CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Prof. Bernd Finkbeiner, Ph.D.
Jana Hofmann, M.Sc.
Reactive Systems Group

# Programmierung 1 (WS 2020/21)
# Aufgaben für die Übungsgruppe C (Lösungsvorschläge)

**Hinweis:** Diese Aufgaben wurden von den Tutoren für die Übungsgruppe erstellt. Sie sind für die Klausur weder relevant noch irrelevant. 🤔 markiert potentiell schwerere Aufgaben.

## Semantische Analyse

**Aufgabe TC.1** *(Konstante Bindungen)*
Welche Bezeichner sind in den folgenden Programmen frei? Welche Programme sind geschlossen und, falls ja, zu was werten `res1` und `res2` aus?

(a)
```
val x = 5
val a = 7
fun test1 () = a + x
fun test2 (a : int) = a + x
val res1 = (test1 (), test2 (2))
val x = 4
val res2 = (test1 (), test2 (2))
```

(b)
```
fun test () : int =
let
val x = 5
in
3
end
val res1 = test ()
val res2 = x
```

(c)
```
val a = 3
fun test (x:int) = if x > 0 then a*x else 0
val res1 =  test(5)
val a = ~3
val res2 = test(2)
```

(d)
```
fun test1 (a:int) = if a > 0 then a*a else ~(a*a)
fun test1 (y:int) = 4*y+5
val res1 =  test1(3y)
val x = ~3
val res2 = test2(2x)
```

*Lösungsvorschlag TC.1*

(a) `res1 = (12, 7)`
`res2 = (12, 7)`

(b) Fehlerhaft, weil `x` ungebunden ist.
ElaborationError: Unbound value identifier x.

(c) `res1 = (15)`
`res2 = (6)`

(d) Fehlerhaft, weil `y` ungebunden ist.
ElaborationError: Unbound value identifier y.

**Aufgabe TC.2** *(Tripeldarstellung)*
Geben Sie die Tripeldarstellungen der folgenden Prozeduren an:

(a)
```
fun double (x : int) : int = if x > 0 then 2 + double (x - 1) else 0
```

(b)
```
fun fak (n : int) : int = if n < 1 then 1
else n * fak (n - 1)
```

(c)
```
(fn (x:int) => fn (b:bool) => if b then x else 7) (2 + 3)
```

*Lösungsvorschlag TC.2*

(a) (fun $double\ x = \text{if } x > 0 \text{ then } 2 + double\ (x-1) \text{ else } 0, int \to int, [])$

(b) (fun $fak\ n = \text{if } n < 1 \text{ then } 1 \text{ else } n * fak\ (n-1), int \to int, [])$

(c) (fn $b \Rightarrow \text{if } b \text{ then } x \text{ else } 7, bool \to int, [x := 5])$

**Aufgabe TC.3** *(gnulletsradlepirT)*
Geben Sie Prozeduren und ggf. zusätzliche Wertdeklarationen an, die zu folgenden Tripeldarstellungen auswerten:

(a) (fun $q\ n = \text{if } n = 0 \text{ then } 0 \text{ else } n \text{ mod } 2 + q\ (n \text{ div } 2), int \to int, [])$

(b) (fun $add\ (x, y) = \text{if } x = n \text{ then } y \text{ else } add\ (x-1, y+1), int * int \to int, [n := 0])$

(c) (fn $z \Rightarrow 10 * f\ (z \text{ div } 10) + f\ (z \text{ mod } 10), int \to int, [f := (\text{fn } n \Rightarrow 2 * n, int \to int, [])])$

*Lösungsvorschlag TC.3*

(a)
```
fun q (n : int) : int = if n = 0 then 0 else n mod 2 + q(n div 2)
```

(b)
```
val n = 0
fun add (x : int, y : int) : int =
  if x <= n then y
  else then add(x - 1, y + 1)
```

(c)
```
(fn (f:int->int) => fn (z:int) => 10 * f(z div 10) + f(z mod 10)) (fn (n:int) => 2 * n)
```

**Aufgabe TC.4** *(Ableitungsbäume)*
Prüfen Sie die semantische Zulässigkeit der folgenden Phrasen durch die Erstellung eines Ableitungsbaums.

