Fachrichtung 6.2 — Informatik
Universität des Saarlandes
Tutorenteam der Vorlesung Programmierung 1

Programmierung 1 (Wintersemester 2012/13)

Lösungsblatt 11

(Kapitel 12)

Hinweis: Dieses Übungsblatt enthält von den Tutoren für die Übungsgruppe erstellte Aufgaben.

Die Aufgaben und die damit abgedeckten Themenbereiche sind für die Klausur weder relevant, noch irrelevant.

1 Zoologie

Aufgabe 11.1 (*Dieters kleine Zoologiestunde*)
Dieter Schlau wurde beauftragt, einige zoologische Experimente zum Fressverhalten von Tigern und Mäusen durchzuführen. Da Dieter jedoch begeisterter Tierschützer ist (und Angst vor den Tigern hat), hat er sich entschieden, seine Experimente in Standard ML zu simulieren.
Sowohl Tiere als auch Futter können eindeutig anhand eines Namens identifiziert werden:

```
type name = string
```

Außerdem hat Dieter bereits eine Möglichkeit entwickelt, die Stärke von Tieren als natürliche Zahl darzustellen. Er kann auch zu jeder Portion Futter angeben, wie viel das Essen die Tiere stärkt.

```
type staerke = int
datatype con = Tiger of staerke | Maus of staerke | Futter of staerke
```

Bei den Experimenten können entweder zwei Tiere miteinander kämpfen (nur das stärkere Tier bleibt übrig) oder ein Tier bekommt Futter.

```
datatype opr = Kaempfen | Essen
```

Tiger sind immer stärker als Mäuse, innerhalb dieser Arten gewinnt das stärkste Tier.
Jetzt möchte Dieter eine Prozedur $elab : exp \to ty$ entwickeln, die bestimmt, welchen Typ das Ergebnis eines Experiments hat. Mögliche Typen sind "Nahrung" oder "Tier":

```
datatype ty = Tier | Nahrung
```

Hier ist ein Beispiel für ein Experiment, bei dem c zu einem Tiger (also einem Tier) auswertet:

```
val a = Con(Tiger 500)
val b = Con(Maus 5)
val c = Opr(Kaempfen, a, b)
```

(a) Erarbeiten Sie in der Gruppe die Inferenzregeln für die statische Semantik.

(b) Schreiben Sie den Elaborierer.

(c) Da Dieter Gerechtigkeit liebt, will er jetzt auch den Mäusen eine Möglichkeit geben, die Tiger zu fressen. Dazu hat er sich eine weitere Operation *Schlachten* ausgedacht, die ein Tier in Futter gleicher Stärke umwandelt.

Dieses Experiment rächt sich beispielsweise am Tiger:

```
val d = UOpr(Schlachten, c)
```

(i) Erweitern Sie die Inferenzregeln um eine Regel zum Schlachten.

(ii) Erweitern Sie *elab* entsprechend. Sie müssen dazu auch *uopr* deklarieren und *exp* anpassen.

(d) Da Dieter sich die häufigsten Schritte in seinen Experimenten erleichtern möchte, möchte er Abstraktion und Applikation einführen.

Hier ist ein Beispiel für eine Abstraktion, die ein Tier füttert, und die Anwendung auf eine neue Maus:

```
val e = Abs("ziel", Tier, Opr(Essen, Name "ziel", Con(Futter 5)))
val f = App(e, Con(Maus 10))
```

(i) Erweitern Sie die Inferenzregeln für *Abs* und *App*.

(ii) Erweitern Sie *elab* entsprechend. Sie müssen dazu auch *exp* und *ty* anpassen.

