Kapitel 6: Interpretation, abstrakte Interpretation und Typinferenz

Lernziele dieses Kapitels

1. Interpreter: Syntax, denotationelle Semantik, Environment
2. Abstrakte Interpretation mit Beispielen
3. Striktheitsanalyse und ihre Anwendung
4. Curry-Howard-Isomorphismus
5. Typinferenz im Hindley-Milner Typsystem
6. Typen mit Größeninformation

Interpretation

- Interpreter :: Syntax → Semantik
  - Syntax: Datenstruktur (Syntaxbaum)
  - Semantik: Haskell-Funktion: Eingabe → Ausgabe
- abstrakte Interpretation
  - reduzierte Semantik: abstrakte Eingabe → abstrakte Ausgabe
  - Beispiele
    - Restklassenarithmetik auf ganzen Zahlen
    - Striktheitsanalyse
- Typinferenz: kann als abstrakte Interpretation angesehen werden

später in der Vorlesung: stufenweise Interpretation durch Metaprogrammierung

Interpreter

1. definiert die denotationelle Semantik eines syntaktischen Ausdrucks
2. transformiert Eingabedaten in Ausgabedaten

empfohlener Aufbau: induktiv definierte Semantikfunktion

- nach dem induktiven Aufbau des Syntaxbaums
- basierend auf Semantik der Teilausdrücke

Behandlung von gebundenen Variablen

- β-Reduktion: (syntaktische) Substitution
- Semantikfunktion: Ablage von Variablenwerten in einem Environment

Semantikfunktion für den einfach getypten λ-Kalkül

abstrakte Syntax:

```
data LExp = V String              -- 1. Variable
          | LExp :@: LExp         -- 2. Applikation
          | L String LExp         -- 3. Abstraktion
```

Semantikfunktion $\mathcal{S}$ (ϵ Environment):

1. $\mathcal{S}\,[\![\texttt{V}\ x]\!]\,\epsilon = \epsilon\,[\![x]\!]$
2. $\mathcal{S}\,[\![f\ \texttt{:@:}\ x]\!]\,\epsilon = (\mathcal{S}\,[\![f]\!]\,\epsilon)\ (\mathcal{S}\,[\![x]\!]\,\epsilon)$
3. $\mathcal{S}\,[\![\texttt{L}\ a\ v]\!]\,\epsilon = \lambda x\,.\,(\mathcal{S}\,[\![v]\!]\,\epsilon')$,
   wobei $\epsilon'\,[\![a]\!] = x$ und $\epsilon'\,[\![y]\!] = \epsilon\,[\![y]\!]$ falls $y \neq a$.

Kodierung der Semantikfunktion in Haskell

```
data Value a = Ground a                                  -- Grundtyp
             | Funct (Value a -> Value a)                -- Funktionstyp

sem :: LExp -> Env (Value a) -> Value a
sem (V x)      env = value env x                         -- Nachsehen des Wertes
sem (f :@: x) env = let (Funct h) = sem f env            -- f muss Funktion sein
                        y         = sem x env
                    in h y                  -- y hier noch nicht ausgewertet
sem (L x v)    env = Funct (\ a -> sem v (addenv (x,a) env))
```

```
Lambda> sem ((L "a" (V "f" :@: V "a")) :@: V "b")
            (Env [("b",Ground (5::Int)),
                  ("f",Funct (\(Ground x) -> Ground (x*x)))])
25
```

Hilfsfunktionen

```
newtype Env a = Env { content::[(String,a)] }   -- Definition Environment

value :: Env a -> String -> a   -- Wert einer Variablen nachsehen
value env key = case lookup key (content env) of
                  Nothing -> error ("unknown value of: " ++ key)
                  Just x  -> x

instance Show a => Show (Value a) where   -- Wert anzeigen
   show (Ground x) = show x
   show (Funct x)  = "<function>"

addenv :: (String,a) -> Env a -> Env a   -- Neueintrag (Variable,Wert)
addenv  x (Env xs) = Env (x:xs)   -- ggfs. Verschattung eines alten Namens
```

