Implementierung von Peirce-Algebren in SML

Seminar „Relationale Grammatik“
CIS, SS 2009

Hans Leiß

Universität München
Centrum für Informations- und Sprachverarbeitung

2. November 2009

SML - Standard ML

- SML ist eine funktionale Programmiersprache
- SML hat ein ausgefeiltes Modulsystem, mit dem man Algebren nahe an der mathematischen Syntax programmieren lassen
- Wir verwenden das freie SML/NJ
- SML/NJ hat einen Parsergenerator ML-Yacc zur Entwicklung eigener (deterministischer) Programmiersprachen
- ML-GLR ist eine am CIS entstandene Variante von ML-Yacc, die lexikalische, syntaktische und semantische Ambiguitäten erlaubt.
- Semantische Annotationen der Syntaxregeln sind Funktionen.

Quellen

1. SML of New Jersey, ML-Lex, ML-Yacc: Binär+Quelldateien
   - `http://www.smlnj.org` (siehe Downloads)
2. User's Guide to ML-Lex and ML-Yacc:
   - `http://rogerprice.org`

   oder in den Quelldateien von SML:
   - `smlnj/ml-lex/lexgen.tex`
   - `smlnj/ml-yacc/doc/mlyacc.tex`
3. ML-GLR: Quelldateien und Installationsanweisung
   - Subversion-Archiv (Rechte fehlen noch)
     `https:svn.cip.ifi.lmu.de/~leiss/svn/hl-yacc`
4. User's Guide to ML-GLR:
   - Homepage des Seminars (.pdf)

Installieren von SML/NJ

Unter `www.smlnj.org`, Downloads, die neueste Version (110.71) holen:

- Unix/Linux: Unix The only file you need to download manually is config.tgz. Unpack, configure by editing config/targets, and install by running config/install.sh. The installer automatically downloads any additional files it might need. For more information, see INSTALL.
- Microsoft Windows: Download smlnj.msi and follow the instructions in WININSTALL.

Das Wichtigste zu SML

- SML Aufrufen

  ```
  > sml
  Standard ML of New Jersey v110.67 [built: ... 2007]
  -
  ```

- SML Beenden

  ```
  - ^D
  ```

- Datei laden

  ```
  - use "<Dateiname>";
  ```

- Compilation Manager (CM)

  ```
  - CM.make "<Dateiname>.cm";
  ```

  Das führt den „Makefile“ `<Dateiname>.cm` aus. (s.u.)

Peirce-Algebra

Eine Peirce-Algebra ist eine (offiziell) eine 2-sortige Struktur

$$\mathcal{P} = \langle \mathcal{B}, \mathcal{R}, :, {}^{c} \rangle,$$

wobei

- $\mathcal{B} = (B, +, 0, \cdot, 1, {}^{-})$ eine Boole'sche Algebra,
- $\mathcal{R} = (R, +, 0, \cdot, 1, {}^{-}, \,;\,, {}^{\breve{}}, I)$ eine Relationenalgebra,
- „$\cdot : \cdot$“ eine Abbildung („Urbild“) aus $R \times B \longrightarrow B$ und
- „$\cdot^{c}$“ eine Abbildung („Rechtszylinder“) aus $B \longrightarrow R$ ist,

die gewissen Gleichungsaxiomen genügen, z.B.

- $a \cdot (b + c) = (a \cdot c) + (b \cdot c)$ für $a, b, c \in B$,
- $(r \,;\, s)^{\breve{}} = (s^{\breve{}} \,;\, r^{\breve{}})$ für $r, s \in R$,
- $(r : 1)^{c} = (r \,;\, 1)$ für alle $r \in R$.

Die Signatur einer Peirce-Algebra

Die *Signatur* einer mathematischen Struktur ist die Angabe der Funktionen und Relationen mit ihren Stelligkeiten.

Die Signatur der Boole'schen Algebra ist

$$\langle +^{(2)}, 0^{(0)}, .^{(2)}, 1^{(0)}, -^{(1)} \rangle$$

Bei einer Struktur mit mehreren Objektsorten oder -Typen muß man auch die Typen angeben.

