Typsysteme Zusammenfassung WS24

> *“viel spaß :))” ~ klausi*

Disclaimer
1. Begriffsbestimmungen und Überblick
    1.1 Was sind Typen?
    1.2 Kategorisierung
        1.2.1 Paradigmen und Sprachklassen
        1.2.2 Einteilung von Typsystemen
        1.2.3 Polymorphe Typsysteme
Einfache theoretische Modelle
    2.1 Der Lambda-Kalkül
        2.1.1 Der untypisierte Lambda-Kalkül
        2.1.2 Der typisierte Lambda-Kalkül
        2.1.3 Strukturierte Typen
    2.2 Logik
        2.2.3 Typisierte logische Programme
    2.3 Algebren
        2.3.1 Abstrakte Datentypen
3 Typen in imperativen Sprachen
    3.1 Datentypen in Ada
        3.1.1 Skalare Typen
        3.1.2 Zusammengesetzte Typen
        3.1.3 Unterbereichstypen, abgeleitete Typen und Zeiger
    3.2 Prozeduren und Prozesse
        3.2.1 Funktionen und Prozeduren
        3.2.2 Inkarnationen und Prozesse
    3.3 Generische Pakete
4 Modelle polymorpher Typsysteme und Typen in funktionalen Sprachen
    4.1 Order-sorted Algebras
        4.1.1 Verbände über Sortenmengen
        4.1.2 Polymorphe Operationen
    4.2 Typinferenz und das Typsystem von ML
        4.2.1 Typinferenz
        4.2.3 Ein Typinferenz-Algorithmus
    4.3 Funktionale Ansätze für Subtyping
        4.3.1 Einfache Untertypbeziehungen
        4.3.2 Rekursive Typen
    4.4 Das PER-Modell
5. Typen in objektorientierten Sprachen
    5.1 Untertypen in Beispielen
        5.1.1 Ada 95
    5.2 Allgemeine Konzepte
        5.2.2 Untertypen und Vererbung
        5.2.4 Untertypen und Generizität
    5.3 Logik und Subtyping
    5.4 Typen für aktive Objekte
        5.4.1 Prozesstypen

5.4.2 Vererbungsanomalie

Disclaimer

Die Zusammenfassung beruht auf dem Typsysteme Scriptum WS24. Es ist bei weitem nicht Vollständig aber die wichtigsten Aspekte sollten in dieser Zusammenfassung zu finden sein. Alle Bilder und Code Beispiele sind ebenfalls dem Scriptum entnommen.

1. Begriffsbestimmungen und Überblick

1.1 Was sind Typen?

Es gibt zahlreiche Definitionen für Typen, folgende Definitionen spiegeln verschiedene Interessen und Sichtweisen wider:

1. *Typen sind Werkzeuge zur Klassifikation von Werten aufgrund ihrer Eigenschaften, ihres Verhaltens und ihrer Verwendungsmöglichkeiten.*
2. *Typen sind Schutzschilder gegen unbeabsichtigte bzw. falsche Interpretation roher Daten.*
3. *Typen sind Werkzeuge des Softwareentwicklers zur Unterstützung der Programmerstellung, Weiterentwicklung und Wartung*
4. *Typen schränken Ausdrücke syntaktisch ein, sodass Kompatibilität zwischen Operatoren und Operanden zugesichert wird.*
5. *Typen bestimmen Speicherauslegungen für Werte.*
6. *Typen legen Verhaltens-Invarianten fest, die alle Instanzen der Typen erfüllen müssen.*
7. *Ein Typsystem ist eine Menge von Regeln, die jedem Ausdruck einen eindeutigen allgemeinsten Type zuordnet. Dieser Typ beschreibt die Menge aller Umgebungen, in denen der Ausdruck vorkommen kann und eine Bedeutung hat.*

Da sich diese Definitionen teilweise widersprechen, daher stellt jedes Typsystem einen Kompromiss dar. Im folgenden wird hauptsächlich die Definition

> *Ein Typ beschreibt eine Menge von Instanzen*

verwendet.

1.2 Kategorisierung

1.2.1 Paradigmen und Sprachklassen

$$\text{Sprache} \begin{cases} \text{imperativ} \begin{cases} \text{prozedural} \\ \text{objektorientiert} \end{cases} \\ \text{deklarativ} \begin{cases} \text{funktional} \\ \text{logisch} \\ \text{algebraisch} \end{cases} \end{cases}$$

Imperative Sprachen:

Bestehen aus *Anweisungen → können sequenziell, parallel bzw. in beliebiger Reihenfolge ausgeführt werden.*

Destruktive Zuweisung → Variable bekommt neuen Wert unabhängig vom vorherigem Wert.

Prozedurale Sprachen:

Wichtigster Abstraktionsmechanismus ist die Routine (=Prozedur). Programm wird auf sich gegenseitig aufrufende Routinen zerlegt.

"Saubere" prozedurale Programme → *strukturierte Programmierung*

Objektorientierte Programmierung:

Weiterentwicklung von strukturierter prozeduraler Programmierung.

Objekt steht im Mittelpunkt → hat *Identität, Zustand* und *Verhalten.*

passives Objekt → Verhalten wird durch Operationen die den Objektzustand verändern definiert.

aktives Objekt → Verständigen sich über (a)synchrones *Message-Passing.* Verhalten wird über Endlosschleife festgelegt.

Deklarative Sprachen:

Beschreiben ein statisches Modell durch Beziehungen zwischen Ausdrücken dieses Modells. Grundlegende Sprachelemente sind Symbole (e.g. Variablensymbole, Funktionssymbole, Prädikate, etc.)

Funktionale Sprachen:

Beruhen auf dem Lambda-Kalkül → Alles ist eine Funktion.

Logische Sprachen:

Beruhen auf Klausel-Logik ($\subseteq$ Prädikatenlogik).
Menge aller wahren Aussagen ind einem Modell wird mittels Fakten und Regeln beschrieben.
Anfrage → Ergebnis besagt ob und unter welchen Bedingungen Aussage wahr ist.

Algebraische Sprachen:

Beruhen auf freien Algebren.
Werden fast ausschließlich als Spezifikationssprache verwendet.

