Repräsentation von Kombinatorischer Logik

- Abstrakter Datentyp Boolesche Logik:

  Konstruktoren: Erzeugung von boolesche Konstanten und Variablen

  Operationen: Konjunktion, Disjunktion, Negation, ...

  Abfragen: Test auf Erfüllbarkeit, Tautologie, ...

- Basis-Datentyp in EDA-Werkzeugen:

  Simulatoren, Optimierer, Compiler

- Trade-Off zwischen effizienten Operationen und Platzverbrauch

Implementierung als Funktionstabelle

			f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

			g
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

			$f \vee g$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

			$\neg f$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Operationen einfach Punktweise ausführen

Ad: Funktionstabelle

- Funktionstabelle ist immer 2^n Zeilen gross bei n Variablen

- Operationen sind alle linear in der Grösse der Argumente:

  z.B. Konjunktion ergibt Funktionstabelle gleicher Grösse

- Darstellung ist kanonisch:

  zwei äquivalente boolesche Formeln haben dieselbe Funktionstabelle

- Abfragen sind auch linear:

  Tautologie: überprüfe Zeilen auf Gleichheit

  Erfüllbarkeit: suche 1-Zeile

DNF

- potentiell kompakter als Funktionstabelle

  nur Anzahl Kernprimimplikanten bestimmt Grösse

- unmittelbare Implementierung als minimale 2-stufige Schaltung (PLA)

- Disjunktion linear (ohne Minimierung)

- Konjunktion linear und Negation exponentiell (auch ohne Minimierung)

- Erfüllbarkeit mit konstantem Aufwand:

  DNF erfüllbar gdw. mind. ein Monom vorhanden

Exponentielle DNF für Parity

a \ b	0	1	0	1
1	0	1	0	
0	1	0	1	
1	0	1	0	

c

d

$a \oplus b \oplus c \oplus d$

- keine Zusammenfassung von Min-Termen im KV-Diagramm möglich
- nur *volle* Min-Terme als Prim-implikanten (bestehend aus maximaler Anzahl von Literalen)
- DNF für Parity von n Variablen hat 2^{n-1} Monome

Konjunktion von DNF

$$\underbrace{(a \cdot \overline{b} \;\vee\; \overline{a} \cdot b \cdot \overline{c})}_{\text{1. Operand}} \;\wedge\; \underbrace{(a \cdot \overline{b} \;\vee\; \overline{b} \cdot \overline{c})}_{\text{2. Operand}}$$

Ausmultiplizieren

$$a \cdot \overline{b} \cdot a \cdot \overline{b} \;\vee\; a \cdot \overline{b} \cdot \overline{b} \cdot \overline{c} \;\vee\; \overline{a} \cdot b \cdot \overline{c} \cdot a \cdot \overline{b} \;\vee\; \overline{a} \cdot b \cdot \overline{c} \cdot \overline{b} \cdot \overline{c}$$

Vereinfachen der einzelnen Min-Terme

$$a \cdot \overline{b} \;\vee\; a \cdot \overline{b} \cdot \overline{c}$$

Minimierung (z.B. mit Quine-McCluskey)

$$a \cdot \overline{b}$$

Konjunktion von DNF

$$\underbrace{(a \vee b \vee c)}_{\text{1. Operand}} \;\wedge\; \underbrace{(d \vee e \vee f)}_{\text{2. Operand}}$$

Ausmultiplizieren

$$a \cdot d \vee a \cdot e \vee a \cdot f \vee b \cdot d \vee b \cdot e \vee b \cdot f \vee c \cdot d \vee c \cdot e \vee c \cdot f$$

Keine weitere Vereinfachung möglich!