Lösung 11.1:

STiger $\dfrac{}{T \vdash Tiger(x) : Tier}$ **SMaus** $\dfrac{}{T \vdash Maus(x) : Tier}$

SFutter $\dfrac{}{T \vdash Futter(x) : Essen}$ **SName** $\dfrac{Tx = t}{T \vdash x : t}$

SKampf $\dfrac{T \vdash e_1 : Tier \quad T \vdash e_2 : Tier}{T \vdash Kämpfen(e_1, e_2) : Tier}$ **SEssen** $\dfrac{T \vdash e_1 : Tier \quad T \vdash e_2 : Nahrung}{T \vdash Essen(e_1, e_2) : Tier}$

SSchlachten $\dfrac{T \vdash e : Tier}{T \vdash Schlachten(e) : Nahrung}$ **SAbs** $\dfrac{T[x := t] \vdash e : t'}{T \vdash fn\, x : t \Rightarrow e : t \to t'}$

SApp $\dfrac{T \vdash e_1 : t' \to t \quad T \vdash e_2 : t'}{T \vdash e_1 e_2 : t}$

```
type name = string

type 'a env = name -> 'a
exception Unbound of name
fun empty x = raise Unbound x
fun update env x a y = if y=x then a else env y

exception Error of string

type staerke = int
datatype con = Tiger of staerke | Maus of staerke | Futter of staerke
datatype opr = Kaempfen | Essen | Schlachten

datatype ty = Tier | Nahrung  | Arrow of ty * ty

datatype exp =
     Con of con
  | Name of name
  |  Opr of opr * exp * exp
  | UOpr of opr * exp
  |  Abs of name * ty * exp
  |  App of exp * exp

fun elabCon (Tiger  _) = Tier
  | elabCon (Maus   _) = Tier
  | elabCon (Futter _) = Nahrung
fun elabOpr Kaempfen Tier Tier    = Tier
  | elabOpr Essen    Tier Nahrung = Tier
  | elabOpr  _        _    _     = raise Error "T Opr"
```

```
fun elabUOpr Schlachten Tier = Nahrung
  | elabUOpr  _          _   = raise Error "R UOpr"
fun elab f (Con  c)           = elabCon c
  | elab f (Name x)           = f x
  | elab f (Opr(opr, e1, e2)) = elabOpr opr (elab f e1) (elab f e2)
  | elab f (UOpr(opr, e))     = elabUOpr opr (elab f e)
  | elab f (Abs(x, t, e))     = Arrow(t, elab (update f x t) e)
  | elab f (App(e1, e2)) = (case elab f e1 of
                            Arrow(t, t') => if t = elab f e2 then t'
                                            else raise Error "T App1"
                          | _            => raise Error "T App2")
```

Aufgabe 11.2 (*Dieters dynamische Zoologiestunde*)
Dieter ist mit Ihrer Arbeit leider noch nicht ganz zufrieden: Er kann nun zwar ermitteln, welchen Ergebnistyp ein Experiment liefert, aber kann nicht ermitteln, welches Tier bzw. welches Futter ausgegeben wird. Natürlich freuen Sie sich wieder darauf, ihm zu helfen.

(a) Geben Sie die nötigen Inferenzregeln für die Experimente ohne Abstraktion und Applikation an.

(b) Schreiben Sie einen Evaluierer, der Experimente ohne Abstraktion und Applikation auswerten kann.

(c) Erweitern Sie Ihre Lösungen um Abstraktion und Applikation.

(i) Erweitern Sie die Inferenzregeln für *Abs* und *App*.

(ii) Erweitern Sie *eval* entsprechend.

Lösung 11.2:

DTiger $\dfrac{x \in \mathbb{Z}}{V \vdash Tiger(x) \rhd TV(x)}$ **DMaus** $\dfrac{x \in \mathbb{Z}}{V \vdash Maus(x) \rhd MV(x)}$

DFutter $\dfrac{x \in \mathbb{Z}}{V \vdash Futter(x) \rhd FV(x)}$ **DName** $\dfrac{Vx = v}{V \vdash x \rhd v}$

DK-TvT $\dfrac{V \vdash e_1 \rhd TV(x) \quad V \vdash e_2 \rhd TV(y)}{V \vdash Kämpfen(e_1, e_2) \rhd TV(max(x, y))}$

DK-TvM $\dfrac{V \vdash e_1 \rhd TV(x) \quad V \vdash e_2 \rhd MV(y)}{V \vdash Kämpfen(e_1, e_2) \rhd TV(x))}$

