

Constraint-Programmierung

Vorlesung

Sommersemester 2009,12,15, WS

2016,22

Johannes Waldmann, HTWK Leipzig

3. Februar 2023

Einleitung

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).

Industrielle Anwendungen der CP

- Verifikation von Schaltkreisen
(*bevor* man diese tatsächlich produziert)
 $F = \text{S-Implementierung}(x) \neq \text{S-Spezifikation}(x)$
wenn F unerfüllbar ($\neg \exists x$), dann Implementierung 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 der CP

- 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 <http://viam.science.tsu.ge/clas2022/isr/termination.html>

CP-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 lineare Funktionen

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

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

- monoton: $x > y \Rightarrow f_a(x) > f_a(y) \wedge f_b(x) > f_b(y)$
- kompatibel mit Programm: $f_a(f_b(x)) > f_b(f_a(x))$
- resultierendes Constraint-System für P, Q, R, S ,
- Lösung mittels Z3

Constraints in der Unterhaltungsmathematik

- Nikoli (1980–, „the first puzzle magazine in Japan.“)
<https://www.nikoli.co.jp/en/puzzles/>
- Erich Friedman: *Math Magic* 1998–
<https://erich-friedman.github.io/mathmagic/>
- 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/2022/>

(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
 - CNF-SAT-Constraints (Normalf., Tseitin-Transformation)
 - DPLL-Solver, Backtracking und Lernen
 - ROBDDs (Entscheidungsdiagramme)
- Prädikatenlogik (konjunktive Constraints)
 - Finite-Domain-Constraints
 - naive Lösungsverfahren, Konsistenzbegriffe
 - lineare Gleichungen, Ungleichungen, Polynomgleichungen
 - Termgleichungen, Unifikation
- Kombinationen
 - Kodierungen nach CNF-SAT (FD, Zahlen)
 - SMT, DPLL(T)

Organisatorisches

- jede Woche 1 Vorlesung + 1 Übung
- Hausaufgaben (*Haus* bedeutet: zuhause bearbeiten, in der Übung diskutieren)
 - Aufgaben im Skript
 - Aufgaben in `autotool`
- Prüfungszulassung: 50 Prozent der autotool-Pflichtaufgaben
- Klausur (2 h, keine Hilfsmittel). — Oder?
- Quelltexte, Planung und Diskussion der Übungsaufgaben
<https://gitlab.imn.htwk-leipzig.de/waldmann/cp-ws22> (Projekt-Mitgliedschaft beantragen, Zugang wird dann auf Mitglieder eingeschränkt)

Literatur

- Krzysztof Apt: *Principles of Constraint Programming*,
<https://www.cambridge.org/catalogue/catalogue.asp?isbn=9780521825832>
- Daniel Kroening, Ofer Strichman: *Decision Procedures*,
Springer 2008.
<https://www.decision-procedures.org/>
- Petra Hofstedt, Armin Wolf: Einführung in die
Constraint-Programmierung, Springer 2007.
<https://link.springer.com/book/10.1007/978-3-540-68194-6>
- Uwe Schöning: Logik für Informatiker, Spektrum Akad.
Verlag, 2000.

Hausaufgaben

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

`https://web.archive.org/web/20200227195925/http://www.nada.kth.se/~viggo/problemist/compendium.html`

Beispiel: Minimum File Transfer Scheduling (node195).
Erläutern Sie die Spezifikation an einem Beispiel.

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: lösen Sie eine der offenen Fragen `https://erich-friedman.github.io/mathmagic/unsolved.html`)

3. Aufgabe: formalisieren Sie `https`:

`//www.janko.at/Raetsel/Wolkenkratzer.`

- **Unbekannte:** $h_{x,y} \in \{0, \dots, n-1\}$ für $x, y \in \{0, \dots, n-1\}$

- **Constraint für eine Zeile x :**

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

Bsp: $n = 4$, Vorgabe links 2, rechts 1, kompatibel sind

$[0, 2, 1, 3], [2, 0, 1, 3], [2, 1, 0, 3], [2, 1, 0, 3]$.

entspr. für Spalten

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

Folge-Aufgabe: *geht das auch polynomiell?*

4. 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: Konjunktion, Disjunktion, Implikation, Äquivalenz

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*:
 $\text{Mod}(M) = \{b \mid \forall F \in M : b \models F\}$

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>

Programmbeispiel zu Ramsey

Quelltext in `Ramsey.hs`

```
num p q = 10 * p + q ; n x = negate x
f = do
  p <- [1..5] ; q <- [p+1 .. 5] ; r <- [q+1 .. 5]
  [ [ num p q, num q r, num p r, 0 ]
    , [ n $ num p q, n $ num q r, n $ num p r, 0 ] ]
main = putStrLn $ unlines $ do
  cl <- f ; return $ unwords $ map show cl
```

Ausführen:

```
runghc Ramsey.hs | minisat /dev/stdin /dev/stdout
```

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.
- Programm = Header

`p cnf <#Variablen> <#Klauseln>`, dann Klauseln

Beispiel

```
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

Löser: `minisat input.cnf output.text`

Modellierung durch SAT: N Damen

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

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

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 ; q <- exists
```

```
        assert $ p && not q
```

```
        return [p,q::Bit]
```

```
    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/

N-Queens mit Ersatz

- solveWith minisat \$ do
 let n = 40
 -- f ist Liste aller Positionen (wie auf Folie)
 f = (,) <\$> [0..n-1] <*> [0..n-1]
 qss <- replicateM n \$ replicateM n (exists @Bit)
 let q (x,y) = qss !! x !! y
 assert \$ and \$ do qs <- qss; return \$ or qs
 assert \$ and \$ do
 a <- f ; b <- f
 guard \$ bedroht a b
 return \$ not (q a) || not (q b)
 return qss
- bedroht a@(ax,ay) b@(bx,by) = a /= b
 && (ax == bx || ay == by || abs (ax-bx) == abs (ay-

Zusammenfassung Ersatz (bisher)

- innerhalb von `solveWith`:
boolesche Operatoren (`not`, `(||)`, usw.)
werden angewendet auf *Formeln* (Typ `Bit`),
nicht auf *Wahrheitswerte* (Typ `Bool`)
- `exists @Bit` konstruiert neue Variable,
ist monadische Aktion (benutze `do`, `replicateM`, usw.)
- innerhalb von `assert`: funktional (nicht monadisch)
- noch nicht behandelt haben wir diesen Widerspruch:
wir konstruieren beliebige Formel, Solver erwartet CNF

Hausaufgaben

1. unterschiedliche Halbringe auf zwei Elementen?
2. für die Formel $S(b, h)$ (abhängig von zwei 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 (a - Sprache/Bibliothek beliebig, b - Haskell/Ersatz)

(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 das angegebene Ersatz-Programm zu N -Queens:

- welche Größe haben die erzeugten Formeln (unter den `assert`)? (in Abhängigkeit von N)
- welche Laufzeit hat das Programm?

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

(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. Realisierung der SAT-Kodierung von $R(a, b)$: benutzen Sie eine Funktion, die alle Teilfolgen gegebener Länge bestimmt.

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
```

Verwenden Sie `subs a [1 .. n]` zur Auswahl des K_a sowie `subs 2 xs` zur Auswahl der Kanten.

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)

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

Spezifikation:

- Gegeben F , gesucht 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`)

```
-- (a || b) wird nur einmal T-transformiert:  
let { s = a || b } in s && (c === s)
```

Tseitin-Transformation in Ersatz (Bsp)

- so ausprobieren:

```
ghci> :set -XTypeApplications
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)
```

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

And-Inverter-Graphen

- Ersatz verwendet diesen Datentyp für Formeln

```
data Bit = Var Literal | Not Bit | And [Bit]
```

mit den Invarianten:

- kein Argument von `And` ist `And`
- das Argument von `Not` ist nicht `Not`

- die Implementierung von `(||)`, `(&&)`, `not` müssen die Invarianten erhalten
- die Tseitin-Transformation wird nur für `And` benötigt (aber mit beliebig vielen Argumenten)

Plaisted-Greenbaum-Transformation

- Tseitin für $a \wedge b$
erzeugt neue Variable c
und Klauseln $(a \wedge b) \rightarrow c, c \rightarrow a, c \rightarrow b$.
- `assert (a && b)` erzeugt zusätzlich Klausel c .
- damit wird Klausel $(a \wedge b) \rightarrow c$ nicht mehr benötigt.
- die Plaisted-Greenbaum-Transformation erzeugt diese unnötigen Klauseln von vornherein *nicht*.
- für moderne SAT-Solver entsteht dadurch keine Einsparung von Lösungszeit. Man spart nur Platz und evtl. Laufzeit bei Formel-Erzeugung.

