Institut für Informatik der Ludwig-Maximilians-Universität München
Prof. Martin Hofmann, PhD
Dr. Steffen Jost

SoSe 2014
21.07.2014

Lösung zur Klausur zur Vorlesung

Programmierung und Modellierung

Die Bearbeitungszeit beträgt 120 min. Hilfsmittel sind nicht erlaubt, auch der Besitz ausgeschalteter elektronischer Geräte wird als Betrug gewertet. Schreiben Sie Ihren vollständigen Namen und Ihre Matrikelnummer deutlich lesbar auf dieses Deckblatt, sowie Ihren Namen in die Kopfzeile auf **jedem Blatt** der Klausurangabe! Geben Sie alle Blätter ab, lassen Sie diese zusammengeheftet! Verwenden Sie nur dokumentenechte Stifte und weder die Farben rot noch grün.

Kontrollieren Sie, ob Sie alle Aufgabenblätter erhalten haben. Aufgabenstellungen befinden sich auf den Seiten 1–10. Sie dürfen die Rückseiten für Nebenrechnungen nutzen. Falls Sie die Rückseiten für Antworten nutzen, so markieren Sie klar, was zu welcher Aufgabe gehört und schreiben Sie in der entsprechenden Aufgabe, wo alle Teile Ihrer Antwort zu finden sind. Streichen Sie alles durch, was nicht korrigiert werden soll!

Lesen Sie die Aufgabenstellungen vollständig durch, bevor Sie mit der Bearbeitung der Aufgaben beginnen. Es gibt 8 unterschiedlich gewichtete Aufgaben zu insgesamt 43 Punkten. Mit 22 Punkten haben Sie sicherlich bestanden. Die Teilaufgaben können unabhängig voneinander bearbeitet werden.

Mit Ihrer Unterschrift bestätigen Sie, dass Sie zu Beginn der Klausur in ausreichend guter gesundheitlicher Verfassung sind, und diese Klausurprüfung verbindlich annehmen.

Nachname:

Vorname:

Matrikelnummer:

Studiengang:

Hiermit erkläre ich die Richtigkeit der obigen Angaben: _____
(Unterschrift)

Aufgabe	1	2	3	4	5	6	7	8	$\sum$	Bonus	Note
Punkte	4	6	5	5	6	6	5	6	**43**		
erreicht											

Lösung Aufgabe 1 (Verschiedenes): **(4 Punkte)**

Richtig angekreuzt:	je	0.5 Punkte
Falsch angekreuzt:	je	−0.5 Punkte
Nicht angekreuzt:	je	0 Punkte
Doppelt angekreuzt:	je	0 Punkte

Ein Typ ist eine Menge von Werten.	[X] ja	[ ] nein
In einer funktionalen Sprache sind Funktionen Werte.	[X] ja	[ ] nein
Der Unterschied zwischen einer rekursiven Funktion und einer endrekursiven Funktion ist, dass die Anwendung einer endrekursiven Funktion immer terminiert.	[ ] ja	[X] nein
Die Typsubstitution $[\texttt{Int} \to \alpha/\beta, \texttt{Int}/\gamma,]$ unifiziert die beiden Typen $\texttt{Int} \to \beta$ und $\gamma \to \gamma \to \alpha$	[X] ja	[ ] nein
Es kann maximal nur einen allgemeinsten Unifikator für eine konkrete Menge von Typgleichungen geben.	[ ] ja	[X] nein
Für Maybe-, Listen- und Baumtypen definiert die GHC Standardbibliothek Instanzen der Typklasse `Monad`	[X] ja	[X] nein
Lazy Evaluation (verzögerte Auswerung) erlaubt die Verwendung zirkulärer Definitionen und unendlicher Datenstrukturen	[X] ja	[ ] nein
In einer denotationellen Semantik wird der Typ `Integer` meist durch die partiell geordnete Menge $(\{\bot\} \cup \mathbb{Z}, \sqsubseteq)$ interpretiert. Wofür steht $\bot$? **a)** Berechnungen, welche einen Fehler liefern **b)** Für aufgrund der verzögerten Auswertung noch unausgewertete Ausdrücke vom Typ `Integer`	[X] a)	[ ] b)

Bei der 5. Frage haben wir die Antwort "ja" ebenfalls noch gelten lassen, wenn angemerkt wurde, dass der allgemeinste Unifikator bis auf Umbenennung von Typvariablen eindeutig ist.