Ein Interpreter für einen angereicherten λ-Kalkül

Hinweis: die momentane Form ist unvollständig,
weil die Vervollständigung eine Übungsaufgabe ist

Haskell-Module:

- `Syntax`: Definitionen der λ-Ausdrücke und Funktionen zur Manipulation von λ-Ausdrücken
- `Reduce`: Anwendung von Reduktionsfolgen auf λ-Ausdrücke
- `Environment`: Abbildung von Namen auf Werte
- `Eval`: semantische Auswertung durch Abbildung von λ-Ausdrücken auf Haskell-Ausdrücke
- `Main`: Test

```
module Syntax(Op(..),LExp(..),substitute) where
import List
infixl 9 :@:
data Op = OAnd | OOr | ONot | OAdd | OSub | OMul | OEq   -- Operatorarten
        deriving (Eq,Show)
data LExp
 = V String                     -- Variable
 | CInt Int                     -- Int  - Konstante
 | CBool Bool                   -- Bool - Konstante
 | LExp :@: LExp                -- Applikation
 | L String LExp                -- Abstraktion
 | Let (String,LExp) LExp       -- [Let (x,y) e]: let x=y in e
 | If LExp LExp LExp            -- [If c t e   ]: if c then t else e
 | Y LExp                       -- Fixpunktoperator
 | Prim Op [LExp]               -- vordefinierter Operator
 deriving (Show)
```

```
module Reduce(reduce_WHNF) where
import Syntax
applyPrim :: Op -> [LExp] -> LExp
applyPrim op xs
 = case (op,xs) of
      (OAnd,[CBool a,CBool b]) -> CBool (a&&b)
      ...
red_Redex_LMHead :: LExp -> Maybe LExp
red_Redex_LMHead = r where
 r (V _)          = Nothing
 r ((L v e):@:x) = Just (substitute v x e)
 r (f :@: x)      = case r f of
                       Just e  -> Just (e :@: x)
                       Nothing -> Nothing
 r (L _ _)        = Nothing
 r (Prim op xs)   = case red_Redex_LMHead_Sequence xs of
                       Nothing -> Just (applyPrim op xs)
                       Just ys -> Just (Prim op ys)          ...
```

```
reduce_WHNF :: LExp -> [LExp]
reduce_WHNF exp = case red_Redex_LMHead exp of
                    Nothing   -> [exp]
                    Just exp' -> (exp:reduce_WHNF exp')

red_Redex_LMHead_Sequence :: [LExp] -> Maybe [LExp]
red_Redex_LMHead_Sequence rs
 = let res = foldl (\ acc x -> case acc of
                Left xs -> case red_Redex_LMHead x of
                             Nothing -> Left (xs++[x])
                             Just x' -> Right (xs++[x']) ;
                Right xs -> Right (xs++[x]))
                    (Left [])
                    rs
   in case res of
         Left _   -> Nothing
         Right ys -> Just ys
```

```
module Environment(Env,value,addEnv,emptyEnv) where

newtype Env a = E { content::[(String,a)] }

value :: Env a -> String -> a
value env key = case lookup key (content env) of
                  Nothing -> error "key not found"
                  Just x  -> x

addEnv :: (String,a) -> Env a -> Env a
addEnv  x (E xs) = E (x:xs)

emptyEnv :: Env a
emptyEnv = E []
```

```
module Eval(Domains(..),eval) where
import Syntax
import Environment
data Domains = I Int | B Bool | F (Domains -> Domains)
instance Show Domains where
   show (I i) = show i
   show (B b) = show b
   show (F x)  = "<function>"

eval :: LExp -> Env (Domains) -> Domains
eval (V x)     env = value env x
eval (f :@: x) env = let (F h) = eval f env
                         y     = eval x env
                     in h y
eval (L x v)   env = F (\ a -> eval v (addEnv (x,a) env))
...
```

```
module Main where
import Syntax / Eval / Environment / Reduce

{- Das Fixpunktfunktional der Fakultätsfunktion in LExp -}
fpfac :: LExp
fpfac = (L "fac" (L "n"
        (If (Prim OEq [V "n",CInt 0])
            (CInt 1)
            (Prim OMul [V "n",(V "fac") :@: (Prim OSub [V "n",CInt 1])])))))

fac5 :: LExp
fac5 = (Y fpfac) :@: CInt 5   -- Die Fakultät von 5 in LExp

test :: String
test =  "Reduktionssemantik: "++show (last (reduce_WHNF fac5))++"\n"
     ++ "denotationelle Semantik: " ++show (eval fac5 emptyEnv)++"\n"

main = putStr test
```