Nehmen Sie an, dass `x` und `y` jeweils vom Typ `int` sind, `g` vom Typ `int * int -> bool` und `q` vom Typ `int -> bool` sind.

Falls die Phrase *nicht* semantisch zulässig ist, markieren Sie die Stelle im Baum bei der die Typprüfung fehlschlägt. Markieren Sie zusätzlich die Typen (unterhalb des Fehlschlages) damit diese nicht inferiert werden.

(a)
```
(3.0 + 4.0) * 3
```

(b)
```
(2, g)
```

(c)

```
if x = y then (2, g (2, 3)) else (2 * 3, q 2.0)
```

(d)

```
(#2(4, q)) 42
```

*Lösungsvorschlag TC.4*

(a)
$$\dfrac{\dfrac{\overline{3.0 : \mathit{real}} \quad \overline{+ : \mathit{real} * \mathit{real} \to \mathit{real}} \quad \overline{4.0 : \mathit{real}}}{3.0 + 4.0 : \mathit{real}} \quad \overline{* : \mathit{real} * \mathit{real} \to \mathit{real}} \quad \overline{3 : \cancel{\mathrm{int}} \text{ Hier schlägt die Typprüfung fehl!}}}{(3.0 + 4.0) * 3 : \cancel{\mathrm{real}}}$$

(b)
$$\dfrac{\overline{2 : \mathit{int}} \quad \overline{g : \mathit{int} * \mathit{int} \to \mathit{bool}}}{(2, g) : \mathit{int} * (\mathit{int} * \mathit{int} \to \mathit{bool})}$$

(c)
$$\dfrac{\dfrac{\overline{x : \mathit{int}} \quad \overline{= : \mathit{int} * \mathit{int} \to \mathit{bool}} \quad \overline{y : \mathit{int}}}{x = y : \mathit{bool}} \quad \dfrac{\overline{2 : \mathit{int}} \quad \dfrac{\overline{g : \mathit{int} * \mathit{int} \to \mathit{bool}} \quad \dfrac{\overline{2 : \mathit{int}} \quad \overline{3 : \mathit{int}}}{(2,3) : \mathit{int} * \mathit{int}}}{g\,(2,3) : \mathit{bool}}}{(2, g\,(2,3)) : \mathit{int} * \mathit{bool}} \quad \dfrac{\dfrac{\overline{2 : \mathit{int}} \quad \overline{* : \mathit{int} * \mathit{int} \to \mathit{int}} \quad \overline{3 : \mathit{int}}}{2 * 3 : \mathit{int}} \quad \dfrac{\overline{q : \mathit{int} \to \mathit{bool}} \quad \overline{2.0 : \cancel{\mathrm{real}} Hierschl\ddot{a}gtdieTyppr\ddot{u}fungfehl!}}{q\,2.0 : \cancel{\mathrm{bool}}}}{(2 * 3, q\,2.0) : \cancel{\mathrm{int} * \mathrm{bool}}}}{\text{if } x = y \text{ then } (2, g\,(2,3)) \text{ else } (2 * 3, q\,2) : \cancel{\mathrm{int} * \mathrm{bool}}}$$

(d)
$$\dfrac{\dfrac{\dfrac{\overline{4 : \mathit{int}} \quad \overline{q : \mathit{int} \to \mathit{bool}}}{(4, q) : \mathit{int} * (\mathit{int} \to \mathit{bool})}}{\#2(4, q) : \mathit{int} \to \mathit{bool}} \quad \overline{42 : \mathit{int}}}{\#2(4, q)\ 42 : \mathit{bool}}$$

## Abstraktionen und kaskadierte Prozeduren

**Aufgabe TC.5** *(Abstraktion vs. Prozedur)*
Welche der folgenden Deklarationen sind gültig in SML.

(a) `val a = fn x => 1`

(b) `val b = fun f x = 1`

(c) `val c = let val x = fn q => 2 in x end`

(d) `val d = let fun x q = 2 in x end`

(e) `val f = (fn g => g 5) (fn x => x + 4)`

(f) `val g = (fn x = x) 5`

(g) `val i = (fun x => x) 42`

*Lösungsvorschlag TC.5*
a), c), d), e)

**Aufgabe TC.6** *(Höherstufigkeit vs. Kaskadierung)*

(a) Erklären Sie den Unterschied zwischen kaskadierten Prozeduren und höherstufigen Prozeduren.