Tseitin-Transformation (Aufgaben)

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 eine 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?

Komplexität der Erfüllbarkeit

Motivation, Begriffe

- Laufzeit-Komplexität eines Programms P
ist die Funktion f mit $f(n) =$ die maximale Laufzeit von P
über alle Eingaben der Größe $\leq n$.
- solche Funktionen vergleicht man *asymptotisch*
(ignoriert Anfangswerte und konstante Faktoren)
- Laufzeit-Komplexität einer Aufgabe A :
die minimale Komplexität aller Programme, die A lösen
- die Erfahrung lehrt: $\text{SAT} \notin \text{P}$, das ist jedoch unbewiesen,
man konnte bisher nur zeigen: SAT ist NP-vollständig.

Wiederholung: NP-Vollständigkeit

- nichtdeterministische Turingmaschine (NDTM)
 - Rechnung ist eine Baum,
 - jeder Knoten ist eine Konfiguration (Bandinhalt, Kopfzustand, Kopfposition),
 - jede Kante ist ein Rechenschritt
 - Rechnung ist erfolgreich, wenn in wenigstens einem Blatt der Zustand akzeptierend ist
- $\text{NP} :=$ die Sprachen, die sich in Polynomialzeit durch NDTM entscheiden lassen
- Reduktion $M \leq_P L \iff \exists f : \forall x : x \in M \iff f(x) \in L$ und f ist P-berechenbar
- L ist NP-vollständig: $L \in \text{NP}$ und $\forall M \in \text{NP} : M \leq_P L$

Wiederholung: SAT ist NP-vollständig

SAT \in NP ist klar (NDTM rät die Belegung)

Sei $M \in$ NP, zu zeigen $M \leq_P$ SAT. Gegeben ist also das Programm einer NDTM, die M in Polynomialzeit akzeptiert

- übersetze eine Eingabe x für diese Maschine in eine Formel $f(x)$ mit $x \in M \iff f(x) \in$ SAT:
- benutzt Unbekannte $C(t, p, a) : \iff$ zur Zeit t steht an Position p das Zeichen a .
- Klauseln legen fest:
für $t = 0$ steht die Eingabe auf dem Band,
 $\forall t$: von t nach $t + 1$ richtig gerechnet (lt. Programm)
schließlich akzeptierender Zustand erreicht

Die Tseitin-Transformation

- die Tseitin-Transformation beweist $\text{SAT} \leq_P \text{3SAT}$
denn sie in erzeugt aus einer beliebigen Formel F
in Polynomialzeit
eine erfüllbarkeitsäquivalente Formel G in konjunktiver
Normalform mit 3 Literalen je Klausel
- vereinfacht den Beweis der NP-Vollständigkeit weiterer
Aufgaben (z.B. Graphenfärbung) 3COL.
zu zeigen ist: $\forall A \in \text{NP} : A \leq_P \text{3COL}$
man benutzt $A \leq_P \text{SAT} \leq_P \text{3SAT} \leq_P \text{3COL}$.

Was nützen diese Aussagen?

- 1. sie beantworten die Frage: welche Aufgaben lassen sich SAT-kodieren: Aufgabe A hat SAT-Kodierung mit polynomieller Formelgröße $\iff A \in \text{NP}$.
- 2. SAT ist NP-vollständig:
falls $\text{SAT} \in P$ (es gibt einen SAT-Solver in Polynomialzeit)
dann folgt $\text{NP} \subseteq P$.
(jede NP-vollständige Aufgabe ließe sich durch SAT-Kodierung in Polynomialzeit lösen)
das ist unwahrscheinlich (nach jetzigem Kenntnisstand)
 - für jedes bekannte SAT-Lösungsverfahren gibt es schwere Eingaben
 - Verfahren sind trotzdem erstaunlich leistungsfähig

SAT-Solver

Überblick

Spezifikation:

- Eingabe: eine Formel in CNF
- Ausgabe:
 - eine erfüllende Belegung
 - *oder* ein Beweis für Nichterfüllbarkeit

Verfahren:

- evolutionär (Genotyp = Belegung)
- lokale Suche (Walksat)
- DPLL (Davis, Putnam, Logeman, Loveland)

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

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: Heuristik, Ergänzungen

- 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 <http://fmv.jku.at/kissat/> (Armin Biere)
- wesentlich für Lernen und Vorverarbeitung ist *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

- 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. $l = \overline{x_2}$, $c = (x_1 \vee \overline{x_2})$, $d = (x_2 \vee x_3)$,
 $\text{Res}_l(c, d) = (x_1 \vee x_3)$.
- Satz: $\{c, d\} \models \text{Res}_l(c, d)$.
- 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

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

- jede partielle Belegung b mit $\text{dom}(b) = \{v_1, \dots, v_k\}$ entspricht einer Konjunktion

$$B = (v_1 \leftrightarrow b(v_1)) \wedge \dots \wedge (v_k \leftrightarrow b(v_k))$$

- beim Lösen der Formel (Klauselmenge) F :

- Konflikt bei partieller Belegung b :

man kann $C = \neg B = (\neg(v_1 \leftrightarrow b(v_1)) \vee \dots \vee \neg(v_k \leftrightarrow b(v_k)))$ lernen

Beweis: zu zeigen ist $F \models C$, folgt aus $\text{Mod}(F \cup \{B\}) = \emptyset$

- ... sollte man aber nicht, denn C ist mglw. zu groß (enthält Literale, die am Konflikt gar nicht beteiligt sind)

Lernen (Implementierung) und Backjump

- benutze (resolviere) Klauseln, 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 \bar{l}_h unit-propagiert.

Variablen-Elimination durch vollst. Resolution

- für Formel (Klauselmenge) F und Variable v :
 $\text{Pos}_v(F) = \{c \mid c \in F, v \in c\}$; $\text{Neg}_v(F) = \{c \mid c \in F, \neg v \in c\}$
 $\text{Res}_v(F) = \bigcup_{p \in \text{Pos}_v(F), n \in \text{Neg}_v(F)} \text{Res}_v(p, n)$
- Satz: F ist erfüllbarkeitsäquivalent zu G
mit $G := F \setminus (\text{Pos}_v(F) \cup \text{Neg}_v(F)) \cup \text{Res}_v(F)$.
- das (iteriert) ist ein vollständiges Lösungsverfahren!
aber unpraktisch, weil $|G| \gg |F|$ möglich ist:
 $|G| = |F| - (|\text{Pos}_v(F)| + |\text{Neg}_v(F)|) + |\text{Pos}_v(F)| \cdot |\text{Neg}_v(F)|$.
- Anwendung: für solche v , für die $|G| \leq |F| + \Delta$
- Quelle: Een und Biere: *Effective Preprocessing ...*, SAT 2005, <http://minisat.se/downloads/SatELite.pdf>
dort weitere Vorverarbeitungs-Verfahren

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. Schleife verlassen, wenn c nur noch ein Literal der aktuellen Entscheidungstiefe ... :

- (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 Klauselmengemenge gehört?

3. für eine Teilformel F wird die Tseitin-Transformation durchgeführt und für die dabei entstehende Variable v_F (mit $v_f \leftrightarrow F$) ein assert.

Was passiert bei vollständiger Elimination von v_F ?

(Es ist anzunehmen, daß das alle Solver tun. Deswegen ist für `ersatz` wohl keine

Plaisted-Greenbaum-Transformation (J. Symb. Comp. 2(3) 1986, pp 293-304) `https:`

`//doi.org/10.1016/S0747-7171(86)80028-1`
notwendig.)

4. Wenn G aus F durch vollständige Resolution (Elimination) von v entsteht: Konstruieren Sie aus einem Modell b für G ein Modell für F (wie ist v zu belegen?)