Beispiel lässt sich leicht generalisieren:

Konjunktion zweier DNF mit n und mit m Monomen hat $O(n \cdot m)$ Monome

Ad: Konjunktion von DNF

- maximal linear grösseres Ergebnis (in beiden Argumenten)
- man verliert Minimalität:

  Ausmultiplizieren min. DNF ergibt nicht notwendigerweise min. DNF
- anschliessende Minimierung hätte wiederum exponentiellen Aufwand
- auch implizite Generierung der Monome der Konjunktion möglich

Syntaktische Darstellungen

- z.B. Netzliste der kombinatorischen Schaltung:

  Hypergraph aus Gattern (Knoten) und Signalen (Hyper-Edges)

  (Hyper-Edge: Menge von mit der Kante verbundenen Knoten)

- z.B. Parse-Tree einer booleschen Formel

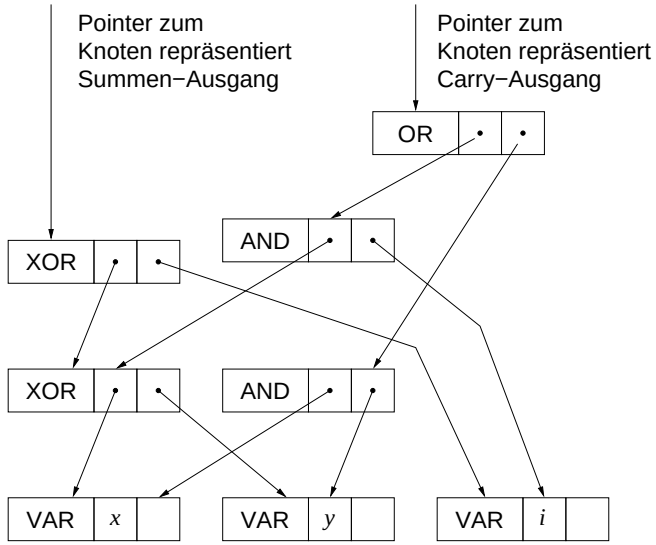
- Sharen gemeinsamer Teil-Formeln ist kompakter:

  Carry-Out eines Ripple-Adders als Baum ist exponentiell gross

  Carry-Out eines Ripple-Adders als Netzliste ist nur linear gross

- z.B. Parse-DAG (Directed Acyclic Graph) für kombinatorische Logik

Parse-DAG eines Voll-Addierers



Knoten-Darstellung in C

```
enum Tag
{
  OR, AND, XOR, NOT, VAR, CONST
};

typedef struct Node Node;
typedef union NodeData NodeData;

union NodeData
{
  Node *child[2];
  int idx;
};

struct Node
{
  enum Tag tag;
  NodeData data;
  int mark;                    /* traversal */
};
```


Ad: Knoten-Darstellung in C

- Ähnliche Darstellung wie von Symbolen in einem Compiler

- Variablen werden durch Integer-Indizes dargestellt

- Boolesche Konstanten werden durch den 0 bzw. 1 Index dargestellt

- Operations-Knoten haben Pointer auf Kinder

- Knoten können mehrfach referenziert werden

  (Speicherverwaltung: Referenz-Counting oder Garbage Collection)

- Keine Zyklen (DAG)!

Erzeugung von Konstanten

```
Node *
new_node_val (int constant_value)
{
  Node *res;

  res = (Node *) malloc (sizeof (Node));
  memset (res, 0, sizeof (Node));
  res->tag = CONST;
  res->data.idx = constant_value;

  return res;
}
```

üblicherweise nur 0 und 1 als Werte für `constant_value`

Erzeugung von Variablen

```
Node *
new_node_var (int variable_index)
{
  Node *res;

  res = (Node *) malloc (sizeof (Node));
  memset (res, 0, sizeof (Node));
  res->tag = VAR;
  res->data.idx = variable_index;

  return res;
}
```

Variable werden an Hand ihres Index unterschieden

Erzeugung von Operations-Knoten

```
Node *
new_node_op (enum Tag tag, Node * child0, Node * child1)
{
  Node *res;

  res = (Node *) malloc (sizeof (Node));
  memset (res, 0, sizeof (Node));
  res->tag = tag;
  res->data.child[0] = child0;
  res->data.child[1] = child1;

  return res;
}
```

Operations-Typ wird als erstes Argument mitübergeben

(Annahme: `child1` ist `0` für Negation)

Erzeugung DAG für Voll-Addierer

```
Node *x, *y, *i, *o, *s, *t[3];

x = new_node_var (0);
y = new_node_var (1);
i = new_node_var (2);
t[0] = new_node_op (XOR, x, y);
t[1] = new_node_op (AND, x, y);
t[2] = new_node_op (AND, t[0], i);
s = new_node_op (XOR, t[0], i);
o = new_node_op (OR, t[1], t[2]);
```