(b) Schreiben Sie eine höherstufige Prozedur, die nicht kaskadiert ist.

(c) Schreiben Sie eine kaskadierte Prozedur, die nicht höherstufig ist.

*Lösungsvorschlag TC.6*

(a) Kaskadierte Prozeduren sind Prozeduren, die Prozeduren als Ergebnisse liefern. Höherstufige Prozeduren nehmen Prozeduren als Argumente.

(b)

```
fun f (x : int -> int) : int = x 3
```

(c)

```
fun f x y = x + y
```

Was äquivalent ist zu:

```
val f = fn x => fn y => x + y
```

**Aufgabe TC.7** *(No Sugar for Dieter)*
Ihr Freund Dieter Schlau hat ein Problem, leider ist auf seinem Laptop die u Taste kaputt gegangen. Trotzdem würde er gerne noch die Aufgaben auf dem letzten Prog Blatt bearbeiten :

Deklarieren sie folgende Prozeduren :

(a) `q : int -> int`, die das Quadrat der Zahl x berechnet.

(b) `add : int * int -> int` , die die Summe zweier Zahlen aus einem Tupel berechnet.

(c) `add' : int -> int -> int`, die auch die Summe zweier Zahlen berechnet.

Können Sie Dieter helfen ? Verwenden sie dabei insbesondere nicht das Schlüsselwort `fun`.

*Lösungsvorschlag TC.7*

(a) `val q = fn x => x * x`

(b) `val add = fn (x,y) => x + y`

(c) `val add' = fn x => fn y => x + y`

**Aufgabe TC.8** *(Bedingte Summe)*
Schreiben sie eine Prozedur `psum: (int -> bool) -> int -> int -> int`, die alle Zahlen von $m$ bis $n$ aufsummiert, welche ein Prädikat erfüllen (d.h. wenn die übergebene Prozedur zu wahr auswertet). Sie können annehmen, dass Ihre Prozedur nur mit Argumenten $n \geq m \geq 0$ aufgerufen wird.

*Lösungsvorschlag TC.8*

```
fun psum p m n = if m > n then 0 else if p m then (psum p (m+1) n + m) else (psum p (m+1) n)
```

**Aufgabe TC.9** *(psum in action)*
Schreiben sie mithilfe von `psum` eine Prozedur, die von 1 bis $n$

(a) alle geraden/ungeraden Zahlen aufsummiert, je nach dem, ob das erste Argument `true` oder `false` ist: `evensum: bool -> int -> int`

(b) alle Primzahlen aufsummiert. Sie dürfen annehmen, dass eine Prozedur `isPrime: (int -> bool)` bereits deklariert ist.

*Lösungsvorschlag TC.9*

(a)
```
fun evensum p = psum (fn x => (x mod 2 = 0) = p) 1
```

(b)
```
val primesum = psum isPrime 1
```

**Aufgabe TC.10** *(Curry, das: eintopfartiges indisches Gericht, oder indische Gewürzmischung)*
Betrachten Sie folgendes Programm:

```
fun add (x : int) (y : int) = x + y
val addme = add 3
val i = addme 5
val j = addme 4
val addmetoo = add 7
val k = addmetoo 5
```

Woran werden die Bezeichner `add`, `i`, `j` und `k` gebunden? Welchen Typ haben die an die Bezeichner `addme` und `addmetoo` gebundenen Werte? Wofür sind sie nützlich? Können Sie ihren Wert angeben?