Die Signatur einer Peirce-Algebra ist also:

$$\langle \sigma, \rho;\ +^{\sigma\times\sigma\to\sigma}, 0^{\sigma}, \ldots, +^{\rho\times\rho\to\rho}, 0^{\rho}, \ldots, ;^{\rho\times\rho\to\rho}, :^{\rho\times\sigma\to\sigma}, {}^{c}\,^{\sigma\to\rho} \rangle$$

wobei σ der Typ der „Mengen“ und ρ der Typ der „Relationen“ ist.

Die Peirce-Algebra $\mathcal{P}(U)$

Die Standardinterpretation der Peirce-Algebra bilden die Teilmengen und zweistelligen Relationen über einem Universum U:

$$\mathcal{P}(U) = \langle \mathcal{B}(U), \mathcal{R}(U), :, {}^{c} \rangle$$

wobei

- $\mathcal{B}(U) = (2^U, \cup, \emptyset, \cap, U, \bar{\cdot})$ die Boole'sche Algebra aller Teilmengen von U ist, mit $\overline{A} := U \setminus A$,
- $\mathcal{R}(U) = (2^{U \times U}, \cup, \emptyset, \cap, U \times U, \bar{\cdot}, ;, \breve{}, I)$ die Relationenalgebra aller 2-stelligen Relationen über U ist, $I = \{(u, u) \mid u \in U\}$,
- $R : A := \{u \in U \mid \exists a \in A \; R(u, a)\}$ das „Urbild der Menge A unter der Relation R“, und
- $A^c := A \times U$ der „Rechtszylinder der Menge A“ ist.

Ein paar zusätzliche Grundoperationen

In einer „Relationalen Grammatik“ sind als Bedeutungen von Ausdrücken nur Objekte einer Peirce-Algebra $\mathcal{P}(U)$ erlaubt.

Als Wahrheitswerte von Aussagen nimmt Böttner U für „wahr“ und $\emptyset$ für „falsch“; er braucht zusätzlich drei Testfunktionen:

$$U(A) := \begin{cases} U & \text{falls } A = U \\ \emptyset & \textit{sonst} \end{cases}$$

$$E(A) := \begin{cases} U & \text{falls } A \neq \emptyset \\ \emptyset & \textit{sonst} \end{cases}$$

$$N(A) := \begin{cases} U & \text{falls } A = \emptyset \\ \emptyset & \textit{sonst} \end{cases}$$

Stattdessen erweitern wir die Peirce-Algebra um die Boole'sche Algebra $\mathbb{B}$ der Wahrheitswerte und Abbildungen $U, E, N : \sigma \longrightarrow \mathbb{B}$:

$$\mathcal{P}(U) = \langle \mathcal{B}(U), \mathcal{R}(U), \mathbb{B}, :, ^c, U, E, N \rangle$$

Signaturen in SML

In SML kann man Module mit der in der Mathematik üblichen Terminologie definieren.

```
signature PEIRCE =
  sig
      type set    (* Mengenbereich B      *)
      type rel    (* Relationenbereich R *)
      type tv     (* Wahrheitswerte IB    *)

      val <Konstante> : <Typ>
      ...
      val <Operation> : <Argumenttyp> -> <Ergebnistyp>
      ...
      val <Operation> : <Argumenttyp> * <Argumenttyp>
                          -> <Ergebnistyp>
  end
```

Strukturen in SML

Wenn man dazu passende Strukturen definieren will, muss man die Definition der Typen und Werte angeben:

```
structure Peirce : PEIRCE =
  struct
      type set = ...   (* Mengenimplementierung *)
      type rel = ...   (* Relationenimplementierung *)
      type tv  = ...   (* Wahrheitswertimplementierung *)

      val <Konstante> = <Angabe des Elements>
      ...
      val <Operation> = <Angabe der Funktion>
      ...
  end
```

Die Signatur PEIRCE

⟨*PA-GLR/peirce.sig*⟩≡

```
signature PEIRCE =
   sig
       type set

       val ConS : string -> set   (* Konstante *)
       val UnitS : set
       val ZeroS : set
       val SumS  : set * set -> set
       val ProdS : set * set -> set
       val CompS : set -> set
```

