{AMO}(x)$: die x in einem Rechteck anordnen,

$$z_i := \bigvee_j x_{ij} \text{ (Zeile } i), s_j := \bigvee_i x_{ij} \text{ (Spalte } j),$$

$$\text{dann } \text{AMO}(x) = \text{AMO}(z) \wedge \text{AMO}(s).$$

Jingchao Chen: *A New SAT Encoding of the At-Most-One Constraint* 10th Workshop Constraint Modeling and Reformulation, 2010 <https://www.it.uu.se/research/group/astra/ModRef10/programme.html>

- Formelgröße $f(n) = \Theta(n) + 2f(\sqrt{n})$, mit $\Theta(\sqrt{n})$ Hilfsvar.

Lineare Faktoren sind klein. Ü: wenn `assert (amo xs)`, kann man einige Klauseln weglassen. Welche?

Listen (Generatoren) zur Programmablaufsteuerung

- `amo_quad bs = and $ do`
 `(b : cs) <- L.tails bs; c <- cs`
 `return $ not b || not c`
- **benutzt**
`L.tails [1,2,3] = [[1,2,3], [2,3], [3], []]`
- **erster (äußerer) Generator** `(b : cs) <- L.tails bs`
es werden der Reihen nach gebunden:
`b=1, cs=[2,3]`, dann `b=2, cs=[3]`, dann `b=3, cs=[]`.
- **zweiter (innerer) Generator** `c <- cs`
für `b=1, cs=[2,3]` werden der Reihe nach gebunden:
`c=2`, dann `c=3`.

Matrizen als Listen von Listen

- `x :: [[Bit]] <- replicateM h $ replicateM b $ exists @Bit`

- `replicateM`: wiederholt Aktion, liefert Liste der Resultate

`exists @Bit`: liefert eine neue Variable

`replicateM b $...`: liefert b Variablen (= eine Zeile)

`replicateM h $...`: liefert h Zeilen (= eine Matrix)

- `L.transpose` transponiert Matrix

`L.transpose [[1,2,3],[4,5,6]] = [[1,4],[2,5],[3,6]]`

Quantifikation über Listen

- Anw.: `flip all x a lo`, bedeutet $\forall e \in x : a \text{ lo}(e)$
- benutzt Funktionen

```
flip :: (a -> b -> c) -> b -> a -> c
```

```
all :: (Boolean b, Foldable t)
```

```
    => (a -> b) -> t a -> b
```

dabei t ein Containertyp (über den man iterieren kann)

das `flip` nur, damit Argumentreihenfolge wie bei Quantor bzw. Zählschleife (erst Bereich, dann Prädikat)

- eine mögliche Implementierung dafür ist

```
all f x = and $ do e <- x ; return f e
```

- entsprechend `any` für endlichen Existenz-Quantor

Iterierte assoziative Verknüpfung

- **Anw.:** `foldMap enc bs`, realisiert $\sum_{b \in bs} enc(b)$
wobei „Addition“ die durch `(<>)` definierte Verknüpfung ist (mit neutralem Element `mempty`)
- **benutzt Funktion und Typklassen**

```
foldMap :: (Foldable t, Monoid m)
  => (a -> m) -> t a -> m
```

```
class Semigroup a where
  (<>) :: a -> a -> a
```

```
class Semigroup a => Monoid a where
  mempty :: a
  mappend :: a -> a -> a
```

- **vgl.** <https://www.imn.htwk-leipzig.de/~waldmann/etc/list/> (**Lists that are really not there**)

Aufgaben

WS 25: 2, 7, 11

1. Modellierung Sudoku: Kodierung eines (ausgefüllten) Schemas durch $9 \times 9 \times 9$ unbekannte Bit mit $u_{x,y,z} \iff$ auf Koordinate (x, y) steht Zahl z .
für welche Teilmengen gelten EXO-Constraints?
Hinweis: für $9^2 + 3 \cdot 9^2$ Mengen, jede mit 9 Elementen.
2. geben Sie Beispiele aus Rätsel-Sammlungen von Nikoli, Janko, Tatham, bei deren SAT-Kodierung AMO- oder EXO-Constraints vorkommen sollten
3. Vergleichen Sie die *commander*-Kodierung für AMO von Klieber und Kwon, Int. Workshop Constraints in Formal Verification, 2007, mit Kodierungen aus dem Skript.
a) auf dem Papier, b) praktisch: mit ersatz

implementieren, Formelgrößen messen, auch nach Vorverarbeitung durch minisat

4. für AMO-log: geben Sie eine Skolem-Funktion für

$\forall x_0, \dots \exists h_0, \dots : \dots$ an.

d.h., die eine erfüllende Belegung der h_i bestimmt, falls $\text{AMO}(x_0, \dots)$ wahr ist.

5. für AMO über 2^k Argumente: verwenden Sie

sqrt-Kodierung für Rechteck mit Abmessungen $2 \times 2^{k-1}$, dann rekursiv über 2^{k-1} .

Vergleichen Sie mit der log-Kodierung.

6. für AMO-lin: untersuchen Sie den Unterschied zwischen der Verwendung von `foldr` (von rechts) und `foldb` (balanciert)

welche Maße der erzeugten Formel stimmen überein,

welche unterscheiden sich?

7. vergleichen Sie Formelgrößen und Solver-Laufzeiten für unterschiedliche AMO-Kodierungen für die Spalten in den unlösbaren Schubfach-Formeln $S(n + 1, n)$.
8. für die AMO-Kodierungen linear und sqrt: wie kann man mit möglichst wenig Zusatz-Aufwand EXO erhalten?
9. ordnen Sie die EXO-Kodierung im Bounded Model Checker (boumchak) in die Systematik der VL ein.
10. Für den Rösselsprung (Code in `fmw-24/leap`)
 - gegebene EXO-Implementierung diskutieren, durch andere ersetzen, Formelgröße/Solverlaufzeit beobachten
 - zusätzliches `assert $ a !! 0 !! 0` diskutieren
 - andere Sprung-Distanzen (Giraffe usw., siehe unten)

- Kamel (1,3)-Sprung (auf nur schwarzen? oder weißen?) Feldern implementieren
- andere Kodierung für Hamiltonkreis: Neng-Fa Zhou: *In Pursuit of an Efficient SAT Encoding for the Hamiltonian Cycle Problem*, CP 2020, ModRef 2019, <https://www.sci.brooklyn.cuny.edu/~zhou/>
- Anwendung: George Jelliss: *Leapers at Large*, 2001, 2022 <http://www.mayhematics.com/t/pl.htm>
 Resultate zur Existenz von Hamilton-Pfaden (HP) und -Kreisen (HC) nachrechnen und ergänzen, Bsp.:
 - HP für Giraffe (1, 4) auf 11×11 ?
(Nein—unsat nach 6 Stunden)
 - HP für Zebra (2, 3) auf 13×13 ?
 - HP für Antilope (3, 4) auf 20×20 ?

vgl. Donald Knuth: *Leaper Graphs*, 1994,

<https://arxiv.org/abs/math/9411240>

1. Plan SAT-Kodierung Skyline (Towers, Wolkenkratzer) Jede Zahl (jedes Haus) wird one-hot-kodiert, dann

```
type Haus=[Bit];type Zeile=[Haus];type Stadt=[Zeile
```

- (ohne Vorgaben) welche AMO-Constraints gelten? Eine Matrix, die diese Constraints erfüllt, heißt *lateinisches Quadrat* (Anzahl der lateinischen Quadrate:

<https://oeis.org/A002860>)

- (mit Vorgaben) Implementieren Sie die Berechnung der (von links) sichtbaren Dächer einer Zeile

`sicht :: Zeile -> [Bit]`. Das Resultat soll der Bitvektor s sein mit $s_i \iff$ Dach der Höhe i ist sichtbar.

- für die anderen Blickrichtungen: die Funktion `sicht`

bleibt, die Stadt wird transformiert. Wie?

- Benutzen Sie eine SAT-Kodierung, um zu beweisen: jedes lateinische Quadrate der Seitenlänge 6 enthält wenigstens eine Zelle, die von keiner Seite sichtbar ist.

2. SAT-Kodierungen für das N -Damen-Problem:

- naiv N^4

für alle a aus Spielfeld: für alle b aus Spielfeld: sich a und b sehen, dann $\neg a \vee \neg b$

- besser N^3

für alle a aus Spielfeld: für alle b , die von a gesehen werden: ...

- noch besser N^2

AMO-Constraints (jeweils lineare Formelgröße) über geeigneten (linear vielen) Mengen

SAT: Normalformen, Transformationen

Normalformen (DNF, CNF)

Definitionen:

- Variable: v_1, \dots Literal: v oder $\neg v$
- DNF-Klausel: Konjunktion von Literalen
- DNF-Formel: Disjunktion von DNF-Klauseln
- CNF-Klausel: Disjunktion von Literalen
- CNF-Formel: Konjunktion von CNF-Klauseln

Disjunktion als Implikation: diese Formeln sind äquivalent:

- $(x_1 \wedge \dots \wedge x_m) \rightarrow (y_1 \vee \dots \vee y_n)$
- $(\neg x_1 \vee \dots \vee \neg x_m \vee y_1 \vee \dots \vee y_n)$

Äquivalenzen

- Def: Formeln F und G heißen *äquivalent*, wenn $\text{Mod}(F) = \text{Mod}(G)$.
- Satz: zu jeder Formel F existiert äquivalente Formel G in DNF.
- Satz: zu jeder Formel F existiert äquivalente Formel G' in CNF.
- aber . . . wie groß sind diese Normalformen?

Erfüllbarkeits-Äquivalenz

- Def: F und G *erfüllbarkeitsäquivalent*, wenn $\text{Mod}(F) \neq \emptyset \iff \text{Mod}(G) \neq \emptyset$.
- Satz: es gibt einen Polynomialzeit-Algorithmus, der zu jeder Formel F eine erfüllbarkeitsäquivalente CNF-Formel G berechnet.
- (Zeit \geq Platz, also auch $|G| = \text{Poly}(|F|)$)
- Beweis (folgt): Tseitin-Transformation
- Vor-Überlegung: warum gibt es keine vergleichbare Aussage für DNF?

Tseitin-Transformation

- Grigori Tseitin, *On the Complexity of Derivation in propositional calculus*, 1966
- Spezifikation:
 - Geg.: F , ges.: erfüllbarkeitsäquivalentes G in CNF.
 - wir verschärfen das zu: $\text{Var}(F) \subseteq \text{Var}(G)$ und
$$\forall b : b \models F \iff \exists b' : b \subseteq b' \wedge b' \models G.$$
- Plan:
 - für jeden nicht-Blatt-Teilbaum T des Syntaxbaumes von F eine zusätzliche Variable n_T einführen,
 - so daß $\forall b' \in \text{Mod}(G) : \text{val}(n_T, b') = \text{val}(T, b)$.
- Realisierung:
 - (Bsp.) $T = L \vee R$, dann $n_T \leftrightarrow (n_L \vee n_R)$ als CNF
 - für jeden der $|F|$ Knoten: ≤ 8 Klauseln mit 3 Literalen

Tseitin-Transf. für Schaltkreise

- beschriebenes Verfahren funktioniert ebenso für Schaltkreise (azyklische gerichtete Graphen, mit Booleschen Operatoren markiert)
- Schaltkreis entsteht aus Baum durch Identifikation (sharing) von Knoten
- für jeden Knoten eine neue Variable angelegen und deren Wert lokal durch eine CNF bestimmen
- Ersatz realisiert *observable sharing* durch Adress-Vergleich von Syntaxbaumknoten (des Typs `Bit`)

```
-- (xor a b) wird nur einmal T-transformiert:  
let {s = (xor a b)} in (c || s ) && (not c || not
```

Visualisierung von Schaltkreisen

- mit `https://git.imn.htwk-leipzig.de/waldmann/ersatz-viz`
- Beispiel

```
import qualified Ersatz.Bit.Display as D
```

```
D.display $ do
```

```
  a <- exists ; b <- exists; c <- exists
```

```
  return $ let {s = (xor a b)}
```

```
            in (c || s ) && (not c || not s)
```

- zu zeichnende Formel steht nach `return`

Tseitin-Transformation in Ersatz (Bsp)

- so ausprobieren:

```
ghci> runSAT' $ do
  x <- exists; y <- exists @Bit; assert (x === y)
  (((), SAT 4 ((1) & (-4 | -3 | -2) & (-4 | 2 | 3)
    & (-2 | 3 | 4) & (-3 | 2 | 4) & (-4)) mempty)
```

- zu transformierende Formel steht nach `assert` (nicht: nach `return`, wie bei `display`)

Tseitin-Transformation nach jedem `assert` (ohne `assert`: keine Ausgabe einer CNF)

- Aufgabe: damit observable sharing bestätigen
- Aufgabe: CNF zu `assert (x /= (y /= z)) (XOR)`

Abmessungen von Schaltkreisen

- (die Frage ist *nicht*, ob AMD Epyc in SP5-Sockel paßt)
- Def. Schaltkreis (circuit): gerichteter kreisfreier markierter Graph, als symbolische Repräsent. einer Booleschen Fkt.
- Def. Abmessung (complexity):
 - Def. Größe (size): Anzahl der Knoten
 - * bei Hardware: Material-Aufwand
 - * für SAT-Kodierung: (nach Tseitin-T.) Größe der CNF
 - Def. Tiefe (depth): Länge eines längsten Pfades
 - * bei (paralleler!) Hardware: Rechenzeit
 - * bei SAT-Kodierung: Länge von Abhängigkeitsketten von (Hilfs)variablen, beeinflußt Solver-Laufzeit
- Bsp: XOR N -stellig: $O(N)$ Größe, $O(\log(N))$ Tiefe

Abmessungen von Schaltkreisen: wozu?