*Hinweis: Schreiben Sie zunächst* `add` *als Abstraktionenkette um!*

*Lösungsvorschlag TC.10*
`add` umgeschrieben: `val add = fn x:int => fn y:int => x+y : int`.

Bindungen:

- $add := (\text{fun } add\ x\ y = x + y, int \to int \to int, [])$

  oder: $add := (\text{fn } x => \text{fn } y \Rightarrow x + y, int \to int \to int, [])$
- $i := 8$
- $j := 7$
- $k := 12$

Die Typen der an `addme` und `addmetoo` gebundenen Werte sind `int -> int`. Sie sind also Prozeduren, die jeweils 3 (bei `addme`) oder 7 (bei `addmetoo`) zum übergebenen Wert addieren. Ihre Tripeldarstellung (also ihr Wert) ist:

- $addme := (\text{fn } y \Rightarrow x + y, int \to int, [x := 3])$
- $addmetoo := (\text{fn } y \Rightarrow x + y, int \to int, [x := 7])$

## Bestimmte und unbestimmte Iteration

**Aufgabe TC.11** *(Add + Mul)*
Schreiben Sie folgende Prozeduren mit Hilfe von `iter`:

(a) `add: int -> int -> int`, welche die Addition zweier Zahlen durch wiederholtes Inkrementieren (+1) berechnet.

(b) `mul: int -> int -> int`, welche die Multiplikation zweier Zahlen mit Hilfe von `add` berechnet.

(c) `pot: int -> int -> int`, welche die Potenzierung zweier Zahlen mit Hilfe von `mul` berechnet.

*Lösungsvorschlag TC.11*

(a)
```
fun add n m = iter n m (fn x => x + 1)
```

(b)
```
fun mul n m = iter n 0 (add m)
```

(c)
```
fun pot n m = iter m 1 (mul n)
```

**Aufgabe TC.12** *(First Second ... n-First)*
Schreiben Sie mit Hilfe von `first` und `iter` eine Prozedur `nth_first : int -> (int -> bool) -> int`, die ein Prädikat $p$ erhält, und die n.-kleinste nichtnegative Zahl zurückgibt, die $p$ erfüllt.

*Lösungsvorschlag TC.12*

```
fun nth_first n f = iter n ~1 (fn x => first (x + 1) f)
```

**Aufgabe TC.13** *(Limited First)*
Schreiben Sie eine Prozedur `limitedFirst : (int -> bool) -> int -> int`, die gegeben eine Zahl $n \geq 0$ die kleinste natürliche Zahl $m \leq n$ liefert, für die die Prozedur `p` *true* liefert und divergiert, falls keine solche Zahl existiert.

**Hinweis:** Eine endrekursive Hilfsprozedur könnte nützlich sein.

*Lösungsvorschlag TC.13*

```
fun limitedFirst' p n m = if m > n then limitedFirst' p n m
        else if p m then m
        else limitedFirst' p n (m + 1)

fun limitedFirst p n = limitedFirst' p n 0
```

**Aufgabe TC.14** *(Kleinstes gemeinsames Vielfaches)*
Bestimmen Sie mit Hilfe von `first` das kleinste gemeinsame Vielfache zweier Zahlen größer 1.

*Lösungsvorschlag TC.14*

```
fun kgv a b = first 2 (fn s => if s mod a = 0 then s mod b = 0 else false)
```

## Typinferenz und Polymorphie

**Aufgabe TC.15** *(Typinstanzen)*
Zeichen Sie Pfeile zwischen den Typen. Ein oranger Pfeil bedeutet „ist Instanz von“ und ein blauer Pfeil „alle Instanzen sind auch Instanzen von“.

