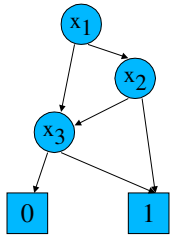
BDDs – Nachtrag



(Existentielle) Quantifizierung

- Berechnung von $(\exists x_i : f(x_1, \ldots, x_n)) = f_{x_i=0} + f_{x_i=1}$
  - f ist eine Boolesche Funktion, die von X_n abhängt
  - $\exists x_i : f$ ist eine Boolesche Funktion, die von $X_n \setminus \{x_i\}$ abhängt
  - Für Belegungen der Variablen aus $X_n \setminus \{x_i\}$ ist $(\exists x_i : f) = 1$ genau dann, wenn $f_{xi=0} = 1$ oder $f_{xi=1} = 1$.
- Genauso gilt $(\forall x_i : f(x_1, \ldots, x_n)) = f_{x_i=0} \cdot f_{x_i=1}$
- Es gibt aber eine effizientere Implementierung...


```
node* exists(node* f, variable v) {
    /* Terminalfälle */
    if (f konstant) return f;
    x := index(f);
    if (level(x) > level(v)) return f;   // f hängt nicht von v ab
    /* Überprüfen der Computed Table */
    if (computed table has entry (f, v, result)) return result;
    if (x == v)      result = f_low + f_high;
          else {     res0 = exists(f_low, v);
                     res1 = exists(f_high, v);
                     result = unique_find_or_add(x, res0, res1);
    }
    insert (f, v, result) into computed_table;
    return result;
}
```


Implementierung: CUDD-Paket

- gehört zu den besten BDD-Paketen
- Autor: Fabio Somenzi (University of Colorado at Boulder)
- Geschrieben in C, es gibt eine C++ Schnittstelle
- Installiert unter /usr/local/ira/DD/CUDD/cudd-2.3.1

CUDD: Knoten

```
struct DdNode {
    unsigned short int  index;     // 16 Bit Variablen-Index
    unsigned short int  ref;       // 16 Bit Referenzzähler
    DdNode* T;                     // Then-Kante
    DdNode* E;                     // Else-Kante
    DdNode* next;                  // pointer für Unique Table
};
```

Größe eines Knotens: 16 Byte.

CUDD: Operationen auf Knoten

- Cudd_Regular(*node*) : // (Komplement)Kante nach Zeiger
  (DdNode*)((unsigned long)*node* & ~01)
- Cudd_Not(*node*) : // Komplement der Funktion
  (DdNode*)((long)*node* ^ 01)
- Cudd_IsConstant(*node*) :
  Cudd_Regular(*node*)->index == CUDD_CONST_INDEX
- Cudd_IsComplemented(*node*) :
  (int)((long)*node* & 01)

CUDD: Manager

```
struct DdManager {
        DdNode* one;   // Terminalknoten 1
        ...
        int*  perm;        // Variablenordnung (index -> Level)
        int*  invperm;     // Inverse Variablenordnung
        DdNode** vars; // Knoten für alle Variablen
        ...
        // Computed Table
        // Unique Tables
        // Statistische Informationen
 };
```


CUDD: Unique Table (1)

```
struct DdManager {
        ...
        // Unique Tables
        int size;                  // Anzahl Variablen
        DdSubtable* subtables;     // Eine Unique Table pro Var.
        unsigned int keys;         // Knoten insgesamt
        unsigned int dead;         // nicht-referenzierte Knoten
        // Statistische Information für „Garbage Collection“
        ...
};
```

CUDD: Unique Table (2)

```
struct DdSubtable {
    DdNode**      nodelist;   // Hashtabelle
    int           shift;      // für Hashfunktion
    unsigned int  slots;      // Größe Hashtabelle
    unsigned int  keys;       // Knoten in Unique Table
    unsigned int  maxKeys;    // slots * 4
    unsigned int  dead;       // Knoten mit Referenzzähler 0
    ...                       // ... und anderes
};
```

- Größe der Hashtabelle ist Zweierpotenz
- ddHash(*f*, *g*, *shift*) : // Hashfunktion
  (((unsigned)*f* * 12582917 + (unsigned)*g*) * 4256249) >> *shift*

Cudd: UniqueFindOrAdd

```
DdNode* cuddUniqueInter(DdManager* dd, int index,
                        DdNode* T, DdNode* E) {
    int level = dd->perm[index];
    DdSubtable* subtable = &(dd->subtables[level]);
    if (Knoten mit (T,E) bereits in subtable) return result;
    // ein neuer Knoten muss angelegt werden
    if (seit letztem Reordering hat sich die Größe verdoppelt) {
        Führe dynamische Reordering durch; return 0;
    }
    if (es gibt zu viele nicht referenzierte Knoten) {
        Führe Garbage Collection durch; return 0;
    }
    lege neuen Knoten an und gebe ihn zurück;
}
```