5. für eine unerfüllbare Schubfachschluß-Formel (in jeder Zeile $1, \dots, H$ höchstens einer wahr, in jeder Spalte $1, \dots, B$ wenigstens einer wahr, z.B. für $B = 3, H = 2$ die Klauselmenge $\{\overline{12}, \overline{13}, \overline{33}, \overline{45}, \overline{46}, \overline{56}, 12, 34, 56\}$

einen möglichst kurzen Beweis der Unerfüllbarkeit

- (a) mit DPLL (ohne Lernen)
- (b) mit DPLL und CDCL
- (c) ... und vorheriger Variablen-Elimination
- (d) ... *nur* mit Variablen-Elimination (ohne DPLL)

6. eine nicht erfüllbare 2SAT-Formel (jede Klausel genau zwei Literale) angeben, für die DPLL möglichst lange braucht. Vergleichen mit DPLL + CDCL, Variablen-Elimination.

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

Für Klauselmengemenge 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 ?

8. *conflict clause minimization* (Sörenson, SAT 2005,

[http://minisat.se/downloads/MiniSat_v1.13_](http://minisat.se/downloads/MiniSat_v1.13_short.pdf)

[short.pdf](http://minisat.se/downloads/MiniSat_v1.13_short.pdf)): Ein Beispiel angeben. Entsprechende

Ergänzung der autotool-Aufgabe vorschlagen.

9. 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 an/abschalten

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)

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

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

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

$$\text{so da\ss } \text{enc}(x + y) = \text{enc}(x) +_{\text{enc}} \text{enc}(y)$$

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

das sind $2n$ Hilfsvariablen, aber bei `assert (amo xs)`

werden n davon eliminiert durch Vorverarbeitung oder UP

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)

Propositionale vs. Funktionale Kodierungen

- Kodierung einer Booleschen Funktion F mit Variablen V durch Formel G mit Variablen $V \cup H$ heißt ...
(Vorsicht: folgende Bezeichnungen sind nicht Standard)
 - *propositional* (ist der Oberbegriff): G und F erfüllbarkeitsäquivalent und belegungs-erhaltend:
für all b gilt: $b \models F \iff \exists b' : b \subseteq b' \wedge b' \models G$
 - *funktional*: propositional und für jedes $b \models F$ gibt es *genau ein* b' ... Belegung von H ist eine *Funktion* der Bel. von V , kann durch *Schaltkreis* realisiert werden
- Negation von Schaltkreis ist Schaltkreis, Negation von $\exists b' : G$ ist $\forall b' : \neg G$, das ist kein Erfüllbarkeitsproblem!
Bsp: $\text{Count}_{>1}(x) = \neg \text{Count}_{\leq 1}(x) = \neg \text{AMO}(x)$
mit log-Kodierung von AMO ergibt das $\forall b_1, b_0 : \dots$

Propositional vs. Funktional

- oft ist $V = E \cup A$ (disjunkt),
 $E =$ Eingabe-, $A =$ Ausgabe-Variablen,
Def: funktional *bezüglich* E : Belegung von A (und H) ist
Funktion der Belegung von E
- mein Eindruck: publiziert werden überwiegend
propositionale Kodierungen (PK). `ersatz` benutzt sehr
oft FK. Das scheint einfacher und paßt besser zu Haskell.
Ist es ein Nachteil beim Constraint-Schreiben/Lösen?
- viele publizierte PK entstehen aus geeigneter FK nach
Vorberarbeitung und UP. (Ü: für log-AMO)
- Forschungsfrage: wo geht das scheinbar nicht?
(und bei genauer Betrachtung dann vielleicht doch?)

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 Eigenschaft für die log-Kodierung von AMO?
Zu betrachten ist eine Belegung $b = \{(x_i, 1), (x_j, 1)\}$.
Wird durch UP ein Konflikt erreicht?

Praktische Eigensch. (II) – Forcing

- für CNF F auf Variablen $V = \{v_1, \dots, v_n\}$ und 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: log-Kodierung von $\text{AMO}(x)$: Betrachte $b = \{(x_i, 1)\}$.

Alle anderen x_j müssen dann falsch sein.

Wird das durch UP erreicht?

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?

- Ü: ist das forcing?

Aufgaben

1. Vergleichen Sie die *commander*-Kodierung für AMO von Klieber und Kwon, Int. Workshop on 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
2. Für die *commander*-Kodierung und die *sqrt*-Kodierung: prüfen Sie, ob die im im Paper angegebene PK tatsächlich aus unserer FK (Beispiel-Quelltexte) entsteht. Sehen Sie sich kleine AMO-Formeln an, vermessen Sie größere mittels mini/kissat (Logging des Preprocessing)

3. Für die sqrt-Kodierung für AMO von Chen 2010:

Benutzen Sie die gleiche Idee für eine höherdimensionale Anordnung, z.B. AMO von 30 Variablen als $2 \times 3 \times 5$.

Teilen Sie AMO für 2^n Variablen in $2 \times \dots \times 2$ ein und vergleichen Sie mit der log-Kodierung.

(sqrt-AMO ist funktional, log-AMO ist es im Original nicht, aber mit dieser Herleitung doch?)

4. für die lineare AMO-Kodierung (Addition auf $\{0, 1, \geq 2\}$):

untersuchen Sie den Unterschied zwischen der Verwendung von `foldr` und `foldb`

5. für jede der betrachteten AMO-Kodierungen: wie kann mit möglichst wenig Zusatz-Aufwand EXO erhalten?

6. Für den Rösselsprung:

- zusätzliches `assert $ a !! 0 !! 0` diskutieren und ausprobieren
- AMO und ALO durch EXO ersetzen
- umbauen, so daß ein kreuzungsfreier Weg der Länge $= l$ (zusätzlicher Eingabe-Parameter) beschrieben wird.
- 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/>

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

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

Desgl. für die Kodierung von $ITE(i, t, e; x)$ (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.

8. (Vorfreude, schönste ...) (ab 1. Dezember) <https://www.mathekalender.de/wp/de/kalender/>

Prädikatenlogik

Plan

(für den Rest der Vorlesung)

- Prädikatenlogik (Syntax, Semantik)
- existentielle konjunktive Constraints
in verschiedenen Bereichen, z. B.
Gleichungen und Ungleichungen auf Zahlen ($\mathbb{Z}, \mathbb{Q}, \mathbb{R}$)
- beliebige Boolesche Verknüpfungen
SAT modulo T (= SMT), DPLL(T)
- Bit-blasting (SMT \rightarrow SAT)

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)

Theorien

Def: $\text{Th}(S) := \{F \mid S \models F\}$

(Die Theorie einer Struktur S ist die Menge der Sätze, die in S wahr sind.)

Bsp: „ $\forall x : \forall y : x \cdot y = y \cdot x$ “ $\in \text{Th}(\mathbb{N}, 1, \cdot)$

Für K eine Menge von Strukturen:

Def: $\text{Th}(K) := \bigcap_{S \in K} \text{Th}(S)$

(die Sätze, die in jeder Struktur aus K wahr sind)

Bsp: „ $\forall x : \forall y : x \cdot y = y \cdot x$ “ $\notin \text{Th}(\text{Gruppen})$

... denn es gibt nicht kommutative Gruppen, z.B. $SL(2, \mathbb{Z})$

Unentscheidbarkeit

- (Alonzo Church 1938, Alan Turing 1937)
Das folgende Problem ist nicht entscheidbar:
 - Eingabe: eine PL-Formel F
 - Ausgabe: *Ja*, gdw. F allgemeingültig ist.
- Beweis: kodiert das Halteproblem für ein universelles Berechnungsmodell als eine Wahrheitsproblem der PL
Rechnung einer Turingmaschine kodiert durch 2-stellige Funktion $f(i, t) =$ der Inhalt von Zelle i zur Zeit t .
- diese mathematische Fragestellung (von David Hilbert, 1928) begründet die Wissenschaft der Informatik.
(die Berechenbarkeits-Theorie)

Folgerungen aus Unentscheidbarkeit

Suche nach (effizienten) Algorithmen für Spezialfälle
(die trotzdem ausreichen, um interessante
Anwendungsprobleme zu modellieren)

- Einschränkung der Signatur (Bsp: keine F.-S., nur einstellige F.-S, nur einstellige Rel.-S.)
- Einschränkung der Formelsyntax
 - nur bestimmte Quantoren, nur an bestimmten Stellen (im einfachsten Fall: ganz außen existentiell)
 - nur bestimmte Verknüpfungen (Bsp: nur durch \wedge)
- Einschränkung auf Theorien von gegebenen Strukturen
Bsp: $F \in \text{Th}(\mathbb{N}, 0, +)$? $G \in \text{Th}(\text{Gruppen})$

Lineare Gleichungen und Ungleichungen

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)

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)

- Ellipsoid-Verfahren (für OP): polynomiell

- Fourier-Motzkin-Verfahren (für EP)

vgl. mit Elimination durch vollständige Resolution

exponentielle Laufzeit (häufig)

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

- mit <https://projects.coin-or.org/Clp> lösen:

```
clp check.lp solve solu /dev/stdout
```

Fourier-Motzkin-Verfahren

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

- die Form „ $x (\leq | \geq)$ (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.