Die 6. Frage wurde aus der Wertung genommen, da die Monaden-Instanz für Bäume in der Vorlesung nicht behandelt wurde.

Lösung Aufgabe 2 (Auswertung): **(6 Punkte)**

a) Rechnen Sie aus, zu welchem Wert die folgenden Haskell-Ausdrücke vollständig auswerten. Geben Sie nur das Endergebnis an, etwaige Nebenrechnungen bitte deutlich abtrennen:

(i) `if True && ((False) || True) then 1:2:[2+1] else ([1+1+1+1])`

`[1,2,3]`

(ii) `[(x,y)| x<-[1..2], y<-[x..3], let z = x+y, odd z]`

`[(1,2),(2,3)]`

(iii) `let f x y = if y/=0 then f (x+1) (y-1) else x in f 3 5`

`8`

b) Werten Sie folgenden Ausdruck schrittweise vollständig aus, unterstreichen Sie dabei den reduzierten Redex. Sie dabei dürfen eine beliebige Auswertestrategie verwenden, für volle Punktzahl müssen Sie aber Call-By-Value oder Call-By-Name verwenden.

Beispiel: $(\backslash x \text{ -> } 4+x)\ (\underline{succ\ 2}) \rightsquigarrow (\backslash x \text{ -> } 4+x)\ \underline{(2+1)} \rightsquigarrow \underline{(\backslash x \text{ -> } 4+x)\ 3} \rightsquigarrow \underline{4+3} \rightsquigarrow 7$

Definitionen:

```
succ x   = x+1
iter f n = f (iter f n)
```

Ausdruck zum Auswerten:
$(\backslash f \text{ -> } (\backslash x \text{ -> } (0+1)*x))$ `(iter succ 2) (succ 3)` $\rightsquigarrow$

LÖSUNG:

$$
\begin{aligned}
& \underline{(\backslash f \text{ -> } (\backslash x \text{ -> } (0+1)*x))\ \texttt{(iter succ 2)}}\ \texttt{(succ 3)} \\
\rightsquigarrow\ & \underline{(\backslash x \text{ -> } (0+1)*x)\ \texttt{(succ 3)}} \\
\rightsquigarrow\ & (0+1)*\underline{\texttt{(succ 3)}} \\
\rightsquigarrow\ & (0+1)*\underline{(3+1)} \\
\rightsquigarrow\ & \underline{(0+1)}*4 \\
\rightsquigarrow\ & \underline{1*4} \\
\rightsquigarrow\ & 4
\end{aligned}
$$

c) Welche Auswertestrategie haben Sie in Aufgabenteil b verwendet?

Call-By-Name (Call-By-Value terminiert bei diesem Ausdruck nicht)

Lösung Aufgabe 3 (Induktion): **(5 Punkte)**

Gegeben sind folgende Definitionen:

```
snoc    []  y = [y]           -- S0
snoc (x:xs) y = x : snoc xs y -- S1

length    []  = 0             -- L0
length (x:xs) = 1 + length xs -- L1
```

Beweisen Sie unter Verwendung von Induktion die folgende Gleichung:

`length (snoc t h)` = `length (h:t)`

Hinweise: Geben Sie in jedem Schritt explizit die verwendete Gleichung (`S0`, `S1`, `L0` oder `L1`) bzw. die Induktionshypothese vollständig an! Anstatt einer einzigen Kette von Gleichungen können Sie natürlich auch nacheinander beide Seiten der zu beweisenden Gleichung zum gleichen Term umformen. Beachten Sie, dass beide Terme exakt gleich sein müssen!

