Ausblicke zu Rekursiven Datenstrukturen

Martin Hofmann

Ludwig-Maximilians-Universität München

TdI'09, 03.07.09

Inhalt

- Ein Mini-Computeralgebra-System
  - Syntaxbäume
  - Polynome als Listen
  - Rekursive Auswertung von Syntaxbäumen
  - Rekursive Implementierung von Algorithmen für Polynome
  - Einsatz von Parsergenerator und Hauptprogramm
- Ausblick auf Ray-Tracing
  - Technik zur Erzeugung von Bildern aus Szenebeschreibungen
  - Zusätzliche Verwendung rekursiver Datenstrukturen
  - Ray-Tracing selbst ist bereits rekursiv.
- Schlussbetrachtung

```
mhofmann@garda:~/work/lehrer09$ maple
    |\^/|     Maple 9.5 (IBM INTEL LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, a division of Waterloo Maple
 \  MAPLE  /  All rights reserved. Maple is a trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> expand((X^2+2*X+1)*(X^2+3*X-1));
                              4      3      2
                             X  + 5 X  + 6 X  + X - 1

> quo(X^2+2*X+1,X+1,X);
                                      X + 1


>

                                        0


> gcd(X^2+2*X+1,X^2-1);
                                      X + 1
```

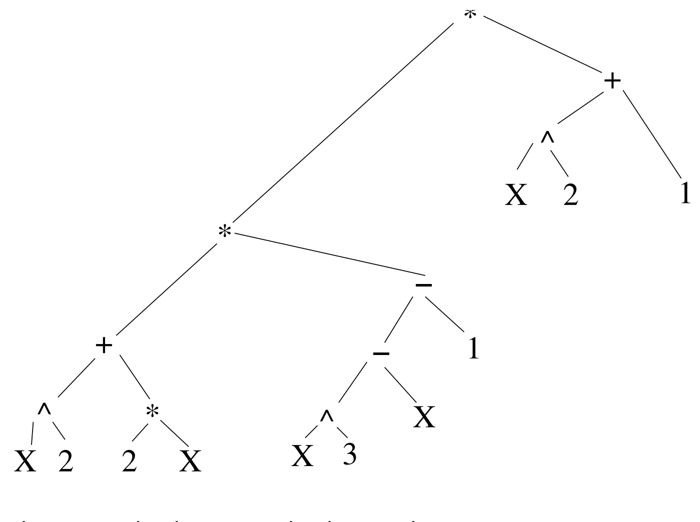
Computeralgebra

```
> sum(i^2+2*i+1,i=0..n);
                                   3           2
                            (n + 1)     (n + 1)
                            -------- + -------- + n/6 + 1/6
                               3           2


> sum(binomial(n,i),i=0..n);
                                             n
                                            2

> sum(binomial(n+i,i)*binomial(n,i),i=0..n);

                                   LegendreP(n, 3)

> sum(binomial(n+i,i),i=0..n);

                               binomial(2 n + 1, n + 1)
```

In diesem Vortrag

```
mhofmann@garda:~/work/lehrer09/alg$ ./main
MHs Polynomalgebrasystem vom 02.07.09

> expand((X^2+2*X)*(X^3-X-1)*(X^2+1));

X^7+2*X^6-X^4-3*X^3-3*X^2-2*X

> poldiv(X^7+2*X^6-X^4-3*X^3-3*X^2-2*X+1,X^3-X-1);

Quotient = X^4+2*X^3+X^2+2*X
Rest = 1

> gcd(X^7+2*X^6-X^4-3*X^3-3*X^2-2*X,(X^2+1)*(X^3-4*X^2+5));

X^2+1
```

Syntaxbäume




```
(X^2+2*X)*(X^3-X-1)*(X^2+1)
```

Syntaxbäume in OCAML

```
type operator = Plus | Minus | Times | Hat
type expression =
       Var |
       Int of int |
       UMinus of expression |
       Binop of operator * expression * expression
```

Unser Ausdruck entspricht:

```
Binop(Times,
 Binop(Times,
  Binop(Plus,Binop(Hat,Var,Int 2),Binop (Times,Int 2,Var)),
  Binop(Minus,Binop(Minus,Binop(Hat,Var,Int 3),Var),Int 1)),
 Binop(Plus,Binop(Hat,Var,Int 2),Int 1))
```