1.2.2 Einteilung von Typsystemen

Kriterium	**Ausprägungen**
Definierbarkeit	vordefiniert, definierbar
Art der Definition	extensional, intensional

Typfestlegung	statisch, stark, schwach, ...
Mächtigkeit	flexibel, sicher, einfach, ...
Typzwang	konservativ, optimistisch
Durchlässigkeit	dicht, löchrig
Typäquivalenz	struktur-, namensgleich
Abstraktionsgrad	abstrakt, konkret
Ausdrücklichkeit	implizit, explizit
Überschneidungen	monomorph, polymorph

Art der Definition:

extensional → zählt Instanzen eines Typs auf. (e.g. $\{2, 3, 5, 7, 11, 13\}$)

intensional → Angabe seiner Eigenschaften relativ zu anderen Typen.
(e.g.
$\{i \in \{1, ..., 16\} | p(i)\}$)

Programmiersprachen sind:

typisiert → es gibt Typen

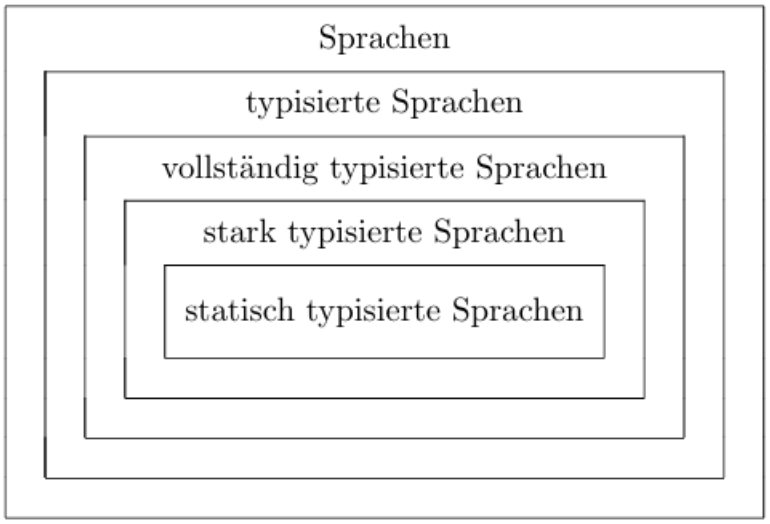
statisch typisiert → Typ jedes Ausdrucks kann statisch ermittelt werden. (zur Compilezeit)

vollständig typisiert → alle Ausdrücke sind garantiert *typkonsistent;* Typen müssen dem Compiler aber nicht bekannt sein. Muss spätestens während Programmausführung überprüft werden.

***stark typisiert* →** Typkonsistenz aller Ausdrücke durch Compiler garantiert.

***dynamisch typisiert* →** typisiert, aber nicht statisch typisiert.

***schwach typisiert* →** typisiert, aber nicht stark typisiert.



Typzwang:

konservativ → Bei Typfehler muss das Programm geändert werden, sodass es keine Typfehler mehr gibt.

optimistisch → Programm geht davon aus, dass es keine Typfehler enthält.

Durchlässigkeit:

löchrig → e.g. Arrays in C (Zugriff kann außerhalb der Grenzen liegen)

dicht → nicht löchrig duh.

Typäquivalenz:

strukturgleich → einfachere Handhabung

namensgleich → zusätzlicher Abstraktionsmechanismus. Erlaubt Einführung von abstrakten Datentypen.

Abstrakte Datentypen (ADT)

Verstecken die interne Darstellung seiner Instanzen. Nur exportierte Operationen sind von außen anwendbar.

Abstrakte Typen (≠ ADT)

Stellen unvollständig spezifizierte Typen dar, aus denen *konkrete Typen* abgeleitet werden.

monomorphe Typsysteme → jede Variable oder Routine hat genau einen Typ.

polymorphe Typsysteme → Variablen und Routinen können mehrere Typen haben.

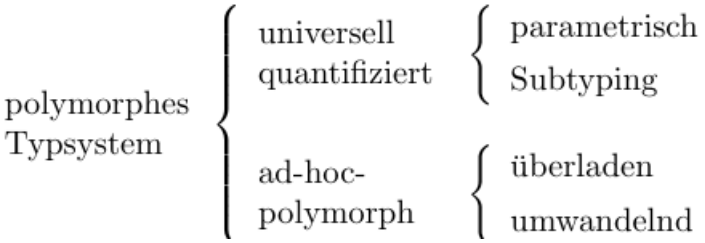
1.2.3 Polymorphe Typsysteme

deklarierter Typ → jener der deklariert wurde

statischer Typ → jener der vom Compiler ermittelt wurde

dynamischer Typ → der speziellste Typ, den der in der Variable gerade gespeicherte Wert hat. (werden für dynamisches Binden verwendet.)

Arten von polymorphen Typsystemen



parametrisch → (= generisch) haben Typparameter.

subtyping → *e.g.* ADT Person hat die beiden Untertypen Student und Angestellter.

überladen → e.g. überladene Methoden/Routinen. Methoden haben gleichen Namen mit verschiedenen Typen. (*dynamisches Binden)*

umwandelnd → casting (implizit (float → int), explizit)

Einfache theoretische Modelle

2.1 Der Lambda-Kalkül

2.1.1 Der untypisierte Lambda-Kalkül

Menge aller Lambda-Ausdrücke *Exp:*

1. $x \in Exp$ wenn $x \in Id$; Id ist eine vorgegebene Bezeichner-Menge (Id sind einfach Namen)
2. $(e_1\ e_2) \in Exp$ wenn $e_1 \in Exp$ und $e_2 \in Exp$ (***Anwendung)***
3. $(\lambda x.e) \in Exp$ wenn $x \in Exp$ und $e \in Exp$ (***Abstraktion)***

Ersetzung:

$$
\begin{aligned}
[e/x_1]x_2 &= \begin{cases} e, & x_1 = x_2 \\ x_2, & x_1 \neq x_2 \land x_2 \in Id \end{cases} \\
[e_1/x](e_2\ e_3) &= ([e_1/x]e_2)\ ([e_1/x]e_3) \\
[e_1/x_1](\lambda x_2.e_2) &= \begin{cases} \lambda x_2.e_2, & x_1 = x_2 \\ \lambda x_2.[e_1/x_1]e_2, & x_1 \neq x_2 \land x_2 \neq free(e_1) \\ \lambda x_3.[e_1/x_1]([x_3/x_2]e_2), & sonst, wobei \\ & x_1 \neq x_3 \neq x_2 \land x_3 \notin free(e_1) \cup free(e_2) \end{cases}
\end{aligned}
$$

