12. Übungsblatt „Programmiersprachen“ SS 99

Leif Kornstaedt, Prof. Gert Smolka

7. Juli 1999

Hinweise

Dieses Übungsblatt bereitet Sie auf die Abschlußklausur am Donnerstag, den 15. Juli 1999 um 14:00 im HS 002 vor. Bitte melden Sie sich per Email an Leif Kornstaedt `<kornstae@ps.uni-sb.de>` zur Klausur an. Für die Klausur gelten die folgenden Regeln:

- Es sind keine Unterlagen oder Hilfsmittel erlaubt.
- Die Lösungen müssen direkt ins Klausurheft geschrieben werden. Sollte der vorgesehene Platz nicht ausreichen, können Sie weitere Papierbögen von uns erhalten.
- Benutzen Sie nur das von uns ausgegebene Notizpapier. Schreiben Sie auf jeden Bogen Ihren Namen und Vornamen sowie Ihre Matrikelnummer.
- Die Bearbeitungszeit beträgt **150 Minuten**.
- Die maximal erreichbare Punktzahl ist 194. Mit mindestens 97 erreichten Punkten gilt die Klausur auf jeden Fall als bestanden.
- Täuschungsversuche führen zum sofortigen Ausschluß von der Klausur. In diesem Fall wird die Klausur als *nicht bestanden* gewertet.
- Sie dürfen die Klausur auch mit Bleistift bearbeiten.

1	2	3	4	5	6	7	8	9	10	11	Σ
6	12	28	20	24	32	16	16	12	18	10	194

XII.1: Polymorphe Typen $(2 + 2 + 2$ Punkte$)$

Geben Sie die allgemeinsten Typen für die folgenden Prozeduren an.

```
fun f nil = nil
  | f [x] = [x]
  | f (x::y::_) = [x, y]
```

```
fun g x = fn y => (f x, f y)
```

```
fun h x = fn y => let val v = f nil
                  in (x::v, y::v)
                  end
```

XII.2: Enumeration mit SML (12 Punkte)

Schreiben Sie eine Prozedur

```
enumerator: 'a list -> unit -> 'a option
```

die zu einer Liste eine Prozedur mit Zustand liefert, die die Elemente der Liste enumeriert. Nachdem alle Elemente der Liste enumeriert wurden, soll `NONE` geliefert werden.

XII.3: Zähler $(12 + 12 + 4$ Punkte)

(a) Deklarieren Sie eine Struktur `Counter` mit der Signatur

```
signature COUNTER =
    sig
        eqtype counter
        val counter: int -> counter
        val tick:    counter -> int
        val set:     counter * int -> unit
    end
```

und Operationen wie folgt:

- `counter` erzeugt bei jedem Aufruf einen neuen Zähler, der wie angegeben initialisiert ist.
- `tick` erhöht den Wert des Zählers um eins und liefert den bisherigen Wert.
- `set` setzt den Zähler auf einen gegebenen Wert.

(b) Sei der Typ

```
type counter =
    {tick:     unit -> int,
     set:      int -> unit,
     identity: unit ref}
```

gegeben. Deklarieren Sie eine Prozedur

```
counter: int -> counter
```

die objekt-orientierte Zähler erzeugt, deren Funktionalität den Zählern aus (a) entspricht.

(c) Deklarieren Sie eine Prozedur

```
equal: counter * counter -> bool
```

die objekt-orientierte Zähler auf Gleichheit testet.

Hinweise:

- Verschiedene Zähler können denselben Wert haben.
- Denken Sie daran, dass Verbundtypen oft explizit spezifiziert werden müssen.

XII.4: Rekursionsoperator $(12 + 8$ Punkte)

(a) Deklarieren Sie eine Prozedur

```
Rek: (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
```

mit der monomorphe Deklarationen der Form

fun f x = $\langle Ausdruck \rangle$

in äquivalente Deklarationen der Form

val f = Rek (**fn** f => **fn** x => $\langle Ausdruck \rangle$)

umgeschrieben werden können.