DK-MvT $\dfrac{V \vdash e_1 \rhd MV(x) \quad V \vdash e_2 \rhd TV(y)}{V \vdash Kämpfen(e_1, e_2) \rhd TV(y))}$

DK-MvM $\dfrac{V \vdash e_1 \rhd MV(x) \quad V \vdash e_2 \rhd MV(y)}{V \vdash Kämpfen(e_1, e_2) \rhd MV(max(x, y))}$

DTEssen $\dfrac{V \vdash e_1 \rhd FV(x) \quad V \vdash e_2 \rhd TV(y)}{V \vdash Essen(e_1, e_2) \rhd TV(x + y))}$

DMEssen $\dfrac{V \vdash e_1 \rhd FV(x) \quad V \vdash e_2 \rhd MV(y)}{V \vdash Essen(e_1, e_2) \rhd MV(x + y))}$

DTSchlachten $\dfrac{V \vdash e \rhd TV(x)}{V \vdash Schlachten(e) \rhd FV(x)}$

DMSchlachten $\dfrac{V \vdash e \rhd MV(x)}{V \vdash Schlachten(e) \rhd FV(x)}$

DAbs $\dfrac{}{V \vdash fn\, x : t \Rightarrow e \rhd \langle x, e, V \rangle}$

DApp $\dfrac{V \vdash e_1 \rhd \langle x, e, V' \rangle \quad V \vdash e_2 \rhd v_2 \quad V'[x := v_2] \vdash e \rhd v}{V \vdash e_1 e_2 \rhd v}$

```
datatype value = TV of int | MV of int | FV of int | Proc of name * exp * value env

fun evalCon (Tiger  x) = TV x
  | evalCon (Maus   x) = MV x
  | evalCon (Futter x) = FV x
fun evalOpr Kaempfen (TV x) (TV y) = if x > y then TV x else TV y
  | evalOpr Kaempfen (TV x) (MV _) = TV x
  | evalOpr Kaempfen (MV _) (TV x) = TV x
  | evalOpr Kaempfen (MV x) (MV y) = if x > y then MV x else MV y
  | evalOpr Essen    (TV x) (FV y) = TV (x+y)
  | evalOpr Essen    (MV x) (FV y) = MV (x+y)
  | evalOpr  _        _      _     = raise Error "R Opr"
fun evalUOpr Schlachten (TV x) = FV x
  | evalUOpr Schlachten (MV x) = FV x
  | evalUOpr  _          _     = raise Error "R UOpr"
fun eval f (Con c) = evalCon c
  | eval f (Name x) = f x
  | eval f (Opr(opr, e1, e2)) = evalOpr opr (eval f e1) (eval f e2)
  | eval f (UOpr(opr, e)) = evalUOpr opr (eval f e)
  | eval f (Abs(x,t,e)) = Proc(x,e,f)
  | eval f (App(e1, e2)) = (case (eval f e1, eval f e2) of
                              (Proc(x, e, f'), v) => eval (update f' x v) e
                            | _                   => raise Error "R App")
```

Aufgabe 11.3
Wie Sie in der konkreten Syntax von SML (Seite 33) sehen können, sind bei Projektionen nur Konstanten als Projektionsindex erlaubt:

```
fun projSecond t = #2 t (* Erlaubt *)
fun proj n t = #n t (* Nicht erlaubt *)
```

(a) In welcher Phase der Analyse wird der Interpreter die zweite Prozedur zurückweisen?

(b) Warum tut der Interpreter uns nicht einfach den Gefallen und erlaubt mehr als nur Konstanten für n? Schließlich wäre ein solches Verhalten sehr praktisch...

Lösung 11.3:

(a) Der Intepreter sollte die zweite Deklaration in der Syntaxanalyse-Phase zurückweisen. Zwar kann er die Deklaration in einzelne Wörter zerlegen, allerdings gibt es keine Regel, die ihm erlaubt, den Prozedurrumpf als Ausdruck zu parsen. Anstelle der Variablen n verlangt die konkrete Grammatik nämlich eine positive Integer-Konstante. Tatsächlich wirft der Moscow ML Interpreter hier eine selbst für Tutoren kryptische Fehlermeldung...

