Heterogene Algebren (Wiederholung)

Eine heterogene Algebra ist ein Tupel $(A_1, \ldots, A_n, \Omega)$, wobei $A_1, \ldots, A_n$ Trägermengen sind und Ω ein System von Operationen ist, deren Bedeutungen durch eine Menge von Gesetzen Δ festgelegt werden.

Jede Trägermenge A_i entspricht genau einer Sorte s_i aus der Menge der Sorten $S = \{s_1, \ldots, s_n\}$.

Die Signatur entspricht der Schnittstelle eines ADT. Sie besteht aus S, Ω, dem Typ und den Indizes der Operationen.

Stelligkeit m und Index einer Operation $\omega \in \Omega$ können in der Form $\omega : s_{i_1} \times \cdots \times s_{i_m} \rightarrow s_j$ dargestellt werden, wobei $j, i_1, \ldots, i_m \in \{1, \ldots, n\}$ und $s_j, s_{i_1}, \ldots, s_{i_m} \in S$.

“Order-sorted Algebras”

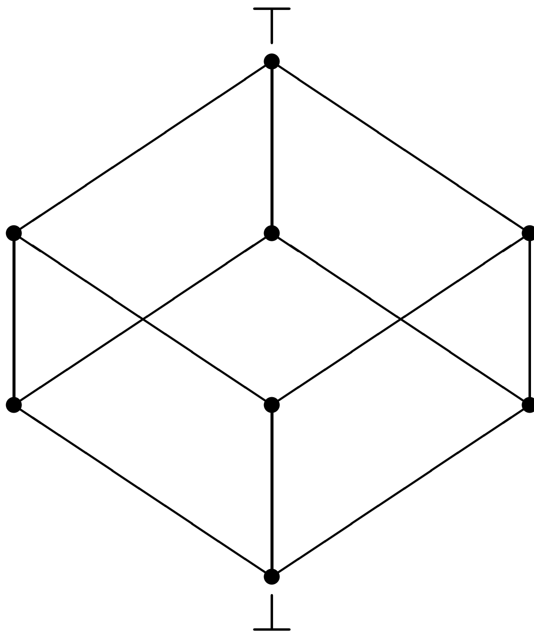
Heterogene Algebren sollen strukturiert werden, um einen Wildwuchs an Sorten und Operationen zu vermeiden.

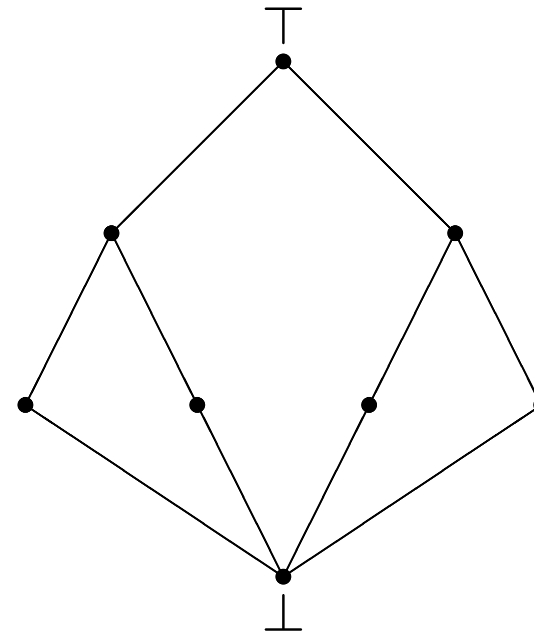
Über der Menge S wird ein *Verband* $\langle S, \leq, \sqcup, \sqcap, \top, \bot \rangle$ eingeführt:

- $\leq$ ist eine reflexive, transitive und antisymmetrische Relation auf S (Halbordnung von S), so dass $\bot \leq s$ und $s \leq \top$ für alle $s \in S$.
- Für jedes $s_1 \in S$ und $s_2 \in S$ sind das Supremum $s_1 \sqcup s_2 = s_3 \in S$ (s_3 ist das kleinste Element von S mit $s_1 \leq s_3$ und $s_2 \leq s_3$) und Infimum $s_1 \sqcap s_2 = s_4 \in S$ (s_4 ist das größte Element von S mit $s_4 \leq s_1$ und $s_4 \leq s_2$) eindeutig definiert. Dabei gilt $s \sqcup s' = s' \Leftrightarrow s \leq s' \Leftrightarrow s \sqcap s' = s$.

