Max Wisniewski , Alexander Steen

Tutor: Ansgar Schneider

Aufgabe 1

Angenommen, dass eine abstrakte Maschine und ihre Überführungsfunktion *delta* gegeben sind.

a) Implementieren Sie die Semantikfunktion O, die jeder Programm-Daten-Kombination die entsprechende Ausgabe zuordnet.

Lösung:
tbd

b) Testen Sie Ihre Semantikfunktion O am Beispiel der WSKEA Maschine und Δ.

Lösung:
tbd

Aufgabe 2

a) Implementieren Sie die Reduktionsfunktion von WHILE in einer Programmiersprache Ihrer Wahl.

Lösung:

```
redT :: (T, (Map I Z, [K], [K])) -> (T, (Map I Z, [K], [K]))
redT ((Z z), t)           = ((Z z), t)
redT ((Id i), (s,e,a))     =
    case Map.lookup i s of
        Just (z)  -> ((Z z), (s,e,a))
        Nothing     -> ((Id i), (s,e,a))
redT ((TApp v1 op v2), t) =
    let (r1, t')                = redT (v1, t)
        (r2, t'')               = redT (v2, t') in
            case (r1,r2) of
                (Z z1, Z z2)    -> ((Z (decodeOP op z1 z2)),
                   t'')
                (lE, rE)        -> ((TApp lE op rE), t'')
redT (TRead, (s,((LiteralInteger z):e), a))  = ((Z z), (s,e,a
   ))
redT (TRead, t)           = (TRead, t)


redB ((Literal b), t)           = ((Literal b), t)
redB ((Not b), t)               =
    case redB (b, t) of
        ((Literal b'), t')   -> ((Literal (not b')), t')
        (b', t')             -> (b', t')
redB ((BApp b1 bop b2), t)      =
    let (r1, t')                = redT (b1, t)
        (r2, t'')               = redT (b2, t') in
            case (r1,r2) of
                (Z z1, Z z2)    -> ((Literal (decodeBOP bop
                   z1 z2)), t'')
```

```
                (lE, rE)          -> ((BApp lE bop rE), t'')
redB (BRead, (s, ((LiteralBool b):e), a)) = ((Literal b), (s,
   e,a))
redB (BRead , t)                  = (BRead, t)

red :: (C, (Map I Z, [K], [K])) -> (C, (Map I Z, [K], [K]))
red (Skip, t)                     = (Skip, t)
red ((Assign i t), (s,e,a))       =
    let (v,(s',e',a')) = redT (t,(s,e,a)) in
        case v of
            (Z z)                 -> (Skip, (Map.insert i z s',
                e', a'))
            _                     -> error ("Couldn't match "
               ++ (show t) ++" with Integer in "++(show i)++"
               :="++(show t))
red ((Seq c1 c2), t)              =
    let (cr, t') = red (c1, t)       -- In our implementation
       could only be Skip
        (cr', t'') = red (c2, t')    -- Same again
    in
        case (cr,cr') of
            (Skip, Skip)           -> (Skip, t'')
            _                      -> error ("Couldn't match"++(
               show c1)++"; "++(show c2)++"to a real
               execution")
red ((If b c1 c2), t)              =
    case redB (b, t) of
        ((Literal True), t')      -> red (c1, t')
        ((Literal False), t')     -> red (c2, t')
        _                         -> error ((show b)++" in the
           expression "++(show (If b c1 c2))++"was no boolean
           ")
red ((While b c), t)              =
    case redB (b,t) of
        ((Literal True), t')      ->
            let (res, t'') = red (c, t') in
                case res of
                    (Skip)        -> red ((While b c), t'')
                    _             -> error ("Couldn't evaluate
                       ("++(show c)++") in ("++(show (While b
                        c))++")")
        ((Literal False), t'')    -> red (Skip, t'')
        _                         -> error ("Couldn't evaluate
           ("++(show b)++") in ("++(show (While b c))++")")
red ((BOut b), t)                 =
    let (r, (s,e,a)) = redB (b,t) in
        case r of
            (Literal bool)  -> (Skip, (s,e,((LiteralBool bool
               ):a)))
            _               -> error ("Type missmatch, 
               expected bool in ("++(show b)++"), in ("++(
               show (BOut b))++")")
red ((TOut v), t)                 =
    let (r, (s,e,a)) = redT (v,t) in
        case r of
            (Z z)             -> (Skip, (s,e,((LiteralInteger z
               ):a)))
            _                 -> error ("Type missmatch, 
               expected bool in ("++(show v)++"), in ("++(
               show (TOut v))++")")
```