⟨*PA-GLR/peirce.sig*⟩+≡

```
type rel

val ConR : string -> rel (* Konstante *)
val UnitR : rel
val ZeroR : rel
val SumR  : rel * rel -> rel
val ProdR : rel * rel -> rel
val CompR : rel -> rel

val I     : rel
val Conv  : rel -> rel
val Prod  : rel * rel -> rel
val Minor : rel -> rel   (* spaeter *)
val Tc    : rel -> rel   (* spaeter *)

val PreIm : rel * set -> set
val Cylin : set -> rel
```

⟨*PA-GLR/peirce.sig*⟩+≡

```
        type tv

        val And : tv * tv -> tv
        val Or  : tv * tv -> tv
        val Neg : tv -> tv

        val E : set -> tv
        val U : set -> tv
        val N : set -> tv

        val ER : rel -> tv
        val UR : rel -> tv
        val NR : rel -> tv
    end
```

Alternative Signatur PEIRCE

die Mengen- und Relationenalgebra als Komponenten vorsieht:

```
signature BOOLEAN_ALGEBRA =
   sig type t
       val Zero : t
       val Unit : t
       val Sum  : t * t -> t
       val Prod : t * t -> t
       val Comp : t -> t
   end
signature RELATION_ALGEBRA = sig type t  ... end
signature PEIRCE =
   sig structure Set : BOOLEAN_ALGEBRA
       structure Rel : RELATION_ALGEBRA
       val PreIm : Rel.t * Set.t -> Set.t
       val Cylin : Set.t -> Rel.t
   end
```

Konstruktion der Peirce-Algebra $\mathcal{P}(U)$

Wir wollen keine *bestimmte* Algebra `P:PEIRCE` implementieren, sondern eine *Konstruktion* $\mathcal{P}$, die zu gegebenem Universum U die volle Peirce'sche Algebra $\mathcal{P}(U)$ liefert.

Ein Universum ist eine Grundmenge U mit benannten Mengen $A \subseteq$ und Relationen $R \subseteq U$ - den Bedeutungen von Nomina, Verben, Adjektiven in einer Anwendung:

⟨*PA-GLR/peirce.sml*⟩≡

```
signature UNIVERSE =
   sig
       eqtype element  (* Typ mit Gleichheitstest = *)
       val elements : element list      (* endlich! *)
       val ConS : string -> element list
       val ConR : string -> (element * element) list
       exception NoValue (* wo ConR,ConS undefiniert *)
   end
```

Beispieluniversum

⟨*PA-GLR/pg1.sml*⟩≡

```
structure Personen : UNIVERSE =
  struct
    datatype element = a | b | c | d | e | f | g | h | m
    val elements = [a,b,c,d,e,f,g,h,m]
    exception NoValue                          (* Synt.Kat. *)
    fun ConS "Anna"      = [a]                      (* EN *)
      | ConS "Emil"      = [e]                      (* CN *)
      | ConS "Frau"      = [a,g,m]
      | ConS "arbeiten"  = [a,e]                    (* IV *)
      | ConS "gesund"    = [a]                      (* CA *)
      | ConS _           = raise NoValue
    fun ConR "lieben"    = [(b,d),(a,a),(e,a)]      (* TV *)
      | ConR "Nachbar"   = [(d,e),(e,a)]            (* RN *)
      | ConR "älter"     = [(d,a),(d,e),(a,e)]      (* IA *)
      | ConR _           = raise NoValue)
  end
```

Mengenimplementierung Set:SET

⟨*PA-GLR/set.sml*⟩≡

```
signature SET = sig
    type 'a set (* 'a El.Typ, ''a mit Gleichheit *)
    val empty : 'a set
    val difference : ''a set -> ''a set -> ''a set
    val equal : ''a set -> ''a set -> bool
    val intersection : ''a set -> ''a set -> ''a set
    val list : 'a set -> 'a list
    val map : ('a -> ''b) -> 'a set -> ''b set
    val member : ''a -> ''a set -> bool
    val mk : ''a list -> ''a set
    val select : 'a set -> ('a -> bool) -> 'a set
    val size : 'a set -> int
    val subset : ''a set -> ''a set -> bool
    val union : ''a set -> ''a set -> ''a set
    val Union : ''a set list -> ''a set
  end
```