Hasse-Diagramme

Jeder Verband lässt sich als Hasse-Diagramm darstellen:





Untertypen und Wertemengen

Zwischen den Trägermengen $A_1, \ldots, A_n$ und den entsprechenden Sorten $s_1, \ldots, s_n$ gelten folgende Beziehungen:

$$\begin{aligned} s_i \leq s_j &\Leftrightarrow A_i \subseteq A_j \\ s_i = s_j \sqcap s_k &\Leftrightarrow A_i \subseteq A_j \cap A_k \end{aligned}$$

$\top$ beschreibt die Menge aller Werte, und $\bot$ beschreibt die Menge der Werte, die in jeder Trägermenge enthalten sind; das heisst, für $s_i = \top$ und $s_j = \bot$ gilt

$$A_i = \bigcup_{k=1,\ldots,n} A_k \qquad \text{und} \qquad A_j = \bigcap_{k=1,\ldots,n} A_k.$$

Meist enthält A_j nur einen Wert, der oft ebenfalls durch $\bot$ dargestellt wird und undefinierte Ergebnisse repräsentiert.

Polymorphe Operationen

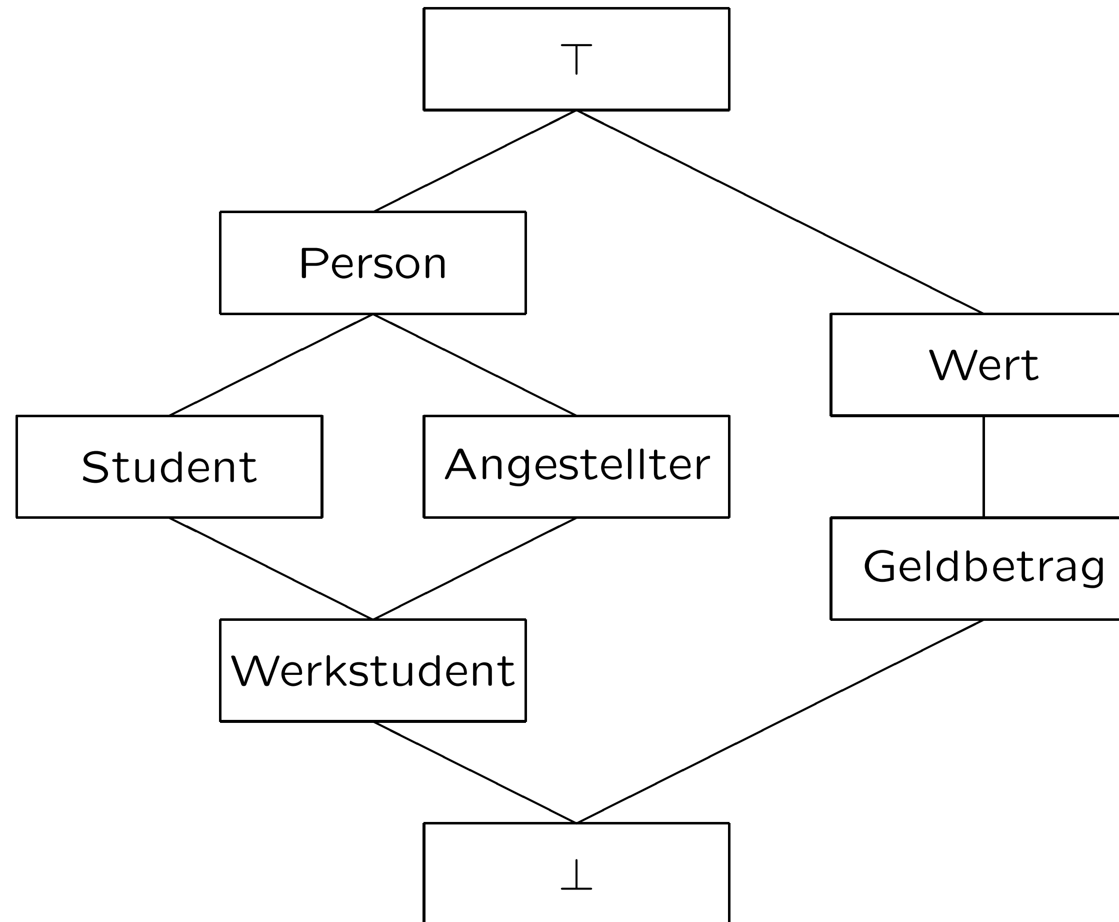
Dadurch, dass Trägermengen ineinander enthalten sind, sind die Operationen automatisch polymorph. Wenn $\omega \in \Omega$ mit $\omega : s_{i_1} \times \cdots \times s_{i_m} \to s_j$ durch Δ vollständig definiert ist, dann ist ω durch Δ gleichzeitig auch als