Aufgaben

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.
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`

Vorverarbeitung eines Terminationsproblems durch

sparse tiling, dann Gewichtsfunktion: siehe Geser, Hofbauer, Waldmann FSCD 2019 <https://drops.dagstuhl.de/opus/volltexte/2019/10528/>

6. cryptominisat benutzen (Beispiele), verstehen (Paper lesen, wie wird DPLL/CDCL für XOR-Klauseln angepaßt?)

(Integer/Real) Difference Logic

Motivation, Definition

- viele Scheduling-Probleme enthalten:
 - Tätigkeit i dauert d_i Stunden
 - i muß beendet sein, bevor j beginnt.
- das führt zu Constraintsystem:
 - Unbekannte: $t_i =$ Beginn von i
 - Constraints: $t_i \leq t_j - d_i$
- das ist Spezialfall eines linearen Ungleichungssystems, mit einfachem Lösungsverfahren,
wird später verwendet als Unterprogramm in DPLL(T)

Constraint-Graphen für IDL

- Für gegebenes IDL-System S konstruiere gerichteten kantenbewerteten Graphen G
 - Knoten $i =$ Unbekannte t_i
 - gewichtete Kante $i \xrightarrow{d} j$, falls Constraint $t_i \leq t_j + d$
- beachte: Gewichte $d \in \mathbb{Z}$ (oder \mathbb{Q} , ist äquivalent)
(Ü: wenn alle ≥ 0 : Problem ist trivial lösbar. Wie?)
- Satz: S lösbar $\iff G$ besitzt keinen gerichteten Kreis mit negativem Gewicht.
(Implikation \Rightarrow ist offensichtlich, wir brauchen \Leftarrow)
- Ansatz: $t_i =$ minimales Gewicht aller Wege von 1 zu i
Diskussion: min existiert nicht? Weg existiert nicht?

Kürzeste Wege in Graphen

- (single-source shortest paths)
 - Eingabe:
 - * gerichteter Graph $G = (V, E)$
 - * Kantengewichte $w : E \rightarrow \mathbb{R}$
 - äquivalent: Matrix $w : V \times V \rightarrow \mathbb{R} \cup \{+\infty\}$
 - * Startknoten $s \in V$
 - Ausgabe: Funktion $D : V \rightarrow \mathbb{R} \cup \{-\infty, +\infty\}$ mit $\forall x \in V : D(x) = \text{minimales Gewicht aller Wege von } s \text{ nach } x$
- der bekannte (und schnelle) Algorithmus von Dijkstra funktioniert nur, falls $\forall i, j : w(i, j) \geq 0$ (Ü: Beispiel)
Lösung: Algorithmus von Bellmann (1958), Ford (1956)

Lösungsidee

iterativer Algorithmus mit *Zustand* $d : V \rightarrow \mathbb{R} \cup \{+\infty\}$.

$$d(s) := 0, \forall x \neq s : d(x) := +\infty$$

while es gibt eine Kante $i \xrightarrow{w_{i,j}} j$ mit $d(i) + w_{i,j} < d(j)$

$$d(j) := d(i) + w_{i,j} \quad // \text{ Kante } ij \text{ wird entspannt}$$

jederzeit gilt die *Invariante*:

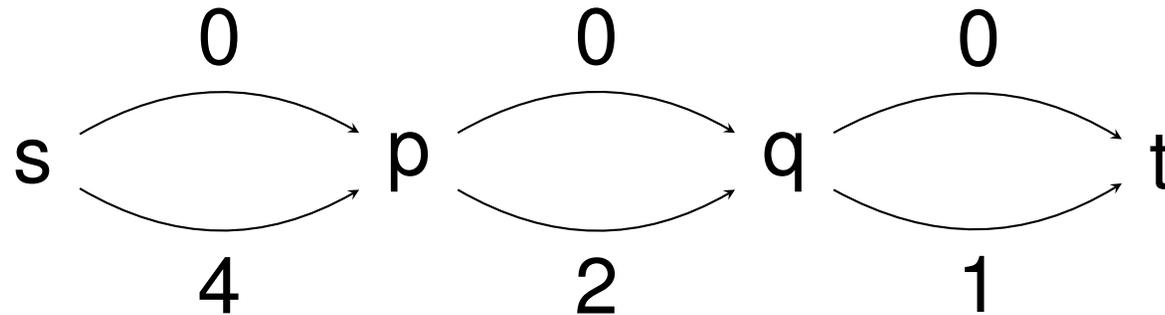
- $\forall x \in V$: es gibt einen Weg von s nach x mit Gewicht $d(x)$
- $\forall x \in V : D(x) \leq d(x)$.

verbleibende Fragen:

- Korrektheit (falls Termination)
- Auswahl der Kante (aus mehreren Kandidaten)
- Termination, Laufzeit

Laufzeit

- exponentiell viele Relaxations-Schritte:



- besser (polynomiell): Bellman-Ford

for i from 1 to $|V|$: jede Kante $e \in E$ einmal entspannen
dann testen, ob alle Kanten entspannt sind.

(d. h. $d(j) \geq d(i) + w_{i,j}$)

Wenn nein, dann existiert negativer Kreis. (Beweis?)

- Dijkstra ist schneller, aber nur bei Gewichten ≥ 0 korrekt

Ganzzahlige lineare Ungleichungen

(Mixed) Integer Programming

- “linear program” (LP): lineares Ungleichungssystem mit Unbekannten aus \mathbb{Q}
- “integer program” (IP): lineares Ungleichungssystem, mit Unbekannten aus \mathbb{Z}
- “mixed integer program” (MIP): lineares Ungleichungssystem, mit Unbekannten aus \mathbb{Q} und \mathbb{Z}
- die Komplexität steigt: $LP \in P$, $MIP \in NP^c$
- mit MIP kann man Boolesche Constraints simulieren
... sollte man aber nicht: Boolesche Lösungsverfahren (DPLL, CDCL) sind besser als numerische

MIP-Beispiel

LP-Format mit Abschnitten

- `General` für ganzzahlige Unbekannte
- `Binary` für Unbekannte in $\{0, 1\}$

Minimize obj: y

Subject To

$$c1: 2x \leq 1$$

$$c2: -2x + 2y \leq 1$$

$$c3: 2x + 2y \geq 1$$

General x y

End

Lösen mit <https://projects.coin-or.org/Cbc>:

```
cbc check.lp solve solu /dev/stdout
```

Ü: ausprobieren und erklären: nur x , nur y ganzzahlig

MIP-Lösungsverfahren

- Ansatz: ein MIP M wird gelöst, indem eine Folge von LP L_1, \dots gelöst wird.
- Def: *Relaxation* $R(M)$: wie M , alle Unbekannten reell.
- *Einschränkung*: für eine ganze Unbekannte x_i falls $\max\{x_i \mid \vec{x} \in \text{Mod}(R(M))\} = B < \infty$, füge Constraint $x_i \leq \lfloor B \rfloor$ hinzu
- *Fallunterscheidung (Verzweigung)*: wähle eine ganze Unbekannte x_i und $B \in \mathbb{R}$ beliebig:
$$\text{Mod}(M) = \text{Mod}(M \cup \{x_i \leq \lfloor B \rfloor\}) \cup \text{Mod}(M \cup \{x_i \geq \lceil B \rceil\})$$
entspricht *decide* in DPLL — aber es gibt kein CDCL