Wir haben uns dafür entschieden, da es nicht anders gebraucht wurde die transitive Hülle in die Reduktion einzubauen. Die Reihenfolge der Ausführung bliebt dadurch erhalten. Desweiteren geben wir in einem Fehlerfall halbwegs detailiert zurück, wo der Fehler aufgetreten ist. Wir könnten zur Sicherheit auch noch einmal den Halbausgewerteten Ausdruck zurückgeben.
Bei redT und redB, werten wir die Terme aus, so wie wir es können. Treffen wir

einmal als Rückgabe auf keinen Grundtyp, so bauen wir unser Statement wieder zusammen und geben es zurück. Im Hauptprogramm nun wird immer wenn so etwas passiert ein Fehler geworfen.

b) Implementieren Sie die Semantikfunktion *eval*, die jeder Programm-Daten-Kombination die entsprechende Ausgabe zuordnet.

Lösung:

```
eval :: C -> [K] -> [K]
eval prog eing  =
    case red (prog, (Map.empty, eing, [])) of
        (Skip, (_,_,aus))   -> aus
        _                   -> error ("The␣programm␣wasn't␣
            evaluated␣complete")
```

Da wir in der Reduktion schon die transitive Hülle gebildet haben, müssen wir an dieser Stelle nur prüfen, ob das Programm wirklich fertig reduziert wurde und können dann die ausgabe zurückgeben.

c) Testen Sie Ihre Funktion *eval* am Beispiel der ganzzahligen Division.

Ergebnis:

```
RedSem> eval divTest [(LiteralInteger 10),(LiteralInteger 2)]
[5]
RedSem> eval divTest [(LiteralInteger 200),(LiteralInteger
   13)]
[15]
```

Die beiden kurzen Testfälle ergben genau das, was wir erwarten, solange wir uns nicht selbst verrechnet haben sollten.

Aufgabe 3

Gegeben sei folgende Syntax

$$\begin{array}{lcl} \text{W} & := & \text{True} \mid \text{False} \\ \text{LOP} & := & \text{AND} \mid \text{OR} \\ \text{LA} & := & \text{W} \mid LA_1 \text{ LOP } LA_2 \mid \text{NOT LA} \end{array}$$

Anmerkung: Uns war es freigestellt uns eine Semantik zu überlegen. Der einfachste Weg wäre es, jeden Ausdruck, den wir bekommen auf True abzubilden. Damit wäre es semantisch völlig spezifiziert. Wir haben uns aber an eine etwas sinnvollere doch nicht zu komplizierte Semantik gesetzt. Da wir uns wörtlich an die Befehle halten, haben wir daher keine Ausgabe. Man könnte sich aber überlegen die Zwischenschritte in die Ausgabe zu schreiben. Dies werden wir im Programmier part auch tun um nachzuvollziehen, ob das richtige passiert.

a) Definieren Sie eine geeignete operationelle Semantik.
Lösung:

Wir führen eine weitere Menge von Kontroll-Symbolen für die abstrakte Maschiene ein $CON = \{\underline{CAND}, \underline{COR}, \underline{CNOT}\}$. Damit ist der Zustandsraum unserer Abstrakten Maschiene $S = CON \cup W \cup LOP \cup LA$. Nun definieren wir die Semantikfunktion Δ. Der Speicher ist die leere Funktion $WK = \emptyset$, da wir keine Werte speichern können. Der Wertekeller S ist eine Liste von Weten W und E, A sind Listen von Werten W. Wobei diese nicht gebaucht werden.

$$
\begin{array}{lcl}
\Delta \ : \ WK \times S \times K \times E \times A & \Longrightarrow & W \times S \times K \times E \times A \\
\Delta \, \langle w|s|\varepsilon|e|a\rangle & \equiv & \langle w|s|\varepsilon|e|a\rangle \\
\Delta \, \langle w|s|True.k|e|a\rangle & \equiv & \langle w|True.s|k|e|a\rangle \\
\Delta \, \langle w|s|False.k|e|a\rangle & \equiv & \langle w|False.s|k|e|a\rangle \\
\Delta \, \langle w|s|(t1ANDt2).k|e|a\rangle & \equiv & \langle w|s|t1.t2.CAND.k|e|a\rangle \\
\Delta \, \langle w|s|(t1ORt2).k|e|a\rangle & \equiv & \langle w|s|t1.t2.COR.k|e|a\rangle \\
\Delta \, \langle w|s|(NOTt1).k|e|a\rangle & \equiv & \langle w|s|t1.CNOT.k|e|a\rangle \\
\Delta \, \langle w|v1.v2.s|CAND.k|e|a\rangle & \equiv & \langle w|(v2 \wedge v1).s|k|e|a\rangle \\
\Delta \, \langle w|v1.v2.s|COR.k|e|a\rangle & \equiv & \langle w|(v2 \vee v1).s|k|e|a\rangle \\
\Delta \, \langle w|v.s|CNOT.k|e|a\rangle & \equiv & \langle w|(\neg v).s|k|e|a\rangle
\end{array}
$$