$$\omega : s_{i'_1} \times \cdots \times s_{i'_m} \to s_{j'} \quad \text{für alle} \quad s_j \leq s_{j'} \quad \text{und} \quad s_{i'_k} \leq s_{i_k}$$

vollständig definiert.

Intuitive Erklärung: Jede Operation bildet eine Menge von Eingabewerten in eine Menge von Ergebniswerten ab. Natürlich ist die Operation auch auf jeder Teilmenge der Eingabewerte definiert, und das Ergebnis liegt stets auch in jeder Obermenge der tatsächlich möglichen Ergebniswerte. Ein Obertyp beschreibt eine Obermenge von Werten, ein Untertyp eine Untermenge.

Beispiel polymorpher Operationen (1)



Beispiel polymorpher Operationen (2)

Damit wäre dieses Programmstück erlaubt (wobei angenommen wird, dass jede Sorte einem Typ entspricht:

```
procedure Beitrag(V:in Person;
                  S:in out Wert;
                  B:out Geldbetrag);
VERSICHERT: Student := ...;
VERSICHERUNGS_LEISTUNGEN: Wert := ...;
BEITRAG: Wert;
Beitrag(VERSICHERT, VERSICHERUNGS_LEISTUNGEN, BEITRAG);
```

Typinferenz

Allgemeinste Typen werden aus untypisierten λ-Ausdrücken ermittelt. Diesen Berechnungsvorgang nennt man Typinferenz.

Der Programmierer verzichtet auf Typangaben. Der Compiler errechnet die Typen und prüft auf Typkonsistenz.

Beispiel in Standard-ML:

Eingabe:	`fun pluszwei x = x + 2 ;`
Ergebnis:	`val pluszwei = fn : int -> int`
Eingabe:	`fun ident x = x ;`
Ergebnis:	`val ident = fn : 'a -> 'a`

Beispiel für explizite Typangabe:

```
fun max (x:int, y) = if x > y then x else y;
```