Im Rechner natürlich wieder als verzeigerter Baum repräsentiert,

Schrittweise Konstruktion

```
# let fac1 =
   Binop(Plus,Binop(Hat,Var,Int(2)),Binop(Times,Int(2),Var));;
val fac1 : expression =
  Binop(Plus,Binop(Hat,Var,Int 2),Binop(Times,Int 2,Var))
# let fac2 =
   Binop(Minus,Binop(Minus,Binop(Hat,Var,Int 3),Var),Int 1);;
val fac2 : expression = ...
# let fac3 =
  Binop(Plus,Binop(Hat,Var,Int(2)),Int(1));;
val fac3 : expression = ...
# let tree = Binop(Times,Binop(Times,fac1,fac2),fac3);;
val tree = ...
```

Polynome

Polynome mit ganzzahligen Koeffizienten repräsentieren wir als Listen ihrer Koeffizienten beginnend mit der niedrigsten Potenz.
Z.B.: $X^2 + 3X + 5$ wird zu `[5;3;1]`
Z.B.: $-7X^5 + 4X^2 - 8X - 3$ wird zu `[-3;-8;4;0;0;-7]`
Nachteile:

- nicht ganz eindeutig wg. führende Nullen.
- Typ `int list` deutet nicht eindeutig auf "Polynom" hin.

Alternativen:

- Koeffizientenlisten andersrum
- Listen von Monomen (Paar Koeff. Exponent),
- Grad, Leitkoeffizient etc. als Attribute mitführen.

Addition von Polynomen

Durch Rekursion über die Listenrepräsentation:

```
let rec poly_add p1 p2 = match (p1,p2) with
      (p,[]) -> p
|     ([],p) -> p
|     (a1::rest1,a2::rest2) -> a1+a2 :: poly_add rest1 rest2
```

Z.B.: `poly_add [4;5;6] [10;11;12;13]` =
`14::poly_add [5;6] [11;12;13]` =
`14::16::poly_add [6] [12;13]` = `14::16::18::poly_add [] [13]`
= `14::16::18::[13]` = `[14;16;18;13]`
Nicht vergessen: `::` ist “cons”, `[]` ist die leere Liste.

Normalisieren von Polynomen

Nullen am Ende eines Polynoms (als Liste) sollen entfernt werden.

```
let remove_leading_zeros p =
  let rec strip l = match l with
        [] -> [] | h::t -> if h=0 then strip t else l in
  List.rev (strip (List.rev p))
```

Die lokale Hilfsfunktion `strip` entfernt Nullen von vorne.
Die Bibliotheksfunktion `List.rev` dreht eine Liste um.
Den gewünschten Effekt erhalten wir durch Hintereinanderschalten von `List.rev`, dann `strip`, dann wieder `List.rev`

Einfache Multiplikationen

Mit `X` multiplizieren:

```
let times_X p = 0::p
```

Mit X^n multiplizieren:

```
let rec times_Xn p n =
   if n=0 then p else times_Xn (times_X p) (n-1)
```

Z.B.: `times_Xn [1;5;9] 2` = `times_Xn (times_X [1;5;9]) 1` = `times_Xn [0;1;5;9] 1` = `times_Xn (times_X [0;1;5;9]) 0` = `times_Xn [0;0;1;5;9] 0` = [0;0;1;5;9]

Mit Skalar multiplizieren:

```
let poly_scalar a p = List.map (fun b -> a * b) p
```

Die Bibliotheksfunktion `List.map` wendet eine Funktion auf alle Einträge einer Liste an: `List.map (fun x -> x*x) [1;2;3;4]` = `[1;4;9;16]`

Multiplikation von Polynomen

```
let rec poly_mult p1 p2 = match p1 with
       [] -> []
|     a1::rest1 -> poly_add (poly_scalar a1 p2)
                   (times_X (poly_mult rest1 p2))
```

- Ist p1 die leere Liste, also Null, dann ist das Ergebnis auch Null
- Ist p1 von der Form a1::rest1, also $p_1 = a_1 + X \cdot rest_1$, so ist $p_1 \cdot p_2 = a_1 \cdot p_1 + X \cdot (rest_1 \cdot p_2)$.