Konversionen

1. α-Konversion (Umbenennung)

$$
\begin{gathered}
\lambda x_1.e \iff \lambda x_2.[x_2/x_1]e \\
\text{wobei } x_2 \in Id \text{ und } x_2 \notin free(e)
\end{gathered}
$$

2. β-Konversion (Anwendung)

$$
(\lambda x.e_1)e_2 \iff [e_2/x]e_1
$$

3. η-Konversion

$$
\lambda x.(e\ x) \iff e \text{ für } x \in Id \text{ und } x \notin free(e)
$$

Reduktionen

1. β-Reduktion

$$
(\lambda x.e_1)\ e_2 \implies [e_2/x]e_1
$$

2. η-Reduktion

$$
\lambda x.(e\ x) \implies e \text{ für } x \in Id \text{ und } x \notin free(e)
$$

Normalform

Der Lambda-Ausdruck kann weder mit β-Reduktionen noch mit η-Reduktion weiter reduziert werden. Es muss nicht für jeden Ausdruck eine Normalform geben. Gegenbeispiel:

$$(\lambda x.(x\ x))\ (\lambda x.(x\ x))$$

!Wenn es eine Normalform gibt ist diese eindeutig! (bis auf α-Konversionen)

Fixpunkt-Theorem (erlaubt Rekursion)

Jeder Lambda-Ausdruck e hat einen Fixpunkt e', sodass $(e\ e')$ zu e' konvertiert werden kann.

$$e' = (Ye)\ (\text{Y-Kombinator})$$

Churchs These

Genau jene Funktionen von den natürlichen Zahlen in die natürlichen Zahlen sind effektiv berechenbar, die im Lambda-Kalkül ausgedrückt werden können.

(⇒ alles was berechenbar ist, ist auch im Lambda-Kalkül berechenbar)

2.1.2 Der typisierte Lambda-Kalkül

Typ:

1. $b \in Typ$ wenn $b \in BasTyp$
2. $t_1 \to t_2 \in Typ$ wenn $t_1 \in Typ$ und $t_2 \in Typ$

Menge aller typisierten Ausdrücke:

1. $x : t \in Exp$ wenn $x : t \in Id$
2. $(e_1 : t_2 \to t_1\ e_2 : t_2) : t_1 \in Exp$ wenn $e_1 : t_2 \to t_1 \in Exp$ und $e_2 : t_2 \in Exp$
3. $(\lambda x : t_2.e : t_1) : t_2 \to t_1 \in Exp$ wenn $x : t_2 \in Id$ und $e : t_1 \in Exp$

Regeln analog zum untypisierten Kalkül.

Im Gegensatz zum untypisierten Kalkül kann jedoch kein Fixpunktoperator definiert werden da e sowohl den Typ $t_2 \to t_1$ als auch t_2 haben muss. Daher definieren wir einen *typisierten Fixpunktoperator* Y_t vom Typ $(t \to t) \to t$.

δ-Konversionsregel

$$(Y_t : (t \to t) \to t\ e : t \to t) : t \iff$$
$$(e : t \to t\ (Y_t : (t \to t) \to t\ e : t \to t) : t) : t$$

2.1.3 Strukturierte Typen

δ-Regeln erlauben die Konstruktion von strukturierten Typen:

- Kartesisches Produkt (Tupel)
- Verbunde (Records)

2.2 Logik

2.2.3 Typisierte logische Programme

Ein logisches Programm kann als eine Menge von Typspezifikationen — also als Typsystem — gesehen werden, und das Ergebnis einer Anfrage besagt, ob und unter welchen Bedingungen ein Ziel den Typspezifikationen entspricht.

2.3 Algebren

Universelle Algebra: $\langle A, \Omega \rangle$, wobei A eine nichtleere Trägermenge und Ω ein System von Operationen auf A ist. Der Typ ist die Stelligkeit der Operationen in Ω.

Varietät: $V(\Omega, \Delta)$ beschreibt Familie von universellen Algebren. Hier sind Δ Gesetze, die gelten.

2.3.1 Abstrakte Datentypen

Universelle Algebren und deren Varietäten werden oft als Grundlage für ADTs verwendet. Ω und deren Typen (=Stelligkeiten) werden als *Signatur bezeichnet.*

Freie Algebren: Hier gelten außer den Gesetzen in Δ und den daraus ableitbaren Gesetzen, keine weiteren.

Heterogene Algebra: ist ein Tupel $\langle A_1, ..., A_n, \Omega \rangle$ wobei $A_1, ..., A_n$ Trägermengen sind und Ω ein System von Operationen ist. Trägermengen haben einen Namen (=*Sorte*).

3 Typen in imperativen Sprachen

3.1 Datentypen in Ada

- Datentypen
  - elementar
    - skalar
      - diskret
        - Aufzählungen
          - Zeichen
          - Wahrheitswerte
          - andere
        - ganze Zahlen
          - vorzeichenbehaftet
          - modular
      - reelle Zahlen
        - Gleitkommazahlen
        - Festkommazahlen
          - gewöhnlich
          - dezimal
    - Zeiger
  - zusammengesetzt

- Verbunde
  - einfache Verbunde
  - diskriminante Verbunde
    - mit Varianten
    - ohne Varianten
- Felder
  - mit spezifizierten Grenzen
  - ohne spezifizierte Grenzen

3.1.1 Skalare Typen

Aufzählungstypen.

Vergleichsoperatoren: $<, \leq, =, \geq, >, \neq \quad : t \times t \to \texttt{BOOLEAN}$

Attribute:

Funktion	Typ	Beschreibung
t'FIRST	t	kleinster Wert in t
t'LAST	t	größter Wert in t
t'SUCC	$t \to t$	nachfolgender Wert
t'PRED	$t \to t$	vorausgehender Wert
t'POS	$t \to$ CARDINAL	Position des Wertes
t'VAL	CARDINAL $\to t$	Wert an Position

Beispiel:

```
type TAG is (MO, DI, MI, DO, FR, SA, SO);
type ARBEITSTAG is (MO, DI, MI, DO, FR);
```

Zahlen.