Einfacher Typinferenz-Algorithmus (1)

$$
\begin{aligned}
\mathsf{PT}(c) &= \emptyset \triangleright c{:}t \quad (t \text{ enthält keine Typparameter}) \\
\mathsf{PT}(x) &= \{x{:}v\} \triangleright x{:}v \quad (v \text{ neu}) \\
\mathsf{PT}(e\,e') &= \text{let } \Gamma \triangleright f{:}t = \mathsf{PT}(e); \\
&\qquad\quad \Gamma' \triangleright f'{:}t' = \mathsf{PT}(e'); \\
&\qquad\quad \theta = \text{unify}(\{s = s' \mid x{:}s \in \Gamma;\ x{:}s' \in \Gamma'\} \\
&\qquad\qquad\qquad\qquad \cup \{t = t' \to v\}) \quad (v \text{ neu}) \\
&\quad\ \text{in } \ \theta\Gamma \cup \theta\Gamma' \triangleright \theta(f\,f'){:}\theta v \\
\mathsf{PT}(\lambda x.e) &= \text{let } \Gamma \triangleright f{:}t = \mathsf{PT}(e); \\
&\quad\ \text{in } \ \text{if } x{:}s \in \Gamma \text{ für ein beliebiges } s \\
&\qquad\quad \text{then } \Gamma \setminus \{x{:}s\} \triangleright (\lambda x{:}s.f){:}s \to t \\
&\qquad\quad \text{else } \Gamma \triangleright (\lambda x{:}v.f){:}v \to t \quad (v \text{ neu})
\end{aligned}
$$

Einfacher Typinferenz-Algorithmus (2)

$$
\begin{aligned}
\text{unify}(\emptyset) &= \emptyset \\
\text{unify}(E \cup \{b = b'\}) &= \text{if } b \neq b' \text{ then fail} \\
&\ \text{else unify}(E) \\
\text{unify}(E \cup \{v = t\}) &= \text{if } v = t \text{ then unify}(E) \\
&\ \text{else if } v \text{ occurs in } t \text{ then fail} \\
&\ \text{else unify}([t/v]E) \circ [t/v] \\
\text{unify}(E \cup \{s{\rightarrow}s' = t{\rightarrow}t'\}) &= \text{unify}(E \cup \{s = t,\ s' = t'\})
\end{aligned}
$$

Beschränkungen bei Typinferenz

Das Typinferenzproblem ist im allgemeinen unentscheidbar.

Eine eingeschränktere Variante ist entscheidbar: Direkt rekursive Aufrufe innerhalb des Rumpfes der Funktionsdefinition dürfen nur monomorph verwendet werden.

Die Komplexität des Algorithmus' zur Berechnung des allgemeinsten Typs kann auch ohne polymorph rekursive Funktionsaufrufe exponentiell sein.

Es gibt aber die Tendenz, sogar unbeschränkte Typinferenz in der Praxis einzusetzen, da die Unentscheidbarkeit bzw. hohe Komplexität nur in seltenen Fällen auftritt.

Typen in ML

Vordefinierte Typen: `bool`, `int`, `real`, `string`

Listen:
```
[1, 2, 3] : int list        ([1,2,3] = 1::[2,3])
["a", "b"] : string list
nil : 'a list               (nil = [])
```

Tupel:
```
(true, "fact", 7) : bool * string * int
(true, nil) : bool * ('a list)
```

Verbunde:
```
{name="Hans", id=9} : {name:string, id:int}
```

Funktionen:
```
identitaetsfunktion : 'a -> 'a
```

Typdef.:
```
type intpair = int * int;
type 'a pair = 'a * 'a;
type boolpair = bool pair;
```

Weiters: Datentypen, ADT, Module und Schnittstellen

Datentypen in ML

```
datatype color = red | green | blue;
datatype publications = nopubs |
                        journal of int |
                        conference of int;
datatype 't stack = empty | push of 't * 't stack;
datatype 't list = nil | :: of 't * 't list;
```

Verwendung:

```
val v = push(7,push(2,empty));
```

Abstrakte Datentypen in ML

```
abstype 'a lifo = Stack of 'a list
with exception error;
     val create = Stack nil;
     fun push(x, Stack xs) = Stack(x::xs);
     fun pop(Stack nil) = raise error;
     |   pop(Stack(x::xs)) = Stack xs;
     fun top(Stack nil) = raise error;
     |   top(Stack(x::xs)) = x;
end;
```