Explizites Sharen durch temporäre Pointer `t[0]`, `t[1]` und `t[2]`

Beispiel-Algorithmus: Input-Cone Variablen

```
void
input_cone_node (Node * node)
{
  if (node->mark)
    return;
  node->mark = 1;
  switch (node->tag)
    {
    case CONST:
      break;
    case VAR:
      printf ("variable %d in input cone\n", node->data.idx);
      break;
    case NOT:
      input_cone_node (node->data.child[0]);
      break;
    default:                    /* assume binary operator */
      input_cone_node (node->data.child[0]);
      input_cone_node (node->data.child[1]);
      break;
    }
}
```


Wiederherstellen des Mark-Flag

```
void
mark_node (Node * node, int new_value)
{
  if (node->mark == new_value)
    return;

  node->mark = new_value;
  switch (node->tag)
    {
    case VAR:
    case CONST:
      return;
    case NOT:
      mark_node (node->data.child[0], new_value);
      break;
    default:
      mark_node (node->data.child[0], new_value);
      mark_node (node->data.child[1], new_value);
      break;
    }
}
```


Ad: Parse-DAG

- Algorithmen basieren auf Depth-First-Search
- Vermeidung von Mehrfach-Traversierung mit `mark`-Flag
- Meist zwei Phasen: Traversierung, Wiederherstellung `mark`-Flag
- Konjunktion, Negation sind schnell (verwende `op`)
- Tautologie und Erfüllbarkeit schwierig
- Nur explizites Sharen: keine kanonische Darstellung

And-Inverter-Graphs (AIG)

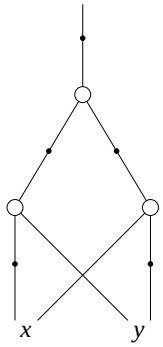
- logische Basis-Operationen: Konjunktion und Negation
- DAG-Darstellung:

  Operations-Knoten sind alles Konjunktionen

  Negation/Vorzeichen als Kanten-Attribut (daher *signed*)

  (platzsparend als LSB im Pointer)
- automatisches Sharen syntaktisch isomorpher Teilgraphen
- Vereinfachungs-Regeln mit konstantem Look-Ahead

XOR als AIG



Negationen/Vorzeichen sind Kantenattribute

(gehören nicht zu den Knoten)

AIGs in C

```
typedef struct AIG AIG;

struct AIG
{
  enum Tag tag;                 /* AND, VAR */
  void *data[2];
  int mark, level;              /* traversal */
  AIG *next;                    /* hash collision chain */
};

#define sign_aig(aig) (1 & (unsigned) aig)
#define not_aig(aig) ((AIG*)(1 ^ (unsigned) aig))
#define strip_aig(aig) ((AIG*)(~1 & (unsigned) aig))
#define false_aig ((AIG*) 0)
#define true_aig ((AIG*) 1)
```

Annahme für Korrektheit:

`sizeof(unsigned) == sizeof(void*)`

Vorteile von Signs

- Alignment moderner Prozessoren “verschwendet” sowieso mehrere LSBs

  Alignment ist typischerweise 4 oder 8 Bytes $\Rightarrow$ 2 oder 3 LSBs übrig

  `malloc` liefert *aligned blocks* (z.B. 8 Byte aligned auf Sparc)

- negierte und unnegierte Formel enspricht einem Knoten

  (potentiell Halbierung des Speicherplatzes)

- maximale Reduktion auf einen einzigen Operations-Typ (AND)

- Negation extrem effizient (Bit im Pointer umsetzen)