- SAT-Kodierung: man möchte immer die (Teil)Formel, für die der Solver am schnellsten die Erfüllbarkeit entscheiden kann
- das läßt sich sehr schwer vorhersagen, abhängig von
 - Lösungsverfahren auf Teilformeln
 - nicht-lokale Kombination von Teil-Lösungen
- betrachten stattdessen die Größe der Eingabe (CNF, Größen sind: Anzahl Variablen, Anzahl Klauseln) in vielen Publikationen so durchgeführt (siehe AMO)
- ... stattdessen Größe eines Schaltkreises
 - mit beschränktem Eingangsgrad (z. B. 2)
 - unbeschränkt nur f. Disjunktion, Konjunktion

Aufgaben zu Tseitin-Transformation

1. für diese Formeln:

- $(x_1 \leftrightarrow x_2) \leftrightarrow (x_3 \leftrightarrow x_4)$
- Halb-Adder (2 Eingänge x, y , 2 Ausgänge r, c)
 $(r \leftrightarrow (\neg(x \leftrightarrow y))) \wedge (c \leftrightarrow (x \wedge y))$
- Full-Adder (3 Eingänge, 2 Ausgänge)

jeweils:

- führe die Tseitin-Transformation durch
- gibt es kleinere erfüllbarkeitsäquivalente CNF? (deren Modelle Erweiterungen der Original-Modelle sind)

2. `data Bit` hat weitere Konstruktoren (Xor, Mux).

Wo werden diese benutzt?

Helfen sie tatsächlich bei der Erzeugung kleiner CNFs?

SAT-Solver

Spezifikation

- Eingabe: eine Formel in CNF

[[1],[-4,-3,11],[-11,3],[-11,4],[-11],[3,4,13],[-13,-3],[-13,-4],[-2,12,13],[-12,2],[-13,-12],[-12],[-7,-6,14],[-14,6],[-14,7],[-14],[6,7,16],[-16,-6],[-16,-7],[-5,15,16],[-15,5],[-16,-15],[-15],[-10,-9,17],[-17,9],[-17,10],[-17],[9,10,19],[-19,-9],[-19,-10],[-8,18,19],[-18,8],[-19,-18],[-18],[2,5,8,20],[-20,-2],[-20,-5],[-20,-8],[-20],[3,6,9,21],[-21,-3],[-21,-6],[-21,-9],[-21],[4,7,10,22],[-22,-4],[-22,-7],[-22,-10],[-22]]

- Ausgabe:
 - eine erfüllende Belegung
 - *oder* ein Beweis für Nichterfüllbarkeit

Implementierung eines naiven SAT-Solvers

- benutzt die Schnittstelle aus `ersatz`

```
minisat :: Solver SAT IO
type Solver s m
  = s -> m (Result, IntMap Bool)
data Result
  = Unsolved   | Unsatisfied | Satisfied
class DIMACS t where
  dimacsNumVariables :: t -> Int
  dimacsClauses     :: t -> Seq IntSet
data SAT; instances DIMACS Sat
```

Lösungsverfahren

- vollständige Suche (alle Belegungen, vollst. Binärbaum)
- unvollständige Suche (einige Belegungen)
 - evolutionär (Genotyp = Belegung)
 - lokale Suche (Selman, Kautz, Cohen 1993: Walksat)
- verbesserte vollständige Suche (Erkennen und Abschneiden sinnloser Teilbäume)
DPLL (Davis, Putnam, Logeman, Loveland 1960/61)
- weitere Verbesserungen durch
 - Lernen (und Vergessen!) von zusätzlichen Klauseln
 - Vorverarbeitung zum Entfernen von Variablen
 - Parallelisierung
(Kommunikation mehrerer Suchverfahren)

Evolutionäre Algorithmen für SAT

- Genotyp: Bitfolge $[x_1, \dots, x_n]$ fester Länge
- Phänotyp: Belegung $b = \{(v_1, x_1), \dots, (v_n, x_n)\}$
- Fitness: z. B. Anzahl der von b erfüllten Klauseln
- Operatoren:
 - Mutation: einige Bits ändern
 - Kreuzung: one/two-point crossover?
Problem: starke Abhängigkeit von Variablenreihenfolge

Lokale Suche (GSat, Walksat)

Bart Selman, Cornell University,

Henry Kautz, University of Washington

https:

//web.archive.org/web/20070311005144/http:

//www.cs.rochester.edu/u/kautz/walksat/

Algorithmus:

- beginne mit zufälliger Belegung
- wiederhole: ändere das Bit, das die Fitness am stärksten erhöht

Problem: lokale Optima — **Lösung:** Mutationen.

DPLL

Davis, Putnam (1960), Logeman, Loveland (1962),

<http://dx.doi.org/10.1145/321033.321034>

<http://dx.doi.org/10.1145/368273.368557>

Zustand = partielle Belegung

- *Decide*: eine Variable belegen
- *Propagate*: alle Schlußfolgerungen ziehen
Beispiel: Klausel $x_1 \vee x_3$, partielle Belegung $x_1 = 0$,
Folgerung: $x_3 = 1$
- bei *Konflikt* (widersprüchliche Folgerungen)
 - (DPLL original) Backtrack (zu letztem Decide)
 - (DPLL mit CDCL) Backjump (zu früherem Decide)

DPLL-Begriffe

für partielle Belegung b (Bsp: $\{(x_1, 1), (x_3, 0)\}$): Klausel c ist

- *erfüllt*, falls $\exists l \in c : b(l) = 1$, Bsp: $(\neg x_1 \vee x_2 \vee \neg x_3)$
- *Konflikt*, falls $\forall l \in c : b(l) = 0$, Bsp: $(\neg x_1 \vee x_3)$
- *unit*, falls $\exists l \in c : b(l) = \perp \wedge \forall l' \in (c \setminus \{l\}) : b(l') = 0$,
Bsp: $(\neg x_1 \vee \neg x_2 \vee x_3)$. Dabei ist $l = \neg x_2$ das Unit-Literal.
- *offen*, sonst. Bsp: $(x_2 \vee x_3 \vee x_4)$.

Eigenschaften: für CNF F und partielle Belegung b :

- wenn $\exists c \in F : c$ ist Konflikt für b , dann $\neg \exists b' \supseteq b$ mit $b' \models F$
(d.h., die Suche kann dort abgebrochen werden)
- wenn $\exists c \in F : c$ ist Unit für b mit Literal l , dann
 $\forall b' \supseteq b : b' \models F \Rightarrow b'(l) = 1$
(d.h., l kann ohne Suche belegt werden)

DPLL-Algorithmus

Eingabe: CNF F ,

Ausgabe: Belegung b mit $b \models F$ oder UNSAT.

DPLL(b) (verwendet Keller für Entscheidungspunkte):

- (success) falls $b \models F$, dann halt (SAT), Ausgabe b .
- (backtrack) falls F eine b -Konfliktklausel enthält, dann:
 - falls Keller leer, dann halt (UNSAT)
 - sonst $v := pop()$ und DPLL($b_{<v} \cup \{(v, 1)\}$).dabei ist $b_{<v}$ die Belegung vor $decide(v)$
- (propagate) falls F eine b -Unitklausel c mit Unit-Literal l enthält: DPLL($b \cup \{(variable(l), polarity(l))\}$).
- (decide) sonst wähle $v \notin \text{dom } b$, push(v), und DPLL($b \cup \{(v, 0)\}$).

DPLL: Eigenschaften

- Termination: DPLL hält auf jeder Eingabe
- Korrektheit: wenn DPLL mit SAT hält, dann $b \models F$.
- Vollständigkeit: wenn DPLL: UNSAT, dann $\neg \exists b : b \models F$

wird bewiesen durch Invariante

- $\forall b' : b' \in \text{Mod}(F) \Rightarrow b \leq_{\text{lex}} b'$
(wenn DPLL derzeit b betrachtet, und wenn F ein Modell b' besitzt, dann ist b' unterhalb oder rechts von b)
- dabei bedeutet: $b \leq_{\text{lex}} b'$:
 $b \subseteq b'$ oder $\exists v : b(v) = 0 \wedge (b_{<v} \cup \{(v, 1)\}) \subseteq b'$

Satz (Ü): für alle endlichen V : $<_{\text{lex}}$ ist eine wohlfundierte Relation auf der Menge der partiellen V -Belegungen:

DPLL-Beispiel

$[[2, 3], [3, 5], [-3, -4], [2, -3, -4]$
 $, [-3, 4], [1, -2, -4, -5], [1, -2, 4, -5]]$

decide belegt immer die kleinste freie Variable, immer
zunächst negativ

DPLL-Beispiel (Lösung)

$[[2, 3], [3, 5], [-3, -4], [2, -3, -4]$
 $, [-3, 4], [1, -2, -4, -5], [1, -2, 4, -5]]$

[Dec (-1), Dec (-2), Prop 3, Prop (-4), Back
, Dec 2, Dec (-3), Prop 5, Prop (-4), Back
, Dec 3, Prop (-4), Back, Back, Back
, Dec 1, Dec (-2), Prop 3, Prop (-4), Back
, Dec 2, Dec (-3), Prop 5]

DPLL: Implement., Heuristik, Ergänzungen

- Grundlage ist effiziente Impl. von UP und Konflikt-Erkennung
- Methoden:
 - Wahl der nächsten Entscheidungsvariablen (kommt am häufigsten in aktuellen Konflikten vor)
 - Lernen von Konflikt-Klauseln (erlaubt Backjump)
 - Vorverarbeitung (Variablen und Klauseln eliminieren)
- alles vorbildlich implementiert und dokumentiert in Minisat <http://minisat.se/> (Niklas Een, Niklas Sorenson) (seit ca. 2005 sehr starker Solver)
später übernimmt diese Rolle: Cadical, Kissat <https://fmv.jku.at/kissat/> (Armin Biere)

Aufgaben

1. Geben Sie eine erfüllbare CNF an, für die monotone lokale Suche

```
improve n cnf b0 = ...
  let b1 = S.insert (negate l) $ S.delete l b0
  if badness cnf b1 <= badness cnf b0
    then improve n cnf b1
    else improve n cnf b0
```

nicht funktioniert: es gibt eine Belegung b_0 , von der aus *keine* zulässige Schrittfolge zu einer erfüllenden Belegung führt. Hinweis: z.B., weil es überhaupt keine erlaubten Schritte gibt.

Wie behandeln gsat/walksat diesen Fall?

2. wenden Sie den SAT-Solver mit lokaler Suche auf

realistische CNFs an, z.B. aus Kodierung Rösselsprung.

3. wie werden in minisat (kissat, ...) Einheits- und Konfliktklauseln erkannt? (Hinweis: two watched literals)

4. unit propagation (UP) implementieren:

- Einheitsklauseln erkennen:

```
type Literal = Int
units :: CNF -> [Literal]
units [[1,2], [-3], [3,4]] = [-3]
```

- ein Literal belegen:

```
assign :: Literal -> CNF -> CNF
assign (-3) [[1,2], [-3], [3,4]] = [[1,2], [4]]
```

Wo stehen die entsprechenden Funktionen im Quelltext der Autotool-Aufgabe zu DPLL?

SAT-Solver (fortgeschrittene Techniken)

Plan

- bisher:
 - (unvollst.) stochastische lokale Suche (gsat, walksat)
 - vollständige Suche (DPLL)
- jetzt und folgend:
 - (jedes Verfahren) beschleunigen durch Vorverarbeitung (preprocessing): Variablen-Elimination
 - DPLL beschleunigen durch Klausel-Lernen (CDCL)
 - Zertifikate für UNSAT
 - gemeinsame Grundlage: Resolution

Semantisches Folgern

- Def: eine Formel F folgt aus einer Formelmenge M , geschrieben $M \models F$, falls $\text{Mod}(M) \subseteq \text{Mod}(F)$.
- Bsp: $\{x_1 \vee \bar{x}_2, x_2 \vee x_3\} \models (x_1 \vee x_3)$,
Beweise (lt. Def.) z.B. durch Vergleich der Wertetabellen
(d.h., explizites Aufzählen der Modellmengen)

Eigenschaften (Übungsaufgaben):

- $M \models \text{True}$
- $(M \models \text{False}) \iff (\text{Mod}(M) = \emptyset)$
- $(M \models F) \iff (\text{Mod}(M \cup \{\neg F\}) = \emptyset)$
- wird bei CDCL benutzt: wir lernen nur Klauseln F , die aus der CNF (Klauselmenge) M folgen:
 $(M \models F) \iff (\text{Mod}(M) = \text{Mod}(M \cup \{F\}))$

Resolution (Syntaktisches Schließen)