Verwendung übersetzter Module im ghci

```
> ghc -c Syntax.hs
> ghc -c Reduce.hs
> ghci Main.hs

...
Loading package base ... linking ... done.
Skipping  Syntax            ( Syntax.hs, ./Syntax.o )
Skipping  Reduce            ( Reduce.hs, ./Reduce.o )
Compiling Environment       ( Environment.hs, interpreted )
Compiling Eval              ( Eval.hs, interpreted )
Compiling Main              ( Main.hs, interpreted )
Ok, modules loaded: Main, Eval, Environment, Reduce, Syntax.
*Main> main
Loading package haskell98 ... linking ... done.
Reduktionssemantik: CInt 120
denotationelle Semantik: 120
```

Abstrakte Interpretation

- Prinzip: Auswertung nur unter Berücksichtigung abstrakter Information
- Hauptziel: automatische Programmanalyse
- Beispiele
  - Vorzeichenregel für Zahlen
  - Restklassenarithmetik
  - Striktheitsanalyse
  - Datenabhängigkeitsanalyse
  - Größenanalyse
  - Typinferenz
- Herausforderung: Analyse rekursiver Funktionen (Behandlung von Fixpunkten)

Vorzeichenregel für die Multiplikation

- konkrete Struktur: $\bar{K} = (\mathbb{Z}, *)$, abstrakte Struktur: $\bar{A} = (\{\boxdot, \boxminus, \boxplus\}, \circledast)$
- Abstraktionsabbildung: $a : \mathbb{Z} \to \{\boxdot, \boxminus, \boxplus\}$

$$a\ x = \begin{cases} \boxminus & , \text{ falls } x < 0 \\ \boxdot & , \text{ falls } x = 0 \\ \boxplus & , \text{ falls } x > 0 \end{cases}$$

- Abstraktion von $*$: $\circledast$ mit

$\circledast$	$\boxdot$	$\boxminus$	$\boxplus$
$\boxdot$	$\boxdot$	$\boxdot$	$\boxdot$
$\boxminus$	$\boxdot$	$\boxplus$	$\boxminus$
$\boxplus$	$\boxdot$	$\boxminus$	$\boxplus$

- damit gilt: $a\ (x * y) = a\ x\ \circledast\ a\ y$

Restklassen als abstrakte Information

```
data Rem10007 = R10007 Int  deriving (Eq,Show)

instance Num Rem10007 where
   R10007 x + R10007 y = R10007 ((x+y) `mod` 10007)
   R10007 x - R10007 y = R10007 ((x-y) `mod` 10007)
   R10007 x * R10007 y = R10007 ((x*y) `mod` 10007)
   fromInteger x = R10007 (fromInteger x `mod`10007)

evalPoly :: Num a => [a] -> a -> a
evalPoly cs x = foldl (\ v c -> x*v+c) 0 cs
```

```
   *Remainder> evalPoly [1..100000] 2 `mod` 10007
   1006
   (42.57 secs, 1593189860 bytes)
   *Remainder> evalPoly (map R10007 [1..100000]) (R10007 2)
   R10007 1006
   (1.77 secs, 25315500 bytes)
```

Übergang zur Potenzmenge (Bsp. Vorzeichenregel)