Mengenimplementierung Set : SET

Endliche Mengen seien Listen von Objekten, die keine Duplikate enthalten und deren Anordnung nicht benutzt werden kann:

⟨*PA-GLR/set.sml*⟩+≡

```
structure Set : SET = struct
    datatype 'a set = Set of 'a list
    val empty = Set []
    fun difference (Set S) (Set T) =
        let fun diff set [] = set
              | diff set (e::E) = diff (drop e set) E
        in Set (diff (Set S) T) end
    and drop e (Set S) = ...
    ...
    fun list (Set S) = S
    fun map f (Set S) = mk (List.map f S)
    fun size (Set S) = List.length S
    ...
end
```

Die Signatur PEIRCEfull

In einer vollen Peirce-Algebra $\mathcal{P}(U)$ sind $\mathcal{B}(U)$ und $\mathcal{R}(U)$ aus demselben U aufgebaut.

Wir erweitern die Signatur PEIRCE um den Typ der Individuen U:

⟨*PA-GLR/peirce.sml*⟩+≡

```
signature PEIRCEfull =
   sig
       include PEIRCE
       type element
   end
```

Wir wollten aber keine feste Peirce'sche Algebra `P:PEIRCEfull` implementieren, sondern die $\mathcal{P}(U)$ für beliebige Beispielmodelle U.

Abhängige Strukturen: Funktoren in SML

Die volle Peirce-Algebra $\mathcal{P}(U)$ über U *hängt* vom Universum U *ab*.

In SML drückt man die Abhängigkeit einer Struktur B von einer anderen Struktur A durch einen Funktor $F : SIG_A \rightarrow SIG_B$ aus:

```
functor F(X:SIGA) =
  struct ... X.f ... end : SIGB

structure B = F(A)
```

Die Konstruktion $\mathcal{P}$, die aus U ein $\mathcal{P}(U)$ bildet, soll als SML-Funktor `Peirce` implementiert werden, der aus einer Struktur $U : UNIVERSE$ eine Struktur $P(U) : PEIRCEfull$ macht:

```
functor Peirce(U : UNIVERSE) : PEIRCEfull =
  struct ... end
```

Der Funktor Peirce : UNIVERSE → PEIRCEfull

⟨*PA-GLR/peirce.sml*⟩+≡

```
functor Peirce(U : UNIVERSE) : PEIRCEfull =
   struct
     type element = U.element
     fun ConS s = Set.mk (U.ConS s)
     fun ConR s = Set.mk (U.ConR s)

     type set = element Set.set      (* BA(U) *)

     val UnitS = Set.mk U.elements
     val ZeroS = Set.empty
     fun SumS(A,B) = Set.union A B
     fun ProdS(A,B) = Set.intersection A B
     fun CompS(A) = Set.difference UnitS A
```

⟨*PA-GLR/peirce.sml*⟩+≡

```
type rel = (element * element) Set.set  (* RA(U) *)

fun Id(A) = Set.map (fn x => (x,x)) A
val I = Id(UnitS)
fun Cart(A,B) =
    let fun f x =
            List.map (fn y => (x,y)) (Set.list B)
    in  Set.mk (List.foldr (op @) nil
                           (List.map f (Set.list A)))
    end
val UnitR:rel = Cart(UnitS,UnitS)
val ZeroR = Set.empty
fun SumR(R,S) = Set.union R S
fun ProdR(R,S) = Set.intersection R S
fun CompR(R) = Set.difference UnitR R
fun Conv(R) = Set.map (fn (x,y) => (y,x)) R
```

⟨*PA-GLR/peirce.sml*⟩+≡

```
fun Prod(R,S) =                    (* Relationenprodukt *)
    let fun f(a,b) =
        Set.map (fn (x,c) => (a,c))
                (Set.select S (fn (x,c) => x = b))
    in  Set.Union (List.map f (Set.list R))
    end
fun Tc(R) =                        (* Transitive Huelle *)
    let val T = SumR(Prod(R,R),R)
    in
        if Set.size(T) = Set.size(R) then R
        else Tc(T)  (* terminiert bei U endlich *)
    end

fun PreIm(R,B) =                              (* R : B *)
    Set.map #1 (Set.select R
                        (fn (a,b) => Set.member b B))
fun Cylin(A) = Cart(A,UnitS)                  (* A^c *)
```