- Definition: für Literal l : die *Resolvente* $\text{Res}_l(c, d)$ der Klausel c mit $l \in c$, und der Klausel d mit $\neg l \in d$: ist die Klausel $(c \setminus \{l\}) \cup (d \setminus \{\neg l\})$.
- Bsp. $l = \bar{x}_2$, $c = (x_1 \vee \bar{x}_2)$, $d = (x_2 \vee x_3)$,
 $\text{Res}_l(c, d) = (x_1 \vee x_3)$.
- Satz (Korrektheit): $\{c, d\} \models \text{Res}_l(c, d)$.
- Beweis: für jede Belegung $b \in \text{Mod}(c, d)$:
vollst. Fallunterscheidung: $b(l) = 0$ oder $b(l) = 1$.
- Anwendung: Hinzufügen einer Resolvente ändert die Modellmenge nicht, kann Propagationen ermöglichen
Bsp: CNF $F = \{1\bar{2}3, 2\bar{4}5, \bar{3}5\}$, partielle Bel. $b = \{\bar{1}, 4\}$
 $\text{Res}_{\bar{2}}(1\bar{2}3, 2\bar{4}5) = 13\bar{4}5$, $\text{Res}_3(13\bar{4}5, \bar{3}5) = \dots$ ist Unit

Variablen-Elimination durch vollst. Resolution

- für Formel (Klauselmenge) F und Variable v :
 $\text{Pos}_v(F) = \{c \mid c \in F, v \in c\}$; $\text{Neg}_v(F) = \{c \mid c \in F, \neg v \in c\}$
 $\text{Res}_v(F) = \bigcup_{p \in \text{Pos}_v(F), n \in \text{Neg}_v(F)} \text{Res}_v(p, n)$
- Bsp: $F = \{12, \bar{1}3, 2\bar{3}, \bar{2}4, \bar{3}4\}$,
 $\text{Pos}_3(F) = \{\bar{1}3\}$, $\text{Neg}_3(F) = \{2\bar{3}, \bar{3}4\}$, $\text{Res}_3(F) = \{\bar{1}2, \bar{1}4\}$.
- Satz: F ist erfüllbarkeitsäquivalent zu G
mit $G := F \setminus (\text{Pos}_v(F) \cup \text{Neg}_v(F)) \cup \text{Res}_v(F)$.
- Bsp (fortgesetzt) $G = \{12, \bar{2}4, \bar{1}2, \bar{1}4\}$.
- Beweis (aus $b' \models G$ konstruiere $b \models F$)
das b ist eines von $b_0 := b' \cup \{(v, 0)\}$, $b_1 := b' \cup \{(v, 1)\}$.
Falls $b_0 \not\models \text{Pos}_v(F)$ und $b_1 \not\models \text{Neg}_v(F)$,
dann Widerspruch zu $b' \models \text{Res}_v(F)$.

SAT-Lösen durch iterierte vollst. Resolution

- ein Eliminationsschritt

$$F_i \rightarrow (F_i \setminus (\text{Pos}_v(F_i) \cup \text{Neg}_v(F_i)) \cup \text{Res}_v(F_i)) = F_{i+1}$$

erhält Erfüllbarkeit, verringert Variablenanzahl

- mit $n = |\text{Var}(F)|$: $F \rightarrow^n F_n$ mit $\text{Var}(F_n) = \emptyset$,

also $F_n = \emptyset$ (SAT) oder $F_n = \{\emptyset\}$ (UNSAT)

- ist vollständiges Lösungsverfahren!

aber unpraktisch, weil $|F_{i+1}| \gg |F_i|$ möglich ist: $|F_{i+1}| = |F_i| - (|\text{Pos}_v(F_i)| + |\text{Neg}_v(F_i)|) + |\text{Pos}_v(F_i)| \cdot |\text{Neg}_v(F_i)|$.

- nur solange, wie $|F_{i+1}| \leq |F_i| + \Delta$ (für passende Variable)

Een und Biere: *Effective Preprocessing ...*, SAT 2005,

<http://minisat.se/downloads/SatELite.pdf>

dort weitere Vorverarbeitungs-Verfahren

DPLL mit CDCL (Plan)

conflict driven clause learning –

bei jedem Konflikt eine Klausel C hinzufügen, die

- aus der Formel folgt (d.h. Modellmenge nicht ändert)
- den Konflikt durch Propagation verhindert

Eigenschaften/Anwendung:

- danach *backjump* zur vorletzten Variable in C .
(die letzte Variable wird dann propagiert, das ergibt die richtige Fortsetzung der Suche)
- C führt hoffentlich auch später zu Propagationen, d.h. Verkürzung der Suche
- ... wenn nicht: gelernte Klauseln kann man auch vergessen

Naives Lernen

- beim Lösen der Formel (Klauselmenge) F :
partielle Belegung b , die eine Konfliktklausel hat,
wurde durch Entscheidungen und Propagationen erreicht
 $B :=$ die Konjunktion dieser Entscheidungsliterale
- man kann die Klausel $C = \neg B'$ = Disjunktion der
negierten Entscheidungsliterale lernen (hinzufügen)
es gilt $\text{Mod}(F \cup B) = \emptyset$, also $F \models C$
backjump für dieses C ist *backtrack* (die letzte
Entscheidung wird durch UP mit C negiert)
- dieses C enthält mglw. Literale, die am Konflikt nicht
beteiligt sind, deswegen besser (folgende Folie)

Lernen (Implementierung) und Backjump

- resolviere Konfliktklausel mit den Kl., die seit der letzten Entscheidung für Propagationen verwendet wurden.
- `c := Konflikt-Klausel // für aktuelle Bel. b`
`while (...) { // inv: für alle l' in c : $b(l')=0$`
 `l := das in c zuletzt belegte Literal`
 `d := Unit-Klausel, durch die $\text{var}(l)$ belegt wurde`
 `c := resolve_l (c, d); }`

diese Resolution ist immer möglich, denn es gilt

$$b(l) = 0, l \in c \text{ (ist Konflikt)}, b(\bar{l}) = 1, \bar{l} \in d \text{ (ist Unit)}$$

- Schleife verlassen (und c lernen), wenn c nur noch *ein* Literal l_h der aktuellen Entscheidungstiefe enthält.
- dann Backjump zu nächst-höherer Entscheidung in c .
von dort wird l_h unit-propagiert.

Beweise der Nichterfüllbarkeit

- für CNF F : Antwort SAT hat Zertifikat (Belegung b), das in Polynomialzeit überprüft werden kann (nachrechnen, daß $b \models F$)
- wie kann man Antwort UNSAT überprüfen, ohne die gesamte (erfolglose) Suche zu wiederholen?
- ein Zertifikat für UNSAT ist eine Resolutionsableitung der leeren Klausel, ein solches muß existieren, denn es gilt Satz (Widerlegungsvollständigkeit der Resolution):
für jede CNF F : $\text{Mod}(F) = \emptyset \iff F \vdash \emptyset$.
- alle Schritte von DPLL und CDCL können durch Resolutions-Schlüsse begründet werden

Widerlegungsvollständigkeit der Resolution

- Satz für jede CNF F : $\text{Mod}(F) = \emptyset \iff F \vdash \emptyset$.
- Beweis durch Induktion nach Variablenanzahl $|\text{Var}(F)|$
 - Anfang: $\text{Var}(F) = \emptyset$, dann $F = \{\emptyset\}$
 - Schritt: wähle beliebig $v \in \text{Var}(F)$.
 - * F UNSAT, also auch $F_0 = F[v := 0]$ (v belegen und Formel vereinfachen) UNSAT, nach I.V. gilt $F_0 \vdash \emptyset$ zu jeder Klausel der Ableitung $F_0 \vdash$ das Literal $\neg v$ hinzufügen: ergibt Ableitung $F \vdash \{\neg v\}$
 - * entstr. $F_1 = F[v := 1]$ ist UNSAT, ... $F \vdash \{v\}$
 - * zusammensetzen zu $\{\{\neg v\}, \{v\}\} \vdash \emptyset$
- auf diese Weise kann ein DPLL-Solver ein UNSAT-Zertifikat aus erfolglosem Decide-Knoten herstellen

Die Größe von Widerlegungen

- es gibt CNF $F \in \text{UNSAT}$, für die jede Resolutions-Ableitung von \emptyset exponentiell lang ist
Armin Haken, TCS 1985, <https://www.sciencedirect.com/science/article/pii/0304397585901446>
- hätte jedes $F \in \text{UNSAT}$ eine polynomiell lange Ableitung von \emptyset , dann wäre $\text{UNSAT} \in \text{NP}$, also $\text{SAT} \in \text{co-NP}$, also $\text{SAT} \in \text{NP} \cap \text{co-NP}$,
das glaubt die Fachwelt nicht, denn für jedes $L \in \text{NP} \cap \text{co-NP}$ wurde schließlich $L \in \text{P}$ gezeigt
(Bsp: Agraval et al.: *PRIMES is in P*, 2004)
- das kann für SAT nicht passieren, denn $\text{SAT} \in \text{NP}^c$, und dann wäre $\text{P} = \text{NP}$, das glaubt kaum jemand.

Aufgaben

1. (Knuth Aufgabe 254) Für $\{12, \bar{1}3, 2\bar{3}, \bar{2}4, \bar{3}4\}$: nach der Entscheidung 1: welche Klausel wird gelernt?
D. E. Knuth, TAOCP Vol. 4, Fasc. 6, *Satisfiability*, 2015.
2. für die vorige CNF: lösen durch iterierte vollständige Elimination
3. Schleife verlassen, wenn c nur noch ein Literal der aktuellen Entscheidungstiefe \dots :
 - (a) wieso ist die Bedingung anfangs falsch? (es kann nicht sein, daß die originale Konflikt-Klausel c gelernt wird)
 - (b) wieso wird diese Bedingung wahr? (es kann nicht sein, daß es immer ≥ 2 solche Literale sind oder plötzlich gar keines)

(c) wieso wird immer eine Klausel gelernt, die bisher nicht zur Klauselmenge gehört?

4. (Knuth Aufgabe 378) *blocked clause elimination*:

Für Klauselmenge F : eine Klausel $C = (l \vee l_1 \vee \dots \vee l_k)$ heißt *blockiert durch Literal* l ,

falls für jede Klausel $D \in F$ mit $\bar{l} \in D$ gilt: $\exists i : \bar{l}_i \in D$.

(a) ein Beispiel angeben.

(b) beweisen: $F \setminus \{C\}$ erfüllbar $\Rightarrow F$ erfüllbar.

(c) ist jedes Modell b von $F \setminus \{C\}$ auch ein Modell von F ?

5. hier besprochene Heuristiken und Ergänzungen

- in den Protokollen von minisat, kissat wiederfinden (z.B. wieviele Klauseln werden gelernt? wie groß sind diese? wann vergessen?)
- durch Optionen von kissat o. cadical an/abschalten

6. *conflict clause minimization* (Sörenson, SAT 2005, http://minisat.se/downloads/MiniSat_v1.13_short.pdf):
Ein Beispiel angeben.
7. einen SAT-Solver, der UNSAT-Beweise ausgeben kann, auf Schubfach-Formeln anwenden und Beweis-Länge betrachten. (Diese sollte exponentiell ansteigen.)
8. *reverse unit propagation* (RUP) für CNF M , Klausel C
 $\bar{C} :=$ die Menge der Negationen der Literale aus C ,
 $(M \cup \bar{C}) \vdash_U \emptyset$, dabei nur unit-propagations-Schritte.
- (a) Beweisen: dann folgt $M \models C$. Beispiel angeben.
- (b) Beispiel angeben für eine gültige Folgerung, die so *nicht* beweisbar ist (UP-Schritte reichen nicht aus)
- NB: in autotool-Aufgabe zu CDCL wird für (behauptete) gelernte Klausel geprüft, ob sie durch RUP folgt.

Bit-Blasting für Arithmetik

Bausteine für Zahlenrechnungen

- `halfAdder`

```
:: Bit -> Bit -> (Bit, Bit)
```

```
halfAdder x y = (xor x y, x && y)
```

- `fullAdder`

```
:: Bit -> Bit -> Bit -> (Bit, Bit)
```

```
fullAdder x y z =
```

```
  let (s1, c1) = halfAdder x y
```

```
      (s2, c2) = halfAdder _ _
```

```
  in  (_, _)
```

Praktische Eigenschaften von Kodierungen (I)

- für CNF F auf Variablen $V = \{v_1, \dots, v_n\}$:

Definition: F erkennt Widersprüche durch UP (unit prop.):

für jede partielle Belegung b gilt:

wenn keine vollst. Belegung $b' \supseteq b$ existiert mit $b' \models F$,

dann führt UP auf F von b aus zu einem Konflikt

- Bsp: $F = \{123, \overline{12}, 1\overline{23}, 2\overline{3}, \overline{23}\}$, $b = \{\overline{1}\}$.
- Ü: gilt diese Eigensch. für Produkt-Kodierung von AMO?

Zu betrachten ist, ob für jede Belegung

$b = \{(x_i, 1), (x_j, 1)\}$ durch UP ein Konflikt erreicht wird.

Praktische Eigensch. (II) – Forcing

- für CNF F auf Variablen $V = \{v_1, \dots, v_n\}$ und evtl. Hilfsvariablen H :

Def.: F ist (*generalized*) *arc-consistent (GAC) (forcing)*:

für jede partielle Belegung b mit $\text{dom } b \subseteq V$

und jedes $v \in V$ mit $v \notin \text{dom } b$:

wenn v in allen Modellen $b' \supseteq b$ von F den gleichen Wert hat, dann folgt dieser Wert bereits durch UP.

- Bsp: Produkt-Kodierung von $\text{AMO}(x)$: Betrachte $b = \{(x_i, 1)\}$.

Alle anderen x_j müssen dann falsch sein.

Wird das durch UP erreicht?

Aufgaben

1. Ist die Kodierung des Halb-Addierers $\text{HA}(x, y; c, r)$ durch $(r \leftrightarrow x \oplus y) \wedge (c \leftrightarrow x \wedge y)$ (Tseitin-Kodierung ohne weitere Hilfsvariablen) forcing?