LÖSUNG: Der Beweis wird mit Induktion über die Länge der Liste `t` geführt.

Wichtig: Die andere Variable `h` verbleibt als Variable von einem beliebigen Wert, denn wir wollen die Gleichung ja auch für beliebige Werte von `h` beweisen!

Induktionsanfang für eine Liste der Länge 0: Es gilt also `t = []`.

1. Teil: `length (snoc [] h) =(S0)= length [h] =(L1)= 1 + length [] =(L0)= 1 + 0 = 1`

2. Teil: `length (h:[]) =(L1)= 1 + length [] =(L0)= 1 + 0 = 1`

wobei man bei `1 + length []` jeweils schon aufhören könnte.

Induktionsschritt für Liste der Länge $n > 0$: Da die Liste `t` in diesem Fall nicht leer ist, können wir dem Kopf und Rumpf von `t` eigene, frische Namen geben: Es sei `t = s:r`. Damit ist `r` eine Liste mit einer Länge kleiner als n. Somit dürfen wir als Induktionshypothese die Gleichung `length (snoc r h)` = `length (h:r)` verwenden.

1. Teil: `length (snoc (s:r) h) =(S1)= length (s : snoc r h) =(L1)=`
`1 + length (snoc r h) =(IH)= 1 + length (h:r) =(L1)= 1 + 1 + length r`

2. Teil: `length (h:s:r) =(L1)= 1 + length (s:r) =(L1)= 1 + 1 + length r`

Damit haben wir für beide Seiten der geforderten Gleichung die Gleichheit zu einem identischen Term gezeigt. *Achtung:* `1 + length (h:r)` ist ein anderer Term als `1 + length (s:t)`, d.h. man muss in beiden Fällen noch einmal `(L1)` anwenden!

Alternative Schreibweise als Kette:

```
length (snoc (s:r) h) =(S1)= length (s : snoc r h) =(L1)=
1 + length (snoc r h) =(IH)= 1 + length (h:r) =(L1)= 1 + 1 + length r
=(L1)= 1 + length (s:r) =(L1)= length (h:(s:r))
```

Lösung Aufgabe 4 (Abstiegsfunktion): **(5 Punkte)**

Wir wollen mithilfe einer geeigneten Abstiegsfunktion zeigen, dass die folgende rekursive Funktion `foo :: (Int,Int,Int) -> Int`, gegeben in Haskell Notation, immer terminiert:

```
foo (x, y, z)
  | even x && w > 10 = foo (x+1, y  , z-1)
  | odd  x && w > 15 = foo (x-1, y-1, z  )
  | otherwise        = x + z
 where w = y + z
```

a) Zeigen Sie, dass die Funktion $m'(x, y, z) = \max(x + y, 0)$ keine geeignete Abstiegsfunktion für den Terminationsbeweis von `foo` ist.

LÖSUNG: Wir wählen ein Argument, so dass wir im ersten Fall sind, da dort $x + y$ nicht kleiner wird, z.B. für $(6, 9, 9)$ haben wir $m'(6, 9, 9) = 15$. Es findet aber ein rekursiver Aufruf mit dem Argument $(7, 9, 8)$ statt. Dafür haben wir $m'(7, 9, 8) = 16 \not< 15$. Wir haben also ein Funktionsargument, für den der Wert von m' für einen rekursiven Aufruf nicht echt kleiner (also größer) wird. Somit ist m' keine geeignete Abstiegsfunktion.

Das Argument "weil z in m' nicht vorkommt" nutzt gar nichts; wie wir im zweiten Aufgabenteil sehen, muss nicht jedes Argument in der Abstiegsfunktion berücksichtigt werden.

b) Finden Sie eine Abstiegsfunktion m und beweisen, dass diese eine tatsächlich eine geeignete Abstiegsfunktion ist, also dass `foo` immer terminiert.
Hinweis: Auf Auf) und Def) verzichten wir hier.