Beispiel: poly_mult [1;2;1] [1;3;1] = [1; 5; 8; 5; 1]

Auswerten von Syntaxbäumen

```
let rec expand expr = match expr with
       Var -> [0;1]
|      Int i -> [i]
|      UMinus expr -> poly_scalar (-1) (expand expr)
|      Binop(op,expr1,expr2) ->
          let p1 = expand expr1 in
          let p2 = expand expr2 in
           (match op with
              Plus -> poly_add p1 p2
           |  Times -> poly_mult p1 p2
           |  Minus -> poly_add p1 (poly_scalar (-1) p2)
           |  Hat -> if not(posint p2) then
                raise (Error "Nur natuerliche Exponenten")
                else ntimes (poly_mult p1) (toint p2) [1])
```

Hilfsfunktionen und Erläuterungen zu Hat

Testen, ob ein Polynom positiver Integer ist:

```
let posint p = match (remove_leading_zeros p) with
    [] -> true
|   [n] -> n >= 0
| _ -> false
```

Eine Funktion n-Mal auf einen Wert anwenden, also $f^{(n)}(x)$ berechnen:

```
let rec  ntimes f n x = if n = 0 then x
       else ntimes f (n-1) (f x)
```

Ein Polynom $p \in \mathbb{Z}$ nach `int` konvertieren.

```
let toint p = match  (remove_leading_zeros p) with
   [] -> 0
| [n] -> n
| _ -> raise (Error "toint")
```

Exceptions

Mit

```
exception Error of string
```

haben wir eine Exception `Error` deklariert. Man kann sie mit `raise` "werfen" und mit `try with` fangen.
Man kann ihr einen `string`-Wert mitgeben.

Eingabe

Lexikalische Analyse: Eingabestring in Tokenstream konvertieren.
Stream = Funktion, die bei jedem Aufruf das jeweils nächste Token zurückgibt. Token = Lexikalische Einheit (Schlüsselwörter, Klammern, Komma, Zahlen, . . .)

Syntaxanalyse: Tokenstream in Syntaxbaum konvertieren.
Wir reichern unsere Syntax noch um Kommandos an. Es gibt zunächst nur ein Kommando; `expand`, welches als Argument *einen* Ausdruck hat.

```
type command = Expand of expression
```

Ocamlyacc Grammatik 1. Teil

```
%{open Syntax%}
%token <int> INT
%token PLUS MINUS TIMES DIV
%token LPAREN RPAREN SEMI COMMA
%token EXPAND
%token VAR HAT
%left PLUS MINUS        /* lowest precedence */
%left TIMES DIV         /* medium precedence */
%nonassoc UMINUS        /* one to highest precedence */
%right HAT              /* highest precedence */
%start main             /* the entry point */

%type <Syntax.command> main
%%
```

Hier sind die Tokens definiert, die Präzedenz und Bindungskraft von Operatoren, das Startsymbol und sein Typ (command)

Ocamlyacc Grammatik 2. Teil

```
main:
    EXPAND LPAREN expr RPAREN SEMI            { Expand($3) }
;
expr:
    INT                      { Int $1 }
  | LPAREN expr RPAREN       { $2 }
  | expr PLUS expr           { Binop(Plus,$1,$3) }
  | expr MINUS expr          { Binop(Minus,$1,$3) }
  | expr TIMES expr          { Binop(Times,$1,$3) }
  | expr HAT expr            { Binop(Hat,$1,$3)  }
  | VAR                      { Var }
  | MINUS expr %prec UMINUS { UMinus $2 }
;
```

Produktionen einer kontextfreien Grammatik.

In den geschw. Klammern steht jeweils die “semantische Aktion”, also was als Wert zurückgegeben wird.

Mit $1, $2, etc. kann man auf die Werte der Komponenten zurückgreifen.

Lexer-Definition

```
{
open Parser exception Eof exception Bad_Token
}
rule token = parse
    [' ' '\n' '\t']     { token lexbuf }      (* skip blanks *)
  | ['0'-'9']+ as lxm { INT(int_of_string lxm) }
  | '+'             { PLUS }     | '-'            { MINUS }
  | '*'             { TIMES }    | '^'            { HAT }
  | '('             { LPAREN }   | ')'            { RPAREN }
  | 'X'             { VAR    }
  | ';'             { SEMI }     | ','            { COMMA }
  | "expand"        { EXPAND }   | "poldiv"       { POLDIV }
  | "gcd"           { GCD }      | eof            { raise Eof }
  | _               { raise Bad_Token }
```

Die Werkzeuge Ocamlyacc und Ocamllex erzeugen aus diesen Spezifikationen automatisch Ocaml-Code, der übersetzt und dann verwendet werden kann:

Read-Eval-Print-Schleife

```
let _ = print_string "MHs Polynomalgebrasystem vom 02.07.09\n'
    try (
      let lexbuf = Lexing.from_channel stdin in
      while true do print_string "> ";flush stdout;
         try let com = Parser.main Lexer.token lexbuf in
           run_command com;
   print_newline(); flush stdout
         with Error err -> print_string ("Fehler: "^err^"\n")
      done
    ) with Lexer.Eof -> exit 0
```

- `while true do` *e* `done` wertet *e* immer wieder aus.
  Dasselbe wie
  `let rec loop f = f();loop f in loop (fun _ -> e)`
- `^` konkateniert Strings
- Das Ganze mehr oder weniger aus dem Ocaml-Referenzmanual kopiert.

Drucken von Polynomen

```
let string_of_poly p =
  let p = remove_leading_zeros p in
  let deg = degree p in
  if deg = -1 then "0" else (
  let q = List.rev p in
  let (_,_,str) = List.fold_left (fun (needsign,exponent,str) coeff
     (true,exponent-1,(str ^ if coeff = 0 then "" else
     ((if needsign && coeff > 0 then "+" else "") ^
     (if coeff = 1 && exponent > 0 then "" else
      if coeff = -1 && exponent > 0 then "-" else string_of_int coe
     (if coeff != 1 && coeff != -1 &&
                      exponent > 0 then "*" else "") ^
     (if exponent > 0 then "X" else "") ^
     (if exponent > 1 then "^" ^ string_of_int
                   exponent else ""))))) (false,deg,"") q
   in str)
```

Im Prinzip trivial, aber erstaunlich zeitaufwendig zu programmieren.

Abarbeiten von Kommandos

```
let run_command com = match com with
   Expand exp -> print_string(string_of_poly (expand exp))
```

Wir führen jetzt noch weitere Kommandos ein:

```
type command = Expand of expression
       | Poldiv of expression*expression
       | Gcd of expression * expression
```

nebst entsprechenden Grammatik- und Lexer-Regeln.

Polynomdivision

```
let rec poly_div p1 p2 =
 let deg1 =degree p1 in
 let deg2 =degree p2 in
 let lc1 = lcoeff p1 in
 let lc2 = lcoeff p2 in
 if isnull p2 then raise (Error "Division durch Null") else
 if deg1 < deg2 then (poly_of_int 0,p1) else
 if lc1 mod lc2 != 0 then
      raise (Error "Rationale Zahl nicht unterstuetzt") else
 if deg1 < deg2 then (poly_of_int 0,p1) else
 let quotmon = times_Xn (poly_of_int (lc1/lc2)) (deg1-deg2) in
 let diff = poly_add p1
                 (poly_scalar (-1) (poly_mult quotmon p2)) in
 let (quot,rem) = poly_div diff p2 in
       (poly_add quotmon quot,rem)
```

Hilfsfunktionen & Idee

```
let degree p =
     let p = remove_leading_zeros p in
     List.length p - 1
let lcoeff p =
     let p = remove_leading_zeros p in
     match List.rev p with [] -> 0 | h::t -> h
let isnull p =
     let p = remove_leading_zeros p in p=[]
let poly_of_int a = if a = 0 then [] else [a]
```

Falls $\deg(p_1) < \deg(p_2)$ so ist Quotient 0 und Rest p_2.

Falls $\deg(p_1) \geq \deg(p_2)$ so bilde $\mathit{diff} = p_1 - X^{\deg(p_1)-\deg(p_2)} p_2$ und dividiere *diff* rekursiv durch p_2. Zum Quotienten wird $X^{\deg(p_1)-\deg(p_2)}$ addiert; Rest ist derselbe.

Euklidischer Algorithmus für ggT

```
let rec int_gcd x y =
 if abs x < abs y then int_gcd y x else
 if y = 0 then x else
 int_gcd y (x mod y)
```

$ggT(24, 32) = ggT(32, 24) = ggT(24, 8) = ggT(8, 0) = 8.$

Euklidischer Algorithmus für Polynome (Erste Version)

```
let rec int_gcd x y =
 if abs x < abs y then int_gcd y x else
 if y = 0 then x else
 int_gcd y (x mod y)

let rec poly_gcd p1 p2 =
     let lc2 = lcoeff p2 in
     let deg1 =degree p1 in
     let deg2 =degree p2 in
     if deg1 < deg2 then poly_gcd p2 p1 else
     if isnull p2 then p1 else
     let (quot,rem) = poly_div p1 p2 in
     poly_gcd p2 rem
```