⟨*PA-GLE/peirce.sml*⟩≡

```
    datatype tv = datatype bool

    fun And(A,B) = A andalso B    (* von SML's bool *)
    fun Or(A,B)  = A orelse B
    fun Neg(A)   = not A

    fun E(A) = (Set.size A > 0)
    fun N(A) = (Set.size A = 0)
    fun U(A) = ((Set.size A) = (Set.size UnitS))

    fun ER(A) = (Set.size A > 0)
    fun UR(A) = ((Set.size A) = (Set.size UnitR))
    fun NR(A) = (Set.size A = 0)
  end
```

Anwendung auf das Beispieluniversum

```
> cd PA-GLR
> sml
Standard ML of New Jersey v110.67 [built: Nov 26 2007]
- use "set.sml";
- use "peirce.sig"; use "peirce.sml";
- use "pg1.sml";
[opening pg1.sml]
structure PersonenU :
  sig
    datatype element = a | b | c | d | e | f | g | h | m
    val elements : element list
    exception NoValue
    val ConS : string -> element list
    val ConR : string -> (element * element) list
  end
- structure Personen = Peirce(PersonenU);
structure Personen : PEIRCEfull
```

```
- PersonenU.ConR "Nachbar";
val it = [(d,e),(e,a)] : (PersonenU.element *
                          PersonenU.element) list
- Personen.ConR "Nachbar";
val it = Set [(d,e),(e,a)] : Personen.rel
- Personen.Tc it;
val it = Set [(d,e),(e,a),(d,a)] : Personen.rel

- open Personen;
opening Personen
  type set = Personen.element Set.set
  val ConS : string -> set
  val UnitS : set
  val ZeroS : set
  val SumS : set * set -> set
  val ProdS : set * set -> set
  val CompS : set -> set
```

```
type rel = (Personen.element * Personen.element) Set.set
val ConR : string -> rel
val UnitR : rel
val ZeroR : rel
val SumR : rel * rel -> rel
val ProdR : rel * rel -> rel
val CompR : rel -> rel
val I : rel
val Conv : rel -> rel
val Prod : rel * rel -> rel
val Minor : rel -> rel
val Tc : rel -> rel

val PreIm : rel * set -> set
val Cylin : set -> rel
```

```
  datatype bool = false | true        (* = tv ! *)
  val And : tv * tv -> tv
  val Or : tv * tv -> tv
  val Neg : tv -> tv
  val E : set -> tv
  val U : set -> tv
  val N : set -> tv
  val ER : rel -> tv
  val UR : rel -> tv
  val NR : rel -> tv

  type element = PersonenU.element
```

Nach dem Öffnen der Struktur entfällt das Präfix `Personen.`:

```
- ConR "Nachbar";
val it = Set [(d,e),(e,a)] : rel
- true : Personen.tv;
val it = true : bool
```

Fazit

- In SML kann man abstrakte Strukturen gut implementieren:

```
structure A = struct
  type t = ...                      (* Objektbereich          *)
  val c : t = ...                   (* Objekt                 *)
  fun f (x:t, y:t) : t = ...        (* Funktion f:t*t -> t    *)
  structure B = ...                 (* Unterstruktur          *)
end : SIGNATURE
```

- Funktoren konstruieren Strukturen aus anderen Strukturen:
  `functor Peirce(U:UNIVERSE)` konstruiert $\mathcal{P}(U)$ aus U

Demnächst:

- Ziel 1: Parsergenerator, der aus einer Grammatik einen Parser erzeugt, der Peirce-Terme zurückgibt
- Ziel 2: Funktor `Semantik(A:PEIRCEfull)`, der Peirce-Terme in der Algebra A auswertet

Compilation Manager: `CM.make`

SML hat ein eingebautes „make“: im Unterschied zu dem von Unix sagt man im „Makefile“ nicht, welche Schritte ausgeführt werden sollen, sondern welche Strukturen gebaut werden sollen.

⟨*PA-GLR/personen.cm*⟩≡

```
Group
 signature PEIRCEfull
 structure Personen
is
 $/basis.cm   (* fuer structure List *)
 set.sml
 peirce.sig
 peirce.sml
 pg1.sml
```