LÖSUNG: Wir wählen als Abstiegsfunktion $m(x, y, z) = \max(y + z, 0)$ mit $m : \texttt{Int} \times \texttt{Int} \times \texttt{Int} \to \mathbb{N}$.

Der erste rekursive Aufruf erfolgt für `even x` und $y + z = w > 10$. Damit gilt in diesem Fall $\max(y + z, 0) = y + z$. Wir rechnen:

$$m(x, y, z) = \max(y + z, 0) = y + z > y + z - 1 = \max(y + (z - 1), 0) = m(x + 1, y, z - 1)$$

Die Gleichung $y+z-1 = \max(y+(z-1), 0)$ gilt, da wegen $y+z > 10$ auch $y+z-1 > 10-1 > 0$ gilt.

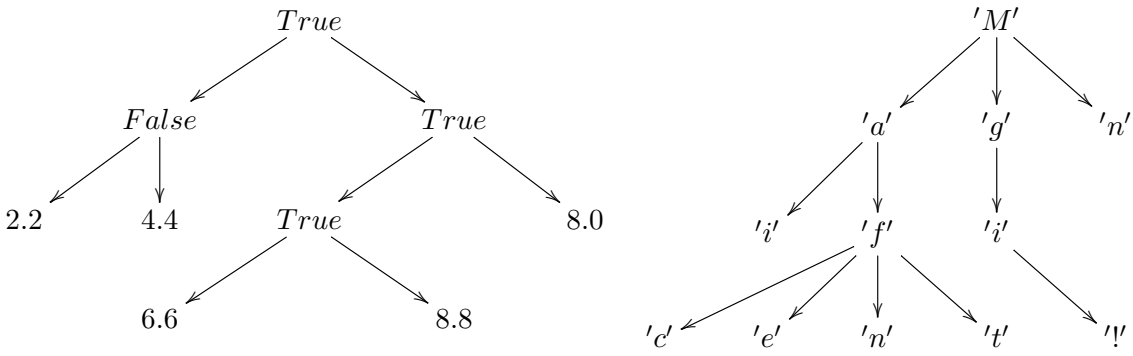
Der zweite rekursive Aufruf erfolgt, falls `odd x` und $y + z = w > 15$ gilt. Wir rechnen völlig analog:

$$m(x, y, z) = \max(y + z, 0) = y + z > y + z - 1 = \max((y - 1) + z, 0) = m(x - 1, y - 1, z)$$

Oft genannte Lösungsversuche wie etwa $m(x, y, z) = \max(x + 2y, 0)$ funktionieren nicht, da bei großen negativen Werten von x die max-Funktion greift und der Wert der Abstiegsfunktion dann immer 0 ist, also nicht echt kleiner wird, wie man z.B. für $x = -99, y = 10, z = 10$ leicht nachprüfen kann. Wird der Wert von x in der Abstiegsfunktion berücksichtigt, kann man im Gegensatz zu unserem Lösungsvorschlag hier die max-Funktion nicht mehr so einfach verschwinden lassen!

Lösung Aufgabe 5 (Bäume): **(6 Punkte)**

a) Geben Sie eine einzelne parametrisierte Datentypdeklaration in Haskell an, mit der man beide Bäume repräsentieren kann:



LÖSUNG: Diese Aufgabe war als A4-1 bekannt. Eine mögliche Lösung ist erneut `data Tree a b = Leaf a | Node b [Tree a b]`

b) Gegeben ist folgende Datentypdeklaration

```
data Baum = Ast Baum Baum | Zweig Baum | Blatt deriving Show
```