- Grund: Umgang mit Informationsverlust
- Bsp.: Vorzeichenregel versagt bei Addition
- $\bar{K} = (\mathcal{P}(\mathbb{Z}), \subseteq)$
- $\bar{A} = (\{\bot, \boxminus, \boxdot, \boxplus, \top\}, \sqsubseteq)$, wobei $\forall x : x \in A : \bot \sqsubseteq x$ und $\forall x : x \in A : x \sqsubseteq \top$
- Abstraktionsfunktion $\alpha : K \to A$ und Konkretisierungsfunktion $\gamma : A \to K$

$$\alpha(X) = \begin{cases} \bot & , X = \emptyset \\ \boxminus & , \forall x : x \in X : x < 0 \\ \boxdot & , X = \{0\} \\ \boxplus & , \forall x : x \in X : x > 0 \\ \top & , \text{sonst} \end{cases}$$

Galoisverbindung

$$\alpha(X) = \begin{cases} \bot & , X = \emptyset \\ \boxminus & , \forall x : x \in X : x < 0 \\ \boxdot & , X = \{0\} \\ \boxplus & , \forall x : x \in X : x > 0 \\ \top & , \text{sonst} \end{cases} \qquad \gamma(M) = \begin{cases} \emptyset & , M = \bot \\ \{\, x \in \mathbb{Z} \mid x < 0 \,\} & , M = \boxminus \\ \{0\} & , M = \boxdot \\ \{\, x \in \mathbb{Z} \mid x > 0 \,\} & , M = \boxplus \\ \mathbb{Z} & , M = \top \end{cases}$$

- Galoisverbindung: $K \rightleftharpoons_{\gamma}^{\alpha} A$
  1. $\forall X : X \in K : X \subseteq \gamma(\alpha(X))$
  2. $\forall M : M \in A : \alpha(\gamma(M)) \sqsubseteq M$
  3. α ist monoton
  4. γ ist monoton
- es gilt: $\alpha \circ \gamma \circ \alpha = \alpha$ und $\gamma \circ \alpha \circ \gamma = \gamma$

Striktheitsanalyse (vorwärts)

- Ziel: in welchen Argumenten ist eine Funktion strikt?
- abstrakte Wertemenge $\{\bot, \top\}$ mit der Ordnung $\bot \sqsubseteq \top$

  $\bot$: konkreter Wert ist garantiert undefiniert

  $\top$: keine Information vorhanden
- $\alpha^\sharp(f)$: abstrakte Funktion zu f

  $$\alpha^\sharp(\texttt{if\_then\_else\_})(z,x,y) = \left\{ \begin{array}{ll} \bot & , \text{falls } z = \bot \\ x \sqcup y & , \text{falls } z = \top \end{array} \right\} = z \sqcap (x \sqcup y)$$
- andere Grundfunktionen sind strikt: $\alpha^\sharp(f)(x_0, ..., x_n) = \bigsqcap i : 0 \leq i \leq n : x_i$
- eine Funktion f ist strikt in Argument i, falls

  $x_i = \bot \wedge (\forall j : j \neq i : x_j = \top) \implies \alpha^\sharp(f)(x_0, ..., x_n) = \bot$
- Funktionale stetig und Verband über $\{\bot, \top\}^n$ endlich: rekursive Funktion mit n Argumenten kann (mit maximal n Approximationen á 2^n abstrakten Auswertungen) analysiert werden

Bedeutung der Striktheitsanalyse

Compiler-Optimierung für nicht-strikte Sprachen:
call-by-value ist effizienter, falls Wert gebraucht wird

1. Analyse: bestimme für alle Funktionen die strikten Argumente
2. Codeerzeugung:
   - Aufruf: Auswertung der strikten Argumente vor Aufruf
   - Verwendung: beim Zugriff kein Test, ob Auswertung nötig

Beispiel

- konkrete Funktion: `f (x,y) = if p x then c else f (g (x,y), y)`
- abstrakte Funktion: $f^\sharp(x,y) = x \sqcap (\top \sqcup f^\sharp(x \sqcap y, y))$
- Fixpunktiteration