(a) $(\texttt{int} \to \texttt{real}) \to (\texttt{int} * \texttt{int}) \to \texttt{real}$

(b) $\forall \alpha, \beta.\ (\alpha \to \alpha) \to (\alpha * \beta) \to \alpha$

(c) $\forall \alpha, \beta.\ ('\alpha \to \beta) \to ('\alpha * \texttt{int}) \to \beta$

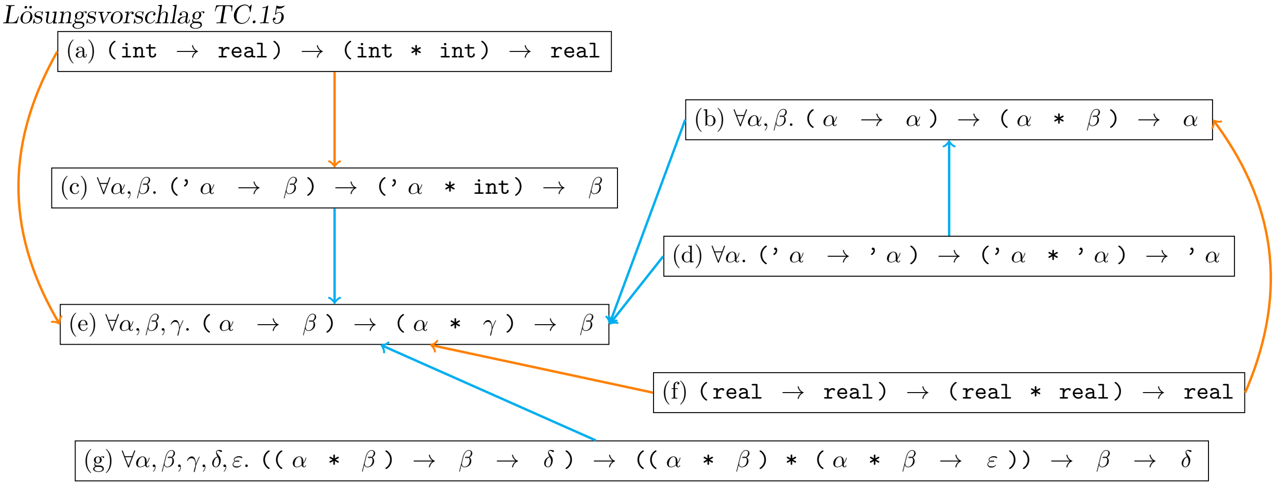
(d) $\forall \alpha.\ ('\alpha \to '\alpha) \to ('\alpha * '\alpha) \to '\alpha$

(e) $\forall \alpha, \beta, \gamma.\ (\alpha \to \beta) \to (\alpha * \gamma) \to \beta$

(f) $(\texttt{real} \to \texttt{real}) \to (\texttt{real} * \texttt{real}) \to \texttt{real}$

(g) $\forall \alpha, \beta, \gamma, \delta, \varepsilon.\ ((\alpha * \beta) \to \beta \to \delta) \to ((\alpha * \beta) * (\alpha * \beta \to \varepsilon)) \to \beta \to \delta$

*Lösungsvorschlag TC.15*




**Aufgabe TC.16** *(Typen, hurra!)*
Bestimmen Sie das Typschema folgender Prozeduren *ohne sie in einen SML-Interpreter einzugeben.*

*Anmerkung:* Nicht wohlgetypte Prozeduren haben keinen Typen.

(a) `fun f x = x`

(b) `fun f x y = x`

(c) `fun f a b = a b b`

(d) `fun f a b c = a b c`

(e) `fun f a b c = a (b c)`

(f) `fun f g s n = if n = 0 then s else f g (g s) (n-1)`

(g) `fun y a (b : real) c = if c then b else 3`

(h) `fun f (a, b, c) d = if c then (a + 4) = b else d (true)`

*Lösungsvorschlag TC.16*

(a) $\forall\alpha.\ \alpha \to \alpha$

(b) $\forall\alpha,\beta.\ \alpha \to \beta \to \alpha$

(c) $\forall\alpha\,\beta.\ (\alpha \to \alpha \to \beta) \to \alpha \to \beta$

(d) $\forall\alpha\,\beta\,\gamma.\ (\alpha \to \beta \to \gamma) \to \alpha \to \beta \to \gamma$