SAT als IP, Komplexität von IP

- es gilt $\text{SAT} \leq_P \text{IP}$

Beweis durch Funktion $T : \text{CNF} \rightarrow \text{IP}$ mit

- T ist in Polynomialzeit berechenbar
- $\forall F \in \text{CNF} : F \text{ erfüllbar} \iff T(F) \text{ lösbar}$

Lösungsidee:

- Variablen von $T(F) =$ Variablen von F
- Wertebereich der Variablen ist $\{0, 1\}$
- Negation durch Subtraktion, Oder durch Addition, Wahrheit durch ≥ 1

- Folgerung: aus $\text{SAT} \in \text{NPc}$ folgt $\text{IP} \in \text{NPc}$,
deswegen kein IP- oder MIP-Solver in Polynomialzeit
(oder $P = \text{NP} = 1$ Million Dollar)

Travelling Salesman als MIP

(dieses Bsp. aus Papadimitriou und Steiglitz:
Combinatorial Optimization, Prentice Hall 1982)

Travelling Salesman:

- Instanz: Gewichte $w : \{1, \dots, n\}^2 \rightarrow \mathbb{R}_{\geq 0} \cup \{+\infty\}$ und Schranke $s \in \mathbb{R}_{\geq 0}$
- Lösung: Rundreise mit Gesamtkosten $\leq s$

Ansatz zur Modellierung:

- Variablen $x_{i,j} \in \{0, 1\}$, Bedeutung: $x_{i,j} = 1 \iff$ Kante (i, j) kommt in Rundreise vor
- Zielfunktion?
- Constraints — reicht das: $\sum_i x_{i,j} = 1, \sum_j x_{i,j} = 1$?

Travelling Salesman als MIP (II)

Miller, Tucker, Zemlin: *Integer Programming Formulation and Travelling Salesman Problem* JACM 7(1960) 326–329

- zusätzliche Variablen $u_1, \dots, u_n \in \mathbb{R}$
- Constraints C : $\forall 1 \leq i \neq j \leq n : u_i - u_j + nx_{i,j} \leq n - 1$

Übung: beweise

- für jede Rundreise gibt es eine Belegung der u_i , die C erfüllt.
- aus jeder Lösung von C kann man eine Rundreise rekonstruieren.

Was ist die anschauliche Bedeutung der u_i ?

min und max als MIP

- kann man den Max-Operator durch lin. Ungln simulieren?
(gibt es äq. Formulierung zu $\max(x, y) = z$?)
- Ansatz: $x \leq z \wedge y \leq z \wedge (x = z \vee y = z)$,
aber das *oder* ist verboten.

Idee zur Simulation von $A \leq B \vee C \leq D$:

- neue Variable $f \in \{0, 1\}$
- Constraint $A \leq B + \dots \wedge C \leq D + \dots$

falls eine obere Schranke S für die Werte von A, B, C, D bekannt ist

Übungen zu DL, MIP (für KW 53)

- Beispiel für Dijkstra, Bellmann-Ford
- Diskussion, Beispiel für TSP als MIP
- Formulierung eines SAT-Problems als IP, Lösung mit CBC.
- Überdeckungsproblem (möglichst wenige Damen, die das gesamte Schachbrett beherrschen) as IP,
Constraint-System programmatisch erzeugen, z.B.
`https://hackage.haskell.org/package/limp,`
Lösen mit `https://hackage.haskell.org/package/limp-cbc`
- Auswertung Adventskalender, Planung Projekte

SMT (Satisfiability Modulo Theories)

Definition, Lösungsverfahren (Plan)

- 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

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: $i(c_1 \dots c_n) := i(c_1) \circ \dots \circ i(c_n)$
- i kompatibel mit R , falls $\forall (l, r) \in R : i(l) > i(r)$.
- 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)
```

Aufgaben

1. 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

Lösen Sie nun die gleiche Aufgabe mit QF_BV (Bitvektoren)

2. Dieses Beispiel in 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/sum-of-three-cubes-problem-solved-for-stubborn-number-33>

3. wählen Sie zufällig in SMTLIB eine (quantorenfreie) Logik und dort eine Benchmark. Erklären Sie die Benchmark. Wenden Sie verschiedene SMT-Solver an (z.B. Z3 und Z3++) und vergleichen Sie Laufzeiten. Ändern Sie die Formel (vorsichtig), erläutern Sie die Änderungen der

Belegung oder Erfüllbarkeit.

4. die zitierte Hamiltonkreis-Kodierung (oder die früher zitierte MIP-Kodierung dafür) in SMTLIB-Syntax hinschreiben (in möglichst einfacher Logik) und ausprobieren.
5. Eine kompatible arktische Matrix-Interpretation bestimmen für $a^2b^2 \rightarrow b^3a^3$.

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

Hinweis: weil diese Regel quadratische lange

Ableitungen gestattet (geben Sie welche an), aber solche können bei arktischen Interpretationen nicht vorkommen (warum?)

Uninterpretierte Funktionen (UF)

Motivation, Definition

Interpretation \models Formel,

Interpretation = (Struktur, Belegung)

Die *Theorie* $\text{Th}(S)$ einer Struktur S ist Menge aller in S wahren Formeln: $\text{Th}(S) = \{F \mid \forall b : (S, b) \models F\}$

Beispiel 1: Formel $a \cdot b = b \cdot a$ gehört zur $\text{Th}(\mathbb{N}$ mit Multipl.), aber nicht zu $\text{Th}(\text{Matrizen über } \mathbb{N} \text{ mit Multipl.})$.

Beispiel 2: Formel

$$(x = y) \Rightarrow f(f(x)) = f(f(y))$$

gehört zu *jeder* Theorie (mit passender Signatur),

Gleichheit von Termen

In jeder Algebra gelten diese Formeln:

$$(t_1 = s_1) \wedge \dots \wedge (t_k = s_k) \rightarrow f(t_1, \dots, t_k) = f(s_1, \dots, s_k)$$

(Leibniz-Axiom für die Gleichheit, *functional consistency*)

- Definition: eine Σ -Algebra A heißt *frei*, wenn die Implikation im Leibniz-Axiom eine Äquivalenz ist
- Beispiel: jede Termalgebra ist frei.
- Nicht-Beispiel: $\Sigma = \{+/2\}$, $D = \mathbb{N}$ ist nicht frei.
- Bsp.: eine freie Algebra auf \mathbb{N} zur Signatur $\{f/2\}$ ist $f(x, y) = 2^x \cdot 3^y$ (Satz von Euklid: jede positive natürliche Zahl besitzt *genau eine* Zerlegung in Primzahlpotenzen)

Anwendungen

Für jede Σ -Algebra S gilt:

- Formel F ist allgemeingültig in der *freien* Σ -Algebra
- \Rightarrow Formel F ist allgemeingültig in S .

Vorteil: kein Entscheidungsverfahren für S nötig

Nachteil: Umkehrung gilt nicht.

Anwendung bei Analyse von Programmen, Schaltkreisen:
Unterprogramme (Teilschaltungen) als *black box*,

Roope Kaivola et al.: *Replacing Testing with Formal Verification in Intel Core™ i7 Processor Execution Engine Validation*, Conf. Computer Aided Verification 2009, http://dx.doi.org/10.1007/978-3-642-02658-4_32

Die Logik QF_UF in SMT-LIB

- Bsp: die Formel $(x = y) \wedge (f(f(g(x))) \neq f(f(g(y))))$