- erlaubt zusätzliche Vereinfachungsregeln

  direkte Erkennung von Argumenten unterschiedlichem Vorzeichens

Vereinfachungs-Regeln

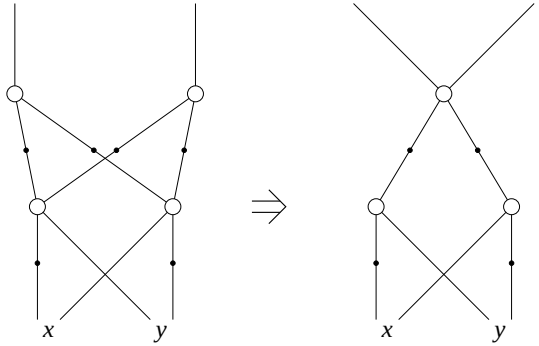
```
int
simp_aig (enum Tag tag, void *d0, void *d1, AIG ** res_ptr)
{
  if (tag == AND)
    {
      if (d0 == false_aig || d1 == false_aig || d0 == not_aig (d1))
        { *res_ptr = false_aig; return 1; }

      if (d0 == true_aig || d0 == d1)
        { *res_ptr = d1; return 1; }

      if (d1 == true_aig)
        { *res_ptr = d0; return 1; }
    }

  return 0;
}
```

Syntaktisches Sharen



Verschmelzen von Knoten mit gleichen Kindern

Ad: Syntaktisches Sharen

- wird auch *Algebraische Reduktion* genannt
- Hauptvorteil ist automatische Kompaktifizierung
- Implementierung:

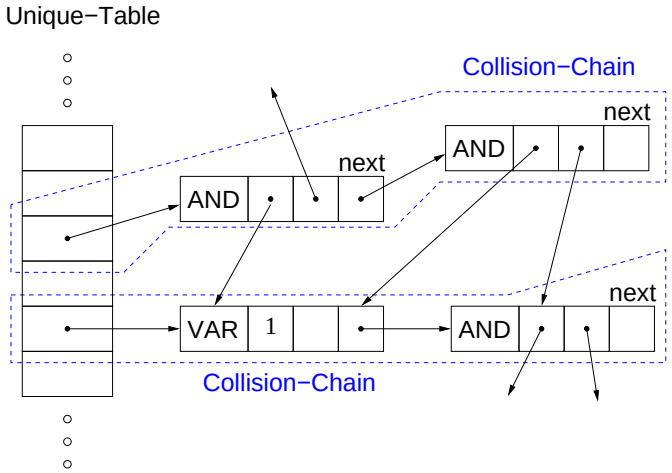
  Knoten in einer Hash-Tabelle (Unique-Table) gespeichert
- On-The-Fly Reduktion:

  Invariante: zwei Knoten haben immer unterschiedliche Kinder

  Erzeugung neuer Knoten: zunächst Suche nach äquivalentem Knoten

  Suche erfolgreich liefert äquivalenten Knoten als Resultat

Unique-Table



Suche nach Äquivalenten Knoten in C

```
#define UNIQUE_SIZE (1 << 20)
AIG *unique[UNIQUE_SIZE];

AIG **
find_aig (enum Tag tag, void *d0, void *d1)
{
  AIG *r, **p;
  unsigned h = (tag + ((int) d0) * 65537 + 13 * (int) d1);
  h = h & (UNIQUE_SIZE - 1);    /* modulo UNIQUE_SIZE */
  for (p = unique + h; (r = *p); p = &r->next)
    if (r->tag == tag && r->data[0] == d0 && r->data[1] == d1)
      break;

  return p;
}
```

Ad: Suche nach Äquivalenten Knoten in C

1. Bilde Hashwert als Kombination des Tags und der Pointer-Werte

   (bzw. der Variablen-Indizes bei Konstanten)

2. Normalisiere Hashwert auf Grösse der Unique-Table

3. Durchsuche Collision-Chain an der Hashwert-Position in der Unique-Table

4. Vergleiche Knoten mit neu zu erzeugendem Knoten

5. Falls gleich gib Pointer auf Link zu diesem Knoten zurück

6. Ansonsten gib Pointer auf letztes leeres Link-Feld in Chain zurück

Speichern von Knoten in Unique-Table

```
void
insert_aig (AIG * aig)
{
  AIG **p;
  int l[2];

  p = find_aig (aig->tag, aig->data[0], aig->data[1]);
  assert (!*p);
  aig->next = *p;
  *p = aig;

  if (aig->tag == AND)
    {
      l[0] = strip_aig (aig->data[0])->level;
      l[1] = strip_aig (aig->data[1])->level;
      aig->level = 1 + ((l[0] < l[1]) ? l[1] : l[0]);
    }
  else
    aig->level = 0;
}
```

