Karlsruher Institut für Technologie

Institut für Programmstrukturen und Datenorganisation
Prof. Dr.-Ing. Gregor Snelting
gregor.snelting@kit.edu
Prof. Dr. Ralf Reussner
reussner@kit.edu

Klausur Programmierparadigmen

SS2011, 29. Juli 2011, 8:00 - 9:30 Uhr

Zugelassene Hilfsmittel: Vorlesungsfolien, Übungsblätter, eigene Aufzeichnungen
Die Verwendung von Büchern, Materialien aus dem Internet und elektronischen Geräten ist verboten.

Bearbeitungszeit: 90 min

Aufgabe	max. Punkte	err. Punkte
1	10	
2	19	
3	14	
4	15	
5	4	
6	4	
7	12	
8	12	
$\sum$	**90**	

Jeder Punkt entspricht ca. 1 min Bearbeitungszeit. Es ist garantiert, dass die Klausur mit 45 Punkten bestanden ist, und dass 85 Punkte für die Note 1,0 ausreichen.

Name: _____

Matrikelnummer: _____

Studiengang: _____

Schreiben Sie Ihre Lösungen direkt in die Klausurangabe. Beschriften Sie **alle verwendeten Blätter** mit Ihrem Namen und Ihrer Matrikelnummer. Trennen Sie die geklammerten Blätter der Klausur nicht auf. Weitere Blätter (auch für die Lückenaufgaben) erhalten Sie bei Bedarf.

Klausureinsicht: Die Klausureinsicht findet am 23.8.2011 von 14 bis 15:30h im R 207 AVG statt.

Aufgabe 1 (lazy evaluation) [10 Punkte]

(a) Definieren Sie eine Haskellfunktion [3 Punkte]