	$(\bot,\bot)$	$(\bot,\top)$	$(\top,\bot)$	$(\top,\top)$	
0	$\bot$	$\bot$	$\bot$	$\bot$	
1	$\bot$	$\bot$	$\top$	$\top$	
2	$\bot$	$\bot$	$\top$	$\top$	Fixpunkt erreicht

- Interpretation des Ergebnisses
  - $f(\bot,\top) = \bot$: `f` ist strikt im ersten Argument
  - $f(\top,\bot) = \top$: man kann nicht sagen, ob `f` im zweiten Argument strikt ist

Curry-Howard-Isomorphismus

Logik zwischen Programmen und ihren Typen

"Formulae as types and proofs as terms"

Bsp.: Modus-Ponens-Regel	Typregel der Applikation
$\frac{P \Rightarrow Q \mid P}{Q}$	$\frac{f :: \alpha \to \beta \mid x :: \alpha}{f\ x\ :: \beta}$

Gedankliche Verbindung:

- x ist ein Beweis für α
- f konstruiert einen Beweis für β unter Voraussetzung α
- $f\,x$ ist ein Beweis für β

Curry-Howard-Isomorphismus, logische Operatoren

Formel	Typ	
false	`Void`	leerer Typ
$a \wedge b$	`(a,b)`	Produkttyp
$a \vee b$	`Either a b`	Summentyp
$a \Rightarrow b$	`a->b`	Funktionstyp
true	`()`	Unit-Typ

Bsp.: $a \wedge b \Rightarrow a \vee b$

```
type Theorem = forall a b. (a,b) -> Either a b
proof :: Theorem
proof (x,y) = Left x
```

Typinferenz als abstrakte Interpretation

- Typinformation der Applikation

$$\frac{f :: \alpha \to \beta \mid x :: \alpha'}{f\ x\ :: \beta'}$$

  wobei $\beta' = \sigma(\beta)$, $\sigma = \text{mgu}(\alpha, \alpha')$, most general unifier (Substitution)

  Bsp.: $f :: \gamma \to \gamma$, $x :: (Int \to Int)$, $\sigma = [\gamma := Int \to Int]$

- Spezialisierung: $\alpha \sqcup \alpha' = \sigma(\alpha) = \sigma(\alpha')$
- Typvariable entspricht $\bot$ (nicht-definierter Typ)
- Typfehler entspricht $\top$

Hindley-Milner Typsystem (1)

Variablen

$$A \uplus \{x :: \tau\} \vdash x :: \tau$$

Bedingungen

$$\frac{A \vdash c :: \texttt{Bool} \mid A \vdash e_1 :: \tau \mid A \vdash e_2 :: \tau}{A \vdash (\texttt{if } c \texttt{ then } e_1 \texttt{ else } e_2) :: \tau}$$

Abstraktionen

$$\frac{A \uplus (x :: \sigma) \vdash e :: \tau}{A \vdash (\backslash x \texttt{ -> } e) :: (\sigma \to \tau)}$$

Applikationen

$$\frac{A \vdash f :: (\sigma \to \tau) \mid A \vdash x :: \sigma}{A \vdash (f\ x) :: \tau}$$

Hindley-Milner Typsystem (2)

`let`-Ausdrücke
$$\frac{A \vdash y :: \sigma \mid A \uplus \{x :: \sigma\} \vdash e :: \tau}{A \vdash (\texttt{let}\ x\ \texttt{=}\ y\ \texttt{in}\ e) :: \tau}$$

Fixpunkt
$$\frac{A \uplus \{x :: \tau\} \vdash e :: \tau}{A \vdash (\texttt{Y}\ (\backslash x\ \texttt{->}\ e)) :: \tau}$$

Generalisierung
$$\frac{A \vdash e :: \tau}{A \vdash e :: \forall \alpha.\tau} \ (\alpha \text{ nicht frei in } A)$$