```
(set-logic QF_UF) (declare-sort U 0)
(declare-fun f (U) U) (declare-fun g (U) U)
(declare-fun x () U) (declare-fun y () U)
(assert (and (= x y)
              (not (= (f (f (g x))) (f (f (g y))))))
(check-sat)
```

ist nicht erfüllbar,

- d. h., das Gegenteil ist allgemeingültig:

$$\forall f, g, x, y : ((x = y) \rightarrow (f(f(g(x))) = f(f(g(y))))))$$

Ackermann-Transformation

- ... von Formel F in $\mathcal{QF_UF}$ nach Formel F' in *equality logic* (= $\mathcal{QF_UF}$ mit nur nullstelligen Symbolen)
 - ersetze jeden vorkommenden Teilterm F_i durch ein neues nullstelliges Symbol f_i
 - füge *functional consistency constraints* hinzu: für alle $F_i = g(F_{i1}, \dots, F_{ik}), F_j = g(F_{j1}, \dots, F_{jk})$ mit $i \neq j$:
 $(f_{i1} = f_{j1} \wedge \dots \wedge f_{ik} = f_{jk}) \rightarrow f_i = f_j$
- Bsp: $\underbrace{g(h(x))}_{f_1} = h(\underbrace{g(g(y))}_{f_3})$ übers. in $(x = f_4 \rightarrow f_2 = f_5)$
 $\wedge (f_1 = y \rightarrow f_2 = f_3) \wedge (f_1 = f_3 \rightarrow f_2 = f_4) \wedge (f_2 = f_5)$
- Satz: F erfüllbar $\iff F'$ erfüllbar.

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)

Theorie-Solver für Gleichheitslogik

- entscheidet Erfüllbarkeit von Konjunktion von T-Literalen,
Bsp: $(x = y) \wedge \neg(x = z) \wedge (y = z)$
- ein Entscheidungsverfahren ist:
 - bestimme die transitive und symmetrische Hülle $=^*$ der Gleichungen. Bsp: $x =^* z$
 - Formel ist erfüllbar \iff es gibt keine Ungleichung $l \neq r$ mit $l =^* r$.

DPLL(T), Beispiel QF_LRA

- T-Solver für Konjunktion von Literalen
z. B. Simplex, Fourier-Motzkin
- T-Konfliktanalyse:
bei Nichterfüllbarkeit liefert T-Solver eine „Begründung“
= (kleine) nicht erfüllbare Teilmenge (von Literalen $\{a_1, \dots, a_k\}$), dann Klausel $\neg a_1 \vee \dots \vee \neg a_k$ lernen
- T-Deduktion, Bsp: aus $x \leq y \wedge y \leq z$ folgt $x \leq z$
neues (!) Atom $x \leq z$ entsteht durch Umformungen
während Simplex oder Fourier-Motzkin
betrachte $\neg x \leq y \vee \neg y \leq z \vee x \leq z$ als Konfliktklausel,
damit CDCL

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.

Aufgaben

1. freie Algebra:

- weitere freie Algebren zu $\Sigma = \{f/2\}$ über \mathbb{N} .
- die von $F(x) = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} x$; $G(x) = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} x$; $I() = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ erzeugte Algebra ist frei?

2. QF_UF:

(a) warum ist diese Formel nicht erfüllbar?

`https://clc-gitlab.cs.uiowa.edu:`

`2443/SMT-LIB-benchmarks/QF_UF/-/blob/master/`

`TypeSafe/z3.1184163.smt2`

(b) **Benchmarks** `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_UF/-/tree/master/eq_diamond` ausprobieren, angegebene Quelle (SMT 2005) lesen

(c) über alle Benchmarks

[https://clc-gitlab.cs.uiowa.edu:](https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_UF/-/tree/master/)

[2443/SMT-LIB-benchmarks/QF_UF/-/tree/master/](https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_UF/-/tree/master/)

welches ist die schwerste (für Z3, für CVC5)?

(d) Für *quantifiers in the UF theory* [https://smt-comp.](https://smt-comp.github.io/2021/system-descriptions/Yices2.pdf)

[github.io/2021/system-descriptions/Yices2.pdf](https://smt-comp.github.io/2021/system-descriptions/Yices2.pdf):

Beschreibung und Beispiele suchen, Yices installieren,
Beispiele ausprobieren

3. inkrementeller Theorie-Solver:

(a) für QF_IDL (Niewenhuis JACM 2006

<https://www.cs.upc.edu/~roberto/papers.html>),

Unterschiede zu nicht inkrementellem Verfahren.

(b) für QF_LRA : wie kann man Fourier-Motzkin inkrementell verwenden? (Der Anwendungsfall ist: es taucht eine

neue Ungleichung auf, in der Variablen vorkommen, die durch Fourier-Motzkin bereits eliminiert wurden)

4. DPLL(T) in autotool-Aufgabe

5. Diskussion zu Projekten

Bitblasting

Motivation: SMT über Bitvektoren

SMT-Logik \mathcal{Q}_F_BV motiviert durch (Rechner-)Schaltkreis- und (Maschinen-)Programmverifikation

Constraint-Bereich: Bit-Vektoren mit statisch bekannter Breite w

- plus, minus, mal (*modulo* 2^w), min, max, ...
- Vergleiche, (parallele) boolesche Verknüpfungen

```
(declare-fun a () (_ BitVec 12))
(declare-fun b () (_ BitVec 12))
(assert (and (bvult (_ bv1 12) a)
             (bvult (_ bv1 12) b)
             (= (_ bv1001 12) (bvmul a b))))
```

Lazy und Eager Approach für QFBV

- Hintergrund-Theorie = T = CNF-SAT:
 - unbekannter Bitvektor = Folge unbekannter Bits
 - atomares Constraint = aussagenlogische Formel (z.B. Formel für Addier- o. Multiplizierschaltkreis)
- weitere spezifische Eigenschaften, z.B. Assoziativität, Kommutativität von Addition, Multiplikation
- DPLL(T): *lazy*
 T -Solver wird „bei Bedarf“ aufgerufen (mehrfach)
- bit-blasting: *eager*
die Aufgabe wird komplett in die Theorie übersetzt und dann komplett durch den SAT-Solver gelöst
 - Nachteil: das ignoriert spezifische Eigenschaften
 - Vorteil: man braucht nur Übersetzer und SAT-Solver

Bit-Blasting für FD-Constraints

(Wertebereich jeder Unbekannten ist endlich, FD = Finite Domain) Ansatz:

- FD: Unbekannte u mit Bereich $\{0, 1, \dots, n - 1\}$
- \Rightarrow unbekannter Bitvektor $b = [b_0, \dots, b_{w-1}]$ mit Constraints

Realisierungen:

- *binär*: b ist Binärdarstellung von u , $(w = \lceil \log_2 n \rceil)$

Constraint: $b < n$

- *unär* ($w = n$)

– one-hot encoding: $\forall i : b_i \iff (i = u)$

Constraint: $\text{Exactly}_1(b)$

– order encoding (Treppen-Kodierung) $\forall i : b_i \iff (i \leq u)$

Constraint: ?

Bit-Blasting für unendliche Bereiche?

- z.B. als Lösungsverfahren für QF_LRA , QF_NIA
... das geht im allgemeinen nicht, aber eventuell folgt aus den Constraints, daß es genügt, endliche Wertebereiche zu betrachten.
- auch wenn es (zunächst) nicht folgt—man kann es trotzdem probieren! z.B. für $P \in QF_NIA$ nacheinander probieren, ob es Lösungen mit Bit-Breite 1, 2, ... gibt.
Wenn man eine Lösung findet, hat man Glück gehabt.
Man kann aber Pech haben (P ist lösbar, aber nicht mit kleinen Zahlen)

Anwendung: Graceful Labeling

- Für ungerichteten Graph $G = (V, E)$:
eine Abbildung $f : V \rightarrow \{0 \dots |E|\}$ heißt *graceful*,
wenn f injektiv und $\{|f(p) - f(q)| : pq \in E\} = \{1 \dots |E|\}$.
- Bsp: eine solche Abb. für $\bullet - \bullet - \bullet - \bullet$
- Vermutung von Ringel, Kotzig, Rosa (1963): jeder Baum besitzt ein graceful labeling.
- für jedes G : Existenz eines graceful labeling für G ist FD-Constraint. Wir betrachten Lösung durch Bit-Blasting.
zu modellieren sind: Subtraktion (Addition) und (Un)gleichheit

one-hot-Kodierung

- SAT-Kodierung von Exactly_1 ist entscheidend und deswegen ausführlich untersucht, Übersicht in: Hölldobler, Van Hau Nguyen:

<http://www.wv.inf.tu-dresden.de/Publications/2013/report-13-04.pdf>,

vgl. <https://github.com/Z3Prover/z3/issues/755>

- Funktion, Relation \Rightarrow Wertetabelle, Beispiel:

$$f(a, b) = c \text{ wird zu } \bigwedge_{i,j} (a_i \wedge b_j) \Rightarrow c_{f(i,j)}$$

- Ü: Relation $<$ (kleiner als)?
- Ü: partielle Funktion? z.B. Addition auf $\{0, \dots, n - 1\}$
- (partielle) Funktion $D^k \rightarrow W$ benötigt $|D|^k$ Klauseln nur für kleine Bereiche, geringe Stelligkeiten praktikabel

One-Hot-Kodierung mit Ersatz

- **Datentyp:** `data OH = OH [Bit]`
- **eine Unbekannte (mit Bereich $\{0, \dots, w\}$) anlegen:**