Ganze Zahlen

```
type SEITENZAHL is range 1..10000;
type CARDINAL is range 0..INTEGER'LAST;
type ZWEI_ZIFFERN is mod 100;
```

Gleitkommazahlen

```
type MY_FLOAT is digits 8 range -1.0..1.0E30;
```

Attribute:

Funktion	Typ	Beschreibung
t'DIGITS	CARDINAL	berechenb. Dezimalstellen
t'MANTISSA	CARDINAL	Länge der Binär-Mantisse
t'EMAX	CARDINAL	maximaler Exponent
t'SMALL	t	kleinste positive Zahl
t'LARGE	t	größte positive Zahl
t'EPSILON	t	max. Fehler/Rechenschritt

Festkommazahlen

```
type SPANNUNG is delta 0.1 range 0.0..10.0;
type PREIS is delta 0.01 digits 5;
```

Attribute:

Funktion	Typ	Beschreibung
t'DELTA	CARDINAL	Mindestgenauigkeit
t'ACTUAL_DELTA	CARDINAL	tatsächl. Genauigkeit
t'BITS	CARDINAL	Speicherbedarf
t'LARGE	t	größte Zahl

3.1.2 Zusammengesetzte Typen

Felder

```
type VEKTOR is array (1..5) of INTEGER;
type MATRIX is array (1..K*J,Func(N)..Func(N*M)) of FLOAT;
type STUNDENTAFEL is array (TAG range MO..SA, 1..8) of TEXT;
```

Attribute:

Funktion	Typ	Beschreibung
t'FIRST(i)	t_i	untere Indexgrenze
t'LAST(i)	t_i	obere Indexgrenze
t'LENGTH(i)	CARDINAL	Anzahl der Werte

Verbundtypen

```
type DATUM is
        record
           TAG:   INTEGER range 1..31;
           MONAT: MONATSNAME;
           JAHR:  INTEGER range 1800..2200;
        end record;
```

Diskriminanten: (=bestimmen Struktur des Verbundtyps)

```
type GERAET is (DRUCKER,PLATTE,TROMMEL);
 type PERIPHERIE (SORTE: GERAET) is
```

```
record
   STATUS: ZUSTAND;
   case SORTE is
      when DRUCKER ⇒
            ZEILE:
            INTEGER;
      when others ⇒
            ZYLINDER: INTEGER;
            SPUR:
            INTEGER;
   end case;
end record;
```

3.1.3 Unterbereichstypen, abgeleitete Typen und Zeiger

Unterbereichstypen.

Enthalten weiter Einschränkungen auf Typen.

```
subtype WERKTAG is TAG(MO)..TAG (SA);
subtype SMALL is INTEGER range -100..100;
subtype BETRIEBS_SPANNUNG is SPANNUNG delta 0.5 range 0.0..5.0;
subtype PP is PREIS range 0.01..999.99;
subtype SMALL_VEKTOR is VEKTOR(2..4);
subtype MATRIX1 is MATR(1..100,1..100);
subtype PER_DR is PERIPHERIE(DRUCKER);
```

Stellen keine neuen Typen dar. Sie legen nur Einschränkungen fest, die zur Laufzeit überprüft werden.

Abgeleitete Typen

Sind eigenständige Typen

```
type APFEL is new INTEGER range 0..100;
type BIRNE is new INTEGER range 0..100;
```

Die beiden Typen sind unterschiedlich, werden jedoch durch explizite Typumwandlung vergleichbar.

Zeiger

Können auf jedem beliebigen Typ definiert werden.

```
type APFEL_P is access APFEL;
MY_APFEL: APFEL_P := new APFEL;
```

3.2 Prozeduren und Prozesse

Auch Funktionen, Prozeduren und Prozessen kann ein Typ zugeordnet werden.

3.2.1 Funktionen und Prozeduren

Beispiel:

```
procedure PUSH (EL: in ELEMENT_T; ST: in out STACK_T);
procedure POP (EL: out ELEMENT_T; ST: in out STACK_T);
function MAX (X, Y: in FLOAT) return FLOAT;
function "*" (X: in MATR(A,B); Y: in MATR(B,C)) return MATR(A,C);
```

Der Kopf einer Routine kann als Typdefinition aufgefasst werden. In Ada 83 stehen Funktions- oder Prozedurnamen in einer strikten eins-zu-eins Beziehung mit ihrer Implementierung.

3.2.2 Inkarnationen und Prozesse

Inkarnation → Instanz (= durch Routinenaufrufe erzeugte “stack frames”)

Prozess → (=tasks) nebenläufige Programmierung

Beispiel:

```
task type DRUCKER_TREIBER is
   entry DRUCK_ZEILE (ZL: in STRING(80));
   entry SEITENVORSCHUB;
   entry DRUCKER_STATUS (F: out STATUS);
end;

DRUCKER_POOL: array(1..MAX_DRUCKER) of DRUCKER_TREIBER;
DRUCKER_POOL(1).DRUCK_ZEILE("Hallo!");
```

Attribute:

Funktion	Typ	Beschreibung
p’TERMINATED	BOOLEAN	p beendet?
p’PRIORITY	CARDINAL	Priorität von p
p’STORAGE_SIZE	CARDINAL	Speicherbedarf für p
e’COUNT	CARDINAL	Anzahl der Aufrufe

Auch hier gibt es eine eins-zu-eins Beziehung zwischen tasks und deren Implementierung.

3.3 Generische Pakete

Das theoretische Konzept der heterogenen Algebren mit Parametersorten wird fast unverändert zur Definition der Schnittstellen von abstrakten Datentypen in Ada übernommen.

Beispiel für generisches Paket:

```
generic
   type ELEMENT_T is private;
   MAX_ELEM: CARDINAL;
package STACK is
   type STACK_T is limited private;
   procedure PUSH (EL: in ELEMENT_T; ST: in out STACK_T);
```

```
    procedure POP (EL: out ELEMENT_T; ST: in out STACK_T);
    function IS_EMPTY (ST: in STACK_T) return BOOLEAN;
    function IS_FULL (ST: in STACK_T) return BOOLEAN;
private
    type STACK_T is
            record
               EINTRAEGE: array(1..MAX_ELEM) of ELEMENT_T;
               INDEX: INTEGER range 0..MAX_ELEM;
            end record;
end STACK;
package body STACK is
...
endSTACK;

package I_STACK is new STACK(INTEGER,100);
MYSTACK: I_STACK.STACK_T;
```

Auch bei Modulen gibt es eine eins-zu-eins Beziehung zwischen Modultyp und dessen Implementierung.