Spezialisierung
$$\frac{A \vdash e :: \forall \alpha.\tau}{A \vdash e :: (\tau[\alpha := \sigma])}$$

Bsp.: Typinferenz der map-Funktion (1)

```
map' = Y h where
 h = \ map f xs
       -> if null xs
             then []
             else f (head xs) : map f (tail xs)
```

$$\frac{\texttt{xs} :: \tau_1 \mid \texttt{null} :: [\tau_2] \to \texttt{Bool}}{\texttt{null xs} :: \texttt{Bool}} \{\tau_1 = [\tau_2]\}$$

$$\frac{\texttt{xs} :: [\tau_2] \mid \texttt{head} :: [\tau_3] \to \tau_3}{\texttt{head xs} :: \tau_2} \{\tau_3 = \tau_2\}$$

$$\frac{\texttt{f} :: \tau_4 \mid \texttt{head xs} :: \tau_2}{\texttt{f (head xs)} :: \tau_5} \{\tau_4 = \tau_2 \to \tau_5\}$$

Typinferenz der `map`-Funktion (2)

$$\frac{\texttt{xs} :: [\tau_2] \mid \texttt{tail} :: [\tau_6] \to [\tau_6]}{\texttt{tail xs} :: [\tau_2]} \{\tau_6 = \tau_2\}$$

$$\frac{\texttt{map} :: \tau_7 \mid \texttt{f} :: \tau_2 \to \tau_5}{\texttt{map f} :: \tau_8} \{\tau_7 = (\tau_2 \to \tau_5) \to \tau_8\}$$

$$\frac{\texttt{map f} :: \tau_8 \mid \texttt{tail xs} :: [\tau_2]}{\texttt{map f (tail xs)} :: \tau_9} \{\tau_8 = [\tau_2] \to \tau_9\}$$

$$\frac{\texttt{(:)} :: \tau_{10} \to [\tau_{10}] \to [\tau_{10}] \mid \texttt{f (head xs)} :: \tau_5}{\texttt{(:) (f (head xs))} :: [\tau_5] \to [\tau_5]} \{\tau_{10} = \tau_5\}$$

$$\frac{\texttt{(:)(f (head xs))} :: [\tau_5] \to [\tau_5] \mid \texttt{map f (tail xs)} :: \tau_9}{\texttt{f (head xs) : map f (tail xs)} :: [\tau_5]} \{\tau_9 = [\tau_5]\}$$

Typinferenz der map-Funktion (3)

$$\frac{\begin{array}{l}\texttt{null xs} :: \texttt{Bool} \\ \mid \texttt{[]} :: [\tau_{11}] \\ \mid \texttt{f (head xs) : map f (tail xs)} :: [\tau_5]\end{array}}{\begin{array}{l}\texttt{if null xs} \\ \quad \texttt{then []} \\ \quad \texttt{else f (head xs) : map f (tail xs)} \\ :: [\tau_5]\end{array}} \{\tau_{11} = \tau_5\}$$

Typinferenz der `map`-Funktion (4)

Abkürzung: `h = \ map ->` R', wobei R' `= \ f xs ->` R

$$\frac{R :: [\tau_5] \mid \texttt{xs} :: [\tau_2]}{(\backslash\,\texttt{xs ->}\, R) :: [\tau_2] \to [\tau_5]}$$

$$\frac{\texttt{f} :: \tau_2 \to \tau_5 \mid (\backslash\,\texttt{xs ->}\, R) :: [\tau_2] \to [\tau_5]}{(\backslash\,\texttt{f xs ->}\, R) :: \sigma} \{\sigma = (\tau_2 \to \tau_5) \to [\tau_2] \to [\tau_5]\}$$

$$\frac{\texttt{map} :: \sigma \mid R' :: \sigma}{(\texttt{Y}\ (\backslash\,\texttt{map ->}\ R')) :: \sigma}$$

Ergebnis: $\boxed{\texttt{map} :: \forall\,\alpha\,\beta\,.\,(\alpha \to \beta) \to [\alpha] \to [\beta]}$