Modul in ML

```
structure S = struct
    exception error;
    type 't Stack = 't list;
    val create = Stack nil;
    fun push(x, Stack xs) = Stack(x::xs);
    fun pop(Stack nil) = raise error;
    |   pop(Stack(x::xs)) = Stack xs;
    fun top(Stack nil) = raise error;
    |   top(Stack(x:xs)) = x;
end;
```

Verwendungsmöglichkeiten:

```
val v = S.push(2, S.create);
```

```
open S;
val v = push(2, create);
```

Schnittstelle in ML

```
signature st = sig
    type q;
    val push: 't * q -> q;
    val pop: q -> q;
end;

structure S1:st = S;
```

Schnittstelle in ML

```
signature stringStack = sig
    exception error;
    type string Stack;
    val create: string Stack;
    val push: string * string Stack -> string Stack;
    val pop: string Stack -> string Stack;
    val top: string Stack -> string;
end;

structure S2:stringStack = S;
```

Subtyping in funktionalen Sprachen

Untertypbeziehungen durch Regelsystem festgelegt.

Ersetzbarkeitsprinzip: Ein Typ s ist ein Untertyp eines Typs t wenn eine Instanz von s überall verwendet werden kann wo eine Instanz von t erwartet wird.

Einfache Untertypbeziehungen:

$$\Gamma \vdash t \leq t$$

$$\Gamma \vdash t \leq \top$$

$$\Gamma \vdash \bot \leq t$$

$$\frac{\Gamma \vdash s \leq t' \quad \Gamma \vdash t' \leq t}{\Gamma \vdash s \leq t}$$

Subtyping für Funktionstypen

$$\frac{\Gamma \vdash s' \leq s \quad \Gamma \vdash t \leq t'}{\Gamma \vdash s \to t \leq s' \to t'}$$

Eingangsparametertypen sind kontravariant.

Ergebnisparametertypen sind kovariant.

Beispiel: Es gilt offensichtlich $\{4, \ldots, 6\} \leq \{3, \ldots, 7\}$.

Eine Funktion vom Typ $\{3, \ldots, 7\} \to \{4, \ldots, 6\}$ kann überall verwendet werden wo eine Funktion vom Typ $\{4, \ldots, 6\} \to \{3, \ldots, 7\}$ erwartet wird:

Argumente werden auf jeden Fall akzeptiert.

Ergebnis liegt auf jeden Fall im erwarteten Bereich.

Subtyping für Verbunde und Varianten

$$\frac{(\forall\, i \le m)\, \Gamma \vdash s_i \le t_i}{\Gamma \vdash \{l_1{:}s_1, \ldots, l_n{:}s_n\} \le \{l_1{:}t_1, \ldots, l_m{:}t_m\}} \quad (m \le n)$$

Beispiel: $\{\texttt{name}{:}\texttt{string}, \texttt{alter}{:}\{18, \ldots, 65\}, \texttt{mnr}{:}\texttt{int}\} \le$
$\{\texttt{name}{:}\texttt{string}, \texttt{alter}{:}\{0, \ldots, 120\}\}$

$$\frac{(\forall\, i \le m)\, \Gamma \vdash s_i \le t_i}{\Gamma \vdash [l_1{:}s_1, \ldots, l_m{:}s_m] \le [l_1{:}t_1, \ldots, l_n{:}t_n]} \quad (m \le n)$$

Beispiel: $[\texttt{Di}{:}\texttt{Tag}, \texttt{Mi}{:}\texttt{Tag}] \le [\texttt{Mo}{:}\texttt{Tag}, \texttt{Di}{:}\texttt{Tag}, \texttt{Mi}{:}\texttt{Tag}, \texttt{Do}{:}\texttt{Tag}]$

Rekursive Typen

Rekursive Typen sind in der Praxis häufig.
Ihre endliche Darstellung ist jedoch problematisch.