```
make w = do
  bs <- replicateM (w+1) $ exists @Bit
  assert $ Ersatz.Counting.exactly 1 bs
  return $ OH bs
```

- **Beziehung zwischen symbolischen und konkreten Daten:**

```
instance Codec OH where
  type Decoded OH = Natural
  decode s (OH bs) = do vs <- decode s bs
    return $ genericLength $ takeWhile not vs
```

One-Hot-Arithmetik (variable Bitbreite)

- instance Equatable OH where

```
OH xs === OH ys =
  let common = min (length xs) (length ys)
  in take common xs === take common ys
```

- variable Bitbreite:

```
instance Num OH where
```

```
OH xs + OH ys = OH $
```

```
flip map [0 .. length xs + length ys] $ \ s ->
```

```
  or $ do -- nicht effizient!
```

```
    (i,x) <- zip [0..] xs
```

```
    (j,y) <- zip [0..] ys
```

```
  guard $ i + j == s
```

```
  return $ x && y
```

Feste (statische) Bitbreite: Zahlen auf Typ-Ebene

- ```
{-# language DataKinds #-} import GHC.TypeLits
data OH (w :: Nat) = OH [Bit] -- Bitbreite im Typ
```

- **Allokation in Ersatz (allgemein)**

```
class Variable t where
 literally :: MonadSAT s m => m Literal -> m t
 exists :: (Variable a, MonadSAT s m) => m a
```

- **Allokation für `OH w`, dabei Typ-Zahl  $\Rightarrow$  Daten-Zahl**

```
instance KnownNat w => Variable (OH w) where
 literally l =
 .. replicateM (natVal (Proxy @w)) l ..
```

danach kann man schreiben `exists @ (OH 3)`

# Reflektion von Zahlen als Typ-Zahlen

- bisher: Zahlen im Typ, Umwandlung zu Daten-Zahlen.  
wir brauchen: die andere Richtung. Bsp: Zahl (Bitbreite) wird im Programm geändert oder ist Benutzer-Eingabe.
- Lösung: Paket `reflection`, enthält Funktion

```
import Data.Reflection
reifyNat :: Integer
 -> (forall (n :: Nat). KnownNat n => Proxy n -> r) -> r
```

$n$  ist *lokal quantifiziert*, kann nicht in  $r$  verwendet werden

- Anw.: Typvar.  $n$  deklariert und mit Wert von  $x$  belegt

```
reifyNat (x :: Integer) $ \ (_ :: Proxy n) ->
 do .. exists @(OH n) ..
```

# Aufgaben

1. Überprüfen Sie *generalized arc-consistency* (forcing) für `sym_diff_eq a b c` (Semantik:  $|a - b| = c$ ) für One-Hot-Kodierung.

Bsp.: Bitbreite 4 (für Bereich  $\{0, 1, 2, 3\}$ ), partielle Belegung  $a_0 = 0, a_1 = 0, b_0 = 0, b_1 = 0$ . Welche  $c_i$  sind dadurch (semantisch) bestimmt? Erhält man diese durch Unit-Propagation? (nehmen Sie dabei an, daß die verwendete Kodierung von exactly-one forcierend ist).

2. zu D. E. Knuth: TAOCP Fasc. 7A (Version 18. Nov. 2022), S. 16 ff: *Graph labeling*

Diskutieren/realisieren Sie Bitblasting (SAT-Kodierung) für das *reverse model* (S. 19)

# Bitblasting für Arithmetik

## Treppen-Kodierung (order encoding)

- Definition:  $\forall i : b_i \iff (i \leq u)$   
Bsp:  $u = 2$  kodiert durch  $b_0 = 1, b_1 = 1, b_2 = 1, b_3 = 0$
- Constraints:  $\forall i : b_i \Leftarrow b_{i+1}$  (Monotonie)
- $\ddot{U}$ : einfache (lineare) Kodierung von  $\leq$   
das erfordert nur linear große Formel  
 $\ddot{U}$ : ist das trotzdem forcierend?
- vergleiche:  $\leq$  ist quadratisch für One-Hot.  
 $\ddot{U}$ : lineare Übersetzung  $f$  von One-Hot nach Treppe,  
dann  $f(x) \leq f(y)$  insgesamt linear für One-Hot  $x, y$ .

# Addition für Treppenkodierung

- (für variable Bitbreite) Addition *genau* wie für one-hot:

```
OH xs + OH ys = OH $
```

```
flip map [0 .. length xs + length ys] $ \ s ->
```

```
 or $ do -- nicht effizient! (kubische Zeit)
```

```
 (i,x) <- zip [0..] xs
```

```
 (j,y) <- zip [0..] ys
```

```
 guard $ i + j == s -- NICHT geändert!
```

```
 return $ x && y
```

Formelgröße: quadratisch. Ü: ist forcierend?

- effizientere Kodierung ( $O(n \log n)$ ) der Addition über Merge-Netze: Een, Sorenson: *Translating Pseudo-Boolean Constraints into SAT, 2007.*

<http://minisat.se/downloads/MiniSat+.pdf>

# Binär-Kodierung, 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
```

# 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.

# Arithmetik für (statisch) fixierte Bitbreite

- `data Bin (w::Nat) = Bin [Bit]`

Gibt es `instance Num (Bin 3)`? Nein:  $5 + 7 \notin \text{Bin } 3$

- relationale Kodierung (Summe raten und überprüfen mit  
`plus_ok :: Bin w -> Bin w -> Bin w -> Bit`)

- funktionale Kodierung mit Überlauf

```
data Bin (w::Nat)
 = Bin { contents::[Bit], overflow::Bit }
```

- Überlauf richtig erzeugen (für Addition trivial, Ü: Multiplikation)
- Überlauf propagieren (aus den Argumenten für `plus` bzw. `mal`)
- Überlauf bei Implementierung von `==`, `<`? beachten

# Aufgaben

1. sind angegeben Implementierungen der binären Arithmetik und Vergleichsrel. fixierter Breite forcierend?
2. Binärzahlen (statt one-hot) für Graceful labeling.
3. Kodierung von vorzeichenbehafteten Zahlen
  - (a) feste Bitbreite: im Zweierkomplement
  - (b) variable 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)
4. Diskutieren Sie richtige Propagation von Überläufen bei Kodierung von Zahlen aus  $\{-B, \dots, B\}$  für Operationen plus, mal, kleiner, gleich.

## 5. weitere Beispiele und Aufgaben zu Bitblasting-Arithmetik:

[https://www.imn.htwk-leipzig.de/~waldmann/talk/22/isr/#solving-qf\\_nia-by-bit-blasting](https://www.imn.htwk-leipzig.de/~waldmann/talk/22/isr/#solving-qf_nia-by-bit-blasting) (ISR 2022)

## 6. Realisieren Sie die Berechnung des Überlaufs bei

(a) Addition

(b) Multiplikation

in  $\text{QF\_BV}$  durch möglichst kleine Formel.

Damit können Sie ein Constraint  $P$  in  $\text{QF\_NIA}$ ,  $\text{QF\_LIA}$  in Constraints  $P_w$  in  $\text{QF\_BV}$  (mit Bitbreite  $w$ ) übersetzen, und evtl. Glück haben, wie in VL beschrieben.