Generische Typvariablen

- Nicht typisierbar im Hindley-Milner System:
  `(\f -> (f 3, f True)) id`
  - Grund: `f` :: $(\alpha \rightarrow \alpha)$, aber α kann nicht gleichzeitig den Typ `Integer` und `Bool` haben
  - *Monomorphismus-Restriktion* von λ-Abstraktionen
- Typisierbar: `let f = id in (f 3, f True)`
  - Grund: `f` :: $\underline{\forall\alpha.}(\alpha \rightarrow \alpha)$
  - α ist eine *generische* Typvariable
- Nicht typisierbar: `(\g -> let f = g in (f 3, f True)) id`
  - Grund: $\forall\alpha$ wird aufgelöst durch λ-Bindung von `g`

Milners $\mathcal{W}$-Algorithmus [Milner, 1978]

- Zuweisung eines eindeutigen, allgemeinsten Typs zu jedem korrekten syntaktischen (ML-)Ausdruck oder Meldung eines Typfehlers.
- Der $\mathcal{W}$-Algorithmus terminiert immer; Typinferenz ist berechenbar (im Ggs. zu der für best. Haskell-Erweiterungen).
- Nur Berechnung von *shallow types*, d.h., keine (All-)Quantoren innerhalb von Typausdrücken (Monomorphismus-Restriktion).
- Lösung von Typgleichungen durch Unifikation mit dem Robinson-Algorithmus [Robinson, 1965]; berechnen des allgemeinsten Unifikators (Substitution).
- Unlösbarkeit von $\alpha = \alpha \rightarrow \alpha$ mit endlichen Termen Grund für das Scheitern der Typinferenz von $(x\ x)$ in $\mathcal{W}$ (die Regeln des Hindley-Milner Systems würden das zulassen).

Unifikation

Kompakte Sicht: Kalkül von Martelli u. Montanari [1982]

1. Dekomposition: $\frac{\Gamma \uplus \{\varphi(\tau_1,...,\tau_n)=\varphi(\tau'_1,...,\tau'_n)\}}{\Gamma \cup \{\tau_1=\tau'_1,...,\tau_n=\tau'_n\}}$ (auch für $n=0$)

2. Variablen-Elimination: $\frac{\Gamma \uplus \{\nu=\tau\}}{\Gamma[\nu:=\tau]}$, falls ν in τ nicht vorkommt (Occurs-Check), dabei Aktualisierung der Substitution durch: $\sigma' = \sigma[\nu := \tau] \cup \{\nu := \tau\}$

3. Umordnung: $\frac{\Gamma \uplus \{\tau=\nu\}}{\Gamma \cup \{\nu=\tau\}}$, falls τ keine Variable ist

4. Elimination trivialer Gleichungen: $\frac{\Gamma \uplus \{\nu=\nu\}}{\Gamma}$

Anreicherung von Typen mit Größeninformation

- Beispiel: zwei Listen sollen nur dann den gleichen Typ haben, wenn sie auch die gleiche Länge haben
- Zweck: bestimmte Laufzeitfehler sollen als Typfehler erkannt werden
- nur für bestimmte Probleme sinnvoll
- erster Ansatz (nicht optimal, monomorph)

```
data N   = N
data C t = C Int t  -- t: Restliste
```

- erwünschtes Element: `C 3 (C 1 (C 2 N)) :: C (C (C N))`
- unerwünscht: `(C 1 "Haha") :: C [Char]`

Typfehler statt Laufzeitfehler

```
hd :: C t -> Int
hd (C x _) = x

tl :: C t -> t
tl (C _ y) = y
```

Beispiele:

- `tl (tl (C 1 (C 2 (C 3 N))))` $\rightsquigarrow$ `C 3 N`
- `hd (tl (C 3 N))` $\rightsquigarrow$

```
Couldn't match 'C t' against 'N'
    Expected type: C t
    Inferred type: N
```