Schreiben Sie eine möglichst endrekursive Funktion `holzhacker :: Baum -> Int` welche die Anzahl der `Ast`- und `Zweig`-Konstruktoren von einem Wert des Typs `Baum` möglichst effizient berechnet, also z.B. `holzhacker $ Ast (Zweig (Zweig Blatt)) Blatt == 3`

LÖSUNG: Wer Pattern-Matching beherrscht, kann hier schon gut Punkte abräumen:

```
holzhacker :: Baum -> Integer
holzhacker  Blatt    = 0
holzhacker (Zweig z) = 1 + holzhacker z
holzhacker (Ast l r) = 1 + holzhacker l + holzhacker r
```

Wer auch den letzten Punkt für die endrekursive Version ergattern will, kann einen gewöhnlichen Akkumulator für das Ergebnis einsetzen. Für den Abstieg in zwei Teilbäume eines Astes nutzen wir dann eine Art "ToDo"-Liste, so wie im Kapitel "Breitendurchlauf" (Folie 8-13) behandelt:

```
holzhacker :: Baum -> Int
holzhacker baum = hh [baum] 0
  where
    hh []             acc = acc
    hh ((Ast l r):ts) acc = hh (l:r:ts) (acc+1)
    hh ((Zweig z):ts) acc = hh (  z:ts) (acc+1)
    hh ( Blatt   :ts) acc = hh (    ts) (acc  )
```

Die Aufgabe wird auf dem nächsten Blatt fortgesetzt.

Fortsetzung von Aufgabe 5:

c) Gegeben ist weiterhin die Datentypdeklaration

```
data Anleitung = Koans | Oans | Zwoa   deriving (Eq, Enum, Show)
```

Schreiben Sie eine Funktion `baumbastler :: [Anleitung] -> (Baum,[Anleitung])` welche eine nicht leere Liste des Typs `Anleitung` in einen Wert des Typs `Baum` umwandelt, und noch den unbenötigten Rest der Anleitung zurück liefert.

Die Anleitung beschreibt den Baum von der Wurzel aus, zuerst links absteigend bis zu einem Blatt, dann jeweils rechtsabsteigend (Tiefendurchlauf). Der Konstruktor `Koans` soll ein `Blatt` generieren, Der Konstruktor `Oans` einen `Zweig`, und `Zwoa` einen `Ast`.

Beispiele:

```
baumbastler [Oans,Koans,Zwoa]       == (Zweig Blatt,[Zwoa])
baumbastler [Zwoa,Oans,Koans,Koans] == (Ast (Zweig Blatt) Blatt,[])
baumbastler [Zwoa,Zwoa,Koans,Oans,Koans,Koans]
        == (Ast (Ast Blatt (Zweig Blatt)) Blatt,[])
baumbastler [Oans,Zwoa,Koans,Zwoa,Koans,Koans,Koans,Koans]
        == (Zweig (Ast Blatt (Ast Blatt Blatt)),[Koans,Koans])
```

LÖSUNG: Bei dieser Aufgabe handelt es sich um eine sehr abgespeckte Version der H6-3: Eine Liste von Tokens wird zu einem Baum geparsed. Der rekursive Abstieg ist hier jedoch deutlich einfacher, da der zu erstellende Knoten direkt anhand des ersten Tokens entscheiden werden kann. Auch die Problematik der Klammerung taucht hier nicht mehr auf.

```
baumbastler :: [Anleitung] -> (Baum,[Anleitung])
baumbastler (Koans:xs) =  (Blatt,xs)
baumbastler (Oans :xs) =  let (b,xs') = baumbastler xs
                          in  (Zweig b,xs')
baumbastler (Zwoa :xs) =  let (bl,xs1) = baumbastler xs
                              (br,xs2) = baumbastler xs1
                          in  (Ast bl br,xs2)
```

Lösung Aufgabe 6 (Funktionen höherer Ordnung): (6 Punkte)

a) Implementieren Sie die Funktion `foldl :: (a -> b -> a) -> a -> [b] -> a` zu Fuss, d.h. ohne Verwendung der Standardbibliothek. Die Funktion `foldl` faltet eine Funktion von links über eine Liste, d.h. `foldl f w [x,y,z]` ergibt das Gleiche wie `((w `f` x) `f` y) `f` z`.