Ist die Kodierung des Voll-Addierers $\text{FA}(x, y, z; c, r)$ durch $\text{HA}(x, y; c_1, r_1) \wedge \text{HA}(r_1, z; c_2, r) \wedge (c \leftrightarrow c_1 \vee c_2)$ forcing?

Desgl. für die Kodierung von $\text{ITE}(i, t, e; x)$ (if-then-else) durch $(i \wedge t \rightarrow x) \wedge (i \wedge \bar{t} \rightarrow \bar{x}) \wedge (\bar{i} \wedge e \rightarrow x) \wedge (\bar{i} \wedge \bar{e} \rightarrow \bar{x})$

Lesen Sie dazu auch Een und Sörenson: *Translating Pseudo-Boolean Constraints into SAT*, JSAT 2006, (<http://minisat.se/Papers.html>) Abschnitt 5.1.

Vergleichen Sie mit den Quelltexten von ersatz.

Binärzahlen, Vergleich, Addition

- in Ersatz: `data Bits = Bits [Bit]`, beginnt mit LSB!
- Ordnung (Ü: fehlende Fälle in `eq`, Ü: `Orderable`)

```
instance Equatable Bits where
  Bits xs == Bits ys = eq xs ys where
    eq [] [] = true
    eq (x:xs) (y:ys) = (x == y) && eq xs ys
```

- Addition (Ü: fehlende Fälle in `add`) Formelgröße linear

```
instance Num Bits where
  Bits xs + Bits ys = Bits $ add false xs ys where
    add cin (x:xs) (y:ys) =
      let (s,cout) = fullAdder cin x y
      in s : add cout xs ys
```

Anw.: Rösselsprung (alternative Kodierung)

- Neng-Fa Zhou: ... Efficient SAT Encoding for the Hamiltonian Cycle Problem, CP 2020, ModRef 2019, <https://www.sci.brooklyn.cuny.edu/~zhou/>
- für jedes Feld p des Schachbretts eine unbekannte Zahl t_p , die den Zeitpunkt angibt, zu dem p besucht wird.
- Hamiltonkreis/pfad in $G = (V, E)$ wird erzwungen durch
$$\forall p \in V : \exists q \in V : pq \in E \wedge (t_p + 1 = t_q)$$

(Vorsicht: das ist nur die Idee, muß modifiziert werden)
- t_p binär repräsentiert, Nachfolger binär implementiert.
- im Vgl. zu Kodierung mit Permutationsmatrix (früher): weniger Variablen/Klauseln, aber schwerer lösbar

Binäre Multiplikation

$$[x_0, \dots]_2 \cdot [y_0, \dots]_2 = [z_0, \dots]_2,$$

- Schulmethode: $z = \sum 2^i \cdot x_i \cdot y$ (sequentielle Summation)
- Verbesserungen: C.S. Wallace (1964), L. Dadda (1965),
benutze *full-adder* für *verschränkte* Summation,
(ähnlich zu carry-save-adder)
verringert Anzahl der Gatter und Tiefe des Schaltkreises
vgl. Townsend et al.: *A Comparison of...*, 2003 <http://www.cerc.utexas.edu/~whitney/pdfs/spie03.pdf>,
- scheint für CNF-Kodierung wenig zu helfen
Testfall: Faktorisierung.

Anwendung: Anzahl-Constraints

- wir kennen: exactly/at-most-one
- wir wünschen: exactly/at-most- k
- (jetzt) triviale Lösung: die Bits binär addieren
aber in der passenden Klammerung (balancierter Additionsbaum), damit die Bitbreite des Resultats vernünftig ist
so realisiert in `Ersatz.Counting`
- geht das besser? (\Rightarrow Bachelorarbeit (wenigstens))

Ausblick: Prädikatenlogik, SMT

- Binärkodierung ist ein Beispiel für *bit blasting*
- dieses Verfahren zerstört semantische Information, diese muß (vom SAT-Solver) teuer rekonstruiert werden, Bsp.

```
x <- unknown 10; y <- unknown 10
```

```
assert $ x * y /== y * x -- > 1 min für UNSAT
```

- semantische Eigenschaften (Assoz., Kommut., Distr.) arithemischer Operatoren können durch Solver nur verwendet werden, wenn Constraint-System diese Operatoren tatsächlich benutzt
- dazu wird Prädikatenlogik benötigt (boolesche Kombination von atomaren Constraints, diese sind Relationen auf Termen)

Aufgaben

WS25: 3,4,6,7, 8

1. zu „alternative Kodierung Rösselsprung“: welches Constraint-System entsteht, wenn die unbekanntes t_p one-hot-kodiert werden?
2. ist Implementierung von `atmost-1`, `atmost-k` durch binäre Addition forcierend?
3. für Binärzahlen (Typ `Bits`): implementieren Sie
 - `min, max`
 - Vorgänger (`:: Bits -> (Bits, Bit)`), zusätzlich ein Bit, das Unterlauf anzeigt, d.h., falls `Argument = 0`)
 - Auswahl (if-then-else
`:: Bit -> Bits -> Bits -> Bits`)

4. Kodierung von vorzeichenbehafteten Zahlen (mit variabler Bitbreite): zur Basis -2, Bsp.

$$-5 = 1 - 2 + 4 - 8 = 1 \cdot (-2)^0 + 1 \cdot (-2)^1 + 1 \cdot (-2)^3 + 1 \cdot (-2)^3$$

Instanzen für Ersatz (Codec, Equatable, Orderable, Num)

5. benutzen Sie `microsat` (Repo dieser VL) (evtl. passend modifiziert), um experimentell zu überprüfen:

- Konflikt wird durch UP erkannt?

```
x <- unknown 10; y <- unknown 10
assert $ x + y /= y + x
```

- Lösung wird durch UP bestimmt?

```
x <- unknown 10;
assert $ encode 13 + x == encode 31
```

ähnlich für Multiplikation

6. Implementieren Sie die Multiplikation für Binarzahlen

nach der Schulmethode. (Anders als `plus`!) Die Funktion `times` ist rekursiv und benutzt keine lokale Hilfsfunktion.

```
times :: Boolean b => [b] -> [b] -> [b]
times [] ys = _
times (x:xs) ys = _
```

```
instance Num Nat where
```

```
...
```

```
Nat xs * Nat ys = Nat (times xs ys)
```

Anwendungen:

- Finden Sie eine ganzzahlige Lösung von

$$x^2 - Dy^2 = 1 \wedge y \neq 0$$

für: $D = 5$, $D = 13$, $D = 61$, $D = 71$.

- Finden Sie eine (oder die kleinste) natürliche Zahl, die auf zwei verschiedene Weisen als Summe von Kuben darstellbar ist.

7. Implementieren Sie die Funktion

`sumBits :: [Bit] -> Nat` mit der Eigenschaft:
`sumBits xs` ist die (Binärdarstellung der) Anzahl der wahren Bit aus `xs`.

Das geht offensichtlich mit `foldl` oder `foldr`, aber ein balanciertes `fold` ist besser, warum? (Betrachten Sie die notwendige und tatsächliche Bitbreite des Resultates.)

8. (ab 1. Dez. 16 Uhr, täglich) `https:`

`//www.mathekalender.de/wp/de/kalender/Adventskalender, Technische Universität Berlin, The`

Berlin Mathematics Research Center MATH+

In der Übung können jederzeit Aufgabenlösungen mit Methoden der Vorlesung präsentiert und diskutiert werden.

Prädikatenlogik

Übersicht

- 1. Prädikatenlogik (Syntax, Semantik)
- 3. existentielle konjunktive Constraints in verschiedenen Bereichen, z. B.
Gleichungen und Ungleichungen auf Zahlen ($\mathbb{Z}, \mathbb{Q}, \mathbb{R}$)
- existentiell mit beliebigen Boolesche Verknüpfungen:
 - 2. hinschreiben: SAT modulo T (= SMT),
 - 4. lösen: DPLL(T)
- 0. Bit-blasting (SMT \rightarrow SAT)
- 5. volle Prädikatenlogik (auch universelle Quant.)

Syntax der Prädikatenlogik

- Signatur: Name und Stelligkeit für
 - Funktions-
 - und Relationssymbole
- Term:
 - Funktionssymbol mit Argumenten (Terme)
 - Variable
- Formel
 - atomar: Relationssymbol mit Argumenten (Terme)
 - Boolesche Verknüpfungen (von Formeln)
 - Quantor Variable Formel
- – gebundenes und freies Vorkommen von Variablen
 - Sätze (= geschlossene Formeln)

Semantik der Prädikatenlogik

- Universum, Funktion, Relation,
- Struktur, die zu einer Signatur paßt
- Belegung, Interpretation
- Wert
 - eines Terms
 - einer Formel

in einer Struktur, unter einer Belegung

die Modell-Relation $(S, b) \models F$ sowie $S \models F$
Erfüllbarkeit, Allgemeingültigkeit (Def, Bsp)

SMT (SAT modulo Theory)

Beispiel

- $P > 0 \wedge Q \geq 0 \wedge R > 0 \wedge S \geq 0 \wedge P \cdot S + Q > R \cdot Q + S$
- ```
(set-logic QF_NIA) (set-option :produce-models true)
(declare-fun P () Int) (declare-fun Q () Int)
(declare-fun R () Int) (declare-fun S () Int)
(assert (and (< 0 P) (<= 0 Q) (< 0 R) (<= 0 S)))
(assert (> (+ (* P S) Q) (+ (* R Q) S)))
```

<https://smt-lib.org/language.shtml>

- ```
setLogic "QF_NIA"
[p,q,r,s] <- replicateM 4 $ var @IntSort
assert $ p >? 0 && q >=? 0 && r >? 0 && s >=? 0
assert $ p * s + q >? r * q + s
```

<https://hackage.haskell.org/package/hasmtlib>

Bitvektor-Logik (QF_BV)

- Grundbereich: Bitfolgen (mit fixierte Länge w), arithmetische Operationen *modulo* 2^w
- mögliche Implementierung: bit-blasting
- Solver benutzt zusätzlich Aussagen und Umformungs-Regeln für Arithmetik
- Bsp: Laufzeit (vgl. mit bit-blasting, vorige Woche)

```
setLogic "QF_BV"  
[x,y] <- replicateM 2 $ var @(BvSort Unsigned 100)  
assert $ not $ x + y === y + x
```

- **Ü/Vorsicht:** `assert $ x /= 0 && 13 * x === x`

Definition SMT, Lösungsverfahren (Plan)

- SMT: Erfüllbarkeitsproblem für beliebige boolesche Kombination von atomaren Formeln aus einer Theorie

Beispiel: $(x \geq 3 \vee \neg(x + y \leq 4)) \leftrightarrow x > y$

- Verfahren: 1. (wirklich) ersetze jedes T-Atom a_i durch eine boolesche Unbekannte u_i , erhalte Formel F
2. (naiv) für jedes boolesche Modell $b \models F$:
entscheide, ob die Konjunktion der entsprechenden (ggf. negierten) Atome in T erfüllbar ist
- (2. besser) *DPLL modulo Theory*: Verschränkung der booleschen Suche mit T-Erfüllbarkeit für partielle Modelle
Hilfsmittel dabei: Tseitig (ähnliche) Transformation

SMT-**{LIB,COMP}**}

- **Standard-Modellierungssprache, Syntax/Semantik-Def:**
`https://smtlib.cs.uiowa.edu/standard.shtml`
- **Aufgabensammlung:** `https://smtlib.cs.uiowa.edu/benchmarks.shtml`
Kombinatorik, Scheduling, Hard- und
Software-Verifikation, ... crafted, industrial, (random?)
- **Wettbewerb:** `https://smt-comp.github.io/`
- **typische Solver (Beispiele)**
 - **Z3** (Nikolas Bjorner, Leo de Moura et al.)
`https://github.com/Z3Prover/z3/`
 - **CVC5** (Clark Barrett, Cesare Tinelli et al.)
`https://cvc5.github.io/`

Beispiel queen10-1.smt2 aus SMT-LIB

```
(set-logic QF_IDL) (declare-fun x0 () Int)
(declare-fun x1 () Int) (declare-fun x2 () Int)
(declare-fun x3 () Int) (declare-fun x4 () Int)
(assert (let ((?v_0 (- x0 x4)) (?v_1 (- x1 x4))
(?v_2 (- x2 x4)) (?v_3 (- x3 x4)) (?v_4 (- x0 x1))
(?v_5 (- x0 x2)) (?v_6 (- x0 x3)) (?v_7 (- x1 x2))
(?v_8 (- x1 x3)) (?v_9 (- x2 x3))) (and (<= ?v_0 3)
(>= ?v_0 0) (<= ?v_1 3) (>= ?v_1 0) (<= ?v_2 3) (>=
?v_2 0) (<= ?v_3 3) (>= ?v_3 0) (not (= x0 x1))
(not (= x0 x2)) (not (= x0 x3)) (not (= x1 x2))
(not (= x1 x3)) (not (= x2 x3)) (not (= ?v_4 1))
(not (= ?v_4 (- 1))) (not (= ?v_5 2)) (not (= ?v_5
(- 2))) (not (= ?v_6 3)) (not (= ?v_6 (- 3))) (not
(= ?v_7 1)) (not (= ?v_7 (- 1))) (not (= ?v_8 2))
(not (= ?v_8 (- 2))) (not (= ?v_9 1)) (not (= ?v_9
(- 1)))))) (check-sat) (exit)
```