(b) Das tut er deshalb nicht, weil er dann Probleme in der statischen Analyse bekommen würde: Da keine Typen angegeben sind, muss der Interpreter sich den Kontext anschauen, um herauszufinden, welchen Typ t haben soll. Offensichtlich muss es sich um einen Tupeltyp handeln, aber es ist unmöglich, herauszufinden, wie viele Stellen das Tupel haben muss, denn wer weiß schon, wie groß

die Werte, mit denen *proj* aufgerufen wird, am Ende höchstens sind? Und Tupeltypen mit variabler Länge gibt es in SML (aus guten Gründen) nicht.

Aufgabe 11.4 (*Verständnis*)
Betrachte die folgende "Lösung" für die *Aufgabe 12.2 a)* vom aktuellen (regulären) Übungsblatt:

```
fun closed' (Con_) l = true
|closed' (Id a) l = List.exists (fn n => n=a) l
|closed' (Opr(_,a,b)) l = closed' a l andalso closed' b l
| closed' (If(a,b,c)) l = closed' a l andalso closed' b l andalso closed' c l
| closed' (Abs(_,_,a)) l = closed' a l
| closed' (App(a,b)) l = closed' a l andalso closed' b l

fun creatlist (Con _) = nil
|creatlist (Id _)= nil
|creatlist (Opr(_,a,b))= creatlist a @ creatlist b
|creatlist (If (a,b,c))= creatlist a @ creatlist b @ creatlist c
|creatlist (Abs(a,_,_))= [a]
|creatlist (App(a,b))= creatlist a @ creatlist b

fun closed e = closed' e (creatlist e)
```

(a) Stellen Sie ihr Verständnis für die Prüfung von geschlossenen Ausdrücken unter Beweis indem Sie erklären, warum die folgende Lösung nicht das gewünschte Resultat berechnet. Versuchen Sie darzustellen, an welcher Stelle der Autor dieses Programmes den Denkfehler gemacht hat.

(b) Betrachten Sie die Ergebnisse, die obiges Program hervorbringen kann. Fällt Ihnen ein Muster auf? Erklären Sie mithilfe von Aufgabenteil (a)

Lösung 11.4:

(a) Der Denkfehler liegt darin, dass der Autor vor der Auswertung des Ausdrucks zuerst sämtliche gebundene Bezeichner in einer Liste sammelt. Tatsächlich müssen bei einer Abstraktion zwar neue Bezeichner eingeführt werden können, allerdings gelten diese nach unseren Regeln für Gültigkeitsbereiche nur für diese Abstraktion und deren Unterausdrücke.
Durch das Einsammeln am Anfang werden sämtliche Gültigkeitsbereiche zu einem großen zusammengeführt, was zu falschen Ergebnissen führt.

(b) Weil jeder vorkommende Bezeichner von Anfang an in der Liste enthalten ist, wird jeder Ausdruck zu true auswerten, sofern keine Bezeichner genommen werden, die niemals vorkommen. Folgendes wäre damit zum Beispiel möglich:

```
If(Id a, App(Abs(Id a, Bool, Id a), Con True), Con False)
```

Zum Zeitpunkt des ersten Auftauchens des Bezeichners a existiert dieser noch garnicht. Dieser wird hypothetisch erst in der tieferliegenden Abstraktion eingeführt. Bestimmt man die Liste aller gebundenen Bezeichner von Anfang an so gilt der obige Ausdruck flschlicherweise als geschlossen.

2 Knobelecke

Aufgabe 11.5 (*Dieters statische Sinnfrage*)
Dieter Schlau fragt sich, ob eine statische Semantik überhaupt sinnvoll ist:

„Die Typsicherheit bei Prozeduranwendungen verbietet sinnvolle und auswertbare Ausdrücke, wie z.B. die *Aufgabe 12.15* zeigt. Die Typsicherheit auf nicht-prozeduralen Werten ist schlicht unnötig, Sprachen wie *JavaScript* zeigen doch, dass es auch ohne geht."