Überladung von Funktionen

```
class List a where
   len   ::                          a       -> Int
   mapL  :: (Int -> Int)          -> a       -> a
   zipW  :: (Int->Int->Int)       -> a -> a -> a
   foldL :: (b -> Int -> b) -> b -> a       -> b
instance List N where
   len       N   = 0
   mapL  f   N   = N
   zipW  f   N N = N
   foldL f e N   = e
instance List t => List (C t) where
   len       (C _ s)          = 1 + len s
   mapL  f   (C x s)          = C (f x)   (mapL f s)
   zipW  f   (C x s) (C y t)  = C (f x y) (zipW f s t)
   foldL f e (C x s)          = foldL f (f e x) s
```

Sortieren

```
class List a where
   ...
   insert :: Int -> a -> C a
   sort   ::         a ->   a  -- Beweis, dass sort Länge erhält!
instance List N where
   ...
   insert  x N   = C x N
   sort      N   = N
instance List t => List (C t) where
   ...
   insert x (C y s) | x<=y = C x (C y s)
                    | x> y = C y (insert x s)
   sort (C x s) = insert x (sort s)
```

Verbesserung

- Verlust an Polymorphie
  - Lösung 1: Verzicht auf Typgleichheit der Elemente, unakzeptabel
  - Lösung 2: zusätzlicher Typparameter (gewählte Alternative): `C t a`
- unerwünschte Elemente, z.B.: `(C 1 "Haha") :: C [Char]`
  - Typklassenkontext `List t` in der Definition des Datentyps `C t a` und nur `N` und `C t` werden als Instanzen von `List t` deklariert.
- Datentypdefinition

```
data              N   a = N
data List t => C t a = C { hd :: a, tl :: t a }
```

- ghc-Flags `-fglasgow-exts -fallow-undecidable-instances` verwenden

Zweiter Ansatz (1)

```
data             N   a = N
data List t => C t a = C { hd :: a, tl :: t a }

class List t where
   len    :: t a -> Int
   zipW   :: (a->b->c) -> t a -> t b -> t c
   ...
instance List N where
   len N = 0
   zipW f N N = N
   ...
instance List t => List (C t) where
   len (C _ s) = 1 + len s
   zipW  f (C x s) (C y t) = C (f x y) (zipW f s t)
   ...
```

Zweiter Ansatz (2)

Da nun Polymorphie möglich ist, wird statt `mapL` die bekannte Funktion `fmap` in entsprechenden Instanzdefinitionen von `Functor` definiert:

```
data N a   = N
data List t => C t a = C { hd :: a, tl :: t a }

instance Functor N where
   fmap f N = N

instance (List t, Functor t) => Functor (C t) where
   fmap f (C x xs) = C (f x) (fmap f xs)
```

Zweiter Ansatz (3)

Bei der polymorphen Variante: nur die Typen ändern sich, die Implementierung des ersten Ansatzes bleibt.

```
data N a   = N
data List t => C t a = C { hd :: a, tl :: t a }

class List t where
   len    :: t a -> Int
   zipW   :: (a->b->c) -> t a -> t b -> t c
   foldL  :: (a->b->a) -> a -> t b -> a
   insert :: Ord a => a -> t a -> C t a
   sort   :: Ord a => t a -> t a
```

Typen mit Größeninformation, Fazit

- Möglichkeiten
  - Beweis von Programmeigenschaften im Typ
  - Kontrolle über Speicherbedarf, Rechenzeit
- Praktische Probleme
  - Einschränkung der Programmiermöglichkeiten
    - Insertion Sort: möglich, aber buchhalterischer Umgang mit Anzahlen
    - Quicksort: nicht möglich, weil Anzahlen erst zur Laufzeit feststehen
  - Keine Standardbibliotheken verwendbar
- Theoretische Grenzen
  bei unbekannten Anzahlen
  - Addition möglich ("=" entscheidbar in der Presburger Arithmetik)
  - Multiplikation nicht möglich ("=" unentscheidbar in der Zahlentheorie)