Umfang der Benchmarks (2014)

http://www.cs.nyu.edu/~barrett/smtlib/?C=S;O=D	
QF_BV_DisjunctiveScheduling.zip	2.7G
QF_IDL_DisjunctiveScheduling.zip	2.4G
incremental_Hierarchy.zip	2.1G
QF_BV_except_DisjunctiveScheduling.zip	1.6G
QF_IDL_except_DisjunctiveScheduling.zip	417M
QF_LIA_Hierarchy.zip	294M
QF_UFLRA_Hierarchy.zip	217M
QF_NRA_Hierarchy.zip	170M
QF_LRA_Hierarchy.zip	160M

- QF: quantifier free,
- I: integer, R: real, BV: bitvector
- D: difference, L: linear, N: polynomial

Anwendung zur Terminations-Analyse

- der *arktische* Halbring: $\mathbb{A} = (\{-\infty\} \cup \mathbb{N}, \max, +, -\infty, 0)$
- $\mathbb{A}^{d \times d}$: quadratische Matizen über \mathbb{A} ,
- $P > Q$ falls $\forall i, j : (P_{i,j} > Q_{i,j}) \vee (P_{i,j} = -\infty = Q_{i,j})$.
- Matrix-Interpretation: $i : \Sigma \rightarrow \mathbb{A}^{d \times d}$ mit $\forall c : i(c)_{1,1} \geq 0$
Interpretation von Wörtern: $i(c_1 \dots c_n) := i(c_1) \circ \dots \circ i(c_n)$
- Bsp: $i : a \mapsto \begin{pmatrix} 0 & 0 \\ 1 & 2 \end{pmatrix}, b \mapsto \begin{pmatrix} 0 & -\infty \\ -\infty & -\infty \end{pmatrix}$
- i kompatibel mit R , falls $\forall (l, r) \in R : i(l) > i(r)$.
Bsp (Fortsetzg.) i kompatibel mit $\{aa \rightarrow aba\}$
- dann terminiert R , denn jede R -Ableitung von w aus hat $\leq i(w)_{1,1}$ Schritte
- für gegebenes R und d : Kompatibilität von i ist Constraint-System in QF_LIA.

e-DSLs für Constraint-Prog.

- die Constraint-Sprache C dient zur Kommunikation mit Solver (nicht: mit Anwender/Anwendungsprogrammierer)
- Programm in einer Gastsprache G für
 - Konstruktion des Constraint-Systems
 - Verarbeitung des Resultates (des Modells)
- dabei müssen übersetzt werden
 - Namen in C , Namen in G
 - Werte in C (symbolisch), Werte in G (tatsächlich)
 - Typen in C (Bsp: `Bit`), in G (Bsp: `Bool`)
- Entwurfs-Ziele:
 - symbolisches Programm (in C) sieht aus wie tatsächliches Programm (in G)
 - notwendige Übersetzungen möglichst unsichtbar

Wiederholung: ersatz als e-DSL für SAT

- ```
(let b = True in solveWith minisat $ do
 p <- exists @Bit; assert (p === encode b)
 return p
) >>= \ case (Satisfied, Just (p :: Bool))
```
- ```
class Codec c where
  type Decoded c
  encode :: Decoded c -> c
  decode :: Belegung -> c -> Decoded c
instance Codec Bit where
  type Decoded Bit = Bool; ...
```
- **Ü:** überprüfen Sie die Design-Ziele, geben Sie die technischen Mittel an, durch die diese erreicht werden

Beispiel: Python-Bindung für Z3

- <https://github.com/Z3Prover/z3/blob/master/examples/python/hamiltonian/hamiltonian.py>

```
L = {0:[1,2], 1:[2], 2:[1,0]} # Beispiel-Graph
cv = [Int('cv%s'%i) for i in range(L)]
s = Solver() ; s.add(cv[0]==0)
for i in range(L):
    s.add(Or([cv[j]==(cv[i]+1)%L for j in gr[i]]))
s.check(); print (s.model())
```

- Design-Ziele überprüfen:
 - Namen, Typen, Ausdrücke, Übersetzungen, Sichtbarkeit
- beachte: hier wird keine SMTLIB-Datei erzeugt, sondern API des Solvers aufgerufen.

Haskell-Bindungen für SMTLIB (Bsp. 1)

- Iavor S. Diatchki:

`https://hackage.haskell.org/package/simple-smt`

- ```
s <- newSolver "cvc4" ["--lang=smt2"] Nothing
setLogic s "QF_LIA"
x <- declare s "x" tInt
assert s (add x (int 2) `eq` int 5)
check s
print =<< getExprs s [x]
```

- $C$ -Namen sind sichtbar
- $C$ -Typen (`tInt`) erscheinen nicht statisch in  $G$ -Typen
- $C$ - und  $G$ -Operatoren: `add`, `+`
- explizite Rück-Übersetzung (`getExprs`)

# Haskell-Bindungen für SMTLIB (Bsp. 2)

- Henning Günther: <https://hackage.haskell.org/package/smtlib2>
- ```
withBackend (createPipe "z3" ["-smt2", "-in"]) $ do
  x <- declareVar int; y <- declareVar int
  assert $ x .+. y .==. cint 5
  assert $ x .>. cint 0; assert $ y .>. cint 0
  checkSat
  IntValue vx <- getValue x; IntValue vy <- getValue
  return (vx, vy)
```
- *C*-Typen erscheinen statisch in *G*-Typen, Bsp:

```
(.>.) :: (Embed m e, IsSMTNumber tp, HasMonad a, HasMonad b, MatchMonad a m, MatchMonad b m, MonadResult a ~ e tp, MonadResult b ~ e tp) => a -> b -> m (e BoolType)
```

Haskell-Bindungen für SMTLIB (Bsp. 3)

- Julian Bruder: Hasmtlib `https://hackage.haskell.org/package/hasmtlib`
... encoding your problem, marshaling the data to an external solver and parsing and interpreting the result into Haskell types. It is highly inspired by `ekmett/ersatz`
- ```
res <- solveWith @SMT (solver z3) $ do
 setLogic "QF_NIA"
 [p,q,r,s] <- replicateM 4 $ var @IntSort
 assert $ p >? 0 && q >=? 0 && r >? 0 && s >=? 0
 assert $ p * s + q >? r * q + s
 return [p,q,r,s]
print res
```

# Aufgaben

WS 25: 1, 2, 3

1. für jede der Logiken QF-LIA, LRA, NIA: ein kleines Constraint-System (mit kleinen Koeffizienten) angeben, das erfüllbar ist, aber nur durch Modelle mit sehr großen Zahlen.
2. Bestimmen Sie:
  - die kleinste natürliche Zahl, die sich auf zwei verschiedene Weisen als Summe von zwei Kuben schreiben läßt
  - mit einem SMT-Solver. Schreiben Sie das Constraint-System von Hand. Benutzen Sie Logiken QF\_NIA (Polynom-Arithmetik) (Warum nicht QF\_LRA?)

Hinweis: die Bedingung *die kleinste* kann man nicht hinschreiben, aber durch systematisches Probieren realisieren

3. die vorige Aufgabe mit  $\mathbb{QF\_BV}$  (Bitvektoren). Dabei müssen Überläufe durch weitere Constraints verhindert werden. Wie geht das?

4. Dieses Beispiel in  $\mathbb{QF\_NIA}$  ist wohl zu schwer für heutige Solver:

Andrew R. Booker, Andrew V. Sutherland: *On a question of Mordell*, <https://arxiv.org/abs/2007.01209>

John Pavlus: *Sum-of-Three-Cubes Problem Solved for 'Stubborn' Number 33*,

<https://www.quantamagazine.org/>

5. wählen Sie zufällig in SMTLIB eine (quantorenfreie) Logik und dort eine Benchmark. Erklären Sie die Benchmark.

Beispiel: Queens (siehe Folie)

Wenden Sie verschiedene SMT-Solver an (z.B. Z3, CVC5, opensmt) und vergleichen Sie Laufzeiten.

Ändern Sie die Formel (vorsichtig), erläutern Sie die Änderungen der Belegung oder Erfüllbarkeit.

Erzeugen Sie das Constraint durch Hasmtlib.

6. die zitierte Hamiltonkreis-Kodierung (oder die früher zitierte MIP-Kodierung dafür) in SMTLIB-Syntax hinschreiben (in möglichst einfacher Logik) und ausprobieren.

7. Eine kompatible arktische Matrix-Interpretation für  $aa \rightarrow aba$  (das Bsp. auf Folie) durch Lösen eines LIA-Systems bestimmen

Desgl. für  $a^2b^2 \rightarrow b^3a^3$  (größere Dimension verwenden).

Warum gibt es keine solche Interpretation für  $ab \rightarrow ba$ ?

Hinweis: weil diese Regel quadratische lange

Ableitungen gestattet (von welchen Startwörtern?),

aber solche können bei arktischen Interpretationen nicht vorkommen (warum?)

8. Adventskalender der DMV [https:](https://www.mathekalender.de/wp/de/kalender/)

[//www.mathekalender.de/wp/de/kalender/](https://www.mathekalender.de/wp/de/kalender/)

Wenden Sie SAT oder SMT an, um Aufgaben zu modellieren und zu lösen.

# DPLL(T) (Modulo Theories)

## DPLL(T), Prinzip

für jedes T-Atom  $A = P(t_1, \dots, t_k)$   
eine boolesche Unbek.  $p_A \leftrightarrow A$ .

- naives Vorgehen:
  - für jede Lösung des SAT-Problem für diese Variablen  $p_*$ :
  - bestimme Erfüllbarkeit dieser Konjunkt. von T-Literalen
- Realisierung mit DPLL(T):
  - decide,  $T$ -solve (Konjunktion von  $T$ -Literalen)
  - Konflikte (logische und  $T$ -Konfl.): backtrack
  - logische Propagationen, Lernen
  - $T$ -Propagation ( $T$ -Deduktion)

# DPLL(T): Datenmodell, Semantik für Formel

- Theorie ist: lineare Ungleichungen über  $\mathbb{Q}$
- Formel  $\in \text{CNF}(T) =$  Konjunkt. v. Disjunkt. von Literalen

```
data Atom = Theory_Atom (FM.Atom Variable)
 | Boolean_Atom Variable
data Literal =
 Literal { polarity :: Bool, atom :: Atom }
type Clause = [Literal]
type CNF = [Clause]
```

- **Beispiel:**  $b \leq x \wedge c \leq x \wedge (b \geq x \vee c \geq x)$

```
[[0 <= -1 * b + x]
, [0 <= -1 * c + x]
, [0 <= + b -1 * x, 0 <= + c -1 * x]]
```

- Beschreibung, Test, Verbesserung : Bachelor-Arbeit

# DPLL(T): Datenmodell, Semantik für Lösung

- Zustand d. Beweis-Suche (wie DPLL): Menge (Konjunktion)  $b$  v. Literalen, Keller  $k$  v. Entscheidungen
- Lösung ist Schrittfolge, jeder Schritt ändert Zustand

```
data Step = Decide Literal
 | Conflict Conflict | Backtrack
 | Propagate { use :: Conflict, obtain :: Literal }
 | SAT | UNSAT
```

- neu (für DPLL(T)): Theorie-Konflikte.

```
data Conflict = Boolean Clause | Theory
```

- Propagation (Wdhlg.) ist eine Entscheidung  $p$  (Belegung eines Atoms), deren Gegenteil  $\neg p$  einen Konflikt  $c$  erzeugt  
wenn  $(b \wedge \neg p) \not\models c$ , dann  $(b \wedge c) \Rightarrow p$ .

# DPLL(T): Einzelheiten, Beispiele

- Literatur: Robert Nieuwenhuis et al.:

<https://www.cs.upc.edu/~roberto/papers/IJCAR2012Slides.pdf>

- Univ. Barcelona, Spin-Off: Barcelologic, Bsp:

<https://barcelologic.com/en/sports-planning/>

... software for professional sports scheduling. It has been successfully applied during the last five years in the Dutch professional football (the main KNVB Ere- and Eerste Divisies).

An adequate schedule is not only important for sportive and economical fairness among teams and for public order. It also plays a very important role reducing costs and increasing revenues, e.g., the value of TV rights.

# Lineare Gleichungen und Ungleichungen

## Beispiel LP: monotone Interpretation

- Beispiel: das Wortersetzungssystem  $R = \{aa \rightarrow bbb, bb \rightarrow a\}$  terminiert.
- Beweis: definiere  $h : \Sigma \rightarrow \mathbb{N} : a \mapsto 5, b \mapsto 3$  und setze fort zu  $h^* : \Sigma^* \rightarrow \mathbb{N} : h(c_1 \dots c_n) = \sum h(c_i)$ .  
Dann gilt  $u \rightarrow_R v \Rightarrow h^*(u) > h^*(v)$  wegen  $\forall (l \rightarrow r) \in R : h^*(l) > h^*(r)$ .
- Die Gewichtsfunktion  $h$  erhält man als Lösung des linearen Ungleichungssystems  $2a > 3b \wedge 2b > a \wedge a \geq 0 \wedge b \geq 0$ .