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c],
```

die zwei möglicherweise unendliche Listen mit einem zweistelligen Operator elementweise zu deren "Summenliste" verknüpft. Verwenden Sie Pattern Matching.

Beispiel:
`zipWith (*) [`x_1`, `x_2`, `x_3`, ...] [`y_1`, `y_2`, `y_3`, ...] = [`x_1`*`y_1`, `x_2`*`y_2`, `x_3`*`y_3`, ...]`

(b) Definieren Sie die unendliche Liste aller Fibonacci-Zahlen. Verwenden Sie `zipWith`. [7 Punkte]

Hinweis: Die Folge der Fibonacci-Zahlen ist $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$

Aufgabe 2 (Listen in Haskell) [19 Punkte]

Polynome $a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1} + a_nx^n$ können durch endliche Listen ihrer Koeffizienten dargestellt werden – zweckmäßigerweise mit a_0 am Listenanfang.

(a) Definieren Sie eine Polynomaddition. [5 Punkte]

(b) Definieren Sie die Auswertung eines Polynoms an einer Stelle x mit dem Horner-Schema. Verwenden Sie **`foldl`** oder **`foldr`**. [8 Punkte]

Zu obigem Polynom ist das Horner-Schema

$$a_0 + x \cdot (a_1 + x \cdot (a_2 + \ldots (a_{n-1} + x \cdot a_n) \ldots))$$

(c) Definieren Sie eine Funktion zum Ableiten eines Polynoms. [6 Punkte]
Zu obigem Polynom ist die Ableitung

$$a_1 + 2a_2 \cdot x + 3a_3 \cdot x^2 + \ldots + n \cdot a_n \cdot x^{n-1}$$

Aufgabe 3 (Typinferenz) [14 Punkte]

Betrachten Sie folgenden Ausschnitt der Typherleitung für den Term λx. **let** f = λy. y x 0 **in** f (...)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1(y) = \beta}{\Gamma_1 \vdash \mathtt{y} : \beta \mid C_1} \quad \dfrac{\Gamma_1(x) = \alpha}{\Gamma_1 \vdash \mathtt{x} : \alpha \mid C_2}}{\Gamma_1 \vdash \mathtt{y\ x} : \delta \mid C_3} \quad \Gamma_1 \vdash \mathtt{0} : \mathsf{int} \mid C_4}{\Gamma_1 \vdash \mathtt{y\ x\ 0} : \gamma \mid C_5}}{\mathtt{x} : \alpha \vdash \lambda\mathtt{y.\ y\ x\ 0} : \beta \to \gamma \mid C_6} \quad \sigma \text{ mgu für } C_6}{\mathtt{x} : \alpha \vdash \textbf{let}\ \mathtt{f} = \lambda\mathtt{y.\ y\ x\ 0}\ \textbf{in}\ \mathtt{f}\ (\ldots) : \ldots} \quad \Gamma \vdash \mathtt{f}\ (\ldots) : \ldots}{\vdash \lambda\mathtt{x.}\ \textbf{let}\ \mathtt{f} = \lambda\mathtt{y.\ y\ x\ 0}\ \textbf{in}\ \mathtt{f}\ (\ldots) : \alpha \to \ldots}$$

wobei $\Gamma_1 = \mathtt{x} : \alpha, \mathtt{y} : \beta$.

(a) Geben Sie die zugehörigen Constraintmengen C_1 bis C_6 an. Geben Sie den mgu σ für C_6 an. [6 Punkte]

(b) Geben Sie den Typkontext Γ für die Typisierung des (teilweise weggelassenen) **in**-Teils des **let**-Ausdrucks an. Verwenden Sie die Typregel mit Typabstraktionen (S. 192 in den Vorlesungsfolien). [2 Punkte]

(c) Entscheiden Sie für die folgenden Terme, ob sie an Stelle von (...) im **in**-Teil stehen können, so dass obige Typherleitung vervollständigt werden kann. Falls dies möglich ist, geben Sie auch den passenden Typ des Terms an; Sie brauchen jedoch *keine Typherleitung* angeben. Andernfalls begründen Sie kurz, warum es nicht möglich ist. [6 Punkte]

- λz. z
- λz. λu. z
- λz. λu. true

Aufgabe 4 (Logische Programmierung) [15 Punkte]

Gegeben seien drei Eimer mit den Fassungsvermögen 8, 5 bzw. 3 Liter. Anfangs sei der Eimer mit 8 Litern voll, die anderen leer. Wie kann man durch Um- und Ausschütten 4 Liter in einem Eimer erhalten?

Der Füllzustand der Eimer wird durch ein Tripel `(X,Y,Z)` repräsentiert. Zum Beispiel bedeutet `(6,1,0)`, dass im 8-Liter-Eimer 6 Liter sind, der 5-Liter-Eimer einen Liter enthält und der 3-Liter-Eimer leer ist.

(a) Das dreistellige Prolog-Prädikat `move(F1,M,F2)` gibt an, dass sich aus dem Füllzustand `F1` der Füllzustand `F2` durch Um- bzw. Ausschüttens eines Eimers ergeben kann. `M` ist ein String, der die Umfüllaktion beschreibt. Geben Sie Regeln für folgende Fälle an: [7 Punkte]

- Ausschütten des nicht-leeren 5-Liter-Eimers,
- Umschütten des 8-Liter-Eimers in den 3-Liter-Eimer.

Beispiel:

```
?- move((7,0,1),M,FL).
FL = 5,0,3
M = 'E8 in E3 umschütten'
```

Hinweis: Man bräuchte weitere Regeln für alle weiteren Eimer(-kombinationen), diese brauchen Sie aber *nicht* anzugeben.

(b) Definieren Sie ein Prolog-Prädikat `solve`, das eine Antwort auf die Frage „Wie erhält man 4 Liter in einem Eimer?“ berechnet. `solve(F0, FL, Ms, Fs)` ist dann erfüllt, wenn sich der Füllzustand `F0` mittels der Füllzustandsübergänge von `move` nach `FL` überführen lässt, ohne dass einer der Zwischenzustände in der Liste `Fs` besucht wird. Die Liste `Ms` sammelt dabei die Beschreibungen der einzelnen `move`s. Die Liste `Fs` der verbotenen Zustände enthält die bereits besuchten Zustände, um Endlosschleifen zu vermeiden. [8 Punkte]

Mit diesem Prädikat lässt sich dann das Anfangsproblem wie folgt lösen:

```
solution(Ms) :- start(F0), solve(F0, FL, Ms, [F0]), final(FL).
start((8,0,0)).
final((_,4,_)).
final((4,_,_)).
```

Hinweis: Verwenden Sie die Prädikate `member` und `not` aus der Vorlesung.

Aufgabe 5 (C) [4 Punkte]

Welche Ausgabe erzeugt das folgende C-Programm? Begründen Sie Ihre Antwort kurz.

Hinweis: `printf("%i",i)` gibt den Zahlenwert der Integer-Variable `i` aus.

```
#include <stdio.h>

int arr[] = {456,2,0,8,1,-2};

int* process(int op_1[], int op_2) {
   printf("x");
   arr[1] = 4;
   arr[5] = 3;
   return &op_1[op_2];
}

int main() {
   printf("%i", *process(&arr[4], arr[4]));
   return 0;
}
```

Aufgabe 6 (X10) [4 Punkte]

Gegeben sei folgende X10-Methode zur Berechnung der Lösungen einer quadratischen Gleichung nach der p-q-Formel mit anschließender Ausgabe der Lösungen:

$$x_{1,2} = -\frac{p}{2} \pm \sqrt{\left(\frac{p}{2}\right)^2 - q}$$

```
def pqFormel(val p:Complex, val q:Complex) {
   var x1 : Complex = Complex(0,0);
   var x2 : Complex = Complex(0,0);
   val h = here;

   val disc = Math.sqrt((p/2)*(p/2) - q);
   x1 = -p/2 + disc;
   x2 = -p/2 - disc;

   Console.OUT.println("x1 = " + x1 + "; x2 = " + x2);
}
```

Man nehme an, dass es zwei Places gibt, und die Arithmetik auf dem zweiten Place sehr viel schneller berechnet werden kann als auf dem ersten. Daher soll das Programm zwischen den beiden Places mithilfe von X10 verteilt werden.

Aufgabe: Ergänzen Sie das untenstehende Methodenfragment an den grau markierten Stellen so, dass nur die Konsolenausgabe und ggf. Zuweisungen auf dem Knoten `here` stattfinden und **sämtliche Berechnungen auf dem anderen** Place.

Hinweis: Der Wert vom Typ Complex kann wie eine gewöhnliche Gleitkommazahl verwendet werden.

```
def pqFormel(val p:Complex, val q:Complex) {
  var x1 : Complex = Complex(0,0);
  var x2 : Complex = Complex(0,0);
  val h = here;
```

```

```

```
  // berechne Diskriminante
```

```
                        Math.sqrt((p/2)*(p/2) - q);
```

```

```

```
  Console.OUT.println("x1 = " + x1 + "; x2 = " + x2);
}
```

Aufgabe 7 (Scala) [12 Punkte]

Zahlen, die auf 2, 3, 7 oder 8 enden, können niemals eine Quadratzahl sein. Dagegen kann eine auf 5 endende Zahl eine Quadratzahl sein, muss aber nicht (z.B. 15). Im nachfolgenden Codegerüst überprüfen `squareCheckPartA` und `squareCheckPartB` anhand der letzten Ziffer, ob die übergebene Zahl eine Quadratzahl sein *könnte*. `squareCheckParallel` soll beide Teile kombiniert auf die Eingabe `input` anwenden.

Vervollständigen Sie das Codegerüst an den grau markierten Stellen, sodass diese Überprüfung auf den Aktor `HelperActor` zurückgreift, um alle relevanten Möglichkeiten für Endziffern zu berücksichtigen. `squareCheckPartA` und `squareCheckPartB` sollen also parallel ausgeführt werden.

Lassen Sie dabei `squareCheckPartA` unverändert; fügen Sie nicht mehr Code als nötig ein. Nehmen Sie an, dass als `input` ausschließlich ganze Zahlen > 0 übergeben werden. Beachten Sie, dass `squareCheckParallel` auch dann **`true`** zurückliefern kann, wenn `input` keine Quadratzahl ist (z.B. bei `input`=15).