Cudd: Garbage Collection

```
int cuddGarbageCollect(DdManager* dd) {
    // Einträge in Computed Table löschen
    int slots = dd->cacheSlots;
    for (i = 0; i < slots; i++) {
      c = &(dd->cache[i]);
      if (c->data != NULL &&
                ein Argument oder Ergebnis hat ->ref == 0)
        c->data = NULL;
    }

    ...
```


Cudd: Garbage Collection (2)

```
    ...
    // Einträge in Unique Table löschen
    for (i = 0; i < dd->size; i++) {
        if (dd->subtables[i].dead == 0) continue;
        for (every entry in dd->subtables[i]) {
            if (node->ref == 0)
                cuddDeallocNode(unique,node);
        }
    }

    dd->subtables[i].keys –= dd->subtables[i].dead;
    dd->subtables[i].dead = 0;
}
```

Beispiel: AND

```
DdNode *
Cudd_bddAnd(DdManager * dd, DdNode * f, DdNode * g)
{
  DdNode *res;

  do {
    dd->reordered = 0;
    res = cuddBddAndRecur(dd,f,g);
  } while (dd->reordered == 1);
  return(res);
} /* end of Cudd_bddAnd */
```



```
DdNode *
cuddBddAndRecur(DdManager * manager,
                DdNode * f, DdNode * g)
{
  DdNode *F, *fv, *fnv, *G, *gv, *gnv;
  DdNode *one, *r, *t, *e;
  unsigned int topf, topg, index;
  one = DD_ONE(manager);

  /* Terminal cases. */
  F = Cudd_Regular(f);
  G = Cudd_Regular(g);
  if (F == G)   { if (f == g) return(f); else return(Cudd_Not(one)); }
  if (F == one) { if (f == one) return(g); else return(f);           }
  if (G == one) { if (g == one) return(f); else return(g);           }

  /* At this point f and g are not constant. */
  ...
```



```
  ...
  /* Try to increase cache efficiency. */
  if (f > g) {
    DdNode *tmp = f;
    f = g;
    g = tmp;
    F = Cudd_Regular(f);
    G = Cudd_Regular(g);
  }

  /* Check cache. */
  if (F->ref != 1 || G->ref != 1) {
    r = cuddCacheLookup2(manager, Cudd_bddAnd, f, g);
    if (r != NULL) return(r);
  }
```



```
  ...
  /* Here we can skip the use of cuddI, because the operands are
  ** known to be non-constant.  */
  topf = manager->perm[F->index];
  topg = manager->perm[G->index];

  /* Compute cofactors. */
  if (topf <= topg) {
    index = F->index;
    fv = cuddT(F);
    fnv = cuddE(F);
    if (Cudd_IsComplement(f)) {
      fv = Cudd_Not(fv);
      fnv = Cudd_Not(fnv);
    }
  } else {
    index = G->index;
    fv = fnv = f;
  }
```

```
    ...
    if (topg <= topf) {
       gv = cuddT(G);
       gnv = cuddE(G);
       if (Cudd_IsComplement(g)) {
          gv = Cudd_Not(gv);
          gnv = Cudd_Not(gnv);
       }
    } else {  gv = gnv = g; }

    t = cuddBddAndRecur(manager, fv, gv);
    if (t == NULL) return(NULL);
    cuddRef(t);

    e = cuddBddAndRecur(manager, fnv, gnv);
    if (e == NULL) {
       Cudd_IterDerefBdd(manager, t);
       return(NULL);
    }
    cuddRef(e);
```



```
    ...
    if (t == e) {
       r = t;
    } else {
       if (Cudd_IsComplement(t)) {
          r = cuddUniqueInter(manager,(int)index,
                                   Cudd_Not(t),Cudd_Not(e));
          if (r == NULL) {
             Cudd_IterDerefBdd(manager, t);
             Cudd_IterDerefBdd(manager, e);
             return(NULL);
          }
          r = Cudd_Not(r);
       } else {
          r = cuddUniqueInter(manager,(int)index,t,e);
          if (r == NULL) {
             Cudd_IterDerefBdd(manager, t);
             Cudd_IterDerefBdd(manager, e);
             return(NULL);
    }  }  }
```



```
    ...

    /* update computed table */
    cuddDeref(e);
    cuddDeref(t);
    if (F->ref != 1 || G->ref != 1)
       cuddCacheInsert2(manager, Cudd_bddAnd, f, g, r);
    return(r);

} /* end of cuddBddAndRecur */
```