Probieren Sie für einfache (handgeschriebene)

Constraints  $P$  aus, ob (z.B. Z3) für  $P$  oder  $P_1, P_2, \dots$  schneller ist.

# Paralleles SAT-Lösen

## Paralleles SAT-Lösen (Grundlagen)

- Youssef Hamadi, Christoph M. Wintersteiger: *Seven Challenges in Parallel SAT Solving*, AAAI 2012, <https://www.microsoft.com/en-us/research/publication/seven-challenges-in-parallel-sat-solving/>
  - (Portfolio) unabhängige Solver
  - Austausch von (kleinen! - M. Soos) gelernten Klauseln
  - Suche in getrennten Unterräumen ( $x_3 = 0$ ,  $x_3 = 1$ )
  - (Propagation läßt sich nicht parallelisieren)
- Mate Soos: *Parallel SAT Solving*, 2018  
<https://www.msoos.org/tag/parallel-sat-solving/>

# Paralleles SAT-Lösen (heute)

- Ludovic Le Frioux, Souheib Baarir, Julien Sopena, Fabrice Kordon: *PainleSS: a Framework for Parallel SAT Solving*, <https://www.lrde.epita.fr/wiki/Publications/le-frioux.17.sat>
- Proc. SAT Competition 2022, <https://helda.helsinki.fi/handle/10138/347211>
- cube-and-conquer: Maximilian Heisinger: Paracooba, tree-based look-ahead, Armin Biere: *treengeling*, <https://github.com/arminbiere/lingeling>
- probabilistic backdoor
- <https://github.com/caiswgroup/ParKissat-RS>

# Anw: Schaltkreisverifikation

- Schaltkreise (Formeln)  $F$  (Muster-Implementierung) und  $G$  (tatsächliche Implementierung) sind äquivalent, wenn  $\neg \exists x : F(x) \neq G(x)$
- Beispiel:  $F(x)$  naives (quadratisches) atmost-one,  $G(x)$  lineares atmost-one
- ähnlich: Kommutativität der Multiplikation von Ersatz.Bits:  $F(x, y) = x \cdot y$ ,  $G(x, y) = y \cdot x$ .  
Quelltexte: `cp-ws22/kw57/equiv`  
verschiedene Solver ausprobieren. Auch für Assoziativität
- Kommutativität ist nicht trivial: Rekursion nach  $x$ , nach  $y$ .
- Übung: wie schwierig sind Komm./Ass. der Addition?  
Ausprobieren, diskutieren. Geht das mit Unit-Prop.?

# 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

# Übung BMC

- **Software:** `https://gitlab.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

# Zusammenfassung, Ausblick

## Kernaussagen

- Constraint-Programmierung =
  - anwendungsspezifische logische Formel,
  - generische bereichsspezifische Such/Lösungsverfahren
- CP ist eine Form der deklarativen Programmierung
- typische Anwendungsfälle für CP mit Formeln ohne Quantoren, mit freien Variablen, Solver sagt:
  - JA: beweist Existenz-Aussage, rekonstruiere Lösung der Anwendungsaufgabe aus Modell
  - NEIN: das beweist All-Aussage (z. B. Implementierung erfüllt Spezifikation für jede Eingabe)

# Kernthemen

## Aussagenlogik (SAT)

- Grundlagen: Formeln, Resolution, DPLL, CDCL
- Anw.: SAT-Kod. für kombinatorische Aufgaben, Forcing,

## Prädikatenlogik: Bereiche und -spezifische Verfahren:

- Zahlen: Differenzlogik, lineare Ungleichungen (Fourier-Motzkin),
- uninterpretierte Funktionen,

## Prädikatenlogik: allgemeine und spezielle Verfahren:

- Kombination von Theorie-Löser und SAT-Löser (DPLL(T))
- Bit-Blasting für finite-domain-Constraints

# Typische Anwendungen

- Ressourcen-Zuordnungs-, -Optimierungs-Aufgaben mit
  - nicht linearen Funktionen
  - nicht nur konjunktiver Verknüpfung von Teilbedingungen
- Hardware-Verifikation: digitale Schaltnetze, Schaltwerke
- Software-Verifikation:
  - bounded model checking,
  - Terminations-Analyse, siehe  
*Constraint Programming for Analysis of Rewriting*, 13th Intl. School on Rewriting, Tbilisi, 2022, <https://www.imn.htwk-leipzig.de/~waldmann/talk/22/isr/>

# Geschichte der Constraint-Progr. (I)

- Lineare Optimierung (Dantzig 1947 et al.)
  - ⇒ Mixed Integer LP (Gomory 195\* et al.)
  - [https://www-03.ibm.com/ibm/history/exhibits/builders/builders\\_gomory.html](https://www-03.ibm.com/ibm/history/exhibits/builders/builders_gomory.html)
- PROLOG (Kowalski 1974)
  - löst Unifikations-Constraints über Bäumen
  - mit fixierter Suchstrategie (SLD-Resolution)
  - ⇒ Constraint Logic Programming (spezielle Syntax und Strategien für Arithmetik, endliche Bereiche)
  - Global constraints in CHIP*: Beldiceanu, Contejean 1994
  - [http://dx.doi.org/10.1016/0895-7177\(94\)90127-9](http://dx.doi.org/10.1016/0895-7177(94)90127-9)

# Geschichte der Constraint-Progr. (II)

- fixierte, bekannte Suchstrategie: PROLOG
- Strategie innerhalb des CP bestimmt: gecode
- (praktisch unbekannte) Strategie im externen Solver:
  - SAT
    - \* DPLL: Martin Davis, Hilary Putnam (1960), George Logemann, Donald W. Loveland (1962),
    - \* SAT ist NP-vollst. (Steven Cook, Leonid Levin, 1971)
    - \* CDCL: J.P Marques-Silva, Karem A. Sakallah (1996)
  - SMT
    - \* DPLL(T): the *lazy* approach to SMT
    - \* Yices 2006,
    - \* Z3 (de Moura, Bjorner, 2008)

# Constraints für (baum)strukturierte Daten

- Sprache CO4, Dissertation von A. Bau  
<http://abau.org/co4.html>
- Constraint  $c :: P \rightarrow U \rightarrow \text{Bool}$  *in Haskell*
- ... das geht in `ersatz` auch?

Ja — aber nur für  $U = \text{Bool}, [\text{Bool}]$ , usw.,  
und auch dafür nicht vollständig:

```
if a == b then c else d && e
```

```
⇒ ite (a == b) c (d && e)
```

- CO4 gestattet für  $P$  und  $U$ :
  - algebraische Datentypen (`data`)
  - pattern matching (`case`)

# SAT-Kodierung von Bäumen

- kodiere (endl. Teilmengen von) algebraischen Datentypen

data U = C1 T11 .. T1i | C2 T21 .. T2j | ..

- durch Baum mit Grad  $\max(i, j, \dots)$ ,  
in jedem Knoten die FD-Kodierung des Konstruktor-Index

- $e = \text{case } (x :: U) \text{ of}$   
    C1 ..  $\rightarrow e1$  ; C2 ..  $\rightarrow e2$

übersetzt in  $\bigwedge_k (\text{index}(x) = k) \Rightarrow (e = e_k)$

Spezifikation, Implementierung, Korrektheit,  
Anwendungsfälle (Terminations-Analyse, RNA-Design),  
Messungen, Verbesserungen, Erweiterungen.

siehe Publikationen auf <http://abau.org/co4.html>

# Themen für Bachelor/Master-Arbeiten

- Dokumentation, Messung und Verbesserung von ersatz:
  - Relationen (Thema ist schon in Arbeit)
  - Zahlen mit type-level Länge und Überlauf
  - Anzahl-Constraints
- Verbesserung/Alternativen Bitblasting in Termination:  
aktuelle parallele SAT-Solver, SMT-Solver (QFBV)
- Methoden zur Analyse von SAT-Kodierungen
  - automat. Prüfen (Herstellen?) d. Forcing-Eigenschaft
  - Rückübertragung von Laufzeitinformation (wer wird eliminiert/propagiert) auf Quelltext-Ebene
- Vergleich von propositionaler und funktionaler Kodierung bei klassischen (publizierten) Testfällen