Hat Dieter Recht (so wie immer)? Diskutieren Sie!

Aufgabe 11.6 (*(für) verrückte Typen*)
Hinweis: Verzichten Sie in der folgenden Aufgabe auf die Verwendung von let-Ausdrücken und Ausnahmen.

(a) Betrachten Sie die folgenden Terme:

(i) $\forall A, B, C : A \to B \to C$
(ii) $\forall A : A \to A$
(iii) $\forall A, B : A \to B \to A$
(iv) $\forall A, B : A \to B \to B$
(v) $\forall A, B : A \to B$
(vi) $\forall A, B : (A \to B) \to A \to B$
(vii) $\forall A, B, C : (A \to B \to C) \to (A \to B) \to A \to C$

Betrachten Sie diese Terme als logische Ausdrücke, wobei Sie Großbuchstaben als logische Teilausdrücke und $\to$ als Implikation betrachten. Welche dieser Terme sind beweisbar wahre Aussagen?

(b) Betrachten Sie die Terme als Typen von SML-Prozeduren, wobei Sie die Großbuchstaben $A, B, C, ...$ als Ersatz für die bekannten griechischen Buchstaben $\alpha, \beta, \gamma, ...$ in Typangaben betrachten.

Für welche Terme können Sie geschlossene Abstraktionen angeben, die den jeweiligen Typ haben? Geben Sie diese an oder begründen Sie kurz, warum dies nicht möglich ist.

(c) Vergleichen Sie Ihre Ergebnisse der beiden Teilaufgaben. Können Sie den Zusammenhang erklären?

(d) Wir wollen nun auch das logische Und und das logische Oder hinzufügen.

(i) Deklarieren Sie einen Datentyp *lAnd*, so dass Sie *genau dann* eine geschlossene Abstraktion des Typs $(\alpha, \beta)\, lAnd$ angeben können, wenn sie sowohl geschlossene Abstraktionen des Typs α als auch geschlossene Abstraktionen des Typs β angeben können.
(ii) Deklarieren Sie einen Datentyp *lOr*, so dass Sie *genau dann* eine geschlossene Abstraktion des Typs $(\alpha, \beta)\, lOr$ angeben können, wenn sie entweder geschlossene Abstraktionen des Typs α angeben können oder geschlossene Abstraktionen des Typs β angeben können.
(iii) Geben Sie je eine geschlossene Abstraktion an, die die folgenden Aussagen darstellen:
i. $\forall A : (A \to A) \land (A \to A \to A)$
ii. $\forall A, B : (A \to B) \lor (A \to A)$

Verwenden Sie explizite Typangaben, falls nötig.
(iv) Begründen Sie kurz, warum Sie keine geschlossene Abstraktion für die Aussage $\forall A, B : (A \to B) \land (A \to A)$ angeben können.

(e) Nun wollen wir die wahre Aussage $\top$ und die falsche Aussage $\bot$ darstellen. Dazu verwenden wir eigene Datentypen.

(i) Deklarieren Sie einen (möglichst einfachen) Datentyp *true* und geben Sie einen Ausdruck des Typs *true* an.
(ii) Deklarieren Sie einen Datentyp $false$, so dass sie keinen Ausdruck des Typs $false$ angeben können. Begründen Sie Ihre Entscheidung kurz.
(iii) Können Sie eine geschlossene Abstraktion des Typs $\forall A : false \to A$ angeben? Ist die Aussage $\bot \to A$ beweisbar wahr?

Dieter Schlau enters the stage.

(f) Dieter Schlau fragt sich, warum immer geschlossene Abstraktionen gefordert waren, warum also Rekursion ausgeschlossen ist.

Deklarieren Sie eine Variable, die einen zur Aussage $\forall A, B : A \to B$ passenden Typ hat.

Lösung 11.6:
Diese Lösung ist unvollständig. Falls Sie Fragen haben, wenden Sie sich an Ihren Tutor.

```
datatype ('a,'b) lOr = Left of 'a | Right of 'b
datatype ('a,'b) lAnd = And of ('a * 'b)

datatype true = I
datatype false = INFINITY of false
```