# Beispiel LP-Solver

- Aufgabenstellung im LP-Format (<http://lpsolve.sourceforge.net/5.0/CPLEX-format.htm>)

Minimize

obj: a + b

Subject To

c1: 2 a - 3 b >= 1

c2: 2 b - a >= 1

End

- lösen mit `glpsol` (GNU Linear Programming Kit, <https://www.gnu.org/software/glpk/>, 2000-2020, Andrew Makhorin)

```
glpsol --lp lin/lpex.cplex
```

# Syntax, Semantik

- lin. (Un-)Gleichungssystem  $\rightarrow \bigwedge_{i=1}^n$  Constraint
- Constraint  $\rightarrow$  Ausdruck Relsym Ausdruck
- Relsym  $\rightarrow = \mid \leq \mid \geq$
- Ausdruck  $\rightarrow$  Zahl  $+$   $\sum_{i=1}^n$  (Zahl  $\cdot$  Unbekannte)
- Zahlenbereich:  $\mathbb{Q}$  (rational)

Beispiel:  $4y \leq x \wedge 4x \leq y - 3 \wedge x + y \geq 1 \wedge x - y \geq 2$

Semantik: Wertebereich für Unbekannte ist  $\mathbb{Q}$  (äquiv:  $\mathbb{R}$ )

# Normalformen

- Beispiel:

$$4y \leq x \wedge 4x \leq y - 3 \wedge x + y \geq 1 \wedge x - y \geq 2$$

- Normalform:  $\bigwedge_i \sum_j a_{i,j} x_j \geq b_i$

$$x - 4y \geq 0$$

...

- Matrixform:  $Ax^T \geq b^T$

$A$  ist linearer Operator.

Lösung von linearen (Un-)Gl.-Sys. mit Methoden der linearen Algebra

# Hintergründe

Warum funktioniert das alles?

- lineares Gleichungssystem:  
Lösungsmenge ist (verschobener) *Unterraum*, endliche Dimension
- lineares Ungleichungssystem:  
Lösungsmenge ist *Simplex* (Durchschnitt von Halbräumen, konvex), endlich viele Seitenflächen

Wann funktioniert es nicht mehr?

- nicht linear: keine Ebenen
- nicht rational, sondern ganzzahlig: Lücken

# Lineare Gleichungssysteme

- Lösung nach Gauß-Verfahren:  
eine Gleichung nach einer Variablen umstellen,  
diese Variable aus den anderen Gleichungen eliminieren  
(= Dimension des Lösungsraumes verkleinern)
- Ü: es gibt kein solches Verfahren für CNF-SAT (es gibt keine Operation, die der Subtraktion entspricht)  
... aber für XOR-SAT (Konjunktion von XOR-Klauseln)
- Mate Soos, Karsten Nohl, Claude Castelluccia:  
*Extending SAT Solvers to Cryptographic Problems SAT*  
2009 <https://github.com/msoos/cryptominisat>  
... we extended the solver's input language to support the XOR operation

# Lineare Ungleichungen und Optimierung

Entscheidungsproblem:

- Eingabe: Constraintsystem,
- gesucht: eine erfüllende Belegung

Optimierungsproblem:

- Eingabe: Constraintsystem und *Zielfunktion* (linearer Ausdruck in Unbekannten)
- gesucht: eine optimale erfüllende Belegung (d. h. mit größtmöglichem Wert der Zielfunktion)

Standard-Form des Opt.-Problems:

$$A \cdot x^T = b, x^T \geq 0, \text{ minimiere } c \cdot x^T.$$

Ü: reduziere OP auf Standard-OP, reduziere EP auf OP

# Lösungsverfahren für lin. Ungl.-Sys.

- Simplex-Verfahren (für OP) (George Dantzig et al., 1947)  
Schritte wie bei Gauß-Verfahren für Gleichungssysteme  
(= entlang einer Randfläche des Simplex zu einer  
besseren Lösung laufen)  
Einzelheiten siehe Vorlesung Numerik/Optimierung  
exponentielle Laufzeit im schlechtesten Fall (selten)
- polynomielle Algorithmen: Leonid Kachiyan, 1979,  
Narendra Karmakar 1984.
- Fourier (1826)-Motzkin (1936)-Verfahren (für EP)  
vgl. mit Elimination durch vollständige Resolution  
exponentielle Laufzeit (häufig)

# Fourier-Motzkin-Verfahren

Def.: eine Ungls. ist in  $x$ -Normalform, wenn jede Ungl.

- die Form „ $x (\leq \mid \geq)$  (Ausdruck ohne  $x$ )“ hat
- oder  $x$  nicht enthält.

Satz: jedes Ungls. besitzt äquivalente  $x$ -Normalform.

Def: für Ungls.  $U$  in  $x$ -Normalform:

$$U_x^\downarrow := \{A \mid (x \geq A) \in U\}, \quad U_x^\uparrow := \{B \mid (x \leq B) \in U\}, \\ U_x^- = \{C \mid C \in U, C \text{ enthält } x \text{ nicht}\}.$$

Def: ( $x$ -Eliminations-Schritt) für  $U$  in  $x$ -Normalform:

$$U \rightarrow_x \{A \leq B \mid A \in U_x^\downarrow, B \in U_x^\uparrow\} \cup U_x^-$$

Satz: ( $U$  erfüllbar und  $U \rightarrow_x V$ )  $\iff$  ( $V$  erfüllbar).

FM-Verfahren: Variablen nacheinander eliminieren.

# Fourier-Motzkin: Datenmodell lineare Fkt.

- ```
import qualified Data.Map as M

data Linear v = Linear (M.Map (Maybe v) Rational)
linf = Linear $ M.fromList
    [(Nothing, 5%2), (Just "x", -3%7), (Just "y", 4%1)]

plus (Linear p) (Linear q) =
    Linear $ M.filter (/= 0) $ M.unionWith (+) p q
```
- ```
https://git.imn.htwk-leipzig.de/waldmann/autotool/
-/tree/master/collection/src/Fourier_Motzkin

$ cabal repl autotool-collection -w /opt/ghc-9.8.4/
ghci> :l Fourier_Motzkin
ghci> scale 2 linf
```

# Fourier-Motzkin: Implementierung

- Datenmodell für Konjunktion von Ungl.

```
data Atom v = NonNegative {linear :: Linear v}
 | Positive {linear :: Linear v}
type Constraint v = [Atom v]
```

- -- |  $f == 0$  , where  $f$  contains  $x$ , is transformed to  
--  $f' == x$  , where  $f'$  does not contain  $x$   
remove :: Ord v => v -> Atom v -> Linear v

- Elimination einer/aller Variablen

```
resolve :: Ord v => v -> Constraint v -> Constraint v
satisfiable :: Ord v => Constraint v -> Bool
```

```
ghci> resolve "y" $ resolve "x" unsat
```

# Aufgaben

WS 24: 1, 2, 5

1. Finden Sie eine monotone Interpretation durch eine Gewichtsfunktion für das Wortersetzungssystem

(RULES

a a a  $\rightarrow$  b b,

b b b  $\rightarrow$  c d ,

c  $\rightarrow$  a a ,

d  $\rightarrow$  c )

(Quelle: SRS/Zantema/z116.srs aus

<https://www.lri.fr/~marche/tpdb/tpdb-2.0/>,

vgl.

<https://termination-portal.org/wiki/TPDB>)

Stellen Sie das passende Ungleichungssystem auf, geben Sie eine (geratene) Lösung an.

2. Führen Sie das Fourier-Motzkin-Verfahren für dieses Ungleichungssystem durch.

3. Bestimmen Sie eine Lösung mit GLPK

4. Bestimmen Sie eine Lösung mit `hmatrix-glpk`.

Alberto Ruiz, Dominic Steinitz, 2010-2018, *Simple interface to linear programming functions provided by GLPK*. `https:`

`//hackage.haskell.org/package/hmatrix-glpk`

5. Finden Sie weitere Systeme aus SRS/Zantema/z101 ... z112 mit Gewichtsfunktion.

Vergleichen Sie mit den Lösungen, die in der letzten

Termination Competition gefunden wurden.

[https://termination-portal.org/wiki/Termination\\_Competition](https://termination-portal.org/wiki/Termination_Competition)

6. Vorverarbeitung eines Terminationsproblems durch *sparse tiling*, dann Gewichtsfunktion: siehe Geser, Hofbauer, Waldmann FSCD 2019 <https://drops.dagstuhl.de/opus/volltexte/2019/10528/> beruht auf einer alten und einfachen Idee, Beispiel: 2-Kachelung für  $aa \rightarrow aba$  ergibt  $\{[aa] \rightarrow [ab][ba]\}$ , Anzahl der  $[aa]$  nimmt ab, das klappt aber nicht immer so einfach (wann nicht?), läßt sich leicht reparieren (wie?) in zitierter Quelle: Einschränkung der Kachelmenge





# Programm = Beweis

## Plan, Motivation

- die Curry (1960)-Howard (1969)-Isomorphie:  
Typ = Behauptung (Proposition), Programm = Beweis  
Behauptung: Typ  $T$  ist nicht leer,  
Beweis dafür: ein Ausdruck (Programm)  $P$  mit Typ  $T$ .
- dann ist Beweis-Prüfung = Typ-Prüfung
- diese kann automatisiert werden (Bsp: Sprache Agda)
- ... die Beweis-Suche aber nicht (volle Prädikatenlogik, Wahrheit ist nicht entscheidbar)
- Unterstützung (interact. proof assistant, Bsp: Agda Emacs mode) für die schrittweise Beweis-Konstruktion

# Konjunktion = Kreuzprodukt

- $(A \wedge B)$  ist wahr gdw.  $A$  wahr und  $B$  wahr  
Typ  $(A \wedge B)$  nicht leer gdw.  $A$  nicht leer und  $B$  nicht leer
- `data Und (a b : Set) : Set where  
 beide : a -> b -> Und a b`
- das ist die generalized algebraic data type (GADT)-Notation für den Haskell-Typ  
`data Pair a b = Beide a b`
- in Agda: Typ-Argumente müssen deklariert werden,  
Groß/Klein-Schreibung ist frei wählbar,  
Operatoren  $(-\>, :)$  benötigen Leerzeichen,  
denn  $a-\>b$  ist *ein* gültiger Name

# Implikation = Funktionstyp

- wenn  $(A \rightarrow B)$  wahr ist, und  $A$  wahr ist, dann ist  $B$  wahr  
modus ponens (Abtrennungsregel)  $\frac{A \rightarrow B, A}{B}$

- Implikation  $A \rightarrow B$  realisiert durch Funktionstyp  $A \rightarrow B$
- Beispiel: wir behaupten und zeigen  $(A \wedge B) \rightarrow (B \wedge A)$

Behauptung:

`prop : {a b : Set} -> Und a b -> Und b a`

- Beweis: eine Implementierung

`prop (beide x y) = ?`

- Syntax: in geschweiften Klammern (im Typ):

*implizite* Argumente (weglassen in Impl.)

# Disjunktion = disj. Vereinigung

- $A \vee B$  ist wahr gdw.  $A$  wahr oder  $B$  wahr ist

Typ  $A \vee B$  ist nicht leer gdw.  $A$  nicht leer oder  $B$  nicht leer

```
data Oder (a b : Set) : Set where
 links : a -> Oder a b
 rechts : a -> Oder a b
```

- ein Distributivgesetz  $(A \vee B) \wedge C \rightarrow (A \wedge C) \vee (B \wedge C)$

```
dist : {a b c : Set} -> Und (Oder a b) c
 -> Oder (Und a c) (Und b c)
```

- Beweis durch vollständige Fallunterscheidung

```
dist (beide (links x) z) = ?
```

...

# Schrittweise Beweis-Konstruktion

- Lücken im Beweis schreiben als ?
- $C-c$   $C-l$  (load) Modul kompilieren
- $C-c$   $C-$ , Ziel (Typ der Lücke) und Kontext (Typen der gebundene Namen) anzeigen  $C-u$   $C-u$   $C-c$   $C-$ , desgleichen, mit expandierten (normalisierten) Typen
- $C-c$   $C-c$  (case) vollständige Fallunterscheidung
- $C-c$   $C-SPC$  (give) Lücke füllen
- $C-c$   $C-r$  (refine) ersetzt ? durch  $f$  ? ... ? (mit passender Anzahl neuer Lücken)
- $C-c$   $C-a$  (auto)

Ziel: Lücken in Programmen durch diese Befehle nach unten (in Teilterme) schieben und schließlich entfernen

# Sprachmodelle und Formale Methoden

- „das ist ja wie eine richtige KI“: Jein. Andersherum ist wahr: funktionale Programmierung und symbolische KI zur Programmkonstruktion seit ca. 1960 (LISP)
- die derzeit gehypten Textmodelle (LLM) „verstehen“ Programmieraufgaben nur so weit, wie sie dazu vermutete Hausaufgabenlösungen auf Stackoverflow memorisiert haben
- das LLM kann einen Agenten steuern (Interpreter oder Compiler) und Glück haben, wenn dabei erzeugte Fehlermeldungen auf Stackoverflow standen.
- das funktioniert um so besser, je genauer die Fehlermeldungen sind. Die Fehlermeldungen werden umso genauer, je genauer die Spezifikation ist.

Wenn die Spezifikation allerdings nur aus vom LLM selbst geratenen Testfällen besteht, nützt sie gar nichts.

- bei ausdrucksstarken statischen Typsystemen steht die Spezifikation (als Typ) exakt da (proposition = type).
- dann ist es auch egal, woher die Implementierungs- bzw. Beweisvorschläge (program = proof) kommen, denn der Compiler prüft diese exakt.
- die Fehler des ahnungslosen Programmierens („vibe coding“) können *nur* durch formale Methoden und Werkzeuge *festgestellt* werden, und auch *vermieden*:
- denn diese machen das Text-basierte Raten überflüssig:  
Bsp. C-c C-a zählt Beweiskandidaten der Größe nach auf und prüft Typkorrektheit.