(b) Sei

fun f x = $\langle Ausdruck \rangle$

eine polymorphe Prozedurdeklaration. Wie kann man diese Deklaration mithilfe des Rekursionsoperators Rek aus (a) in eine äquivalente Deklaration

val f = ...

umschreiben?

XII.5: Funktoren (20 + 4 Punkte)

Seien die Signaturen

```
signature KEY_ENTRY =
    sig
        eqtype key
        type entry
    end

signature TABLE =
    sig
        eqtype key
        type entry
        type table
        val  empty:  unit -> table
        val  insert: table * key * entry -> unit
        val  lookup: table * key -> entry option
    end
```

gegeben. Strukturen mit der Signatur `TABLE` sollen Tabellen als imperative Datenstrukturen realisieren.

(a) Schreiben Sie einen Funktor `MakeTable`, der Strukturen, die der Signatur `KEY_ENTRY` genügen, auf Strukturen abbildet, die der Signatur `TABLE` genügen.

(b) Deklarieren Sie mit Hilfe des Funktors `MakeTable` eine Struktur `Table` mit Schlüsseln des Typs `string` und Einträgen des Typs `int`.

XII.6: Dynamische Semantik (32 Punkte)

Sie sollen die dynamische Semantik einer SML-artigen Sprache R mit Inferenzregeln spezifizieren. Die abstrakte Syntax von R ist:

$$
\begin{array}{lll}
x & \in & \text{Bez} \\
a & \in & \text{Aus}
\end{array}
$$

$$
\begin{array}{lllll}
a & ::= & \texttt{()} & & \text{Unit} \\
 & | & x & & \text{Bezeichner} \\
 & | & \texttt{fn}\ x\ \texttt{=>}\ a & & \text{Abstraktion} \\
 & | & a_1\ a_2 & & \text{Applikation} \\
 & | & \texttt{ref}\ a & & \text{Referenzerzeugung} \\
 & | & \texttt{!}a & & \text{Dereferenzierung} \\
 & | & a_1\ \texttt{:=}\ a_2 & & \text{Zuweisung}
\end{array}
$$

Die dynamische Semantik verwendet Werte, Umgebungen und Zustände wie folgt:

$$
\begin{array}{lllll l}
w & \in & \text{Wert} & = & \{\emptyset\} \cup \text{Ref} \cup \text{Pro} & \text{Wert} \\
r & \in & \text{Ref} & & & \text{Referenz} \\
 & & \text{Pro} & = & \text{Bez} \times \text{Aus} \times \text{WU} & \text{Prozedur} \\
W & \in & \text{WU} & = & \text{Bez} \xrightarrow{\text{fin}} \text{Wert} & \text{Wertumgebung} \\
\sigma & \in & \text{Zust} & = & \text{Ref} \xrightarrow{\text{fin}} \text{Wert} & \text{Zustand}
\end{array}
$$

Dabei ist eine Funktion

$$\text{Next} : \text{Fin}(\text{Ref}) \to \text{Ref}$$

mit

$$\forall R \in \text{Fin}(\text{Ref}) : \text{Next}(R) \notin R$$

für die Erzeugung von Referenzen gegeben. Fin(Ref) ist die Menge aller endlichen Teilmengen von Ref.

Die dynamische Semantik wird mit Transitionen der Form

$$\sigma, W \vdash a \Rightarrow w, \sigma'$$

formuliert. Ergänzen Sie die Prämissen der nachfolgenden Inferenzregeln so, dass diese eine SML-artige dynamische Semantik für R spezifizieren.

$$\frac{}{\sigma, W \vdash \texttt{()} \Rightarrow \emptyset, \sigma}$$

$$\frac{}{\sigma, W \vdash x \Rightarrow w, \sigma}$$

$$\frac{}{\sigma, W \vdash \texttt{\textbf{fn}}\ x\ \texttt{\textbf{=>}}\ a \Rightarrow (x, a, W), \sigma}$$

$$\frac{}{\sigma, W \vdash a_1\ a_2 \Rightarrow w', \sigma'''}$$

$$\frac{}{\sigma, W \vdash \texttt{\textbf{ref}}\ a \Rightarrow r, \sigma''}$$

$$\frac{}{\sigma, W \vdash \texttt{!}a \Rightarrow w, \sigma'}$$

$$\frac{}{\sigma, W \vdash a_1\ \texttt{\textbf{:=}}\ a_2 \Rightarrow \emptyset, \sigma'''}$$