LÖSUNG: Siehe Folie 7-18:

```
foldl _ y   []  = y
foldl f y (h:t) = foldl f (f y h) t
```

b) Schreiben Sie eine Funktion `foo :: [Double] -> (Double,Double)` welche das kleinste negative Element und die Summe einer Liste von Fliesskommazahlen berechnet. Gibt es kein kleinstes negatives Element, weil die Liste leer ist oder nur positive Zahlen enthält, dann soll 0 zurückgegeben werden. Sie dürfen dabei keine Rekursion und keine Funktionen der Standardbibliothek verwenden; Sie dürfen lediglich jeweils einmal `foldl`, `min` und `(+)` aufrufen und sonst keine anderen Funktionen der Standardbibliothek verwenden!
Beispiel: `foo [3,-2,5,-4] == (-4,2)`

LÖSUNG:

```
foo xs = foldl g (0,0) xs
  where
    g (mi,sm) d = (min mi d, sm + d)
```

Einzeilige punktfreie Alternative: `foo = foldl (\(m,s) d -> (min m d, s + d)) (0,0)`

c) Schreiben Sie eine Funktion `maxf`, welche zwei Funktionen f und g des Typs `Double -> Double` zu einer Funktion des gleichen Typs kombiniert, deren Anwendung das Maximum der beiden Funktionsergebnisse zurück gibt. Sie dürfen alle Funktionen der Standardbibliothek benutzen.
Beispiele: `(maxf (+1) negate) 1 == 2` und `(maxf (+1) negate) (-1) == 1`

`maxf :: (Double -> Double) -> (Double -> Double) -> (Double -> Double)`

LÖSUNG:

```
maxf f g x = max (f x) (g x)
```

Wer sich an `max` nicht mehr erinnert, kann das natürlich auch schnell selbst definieren:

```
mymax x y | x > y    = x
          |otherwise = y
```

Lösung Aufgabe 7 (Monaden): (5 Punkte)

a) Schreiben Sie eine Funktion `main :: IO ()`, welche eine Zeile von der Tastatur einliest und anschließend rückwärts ausgibt und beendet, also bei Eingabe von `"evil"` wird `"live"` ausgegeben. Sie dürfen alle Funktionen der Standardbibliothek benutzen. DO-Notation ist erlaubt.

LÖSUNG: Die Lösung ist ähnlich zu Übung A9-1:

```
main = do i <- getLine
          putStrLn $ reverse i
```

Wem `reverse` nicht einfällt, kann das auch selbst hinschreiben. Die vielleicht einfachste Möglichkeit wäre:

```
myreverse    [] = []
myreverse (h:t) = myreverse t ++ [h]
```

Vielleicht nicht besonders elegant oder effizient, aber es genügt den Anforderungen der Aufgabe.

b) Implementieren Sie die Funktion `forM :: Monad m => [a] -> (a -> m b) -> m [b]` zu Fuss, d.h. ohne Verwendung von Funktionen der Standardbibliothek. DO-Notation ist erlaubt.

LÖSUNG: Analog zur H9-2, nur das dieses mal das andere Argument als Liste vorliegt:

```
forM    []  _ = return []
forM (x:xs) f = do b  <- f x
                   bs <- forM xs f
                   return (b:bs)
```

Lösung Aufgabe 8 (Typen): **(6 Punkte)**

a) Geben Sie jeweils den allgemeinsten Typ des gegebenen Haskell-Ausdrucks an, inklusive etwaiger Typklassen Einschränkungen. Bitte nur das Ergebnis hinschreiben. Nebenrechnungen bitte deutlich abtrennen.