```
class SquareChecker extends Actor {
  def squareCheckPartA(input: Int): Boolean = {
    var resultA = (input % 10 != 2 && input % 10 != 3)
    return resultA
  }
  def squareCheckParallel(input: Int): Boolean = {
                                                    
                                                    
                                                    
                                                    
     val resultA = squareCheckPartA(input)
     var resultHelper = true
                                                    
                                                    
                                                    
                                                    
                                                    
                                                    
    return resultA && resultHelper
  }
  def act() { }
}

class HelperActor(parent: SquareChecker) extends Actor {
  def act = receive { case x: Int => squareCheckPartB(x)}
  def squareCheckPartB(input: Int) {
    var resultB = input % 10 !=            && input % 10 !=
                                                    
                                                    
                                                    
  }
}
```

Aufgabe 8 (Compilerbau) [8 Punkte]

Übersetzen Sie folgenden Java-Programmausschnitt in Java-Bytecode.

```
if ((a > b) || (b > 0))
    a = a - this.f(3 + b * a);
```

Gehen Sie dabei davon folgenden Rahmenbedingungen aus:

- Der `this`-Pointer ist in Variable 0 gespeichert.
- Die `int`-Variablen `a` und `b` sind als lokale Variablen 1 und 2 verfügbar.
- Im Konstantenpool steht an vierter Stelle die Information für die nicht-statische Methode `f`.

Hinweis: Verwenden Sie das Übersetzungsschema der Vorlesungsfolien S. 363–370.

Aufgabe 1 (lazy evaluation) [10 Punkte]

Beispiellösung:

```
zipWith op [] ys = []
zipWith op xs [] = []
zipWith op (x:xs) (y:ys) = op x y : zipWith op xs ys

fib = 0 : 1 : zipWith (+) fib (tail fib)
```

Aufgabe 2 (Listen in Haskell) [19 Punkte]

Beispiellösung:

```
type Polynom = [Double]

polyAdd :: Polynom -> Polynom -> Polynom
polyAdd [] q = q
polyAdd p [] = p
polyAdd (a:p) (b:q) = a + b : polyAdd p q

polyEval :: Polynom -> Double -> Double
polyEval p x = foldr (\a v -> v * x + a) 0 p

polyDeriv :: Polynom -> Polynom
polyDeriv [] = []
polyDeriv p  = zipWith (*) [1..] (tail p)
```

Aufgabe 3 (Typinferenz) [14 Punkte]

Beispiellösung:

$C_1 = C_2 = C_4 = \emptyset$, $C_3 = \{\beta = \alpha \to \delta\}$, $C_5 = C_6 = \{\beta = \alpha \to \delta, \delta = \mathsf{int} \to \gamma\}$
$\sigma = [\beta \mapsto \alpha \to \mathsf{int} \to \gamma, \delta \mapsto \mathsf{int} \to \gamma]$

$\Gamma = \mathrm{x} : \alpha, f : ta(\sigma(\beta \to \gamma), \mathrm{x} : \alpha) = \forall\gamma.\ (\alpha \to \mathsf{int} \to \gamma) \to \gamma$

λz. z: geht nicht. Typen von λz. z sind von Gestalt $(\tau_1 \to \tau_1)$, aber $\forall\gamma.\ (\alpha \to (\mathsf{int} \to \gamma)) \to \gamma$ lässt sich mit keinem Typterm der Gestalt $(\tau_1 \to \tau_1) \to \tau_2$ instanziieren, da α im Typschema nicht gebunden wird.
$\Gamma \vdash \lambda$z. λu. z : $\alpha \to \mathsf{int} \to \alpha$
$\Gamma \vdash \lambda$z. λu. `true` : $\alpha \to \mathsf{int} \to \mathsf{bool}$

Aufgabe 4 (Logische Programmierung) [15 Punkte]

Beispiellösung:

```
move((E8,E5,E3), 'E5 ausschuetten', (E8,0,E3)) :- E5 > 0.

move((E8,E5,E3), 'E8 -> E3', (0,E5,R3)) :- E8 > 0, E8 + E3 < 3, R3 is E8 + E3.
move((E8,E5,E3), 'E8 -> E3', (R8,E5,3)) :- E8 > 0, 3 =< E8 + E3, R8 is E8-(3-E3).

solve(X,X,[],_).
solve(X,Y,[M|Ms],Fs) :- move(X,M,F),not(member(F,Fs)),solve(F,Y,Ms,[F|Fs]).
```


Aufgabe 5 (C) [4 Punkte]

Beispiellösung:

Die Ausgabe ist: `x3`

Erklärung:

- Initialisierung der `process` Parameter:
  `op_1 = [1,-2]` (gealiast mit `arr` ab dem 4. Element) `op_2 = 1`
- Ausgabe `x`; wegen des Alias mit `op_1` überschreibt die zweite Zuweisung die `-2` in `op_1` mit `3`.
- Adresse des zweiten Werts (`3`) von `op_1` wird zurückgegeben
- `printf` dereferenziert die übergebene Adresse und gibt den Wert `3` aus.

Aufgabe 6 (X10) [4 Punkte]

```
def pqFormel(val p:Complex, val q:Complex) {
  var x1 : Complex = Complex(0,0);
  var x2 : Complex = Complex(0,0);
  val h = here;
  val o = here.next();
  // berechne Diskriminante
  val disc = at (o) Math.sqrt((p/2)*(p/2) - q);
  x1 = at (o) -p/2 + disc;
  x2 = at (o) -p/2 - disc;
  Console.OUT.println("x1 = " + x1 + "; x2 = " + x2);
}
```

Aufgabe 7 (Scala) [12 Punkte]

```
class SquareChecker extends Actor {
  def squareCheckPartA(input: Int): Boolean = {
    var resultA = (input % 10 != 2 && input % 10 != 3)
    return resultA
  }
  def squareCheckParallel(input: Int): Boolean = {
    val firstHelper = new HelperActor(this).start
    firstHelper ! input
    val resultA = squareCheckPartA(input)
    var resultHelper = true
    receive { // default "case _" not used here
      case x : Boolean => resultHelper = x
    }
    return resultA && resultHelper
  }
  def act() { }
}

class HelperActor(parent: SquareChecker) extends Actor {
  def act = receive { case x: Int => squareCheckPartB(x)}
  def squareCheckPartB(input: Int) {
    var resultB = input % 10 != 7 && input % 10 != 8
    parent ! resultB
  }
}
```

Aufgabe 8 (Compilerbau) [8 Punkte]

```
        iload_1             // a
        iload_2             // b
        if_icmpgt then      // if (a > b) goto then
        goto or             // nicht nötig, aber übersichtlicher
or:     iload_2             // b
        iconst_0            // 0
        if_icmpgt then      // if (b > 0) goto then
        goto else
then:   iload_1             // a
        aload_0             // this
        iconst_3            // 3
        iload_1             // a
        iload_2             // b
        imult               // (a * b)
        iadd                // 3 + (a * b)
        invokevirtual #4    // f(3 + (a * b))
        isub                // a - f(3 + a * b)
        istore_1            // a = a - f(3 + a * b)
else:
finish:
```