Problem mit der ersten Version

```
let rec poly_gcd p1 p2 =
     let lc2 = lcoeff p2 in
     let deg1 =degree p1 in
     let deg2 =degree p2 in
     if deg1 < deg2 then poly_gcd p2 p1 else
     if isnull p2 then p1 else
     let (quot,rem) = poly_div p1 p2 in
     poly_gcd p2 rem
```

Problem: “Rationale Zahl nicht unterstuetzt”

Beispiel: $ggT(X^2 - X + 1, X + 1) = ggT(X + 1, -2X + 1) = \texttt{Exception}$

Abhilfe

Multipliziere p_1 mit $\mathit{lcoeff}(p_2)^{\deg(p_1)-\deg(p_2)+1}$.
Dividiere aber vorher p_2 durch ggT seiner Koeffizienten, z.B.:
$-2X^2 - 4X + 6 \rightsquigarrow -X^2 - 2X + 3$
Dieser ggT heißt *Inhalt* (engl.: content) von p_2.
ggT von Polynomen ist ohnehin nur bis auf ganzz. Vielfache bestimmt.

Endgültige Lösung

```
let rec poly_gcd p1 p2 =
     let p2 = remove_content p2 in
     let lc2 = lcoeff p2 in
     let deg1 =degree p1 in
     let deg2 =degree p2 in
     if deg1 < deg2 then poly_gcd p2 p1 else
     if isnull p2 then p1 else
     let p1 = poly_scalar
          (int_pot lc2 (deg1-deg2+1)) p1 in
     let (quot,rem) = poly_div p1 p2 in
     poly_gcd p2 rem
```

Zusammenfassung Polynomalgebra

- Syntax von Ausdrücken intern als Bäume, dadurch rekursive Auswertung sehr einfach.
- Umwandlung von konkreter Syntax in Bäume mit Parsergenerator (Ocamlyacc & Ocamllex).
- Polynome als Werte von Ausdrücken. Repräsentiert als Listen.
- Rekursion über Listen und Hilfsfunktionen wie `List.map` erlauben kompakte Definition der Funktionen auf Polynomen.
- Read-Eval-Print-Loop zur wiederholten Abarbeitung von Befehlen.
- GgT-Berechnung durch Beschränkung auf ganze Zahlen etwas aufwendig. Alternative: floats oder rationale Arithmetik.

Ausblick auf Raytracing

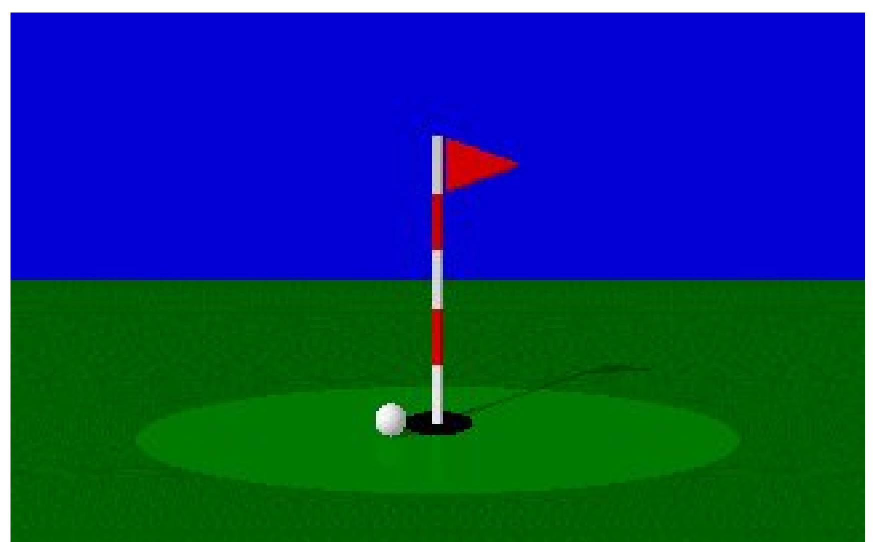
ICFP Programming Contest 2000:

Erstelle in 72Std ein Programm, welches Befehle einer Szenebeschreibungssprache einliest und eine entsprechende Bilddatei erzeugt.

Sprache und Lösungsstrategie war in einem sehr gut lesbaren und hochinteressanten 20S. Dokument dargestellt.