4 Modelle polymorpher Typsysteme und Typen in funktionalen Sprachen

4.1 Order-sorted Algebras

Heterogene Algebren erlauben hohe Flexibilität bei Überladung von Routinen. Um diesen "Wildwuchs" einzuschränken wird eine Ordnung auf Sorten eingeführt. Man spricht von *order-sorted algebra".* Dies ermöglicht Subtyping.

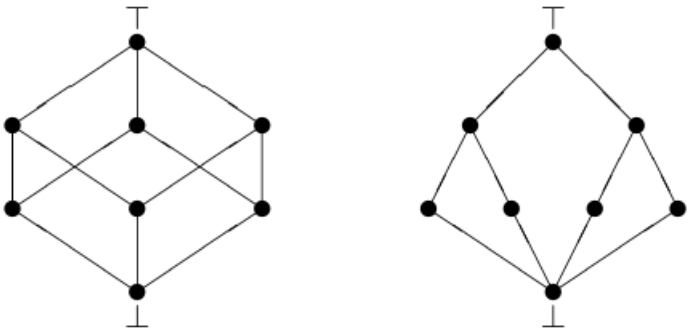
4.1.1 Verbände über Sortenmengen

Angenommen wir haben die Sorten $\top$ und $\bot$ in S. Wir führen den Verband $\langle S, \leq, \sqcup, \sqcap, \top, \bot \rangle$ über S ein.

Eigenschaften:

1. $\leq$ ist eine reflexive, transitive und antisymmetrische Relation auf S (=Halborndung von S), sodass $\bot \leq s$ und $s \leq \top$ für alle $s \in S$ gilt.
2. 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:



4.1.2 Polymorphe Operationen

Da Trägermengen teilweise ineinander enthalten sind, sind auch die Operationen automatisch polymorph.

→ Jede Operation bildet eine Menge von Eingangswerten in eine Menge von Ergebniswerten ab. Auch die Operation auf jeder beliebigen Teilmenge der Eingangswerte ist definiert, und das Ergebnis liegt stets in jeder Obermenge der tatsächlich möglichen Ergebniswerte.

→ Wenn man statt Trägermengen typisierte Variablen betrachtet, bedeutet das folgendes:

> Als Eingangsparameter kann jede Variable stehen deren statischer Typ ein Untertyp des statischen Typs des formalen Parameters ist. Als Ausgangsparameter kann jede Variable stehen deren statischer Typ ein Obertyp des entsprechenden Parametertyps ist. Ein Durchgangsparameter muss genau denselben statischen Typ haben, da sie einen Eingangs- als auch einen Ausgangs-Parameter darstellt.

4.2 Typinferenz und das Typsystem von ML

Im typisierten Lambda-Kalkül ist es leicht möglich monomorphe Funktionen aus zu drücken.

e.g.:

```
Float → Float
Integer → Integer
```

Es ist aber nicht möglich generische Funktionen wie zum Beispiel

```
a → a
```

aus zu drücken. Es ist nicht möglich einfach Konversionsregeln einzuführen, da Typkonsistenz für ein gegebenes Programm im Allgemeinen nicht entscheidbar ist.

4.2.1 Typinferenz

Statt zu versuchen das Typüberprüfungsproblem direkt zu lösen, lässt man die Typen weg (verwendet also den untypisierten Lambda-Kalkül) und dann wird versucht, die Typen der Lambda-Ausdrücke zu berechnen. (=Typinferenz)

4.2.3 Ein Typinferenz-Algorithmus

- c ... Konstante
- x ... Parametername

- v ... Typparameter
- s, t ... Typ
- e ... untypisierter Lambda-Ausdruck
- f ... typisierter Lambda-Ausdruck
- Γ ... Umgebung mit Typzuweisungen an Parameternamen.
- $PT()$ ... Polymorph Type
- $x \triangleright y$ ... gelesen als: "unter der Annahme x gilt y"

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

Gleichungen:

1. Ordnet Konstanten den Typ der Konstante zu. (e.g. $1 \Rightarrow int$, $+ \Rightarrow int \to (int \to int)$)
2. Weist Parameternamen einen beliebigen neuen Typparameter als Typ zu. Wenn dieser Typ später im Algorithmus nicht weiter eingeschränkt wird, kann der Parameter durch beliebige Ausdrücke ersetzt werden.
3. Behandelt Anwendungen (=Funktionsaufrufe). Für die Funktion und deren Parameter werden rekursiv die Typen ermittelt. Die Unifikation liefert eine Menge an Ersetzungsregeln (siehe unten)
4. Hier werden Abstraktionen behandelt. Wenn x im Funktionsrumpf e vorkommt, dann wird x dann ist x gebunden und x wird aus der Umgebung Γ entfernt. Ansonsten hat x einen beliebigen Parametertyp.

Unifikation

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

1. Unter der leeren Menge wird abgebrochen.
2. Wir haben eine Menge E und zwei Basistypen b, b', welche einer Variable zugeordnet werden sollen. Wenn die Basistypen nicht gleich sind (e.g. Float und Integer) dann haben wir einen Typfehler und brechen ab. Ansonsten müssen wir nichts weiter tun und gehen rekursiv weiter.
3. Wir haben wieder eine Menge E und zwei Typparameter v, t, welche einer Variable zugeordnet werden sollen. Wenn diese übereinstimmen müssen wir nichts weiter tun. Falls t ein Verbund o.Ä. ist, welcher den Typparameter v enthält müssen wir abbrechen, da wir uns sonst in einer Endlosschleife befinden. Ansonsten können wir alle vorkommen von v durch t ersetzten.
4. Hier wird mit Funktionstypen umgegangen. Es werden die beiden Eingangs- und Ausgangsparameter gleich gesetzt.