Beispiel: $\texttt{baum} = \{\texttt{i}{:}\texttt{int}, \texttt{l}{:}\texttt{baum}, \texttt{r}{:}\texttt{baum}\}$

Darstellung mittels Fixpunktoperator: $\mu v.\{\texttt{i}{:}\texttt{int}, \texttt{l}{:}v, \texttt{r}{:}v\}$

α-Konversionsregel (Umbenennung):

$$\mu v.t = \mu u.[u/v]t \quad (u \notin \mathit{free}(\mu v.t))$$

Faltungsregel:

$$\mu v.t = [\mu v.t/v]t$$

Beispiel für rekursive Typen

Die folgenden Typausdrücke sind äquivalent:

- $\mu v.\{\mathtt{i}{:}\mathtt{int}, \mathtt{l}{:}v, \mathtt{r}{:}v\}$
- $\{\mathtt{i}{:}\mathtt{int}, \mathtt{l}{:}\mu u.\{\mathtt{i}{:}\mathtt{int}, \mathtt{l}{:}u, \mathtt{r}{:}u\},$
  $\mathtt{r}{:}\mu v.\{\mathtt{i}{:}\mathtt{int}, \mathtt{l}{:}v, \mathtt{r}{:}\mu w.v\}\}$
- $\{\mathtt{i}{:}\mathtt{int}, \mathtt{l}{:}\mu u.\{\mathtt{i}{:}\mathtt{int}, \mathtt{l}{:}u, \mathtt{r}{:}u\},$
  $\mathtt{r}{:}\{\mathtt{i}{:}\mathtt{int}, \mathtt{l}{:}\mu v.\{\mathtt{i}{:}\mathtt{int}, \mathtt{l}{:}v, \mathtt{r}{:}v\},$
  $\mathtt{r}{:}\mu v.\{\mathtt{i}{:}\mathtt{int}, \mathtt{l}{:}v, \mathtt{r}{:}v\}\}\}$

Subtyping für rekursive Typen

Subtyping-Annahmen auf freien Typparametern:

$$\Gamma \cup \{u \leq v\} \vdash u \leq v$$

Subtyping-Regel:

$$\frac{\Gamma \cup \{u \leq v\} \vdash s \leq t}{\Gamma \vdash \mu u.s \leq \mu v.t} \quad (u \notin \mathit{free}(t); v \notin \mathit{free}(s); u, v \notin \mathit{free}(\Gamma))$$

Beispiel: $\mu u.\{\mathtt{i}{:}\mathtt{int}, \mathtt{f}{:}\mathtt{int}{\rightarrow}u, \mathtt{next}{:}u\} \leq \mu v.\{\mathtt{i}{:}\mathtt{int}, \mathtt{f}{:}\mathtt{int}{\rightarrow}v\}$

Gegenbeispiel: $\mu u.\{\mathtt{i}{:}\mathtt{int}, \mathtt{f}{:}u{\rightarrow}u, \mathtt{next}{:}u\} \not\leq \mu v.\{\mathtt{i}{:}\mathtt{int}, \mathtt{f}{:}v{\rightarrow}v\}$

Das PER-Modell

Eine partielle Äquivalenzrelation (PER) ist eine symmetrische und transitive Relation.

Jeder Typ A entspricht einer PER, und die Menge der durch A beschriebenen Werte ist die Menge der Äquivalenzklassen von A, das ist die Menge $\{\{w\}_A \mid w\,A\,w\}$, wobei $\{w\}_A = \{v \mid v\,A\,w\}$. B ist ein Untertyp von A ($B \leq A$) wenn $B \subseteq A$.

Motivation: zwei Möglichkeiten für Untertypbeziehungen:
- Instanzen des Untertyps sind Teilmenge der des Obertyps,
- der Untertyp eines Verbundtyps enthält zusätzliche Felder.

Im PER-Modell ist die Menge der Instanzen eines Untertyps nicht notwendigerweise eine Untermenge der seiner Obertypen.