- `http://de.wikipedia.org/wiki/ICFP`
- `http://www.cs.cornell.edu/icfp/`
- `http://www.cs.cornell.edu/icfp/task.pdf`

Eines der Testbilder



Auszug aus der Szenebeschreibung

```
{ /v /u /face
  white 1.0 0.0 1.0
} sphere
0.10 uscale
/ball

{ /v /u /face
  black 1.0 0.0   1.0
} cylinder
0.25 uscale
/hole
[...]
```

Forts.

```
field
hole
0.0 -0.25 2.0 translate
difference
sky union
post 0.0 0.0 2.0 translate
union
ball -0.3 0.1 1.75 translate
union
flag
0.45 1.8 2.0 translate
union
0.0 -1.0 0.0 translate
/scene
```

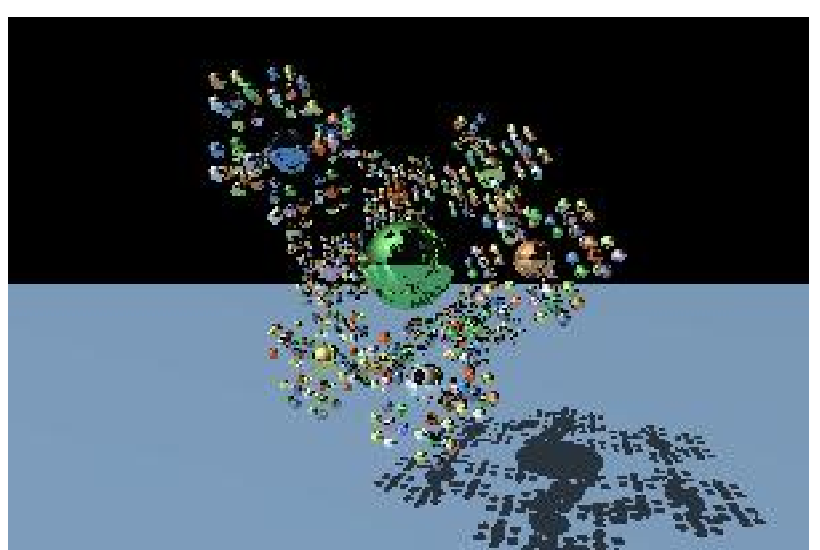
Forts.

```
1.0 -1.0 1.0 point %% position
0.4 0.4 0.4 point   %% intensity
light
/sun

0.6 0.6 0.6 point %% Ambient
[ sun ] %% Lights
scene
2 %% Depth
90.0 %% fov
320 %% width (pixels)
200 %% height (pixels)
"golf.ppm" %% filename
render
```

Funktionsweise

- Für jeden Pixel einen gedachten Strahl in der entsprechenden Richtung in die Szene schicken.
- Trifft der Strahl ein Objekt, in Reflektionsrichtung rekursiv weiterverfolgen.
- Schattenstrahlen in Richtung der Lichtquellen schicken. Falls verdeckt, kein Beitrag der Lichtquelle.
- Farbe und Intensität des ursprünglichen Pixels aus den Farben und Intensitäten der (rekursiven) Hilfsstrahlen berechnen.
- Rekursionstiefe beschränkt auf z.B. 2-4.
- Objekte in Bounding-Boxen packen und diese in Bäume zu organisieren hilft bei der schnellen Ermittlung von Verdeckungen.

Ein weiteres Testbild



Zusammenfassung

- Computeralgebra und Raytracing als Anwendungen rekursiver Datenstrukturen
- Syntaxbäume, Verdeckungshierarchien, Objektlisten, Koeffizientenlisten
- Rekursive Kontrollstruktur bei Auswertung, Raytracing und bei Polynomalgorithmen.
- Verwendung einer funktionalen Sprache, hier OCAML, erlaubt relativ schnelles Prototyping, bei gleichzeitig hoher Leistung. Ist für Problemstellungen der Form

  Eingabe $\rightarrow$ Parser $\rightarrow$ Syntaxbaum $\rightarrow$ Verarbeiten $\rightarrow$ Ausgabe

  hervorragend geeignet. Dokumentation & Download `http://caml.inria.fr/ocaml/`.
- Wir verwenden Ocaml zur Implementierung von
  - Ressourcenanalysen (Eingabe=Programm, Ausgabe=Ressourceninformation),
  - Entscheidungsverfahren (Eingabe=logische Formeln, Ausgabe=Beweis/Modell)