(i) `( 2>3, show True, '0')`

`(Bool, String, Char)`

(ii) `Just Nothing`

`Maybe (Maybe a)`

(iii) `(\f x y -> x == f y)`

`Eq b => (a -> b) -> b -> a -> Bool`

b) Beweisen Sie folgendes Typurteil unter Verwendung der zur Erinnerung auf Seite 10 angegeben Typregeln in einer der beiden in der Vorlesung behandelten Notationen (Herleitungsbaum oder lineare Schreibweise).

(iv) $\Gamma \vdash \texttt{f}\ (\backslash x\ \texttt{->}\ 3)\ z :: \gamma$ wobei $\Gamma = \{\texttt{f}::(\beta\ \texttt{->}\ \texttt{Int})\ \texttt{->}\ (\alpha\ \texttt{->}\ \gamma),\ \ z::\alpha\}$

LÖSUNG: Als Herleitungsbaum:

$$\dfrac{\dfrac{}{\Gamma \vdash f::(\beta\ \texttt{->}\ \texttt{Int})\ \texttt{->}\ (\alpha\ \texttt{->}\ \gamma)}\ (\textsc{Var}) \quad \dfrac{\dfrac{}{\Gamma, x::\beta \vdash 3::\texttt{Int}}\ (\textsc{Int})}{\Gamma \vdash \backslash x\ \texttt{->}\ 3::(\beta\ \texttt{->}\ \texttt{Int})}\ (\textsc{Abs})}{\dfrac{\Gamma\ \vdash\ \texttt{f}\ (\backslash x\ \texttt{->}\ 3) :: \alpha \to \gamma \qquad \dfrac{}{\Gamma \vdash z::\alpha}\ (\textsc{Var})}{\Gamma\ \vdash\ \texttt{f}\ (\backslash x\ \texttt{->}\ 3)\ z :: \gamma}\ (\textsc{App})}\ (\textsc{App})$$

Typregeln:

$$\frac{}{\Gamma \vdash x :: \Gamma(x)} \quad \text{(Var)}$$

Alternative Schreibweise für Var:

$$\frac{x{::}A \in \Gamma}{\Gamma \vdash x :: A} \quad \text{(Var)}$$

$$\frac{c \text{ ist eine Integer Konstante}}{\Gamma \vdash c :: \texttt{Int}} \quad \text{(Int)}$$

$$\frac{c \in \{\texttt{True}, \texttt{False}\}}{\Gamma \vdash c :: \texttt{Bool}} \quad \text{(Bool)}$$

$$\frac{\Gamma \vdash e_1 :: A \to B \qquad \Gamma \vdash e_2 :: A}{\Gamma \vdash e_1\ e_2 :: B} \quad \text{(App)}$$

$$\frac{\Gamma, x{::}A \vdash e :: B}{\Gamma \vdash \backslash x \to e :: A \to B} \quad \text{(Abs)}$$

$$\frac{\Gamma \vdash e_1 :: A \qquad \Gamma \vdash e_2 :: B}{\Gamma \vdash (e_1, e_2) :: (A, B)} \quad \text{(Pair-Intro)}$$

$$\frac{\Gamma \vdash e_1 :: (B, C) \qquad \Gamma, x{::}B, y{::}C \vdash e_2 :: A}{\Gamma \vdash \texttt{let}\ (x, y)\ \texttt{=}\ e_1\ \texttt{in}\ e_2 :: A} \quad \text{(Pair-Elim)}$$

$$\frac{\Gamma \vdash e_1{::}A \qquad \Gamma,\ x{::}A \vdash e_2 :: C}{\Gamma\ \vdash\ \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 :: C} \quad \text{(Let)}$$

$$\frac{\Gamma \vdash e_1 :: \texttt{Bool} \qquad \Gamma \vdash e_2 :: A \qquad \Gamma \vdash e_3 :: A}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: A} \quad \text{(Cond)}$$