Die Jugend Von Heute kann das gern *agentic coding* nennen, Hauptsache, sie kommt in die FMW-Vorlesung.

# Das Falsche, das Wahre

- Aussage  $A$  wahr gdw. Type  $A$  nicht leer,  
also:  $A$  falsch gdw.  $A$  leer, d.h., ohne Konstruktoren  
`data Absurd : Set`, Bezeichnung auch  $\perp$  (bottom).
- dieses Programm hat *nicht* den Typ `Absurd`  
`f : Absurd ; f = f` (in Haskell: statisch korrekt, in  
Agda: wegen Nichttermination abgelehnt)
- vgl. das Wahre: `data Unit : Set where unit : Unit` ,  
auch  $\top$  (top).
- `exfalso : Absurd -> Unit`  
**Beweis:** `exfalso ()` das ist keine Gleichung  
(Implementierung), sondern die Behauptung, daß es  
keine gibt.

# Die Negation

- eine Art der Negation  $\neg A$  ist definiert als  $(A \rightarrow \perp)$ .

`not : Set -> Set ; not a = a -> Absurd`

Bezeichnung: das ist *intuitionistische* Logik

Vorsicht, dafür gilt  $A \rightarrow \neg\neg A$  (Ü), aber nicht  $\neg\neg A \rightarrow A$

- Ü:  $\neg\neg\neg A \rightarrow \neg A$
- de Morgan:  $\neg(A \vee B) \rightarrow (\neg A \wedge \neg B)$
- Ü: andere Varianten dieser Aussage. nicht alle klassische wahren sind auch intuitionistisch wahr.

# Die Peano-Zahlen

- den Typ schreiben wir genauso wie früher in Haskell

```
data Nat : Set where zero : Nat ; succ : Nat -> Nat
```

- Funktionen auch (Fallunterscheidung, Rekursion)

```
plus : Nat -> Nat -> Nat
plus zero y = y ; plus (succ x) y = succ (plus x y)
```

- Unterschiede zu Haskell: Vollständigkeit, Termination (Beispiele!)

# Eigenschaften von Zahlen (Bsp. 1)

- Bsp: die Eigenschaft „ist gerade“.  
wir rechnen nicht einen Wahrheitswert aus,  
sondern geben Beweismöglichkeiten an:

```
data even : Nat -> Set where
 even-zero : even zero
 even-succ : {x : Nat}
 -> even x -> even (succ (succ x))
```

- Typ v. `even-succ` ist *dependent* (daten-abhängig, von `x`)
- Beweis (vollständige ! Fallunterscheidung)

```
even-plus
 : (x y : Nat) -> even x -> even y -> even (plus x
even-plus zero y ex ey = ey
even-plus (succ (succ x)) y (even-succ ex) ey = ?
```

# Die Sprache/Das System Agda

- *dependently typed functional programming language*: has inductive families (data types which depend on values)  
*proof assistant*: interactive system for writing and checking proofs. based on intuitionistic type theory,  
<https://wiki.portal.chalmers.se/agda/>
- Per Martin-Löf, *Constructive Mathematics and Computer Programming*, 1979 <http://intuitionistic.files.wordpress.com/2010/07/martin-lof-computer.pdf>
- Wadler, Kokke, Siek: *Programming Language Foundations in Agda* <https://plfa.github.io/>
- Samuel Mimram: *Program = Proof* <https://www.lix.polytechnique.fr/Labo/Samuel.Mimram/>

# Aufgaben

exakt formalisieren, Beweis schrittweise entwickeln  
Code aus VL verwenden (d.h., keine Standardbibliothek)  
(möglichst viel `C-c` . . ., möglichst wenig selbst schreiben)

1. Assoziativität des Oder

2.  $\neg\neg\neg a \rightarrow \neg a$

3. Varianten von de Morgan

4. für `Nat`: definiere `odd` (ungerade) (direkt, nicht als Negation von gerade), zeige:

`{x : Nat} -> even x -> odd x -> Falsch`

5. zeige: ungerade plus ungerade ergibt gerade

6. definiere Multiplikation,

zeige: ungerade mal ungerade ergibt ungerade

Note: Assoziativität der Addition (usw.) noch nicht, weil wir die Gleichheit noch nicht definiert haben.

# Dependent Typen und Muster

## Datentypen

- ohne Parameter, ohne Index:

```
data T : Set where A : T ; B : T
```

alle Konstruktoren liefern gleichen Typ

- mit Parameter (generische Polymorphie)

```
data M (a : Set) : Set where N : M a; J : a -> M a
```

Parameter vor Doppelpunkt, in allen Konstruktoren sichtbar, alle Konstruktoren liefern gleichen Typ

- mit Index (dependent Type)

```
data D : T -> Set where E : D A; F : (x : T) -> D x
```

Index nach Doppelpunkt, kann in Konst. unterschiedlich sein

# Pattern Matching in Funktions-Def.

- definierte Funktion soll total sein  $\Rightarrow$  jede Fallunterscheidung muß vollständig sein
- match auf Argument mit nicht indiziertem Typ:  
für jeden Konstruktor eine Klausel
- match auf Argument mit indiziertem Typ:
  - für jede Klausel: Index des Musters *unifizierbar* mit Index des Arguments
  - für jede fehlende Klausel: nicht unifizierbar

Bsp: ist vollständig, Muster  $f\ A\ (F\ B)$  fehlt:

$f : (x : T) \rightarrow (y : D\ x) \rightarrow \text{Unit}$

$f\ A\ E = ? ; f\ A\ (F\ A) = ? ; f\ B\ y = ?$

# Anwendung: langen-indizierte Vektoren

- naturliche Zahlen als Peano-Zahlen

```
data Nat : Set where zero : Nat; succ : Nat -> Nat
```

- Vektor mit Inhaltstyp als Parameter, Lange als Index

```
data Vec (a : Set) : Nat -> Set where
 nil : Vec a zero
 cons : {n : Nat} -> a -> Vec a n -> Vec a (succ n)
```

- Anwendung: diese Funktion ist total

```
head : {a : Set} {n : Nat} -> Vec a (succ n) -> a
head (cons x v) = x
```

die Klausel fur `head nil` fehlt, denn die Indizes sind nicht unifizierbar (aus Typ von `head`: `Index succ n`, aus Typ von `nil`: `Index zero`)

# Die Gleichheit

- zwei Dinge sind gleich—wenn sie identisch sind

```
data Eq {a : Set} : (x y : a) -> Set where
 refl : {x : a} -> Eq x x -- reflexiv
```

der Konstruktor (`refl`) hat einen eingeschränkten Typ  
(nicht `Eq x y`)

- Bsp: diese Gleichheit ist transitiv:

```
trans : {a : Set} -> {x y z : a}
 -> Eq x y -> Eq y z -> Eq x z
trans refl refl = ?
```

das erst Arg. hat Typ `Eq x y` und Konstruktor `refl`  
(vollst. Falluntersch.: es gibt keinen anderen). Also folgt  
`x = y`. Entspr. `y = z`,

# Anwendung Gleichheit

- Funktionsanwendung: gleiche Argumente ergeben gleiches Resultat

```
cong-succ
```

```
 : (x y : Nat) -> Eq x y -> Eq (succ x) (succ y)
```

```
cong-succ x y refl = refl
```

- Assoziativität der Addition, Beweis durch Induktion

```
plus-assoc : (x y z : Nat)
```

```
 -> Eq (plus x (plus y z)) (plus (plus x y) z)
```

```
plus-assoc zero y z = refl
```

```
plus-assoc (succ x) y z = cong-succ ? ? ?
```

im I.S. wird die Kongruenz angewendet auf die I.V.

# Notation für Gleichungsketten

- so sieht man besser, was passiert:

```
plus-assoc-v (succ x) y z = begin
 plus (succ x) (plus y z)
 =[] succ (plus x (plus y z))
 =[cong succ (plus-assoc-v x y z)]
 succ (plus (plus x y) z)
 =[] plus (succ (plus x y)) z
 =[] plus (plus (succ x) y) z end
```

- mit diesen Operatoren  
(Argument  $x$  wird nicht benutzt, nur hingeschrieben)

```
infix 1 begin_ begin refl = refl
infixr 2 _=[]_ x=[] refl = refl
infixr 2 _=[_]_ x=[exy] eyz = trans exy eyz
infix 3 _end x end = refl
```

# Operator-Syntax in Agda

- mehrstellige benutzerdefinierte Operatoren
- Operator-Namen: beliebige Zeichenfolge (keine Klammern)
  - keine Leerzeichen,
  - jeder Unterstrich bezeichnet eine Argumentposition
- ein Op. aus vorigem Beispiel, vollständig definiert:
  - Typ:  
$$\_=[\_]\_ : \{A : \text{Set}\} (x : A) \{y z : A\} \\ \rightarrow \text{Eq } x \ y \rightarrow \text{Eq } y \ z \rightarrow \text{Eq } x \ z$$
  - Implementierung:  
$$x \ = \ [ \ exy \ ] \ eyz = \ \text{trans} \ exy \ eyz$$
  - Präzedenz, Assoziativität: `infixr 2 _=[_]_`
- Beispiele: `_+_`, `_<_`, `if_then_else_`

# Aufgaben

exakt formalisieren, Beweis schrittweise entwickeln  
Code aus VL verwenden (d.h., keine Standardbibliothek)  
(möglichst viel C-C . . ., möglichst wenig selbst schreiben)

1.  $x + 0 = x$
2.  $x + y = y + x$
3. Multiplikation definieren (Induktion über 1. Arg.)
4. gerade mal beliebig ergibt gerade
5. Multiplikation ist assoziativ, kommutativ, distributiv (rechts und links) bzgl. Addition
6. Potenzfunktion (Induktion über 2. Arg.), Potenzgesetze





# Verifizierte binäre Arithmetik

## Binärzahlen

- $25_{10} = 11001_2 = 1 \cdot 2^0 + 0 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4$   
 $= 1 + 2 \cdot (0 + 2 \cdot (0 + 2 \cdot (1 + 2 \cdot (1 + 2 \cdot 0))))$
- $= o \quad (e \quad (e \quad (o \quad (o \quad z)))$

dieser Typ beschreibt nur die Form der Daten:

```
data Bin : Set where
 z : Bin; e : Bin -> Bin; o : Bin -> Bin
```

- beschreibt zusätzlich die Semantik im Typ-Index:

```
data Bin : Nat -> Set where
 z : Bin zero
 e : {a : Nat} -> Bin a -> Bin (double a)
 o : {a : Nat} -> Bin a -> Bin (succ (double a))
```

# Korrekt getypt = richtig gerechnet

- `data Bin : Nat -> Set where`
  - `z : Bin zero`
  - `e : {a : Nat} -> Bin a -> Bin (double a)`
  - `o : {a : Nat} -> Bin a -> Bin (succ (double a))`

`double : Nat -> Nat ; double zero = zero`  
`double (succ x) = succ (succ (double x))`

- **Compiler (Mimer, C-c C-a) inferiert Binärdarstellung:**

`b5 : Bin (succ (succ (succ (succ (succ zero)))))`  
`b5 = ?`

- **...inferiert Programm:**

`bsucc : {a : Nat} -> Bin a -> Bin (succ a)`  
`bsucc z = ? ; bsucc (e x) = ? ; bsucc (o x) = ?`

# Umformung von Typ-Indizes

- `data Bin : Nat -> Set where`  
    `z : Bin zero`  
    `e : {a : Nat} -> Bin a -> Bin (double a)`  
    `o : {a : Nat} -> Bin a -> Bin (succ (double a))`
- `bplus : {a b : Nat}`  
    `-> Bin a -> Bin b -> Bin (plus a b)`  
    `bplus z y = y -- ist richtig typisiert`  
    `bplus (e x) z = (e x) -- ist es nicht (!)`
- `cast : {x y : Nat} -> x == y -> Bin x -> Bin y`  
    `...`  
    `bplus (e {a} x) z =`  
        `cast (sym (plus-0 (double a))) (e x)`
- **Ü:** ergänze fehlende Gl. der Addition,  
    implementiere Multipl. (autotool), Potenz (Projekt?)
- ***Don't Touch The Green Slime!*** D. O. Kidney 2018 <https://doisinkidney.com/posts/2018-09-20-agda-tips.html>

# Ausnutzen und Ignorieren von Typ-Indizes

- `data Bin : Nat -> Set where ...`  
`bplus : {a b : Nat}`  
`-> Bin a -> Bin b -> Bin (plus a b)`
- Typ-Indizes (`a, b, plus a b`)  
erzwingen Korrektheit der Implementierung,  
Beweise sind *intrinsisch* (= Teil des Programms).  
(Gegensatz: *extrinsische* Beweise separat von Prog.)
- Typ-Indizes sind zur Laufzeit *nicht* nötig,  
können *ausradiert* werden (Annotation `@0`)  
`{-# OPTIONS --erasure #-}`  
`data Bin : @0 Nat -> Set where ...`
- nicht nur „nicht nötig“, sondern schädlich,  
denn die Berechnung (im Unär-System!) ist teuer

# Paare, Records, Spezifikationen

## Paare

- bisher: unabhängige Typen der Komponenten (vgl. `Und`)

```
data Pair (A B : Set) : Set
 where pair : (x : A) -> (y : B) -> (Pair A B)
```