(e) $\forall\alpha\,\beta\,\gamma.\ (\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta$

(f) $\forall\alpha.\ (\alpha \to \alpha) \to \alpha \to \texttt{int} \to \alpha$

(g) Nicht wohlgetypt (`int` und `real` als Rückgabetypen)

(h) $\texttt{int} * \texttt{int} * \texttt{bool} \to (\texttt{bool} \to \texttt{bool}) \to \texttt{bool}$

**Aufgabe TC.17** *(Das griechische Alphabet in SML)*
Geben sie Prozeduren mit folgenden polymorphen Typschemata an, ohne explizite Typangaben wie `(x:int)` zu benutzen:

(a) $\forall\alpha,\beta.\ \alpha \to \beta \to \alpha * \beta$

(b) $\forall\alpha,\beta,\gamma.\ \alpha \to \beta \to \gamma \to \alpha$

(c) $\forall\alpha,\beta,\gamma.\ (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$

(d) $\forall'\alpha,\beta.'\ \alpha \to ('\ \alpha \to \beta) \to \beta \to \beta$

(e) $\forall\alpha,\beta,\gamma.((\alpha \to \gamma) \to \gamma) \to (\alpha \to \beta) \to (\beta \to \gamma) \to \gamma$

(f) $\forall\alpha,\beta,\gamma,\delta.((\alpha \to \beta \to \gamma) \to (\alpha \to \gamma) \to \delta) \to (\alpha \to \gamma) \to \delta$

*Lösungsvorschlag TC.17*

(a) `fun f a b = (a, b)`

(b) `fun g a b c = a`

(c) `fun S f g a = f a (g a)`

(d) `fun Eq a b c = if a = a then b a else c`

(e) `fun app c f g = c (fn a => g (f a))`

(f) `fun callcc f g = f (fn a => (fn _ => g a)) g`

**Aufgabe TC.18** *(Typen mit Abstraktionen)*
Geben Sie geschlossene Abstraktionen an, welche folgenden Typen haben:

(a) `(int -> bool -> real) -> int -> bool -> real`

(b) `(int -> bool) -> (bool -> real) -> int -> real`

Die Abstraktionen sollen nur mit Prozeduranwendungen, Tupeln und Bezeichnern gebildet werden. Konstanten und Operatoren sollen nicht verwendet werden.

*Lösungsvorschlag TC.18*

(a) `fn f:int->bool->real => fn x:int => fn b:bool => f x b`

(b) `fn f:int->bool => fn g:bool->real => fn x:int => g(f x)`

**Aufgabe TC.19** *(Polymorphes Tripel)*
Schreiben Sie eine Prozedur `sharp3` : $\alpha * \beta * \gamma \to \gamma$, die die Komponente der 3. Position eines Tripels zurückgibt. Verwenden Sie dabei möglichst nur eine Argumentvariable und keine Projektion. Machen Sie sich klar, warum die Prozedur genau so getypt ist und warum hier nur $\gamma$ als Rückgabetyp infrage kommt.

*Lösungsvorschlag TC.19*
Mit kartesischem Argumentmuster

```
fun sharp3 (a, b, c) = c
```

oder mit Wildcard (siehe Kapitel 3.10)

```
fun sharp3 (_, _, c) = c
```

Da auf explizite Typangaben verzichtet wird, wird die Prozedur polymorph getypt. Insbesondere können die Typen von a, b und c verschieden sein. Deshalb wird der Typ von a mit $\alpha$, der Typ von b mit $\beta$ und der Typ von c mit $\gamma$ quantifiziert. Es wird immer die dritte Komponente des Tupels zurückgegeben. Deren Typ wurde mit $\gamma$ quantifiziert. Daraus ergibt sich das Typschema `sharp3` : $\alpha * \beta * \gamma \to \gamma$.

**Aufgabe TC.20** *(Typen II: Die Rückkehr des Deltas)*
Bestimmen Sie das Typschema folgender Prozeduren *ohne sie in einen SML-Interpreter einzugeben.*

*Anmerkung:* Nicht wohlgetypte Prozeduren haben keinen Typen.