`find_aig` gibt (neue) Position des gehashten Knotens zurück

AIG Konstruktor

```
AIG *
new_aig (enum Tag tag, void *data0, void *data1)
{
  AIG *res;
  if (tag == AND && data0 > data1)
    SWAP (data0, data1);
  if (tag == AND && (simp_aig (tag, data0, data1, &res)))
    return res;
  if ((res = *find_aig (tag, data0, data1)))
    return res;
  res = (AIG *) malloc (sizeof (AIG));
  memset (res, 0, sizeof (AIG));
  res->tag = tag;
  res->data[0] = data0;
  res->data[1] = data1;
  insert_aig (res);

  return res;
}
```


Abgeleitete Konstruktoren

```
AIG *
var_aig (int variable_index)
{
  return new_aig (VAR, (void *) variable_index, 0);
}

AIG *
and_aig (AIG * a, AIG * b)
{
  return new_aig (AND, a, b);
}

AIG *
or_aig (AIG * a, AIG * b)
{
  return not_aig (and_aig (not_aig (a), not_aig (b)));
}

AIG *
xor_aig (AIG * a, AIG * b)
{
  return or_aig (and_aig (a, not_aig (b)), and_aig (not_aig (a), b));
}
```

Zählen von AIG Knoten

```
int
count_aig (AIG * aig)
{
  if (sign_aig (aig))
    aig = not_aig (aig);
  if (aig->mark)
    return 0;
  aig->mark = 1;
  if (aig->tag == AND)
    return count_aig (aig->data[0]) + count_aig (aig->data[1]) + 1;
  else
    return 1;
}
```

Man muss explizit die Vorzeichen behandlen

aber sonst genauso DFS-orientiert wie bei DAG-Darstellung

Markieren von AIG Knoten

```
void
mark_aig (AIG * aig, int new_value)
{
  if (sign_aig (aig))
    aig = not_aig (aig);
  if (aig->mark == new_value)
    return;
  aig->mark = new_value;
  if (aig->tag == AND)
    {
      mark_aig (aig->data[0], new_value);
      mark_aig (aig->data[1], new_value);
    }
}
```

Weniger Fälle und weniger Code als bei DAG Darstellung!

Übersetzung von DAG nach AIG

```
AIG *
node2aig (Node * node)
{
  switch (node->tag)
    {
    case VAR:
      return new_aig (VAR, (void *) node->data.idx, 0);
    case CONST:
      return node->data.idx ? true_aig : false_aig;
    case AND:
      return and_aig (node2aig (node->data.child[0]),
                      node2aig (node->data.child[1]));
    case OR:
      return or_aig (node2aig (node->data.child[0]),
                     node2aig (node->data.child[1]));
    case XOR:
      return xor_aig (node2aig (node->data.child[0]),
                      node2aig (node->data.child[1]));
    default:
      assert (node->tag == NOT);
      return not_aig (node2aig (node->data.child[0]));
    }
}
```


Visualisierung

```
void
vis_aig_aux (AIG * aig, FILE * file, int max_level)
{
  assert (!sign_aig (aig));
  if (aig->mark)
    return;
  aig->mark = 1;
  switch (aig->tag)
    {
    case VAR:
      fprintf (file, "%p@%d:c'%d\n", aig, 2 * max_level, (int) aig->data[0]);
      break;
    default:
      assert (aig->tag == AND); /* TODO: constants */
      vis_aig_edge (aig, aig->data[0], file, max_level);
      vis_aig_edge (aig, aig->data[1], file, max_level);
      vis_aig_aux (strip_aig (aig->data[0]), file, max_level);
      vis_aig_aux (strip_aig (aig->data[1]), file, max_level);
      break;
    }
}
```

Simple DFS: Graph wird rekursiv in Datei geschrieben

AIG Missing Features

- robusterer C-Code notwendig

  (z.B. 64 Bit-Anpassung)

- Speicher-Management fehlt ganz

  (z.B. Reference-Counting oder GC)

- Eingabe-Format und Parser fehlen

- Vereinfachungsregeln für Enkel fehlen

  (z.B. Distributiv-Gesetz)