4.3 Funktionale Ansätze für Subtyping

4.3.1 Einfache Untertypbeziehungen

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.

Regeln.

Γ ... Umgebung mit Subtyping-Annahmen

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

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

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

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

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

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

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

1. t ist Untertyp von sich selbst (=reflexiv).
2. Jeder Typ t ist Untertyp des allgemeinsten Typs $\top$.

3. $\bot$ ist Untertyp jedes Typs t.
4. Untertyprelationen sind transitiv.
5. Beschreibt Untertyprelationen für Funktionen. Der Eingangstyp muss allgemeiner sein (=**kontravariant).** Der Ergebnistyp muss spezieller sein (=**kovariant).**
6. Beschreibt Untertyprelationen für Verbundtypen.
   Beispiel:

   ```
   t = {name:string, alter:{0,...,120}}
   s = {name:string, alter:{18,...,65}, mnr:int}}
   ```

7. Beschreibt Untertyprelationen für Variantentypen (=äquivalent zu 6.).

4.3.2 Rekursive Typen

Beispiel:

```
baum = {i:int, l:baum, r:baum}
```

Hier gibt es ein Problem mit der Typdarstellung: Es ist nicht möglich alle Vorkommen von $baum$ durch dessen Typ zu erstetzen, da der Typ sonst unendlich groß werden würde.

Daher führen wir einen Typparameter v ein und binden diesen mit dem Fixpunktoperator μ.

$$\mu v.\{i : int, l : v, r : v\}$$

Regeln für Äquivalenz rekursiver Typen:

1. α-Konversionsregel:

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

2. Faltungsregel:

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

Die erste Regel ist gleich wie im Lambda-Kalkül. Die zweite Regel hat mehrere Bedeutungen:

- Wenn der Typparameter v in t nicht vorkommt, ist $\mu v.t$ äquivalent zu t.
- Sonst können alle Vorkommen von v in t beliebig durch den Typ selbst $(\mu v.t)$ ersetzt werden.

Subtyping-Regel für rekursive Typen:

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

Diese Regel besagt, dass zwei rekursive Typen in einer Untertypbeziehung stehen, wenn die rekursionsfreien Varianten dieser Typen zueinander in einer Untertypbeziehung stehen, wobei Subtyping-Annahmen für die Typparameter gelten.

Beispiel:

$$\mu u.\{i : int, f : int \to u, next : u\} \leq \mu v.\{i : int, f : int \to v\}$$

4.4 Das PER-Modell

(= *"partial equivalence relation".)*

Das PER-Modell beschreibt Relationen R auf einer Menge an Werten W. Eine Relation R ist eine partielle Äquivalenzrelation (PER) auf W genau dann wenn R eine symmetrische $(vRw \Leftrightarrow wRv)$ und transitive $(uRv \wedge vRw \Rightarrow uRw)$, aber nicht notwendigerweise reflexive (wRw) Relation auf W ist.

Jeder Typ A entspricht einer PER, und die Menge der durch A beschriebenen Werte ist die Menge der Äquivalenzklassen von A. $(\{\{w\}_A | wAw\}$, wobei $\{w\}_A = \{v | vAw\})$

Bei Verbunden ist ein Typ ein Untertyp eines anderen Typs wenn

1. die Instanzen des Untertyps eine Teilmenge der des Obertyps sind und/oder
2. der Untertyp alle Komponenten seines Obertyps und möglicherweise zusätzliche Komponenten hat.

Diese beiden Fälle lassen sich leicht mit dem PER-Modell beschreiben:

1. Es gilt $B \leq A$ (aufgrund erster Regel)

$$B = \{(a,a),(a,b),(b,a),(b,b)\} = \{\{a,b\}\}$$
$$A = B \cup \{(c,c),(c,d),(d,c),(d,d)\} = \{\{a,b\},\{c,d\}\}$$

2. Es gilt $B \leq A$ (aufgrund zweiter Regel)

$$B = A \setminus \{(c,d),(d,c)\} = \{\{a,b\},\{c\},\{d\}\}$$
$$A \text{ wie oben}$$

   Hier trifft B eine genauere Einteilung in Äquivalenzklassen als A.

5. Typen in objektorientierten Sprachen

5.1 Untertypen in Beispielen

5.1.1 Ada 95

$$\begin{aligned}\text{Vererbung} = &\ \text{Typableitung} \\ &+ \ddot{\text{U}}\text{berschreiben} \\ &+ \text{Typerweiterung} \\ &+ \text{dynamische Typerkennung}\end{aligned}$$

1. Typableitung

```
type Int is range 0..10000;
function "+"(Links, Rechts: Int) return Int;
```

```
type Laenge is new Int;
-- function "+"(Links, Rechts: Laenge) return Laenge;
```

Die zweite Funktion “+” ist obsolet (und deswegen auskommentiert), da diese implizit auch für *Laenge* deklariert ist. Auch die Implementierung der Funktion wird geerbt.

2. Überschreiben

```
type Winkel is new Int;
function "+"(Links, Rechts: Winkel) return Winkel;
```

Hier wird im Gegensatz zu oben die Funktion “+” überschrieben.

Wie wir sehen werden alle Typbezeichner der ursprünglichen Routine auf jene des Untertyps geändert. Sowohl Eingangs- als auch Ergebnistypen sind Untertypen von *Int* und somit **kovariant**. **Dies Widerspricht dem Ersetzbarkeitsprinzip!**

3. Typerweiterung

```
type Person is tagged record
      Vorname: String (1..20);
end record;
type Mann is new Person with record
      Barttraeger: Boolean;
end record;
type Frau is new Person with null record;
```

Typerweiterung in Ada beschreibt das Hinzufügen neuer Komponenten zu abgeleiteten Verbundtypen. Die erweiterbaren Verbunde müssen mit *“tagged”* gekennzeichnet sein.

Typumwandlungen sind hier nur in Richtung Vorgängertyp möglich.

Erweiterbare Verbundtypen sind Voraussetzung für dynamische Typerkennung.

Klassenweiter Typ.

Jeder erweiterbare Verbundtyp `t` definiert implizit einen klassenweiten Typ `t'Class`. Dieser Umfasst `t` und alle von `t` abgeleiteten Typen. (e.g. zu `Person'Class` gehören `Person, Mann, Frau` und zu `Frau'Class` nur `Frau`)

Spezifischer Typ.