XII.7: Enumeration mit Java (16 Punkte)

Vervollständigen Sie die folgenden Deklarationen um die Implementation der dynamischen Methode `enumerator`, die einen Enumerator für `this` liefern soll. Nachdem alle Elemente der Liste enumeriert wurden, soll `null` geliefert werden.

```
interface Enumerator {
    Object nextElement();
}

class List {
    final Object head;
    final List tail;
    List(Object o, List l) { head = o; tail = l; }
    Enumerator enumerator() {









    }
}
```

XII.8: Ausdrücke in Java (16 Punkte)

Vervollständigen Sie die nachfolgende Klassendeklaration so, dass damit arithmetische Ausdrücke mit Summen erzeugt, zerlegt und evaluiert werden können.

```
abstract class Expression {
    abstract int eval();

    static class Con extends Expression {


        Con(int x) {


        }







    }

    static class Sum extends Expression {




        Sum(Expression x, Expression y) {




        }






    }
}
```

XII.9: Casts (12 Punkte)

Seien die folgenden Typdeklarationen gegeben:

```
interface A {}
class B implements A {}
class C implements A {}
class D extends B {}

class Test {
    public static void main(String[] args) {
        B b  = new B();
        A a  = (A) b;
        B b1 = (B) a;
        C c  = (C) b;
        C c1 = (C) a;
        D d  = (D) b;
        D d1 = (D) a;
    }
}
```

Tragen Sie in der folgenden Tabelle für jeden der obigen Casts ein, ob er gemäß der

(a) statischen Semantik überflüssig, notwendig oder illegal ist (exklusives Oder). Dabei sollen überflüssige und notwendige Casts natürlich legal sein.

(b) dynamischen Semantik scheitert oder erfolgreich ist.

Cast	statisch überflüssig/notwendig/illegal	dynamisch erfolgreich/scheitert
`A a  = (A) b;`		
`B b1 = (B) a;`		
`C c  = (C) b;`		
`C c1 = (C) a;`		
`D d  = (D) b;`		
`D d1 = (D) a;`		

XII.10: Konstruktoren ($4 + 6 + 4 + 4$ Punkte)

Seien die folgenden Klassendeklarationen gegeben:

```java
class A {
    A() { println("A"); }
    void println(String s) {
        System.out.println(s);
    }
}

class B extends A {}

class C extends B {
    C() { println("C"); }
    public static void main(String[] args) {
        new C();
    }
}
```

(a) Was wird bei der Ausführung von `main` ausgegeben?

(b) Erweitern sie die obigen Klassendeklarationen so, dass implizite Konstruktoraufrufe durch explizite Konstruktoraufrufe ersetzt werden. (Schreiben Sie die Erweiterungen direkt in die obigen Deklarationen.)

(c) Warum sind die folgenden Klassendeklarationen statisch unzulässig?

```java
class A {
    A(String s) { println(s); }
    void println(String s) {
        System.out.println(s);
    }
}

class B extends A {}

class C extends B {
    C() { println("C"); }
    public static void main(String[] args) {
        new C();
    }
}
```

(d) Erweitern sie die Klassendeklarationen in (c) durch eine Konstruktordeklaration für `B` so, dass die Deklarationen statisch zulässig werden. (Schreiben Sie die Erweiterungen direkt in die obigen Deklarationen.)

XII.11: Erweiterung, Verdeckung, Überladung, Überschreibung (10 Punkte)

Seien folgende Klassendeklarationen gegeben:

```
class A {
    int a = 1;
    void v() {}
    static void u() {}
    void w() {}
}
class B extends A {
    int a = 2;

    int b = 3;

    void v() {}

    static void u() {}

    void w(int x) {}
}
```

Geben Sie für jede Deklaration von `B` an, ob es sich um eine Erweiterung, Verdeckung, Überladung oder Überschreibung handelt (exklusives Oder).