(a) `fun fortytwo f p s = if p s then fortytwo f p (f s) else s`

(b) `fun f x y z = x z (y z)`

(c) `fun f a (b, c) d = a (b (c d) c a)`

(d) `fun f x = x f`

(e) `fun f a (b, c) d (e, f) = a (f e) ((a b) (c d))`

(f) `fun f a (b, c) = (a b, a c, a b c)`

*Lösungsvorschlag TC.20*

(a) $\forall \alpha.\ (\alpha \to \alpha) \to (\alpha \to \texttt{bool}) \to \alpha \to \alpha$

(b) $\forall \alpha, \beta, \gamma.\ (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$

(c) $\forall \alpha \beta \gamma \delta.\ (\alpha \to \beta) \to (\gamma \to (\delta \to \gamma) \to (\alpha \to \beta) \to \alpha) * (\delta \to \gamma) \to \delta \to \beta$

(d) Nicht wohlgetypt: `x` muss den Typ von `f` als Rückgabewert beinhalten, und `f` den Typ von `x` als Argument - das wird schwierig.

(e) $\forall \alpha \beta \gamma \delta.\ (\alpha \to \beta \to \beta) \to \alpha * (\gamma \to \beta) \to \gamma \to \delta * (\delta \to \alpha) \to \beta$

(f) $\forall \alpha \beta.\ (\alpha \to \alpha \to \beta) \to \alpha * \alpha \to (\alpha \to \beta) * (\alpha \to \beta) * \beta$

**Aufgabe TC.21** *(... ooh)*
Bestimmen Sie das Typschema von `Y`.

```
fun Y f x = f (Y f) x
```

*Lösungsvorschlag TC.21*
$\texttt{Y} : ((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta$

Vorgehen: Wir legen die Typen zunächst möglichst allgemein fest und bestimmen - den Syntaxbaum von unten aufarbeitend - den finalen Typ. Wir halten zunächst fest: $Y : \alpha \to \beta \to \gamma, f : \alpha, x : \beta$, da $f$ und $x$ die Argumente von $Y$ sind.

Nun wird zunächst der Typ von $Y f$ bestimmt, welcher $\beta \to \gamma$ ist. Da $f$ darauf angewendet wird, ist $f$ eine Funktion mit Argument $\beta \to \gamma$ und einem bisher nicht weiter bestimmten Rückgabetyp $\delta$, sodass $\alpha = (\beta \to \gamma) \to \delta$. Somit haben wir jetzt $Y : ((\beta \to \gamma) \to \delta) \to \beta \to \gamma, f : (\beta \to \gamma) \to \delta, x : \beta$ bestimmt.

Nun müssen wir noch die Prozeduranwendung von `f (Y f)` auf `x` betrachten: Der Typ von $f(Y f)$ ist $\delta$, der Typ von $x$ ist $\beta$, wir stellen also wieder fest, dass $\delta$ ein Prozedurtyp mit Argument $\beta$ und bisher nicht näher bestimmbaren Rückgabetyp $\epsilon$ ist, also $\delta = \beta \to \epsilon$. Wir wissen nun also $Y : ((\beta \to \gamma) \to (\beta \to \epsilon)) \to \beta \to \gamma, f : (\beta \to \gamma) \to (\beta \to \epsilon), x : \gamma$.

Zuletzt muss noch beachtet werden, dass der Rückgabetyp von $Y$ dem Typ des gesamten Ausdruckes entsprechen muss. Der Rückgabetyp ist dabei $\gamma$, der Typ des gesamten Ausdruckes ist $\epsilon$, somit muss $\gamma = \epsilon$ sein und wir erhalten als finalen Typ $Y : ((\beta \to \epsilon) \to (\beta \to \epsilon)) \to \beta \to \epsilon$, was nach Anwendung der Klammersparregeln und Umbenennung mittels $\beta \mapsto \alpha, \epsilon \mapsto \beta$ dem obigen Typschema entspricht.