Jede Instanz eines klassenweiten Typs enthält eine unsichtbare Komponente *“tag”*, das den spezifischen Typ der Instanz angibt. Diese Komponente wird während der Programmausführung gebraucht. Daher muss bei der Initialisierung der spezifische Typ verwendet werden:

```
P: Person'Class := Frau'(Vorname ⇒ "Anna");
```

Der spezifische Typ von P kann zur Laufzeit nur mit Zeigern geändert werden, aufgrund des unterschiedlichen Platzbedarfs.

```
type PersonRef is access Person'Class;
P_Ref: PersonRef;
P_Ref := new Frau'(Vorname ⇒ "Anna");
P_Ref := new Mann'(Vorname ⇒ "Stefan",
                   Barttraeger ⇒ False);
```

4. Dynamische Typerkennung

```
function Anrede (P: in Person) return; -- ""
function Anrede (M: in Mann) return String; --"Herr"
function Anrede (F: in Frau) return String; --"Frau"

procedure Display (P: in Person'Class) is
begin
    Put(Anrede(P));
    Put(P.Vorname);
    if P in Mann'Class then
        Put("maennlich")
    elsif P in Mann'Class then
        Put("weiblich")
    end if;
end Display;
```

Über dynamisches Binden wird hier entsprechend des spezifischen Typs von P die richtige Anrede verwendet. Weiter wird die oben genannte *“tag”* Komponente bei den `if's` verwendet.

Abstrakte Datentypen

```
type A is abstract tagged null record;
procedure Q (X: in A) is abstract;

type B is new A with record
    I: Integer;
end record;
procedure Q (X: in B) is . . .
```

Generizität

```
generic
  type T is private;
  with funtion "<"(X, Y:T) return Boolean;
function Max(X, Y:T) return T is
begin
  if X < Y
```

```
      then return Y;
      else return X;
    end if;
end Max;
```

5.2 Allgemeine Konzepte

5.2.2 Untertypen und Vererbung

Untertypen bieten mehr Möglichkeiten in der Schnittstelle und beschreiben das Verhalten genauer als ein entsprechender Obertyp.

Vererbung bezieht sich dagegen auf die Implementierung. Sie wird geteilt in *Vererbung von Konzepten* (=Subtyping) und *Vererbung von Implementierung*. Letzteres zielt auf Wiederverwendung des Codes ab, während Subtyping sich mit gleichen/ähnlichen Konzepten beschäftigt.

5.2.4 Untertypen und Generizität

Subtyping würde nach dem Ersetzbarkeitsprinzip so definiert, dass Instanzen eines Untertyps überall eingesetzt werden können, wo Instanzen eines entsprechenden Obertyps erwartet werden. Generizität ist ein Mechanismus zur Beschreibung einer Familie von Typen, die sich nur durch die für Typparameter eingesetzten Typen unterscheiden.

In manchen Fällen sind Subtyping und Generizität gegeneinander austauschbar. Jedoch ist es ohne Subtyping zum Beispiel nicht möglich heterogene Listen auszudrücken. Ohne Generizität ist es nicht möglich müsste oft kovariantes Subtyping (=Eingangsparameter werden auch spezieller) verwendet werden, was unerwünschten Einschränkungen mit sich bringt. Diese Einschränkungen gibt es bei Generizität nicht.

Subtyping und Generizität ergänzen einander.

Gebundene Generizität.

```
class List<? extends T>
```

Gebundene Generizität verwendet Subtyping als Hilfsmittel.

Generische Untertypbeziehungen.

Bei kovariantem Subtyping (e.g. in Eiffel) ist dies leicht festzustellen:

> $x[a_1 \to s_1, ..., a_n \to s_n]$ ist ein Untertyp von $y[b_1 \to t_1, ..., b_n \to t_2]$, wenn für alle $1 \leq i \leq n$ gilt, dass s_i ein Untertyp von t_i und $x[s_1, ..., s_n]$ ein Untertyp von $y[s_1, ..., s_n]$ ist.

Für Sprachen mit kontravariantem Subtyping braucht es mächtigere Mechanismen:

> Es gibt im Wesentlichen zwei Möglichkeiten, Klassen bzw. generische Typen zueinander in Beziehung zu stetzen, nämlich **F-gebundene Generizität** und **Subtyping höherer Ordnung**.

F-gebundene Generizität

Sei T ein generischer Typ und $T[S]$ ein Typ, der durch Ersetzung eines Typparameters in T durch den Typ S aus T abgeleitet wird. Im *F-gebundenen Polymorphismus* gilt die Beziehung $S \leq T[S]$. Dadurch wird in einigen Sprachen rekursive Definitionen wie in $S \leq T[S]$ ermöglicht.

Beispiel

```
interface Comparable<T> {
   int compareTo (T that);
}

class Integer implements Comparable<Integer> {
   public int compareTo (Integer that) {...}
}
```

Aber auch dieser Ansatz hat ein Problem:

> Wenn $S \leq T[S]$ gilt und ein Untertyp $U \leq S$ existiert, kann $U \leq T[U]$ nicht mehr gelten, sondern nur $U \leq T[S]$. Wildcards (e.g $S \leq T[\texttt{? extends } S]$ statt $S \leq T[S]$) können helfen, schränken aber die Art der Zugriffe (lesend oder schreibend) stark ein.

Subtyping höherer Ordnung

Hat eine ähnliche Ausdrucksstärke wie F-gebundene Generizität.

Zwei generische Typen S und T sind in einer Untertypbeziehung höherer Ordnung, wenn $S[U] \leq T[U]$ für alle Typen U gilt.

Nicht-generische Typen können als Spezialfall von generischen Typen gesehen werden. Als Grundlage für Vererbung kann dann $<\#$ verwendet werden. Diese Beziehung wird *Matching* genannt und ist definiert durch $S <\# T$ wenn $S[U] \leq T[U]$ für alle Typen U. Das Eersetzbarkeitsprinzip ist aber durch $<\#$ verletzt, da in rekursiven Typen kovariante Eingangsparameter verwendet werden dürfen.