- Typ der zweiten Komp. hängt vom Wert der ersten ab:

```
data Pair (A : Set) (P : A -> Set) : Set
 where pair : (x : A) -> (y : P x) -> (Pair A P)
```

- `solches pair x y : Pair A P` ist Beweis für die Aussage  $\exists x \in A : P(x)$
- Standardbibliothek: Konstruktor ist Operator `_, _`

# Records

- bisher:

```
data Pair (A : Set) (P : A -> Set) : Set
 where pair : (x : A) -> (y : P x) -> (Pair A P)
```

- alternative Notation:

```
record Pair (A : Set) (P : A -> Set) : Set where
 constructor pair
 field
 first : A ; second : P first
```

- Konstruktion eines Records:

```
p4 : Pair Nat even
p4 = pair { first = zero; second = even-zero }
```

# Spezifikation algebraischer Strukturen

- Bsp:  $(A, e, f)$  heißt *Monoid*, wenn  $A$  Menge,  $e \in A$ ,  $f : A^2 \rightarrow A$ ,  $f$  assoziativ,  $e$  links und rechts neutral für  $f$
- Record mit Daten und Beweisen der Eigenschaften

```
record Monoid (A : Set)
 (unit : A) (_o_ : A -> A -> A) : Set where field
 left-unit : (x : A) -> Eq (unit o x) x
 right-unit : (x : A) -> Eq (x o unit) x
 assoc : (x y z : A) -> Eq (x o (y o z)) ((x o y) o z)
```

- plus-monoid : Monoid Nat zero plus

```
plus-monoid = record
 { assoc = plus-assoc
 ; left-unit = \ x -> refl ; right-unit = plus-x-0 }
```

# Spezifikation = Abstrakter Datentyp

- `record Monoid (A : Set)` ist abstrakter Datentyp:
  - Typen für *Operationen* (`unit, _o_`)
  - Spezifikationen (Axiome) (`assoc, ...`)
- `plus-monoid : Monoid Nat` ist konkreter Datentyp:
  - Implementierungen für Operationen
  - Beweise für Axiome
- in weniger ausdrucksstarken Sprachen: Spezifikation und Beweis nur als Kommentar oder Laufzeittest

# Zusammensetzen von Spezifikationen

- `record Semiring (A : Set) (zero one : A)`  
  `(+_ *_ : A -> A -> A) : Set where field`  
  `m-plus : Monoid A zero _+_`  
  `comm : (x y : A) -> (x + y) == (y + x)`  
  `m-times : Monoid A one _*_`  
  `dist-l : (x y z : A)`  
    `-> ((x + y) * z) == ((x * z) + (y * z))`  
  `dist-r : ...`
- `plus-times-semiring`  
  `: Semiring Nat zero (succ zero) plus times`  
`plus-times-semiring = record`  
  `{ m-plus = plus-monoid ; m-times = times-monoid`  
  `; comm = ? ; dist-l = ? ; dist-r = ? }`

# Bibliotheken für ADTs (Bsp.)

- für jeden Halbring  $A$ : Konstruktion des Halbrings der quadratischen Matrizen über  $A$  (fixierter Dimension  $\geq 1$ )

```
Mat A d = Vec (Vec A d) d
```

```
mat-semi : {A : Set} {d : Nat} {d >= 1}
```

```
 -> Semiring A _ _ _ _
```

```
 -> Semiring (Mat A d) _ _ _ _
```

realisiert werden müssen Nullmatrix, Einheitsmatrix, Matrix-Addition, -Multiplikation, Assoziativität,...

- geht besser, wenn man solange wie möglich funktional und koordinatenfrei rechnet, vgl. Bradley Saul: *Carya, Linear Algebra in Agda*, 2023–

```
https://git.sr.ht/~bradleysaul/carya/
```

# Aufgaben–Dependent Paare

1. `data Exists {A : Set} (P : A -> Set) : Set`  
`where exists : (x : A) -> (y : P x) -> (Exists P)`

(ist nur eine Umbenennung von `Pair`) beweise

```
{A : Set} (P Q : A -> Set)
 -> Exists (\ x -> Und (P x) (Q x))
 -> Exists (\ x -> P x)
```

## 2. Vertauschen von Quantoren

```
p0 : {A B : Set} {P : A -> B -> Set}
 -> Exists (\ x -> Exists (\ y -> P x y))
 -> Exists (\ y -> Exists (\ x -> P x y))
```

3. Wiederholung: Die All-Quantifikation wird durch den Funktionspfeil ausgedrückt. Anwendung: Skolemisierung ( $f$  ist die Skolem-Funktion für  $y$  unter  $x$ )

```
p1 : {A B : Set} {P : A -> B -> Set}
 -> ((x : A) -> Exists (\ y -> P x y))
 -> Exists {A -> B} (\ f -> (x : A) -> P x (f x))
```

4. für `even` wie früher (induktiv definiert)

```
{x : Nat} -> even x
-> Exists (\ y -> times y (succ (succ zero)) == x
```

5. `not (Exists (\ x -> Und (even x) (even (succ x))))`

# Aufgaben–Abstrakte Datentypen

1. `nat-times-monoid` : `Monoid Nat _ times`
2. für jeden Typ  $A$ : Menge der Funktionen  $A \rightarrow A$  ist Monoid bezüglich Komposition (Nacheinanderausführung)
3. Gruppen-Axiome und abgeleitete Aussagen

```
record Gruppe {A : Set}
 (e : A) (_o_ : A -> A -> A) (i : A -> A) : Set
field
 left-id : {x : A} -> (e o x) == x
 left-inv : {x : A} -> (i x o x) == e
 assoc : {x y z : A}
 -> (x o (y o z)) == ((x o y) o z)

right-id : {A : Set} {e : A} {_o_ : A -> A -> A} {
 -> Gruppe e _o_ i -> {x : A} -> (x o e) == x
```

`right-inv : {A : Set} {e : A} {_o_ : A -> A -> A} {`  
`-> Gruppe e _o_ i -> {x : A} -> (x o i x) == e`

4. Begriff Halbordnung (mit `_<=_` : `A -> A -> Set`) als Record spezifizieren (allgemein),

konkrete Implementierungen für `Nat` für `x <= y`

- Fallunterscheidung für  $x, y$ , Rekursion
- `Exists \ z -> plus x z == y` (kann Eigenschaften von `plus` nachnutzen, auf andere Monoide verallgemeinert werden)

5. Wenn Relation  $R$  Halbordnung auf Menge  $A$ , dann ist auch Relation  $R^-$  Halbordnung auf  $A$





# Entscheidbarkeit, Programme

## Motivation

- intuitionistischer Wahrheitsbegriff:  
Aussage  $A$  ist wahr, wenn sie einen Beweis hat;  
Aussage  $A$  ist falsch, wenn  $A \rightarrow \perp$  einen Beweis hat.  
`data _<=_ : Nat -> Nat -> Set where ...`
- klassische Wahrheitswerte  
`data Bool : Set where false : Bool; true : Bool`
- Prädikat  $P$  auf  $A$  ist entscheidbar, wenn  $P : A \rightarrow \text{Bool}$  berechenbar, Bsp. `_<=b_ : Nat -> Nat -> Bool`
- Semantik: Beziehung zw. `<=` und `<=b`,  
Pragmatik: Vermeidung doppelten Codes (mit `<=`?)

# Rechnung und separater Beweis

- Korrektheit der Implementierung:

```
T : Bool -> Set; T false = Absurd; T true = Unit
```

```
F : Bool -> Set; F false = Unit; F true = Absurd
```

```
q0 : (x y : Nat) -> T (x <=b y) -> x <= y
```

```
q1 : (x y : Nat) -> T (x <=b y) -> x <= y
```

```
q1 zero y leb = le-zero
```

```
q1 (succ x) (succ y) leb = le-succ (q1 x y leb)
```

```
q2 : (x y : Nat) -> F (x <=b y) -> not (x <= y)
```

- in den Beweisen dieser Aussagen wird die Programmstruktur von `_<=b_` wiederholt

# Rechnung, mit Beweis verbunden

- in den Beweisen der Aussagen über `__<=b__` wird die Programmstruktur von `__<=b__` wiederholt
- besser: berechne nicht `Bool`, sondern  
`Oder (not (x <= y)) (x <= y)`  
(Beweis für Aussage oder Beweis für ihre Negation)
- kürzere Notation dieses Typs durch `Dec (x <= y)` mit

```
data Dec (A : Set) : Set where
 no : not A -> Dec A ; yes : A -> Dec A
```

- anstelle von `__<=b__` verwende

```
__<=?__ : (x : Nat) -> (y : Nat) -> Dec (x <= y)
```

# Ergänzungen, Aufgaben

1. Die Dyck-Sprache ist definiert durch kontextfreie Grammatik mit Regeln  $D \rightarrow \epsilon \mid aDbD$ .

$w \in D$  wird durch `Dyck w` modelliert:

```
data List (E : Set) : Set -- wie gehabt
data Sigma : Set where a : Sigma ; b : Sigma
data Dyck : List Sigma -> Set where
 d0 : Dyck nil
 d1 : {u v : List Sigma}
 -> Dyck u -> Dyck v -> Dyck (cons a (app u (con
```

Formalisieren und beweisen Sie  $D \cdot D \subseteq D$

Formalisieren Sie die Lukasiewicz-Sprache  $L \rightarrow b \mid aLL$ .

```
data Lukas : List Sigma -> Set where ...
```

Beweisen Sie  $D \cdot b \subseteq L$ , formalisiert als

```
prop1 : {w : List Sigma}
 -> Dyck w -> Lukas (app w (cons B nil))
```

Formalisieren Sie die Behauptung  $D \cdot b \supseteq L$ .

Beweis ist (m.E.) etwas umständlicher.

# Code-Erzeugung, Beweis-Irrelevanz

## Motivation, Beispiel

- Jesper Cockx et al., *Reasonable Agda is Correct Haskell*, Haskell Symposium, 2022. (extracting verified and readable Haskell code from an Agda program)

<https://dl.acm.org/doi/10.1145/3546189.3549920>

<https://agda.github.io/agda2hs/>

- ```
data BST (a : Set) {{@0 _ : Ord a}} (@0 lower upper
  Leaf : (@0 pf : lower <= upper) → BST a lower upper
  Node : (x : a) (l : BST a lower x) (r : BST a x upper)
        → BST a lower upper
```

- der Beweis in `Leaf` sowie Typ-Argumente `lower`, `upper` sind zur Laufzeit *irrelevant*, werden *ausradiert (erased)*

Compile-Zeit-Irrelevanz

- anything prepended by a dot (.) is marked irrelevant, which means that it will only be typechecked but never **evaluated** <https://agda.readthedocs.io/en/latest/language/irrelevance.html>

- **Anwendung: intrinsische Beweise in Datenstrukturen.**

```
data Exists {A : Set} (P : A -> Set) : Set where
  exists : (x : A) -> .(y : P x) -> Exists {A} P
```

Existenz genügt für Korrektheit, Beweis-Form ist egal

- ... außer für die Konstruktion anderer Beweise (= Verwendung eines irrelevanten Arguments in einem irrelevanten Kontext)
- irrelevante Argumente werden bei Unifikation ignoriert

Laufzeit-Irrelevanz

- Values marked as erased ($@0$) are not present at run time, type checker enforces that no [hinzugefügt: *run time*] computations depend on erased values.

<https://agda.readthedocs.io/en/latest/language/runtime-irrelevance.html>

- $\text{data Vec (A : Set a) : @0 Nat} \rightarrow \text{Set a where}$
 $[\] : \text{Vec A 0}$
 $_ : \{ @0 n \} \rightarrow A \rightarrow \text{Vec A n} \rightarrow \text{Vec A (suc n)}$

- Compile-Zeit-Irrelevanz ist die strengere Forderung (mit ausradierten Werten kann zur Compile-Zeit gerechnet werden)

Anwendung: Terminations-Zertifikate

- (in Bearbeitung) Heinke, Hofbauer, Waldmann: *Certified Termination in Agda*, 2023–

<https://git.imn.htwk-leipzig.de/waldmann/cetera>

die Vorlage ist: Thieman et al., *IsaFoR/CeTA: An Isabelle/HOL Formalization of Rewriting*, 2009–

- Ziel ist ein *formal verifiziertes* Programm
 $P : (R : \text{Terminationsproblem}) \rightarrow (B : \text{Terminationszertifikat}) \rightarrow \text{Dec}(R \text{ terminiert wegen } B)$

Bsp: $R = \{ab \rightarrow ba\}$, $B = \{a : \lambda x.2x, b : \lambda x.x + 1\}$

- Agda-Compiler prüft `P.agda`, erzeugt `P.hs`, dabei wird `Dec (@0 _)` ausradiert zu `Bool`

Certified Termination Competition

- (jährlich) <https://termcomp.github.io/Y2024/SRS>, show scores, matchbox, s6: 1406, s6-cert: 1304
- was fehlt (in CeTA derzeit)? Verifikation der Methode *Sparse Tiling*, Geser, Hofbauer, Waldmann, FSCD 2020
<https://arxiv.org/abs/2003.01696>
warum ist das aufwendig? siehe Konstruktion mit 55 Regeln auf Seite 27
- warum sollte man Terminationsverfahren verifizieren?
Bsp: Tippfehler in Publikation KKST01, vgl.
<https://git.imn.htwk-leipzig.de/waldmann/pure-matchbox/-/blob/master/src/Matchbox/KKST01.hs>
- Nebenwirkung: Testfall für Agda2hs, MAlonzo