Beispiel:

$$\{i : int, f : u \to u, next : u\} <\# \{i : int, f : v \to v\}$$

Zusammenfassend unterstütz Subtyping höherer Ordnung auch binäre Methoden (= Methoden die mindestens einen Parameter haben, der der eigenen Klasse entspricht). Jedoch muss entweder statisch sichergestellt werden, dass bestimmte übergebene Argumente gleiche Typen haben, oder akzeptieren, dass bei *Matching* Typfehler erst zur Laufzeit erkannt werden.

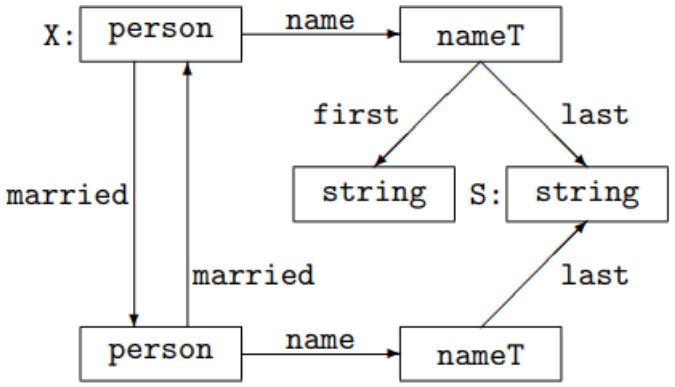
5.3 Logik und Subtyping

ψ-Terme kombinieren logische Terme mit Typen, auf denen ein Verband definiert ist.

Beispiel:

```
X:person(name ⇒
                nameT(first ⇒ string,
                      last ⇒ S:string),
```

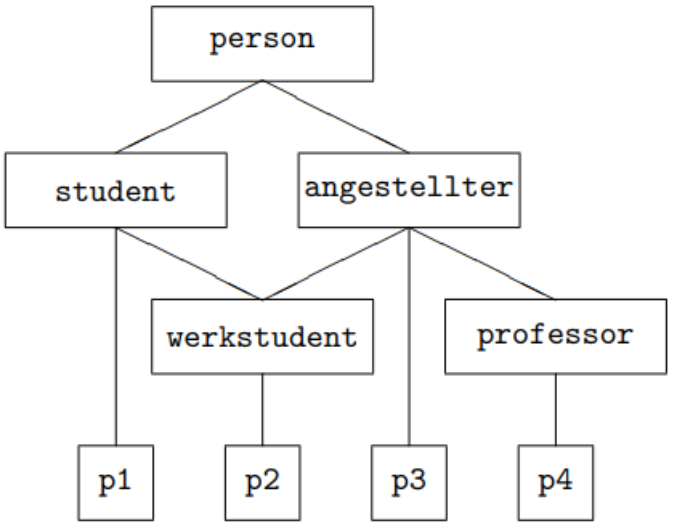
```
married ⇒
    person(name ⇒ nameT(last ⇒ S),
           married ⇒ X))
```



Auf ψ-Termen gibt es eine Halbordnung $\preceq$, die sich aus der Halbordnung $\leq$ auf der Menge der Sorten ableiten lässt. $t \preceq t'$ zwischen zwei ψ-Termen t und t' gilt genau dann, wenn

1. $s \leq s'$ für die äußersten Sorten s und s' von t bzw. t' gilt, und
2. alle Attribute $a_1, ..., a_n$ von t' auch Attribute von t sind, und
3. $t_i \preceq t'_1$ für $i \in \{1, ..., n\}$ und jedes $a_i \Rightarrow t_i$ in t und $a'_i \Rightarrow t'_i$ gilt, und
4. Teilausdrücke in t identisch sind falls die entsprechenden Teilausdrücke in t' identisch sind.

Es gibt für zwei ψ-Terme auch ein eindeutig bestimmtes Infimum:



Die Unifikation von

```
student(betreuer ⇒
            professor(assistent ⇒
                          angestellter),
        freund ⇒ person)
```

und

```
X:person(betreuer ⇒
                p4(assistent ⇒ X),
          vermieter ⇒ person)
```

liefert das Ergebnis

```
X:werkstudent(betreuer ⇒
                   p4(assistent ⇒ X),
                freund ⇒ person,
                vermieter ⇒ person)
```

5.4 Typen für aktive Objekte

5.4.1 Prozesstypen

```
task type Buffer is
      entry Put (E: in Element);
      entry Get (E: out Element);
end Buffer;

task body Buffer is
      X: Element;
begin
      loop
            accept Put (E: in Element)
                  do X := E;
            end;
            accept Get (E: out Element)
                  do E := X;
            end;
      end loop;
end Buffer;
```

```
Buffer = Put (in Element) *
           Get (out Element) *
           Buffer
```

Bei diesem Typ kann ein Element in den Buffer gelegt werden und anschließend ausgelesen werden (! in dieser Reihenfolge, aufgrund des sequentiellen Operators “*”). Durch den rekursiven Buffer Aufruf kann diese Sequenz unbeschränkt wiederholt werden.

```
Buffer = Put (in Element) *
           Get (out Element) *
           Put (in Element) *
```

```
            Get (out Element) *
            ...
```

Ist somit äquivalent zu obigem Beispiel.

```
IBuffer = ( Put (in Element) *
                Get (out Element) ) ||
            IBuffer
```

Dieser Typ beschreibt einen Buffer der gleichzeitig beliebig viele **Put**-Nachrichten, aber höchstens so viele **Get**-Nachrichten wie gerade Elemente im Buffer sind.

```
UBuffer = Put (in Element) ||
            Get (out Element) ||
            UBuffer
```

Dieser Typ kann beliebig viele **Put**- und **Get**-Nachrichten empfangen.

5.4.2 Vererbungsanomalie

(Problem bei nebenläufiger Programmierung im Zusammenhand mit Vererbung)

Wenn eine Unterklasse eine Klasse durch das Dazufügen neuer Operationen erweitert, ergeben sich meist gänzlich neue Synchronisationsbedingungen, sodass der Synchronisationscode für viele ansonsten nicht veränderte Operationen umgeschrieben werden muss. Dieses Problem wird häufig als *Vererbungsanomalie* in nebenläufigen, objektorientierten Sprachen bezeichnet.

Um dies zu Umgehen kann der Synchronisationscode vom restlichen Code getrennt werden.

luv u ❤