Dies sind alle Überführung die wir benötigen. Da die Syntax und die Semantik keine Ausgabe vorsieht , ist die Semantikfunktion einfach eine Funktion, die alles auf das leere Wort schickt.
$O \ : \ LA \times W \Rightarrow W, \ (prog, eing) \mapsto \varepsilon.$

b) Definieren Sie eine geeignete Reduktionssemantik.
Lösung:

Wir übernehmen den Speicher, Ein- und Ausgabe aus Aufgabenteil a). Es bleibt zu erwähnen, dass wir uns, wie in Aufgabe 2 dafür entschieden haben, den transitiven Abschluss (transitive Hülle) gleich mit in die Definition aufzunehmen. Die Auswertung ist hier wieder links vor rechts.

$$
\begin{array}{rcl}
(True, (s, e, a)) & \Longrightarrow & (True, (s, e, a)) \\
(False, (s, e, a)) & \Longrightarrow & (False, (s, e, a)) \\
(t1ANDt2, (s, e, a)) & \Longrightarrow & (v1 \wedge v2, (s'', e'', a'')) \\
& \text{where} & (t1, (s, e, a)) \overset{*}{\Longrightarrow} (v1, (s', e', a')) \\
& & (t2, (s', e', a')) \overset{*}{\Longrightarrow} (v2, (s'', e'', a'')) \\
(t1ORt2, (s, e, a)) & \Longrightarrow & (v1 \vee v2, (s'', e'', a'')) \\
& \text{where} & (t1, (s, e, a)) \overset{*}{\Longrightarrow} (v1, (s', e', a')) \\
& & (t2, (s', e', a')) \Longrightarrow (v2, (s'', e'', a'')) \\
(NOTt, (s, e, a)) & \Longrightarrow & (\neg v, (s', e', a')) \\
& \text{where} & (t, (s, e, a)) \overset{*}{\Longrightarrow} (v, (s', e', a'))
\end{array}
$$

Wie in a) macht die Ausgabe bei uns nichts. Daher wird die Semantikfunktion *eval* auch ε ausgeben.

c) Beweisen Sie die Äquivalenz der Lösungen von 3 a) und b).
Lösung:

Wir zeigen Induktiv, dass der erste Fixpunkt von Δ und der transitive Abschluss von $\Longrightarrow$, die selbe Variablenbelegung, die selbe Ausgabe und die selbe restliche Eingabe hat. Die Spitze des Wertekellers muss dabei den selben Wert haben, wie der Zustand des Programms nach der Ausführung ist.

I.A. Auswertung von Konstanten W:

$$
\begin{array}{rcl}
\Delta \, \langle w|s|True.\varepsilon|e|a\rangle & = & \langle w|True.s|\varepsilon|e|a\rangle \\
(True, (w, e, a)) & \Longrightarrow & (True, (w, e, a))
\end{array}
$$

Wir sind im selben Zustand gestartet mit dem selben Programm. Zurück kam beide male der selbe Wert.

$$\begin{array}{rcl} \Delta\,\langle w|s|False.\varepsilon|e|a\rangle & = & \langle w|False.s|\varepsilon|e|a\rangle \\ (False,(w,e,a)) & \Longrightarrow & (False,(w,e,a)) \end{array}$$

Hier, wie im anderen Fall, bekommen wir den selben Rückgabewert. Eingabe, Ausgabe und Wörterbuch unterscheiden sich nicht.

I.V. Wir können Ausdrücke LA der Tiefe (Abstrakte Syntax) $n-1$ auswerten.

I.S. $n \rightsquigarrow n+1$ $\Delta\,\langle w|s|(t1opt2).k|e|a\rangle = \langle w|s|t1.t2.op'.k|e|a\rangle$, da $t1$, nun ein Syntaxbaum maximal die Tiefe $n-1$ hat, wissen wir, dass es eine Funktionsfolge, der länge $l1$ gibt, so dass $\Delta^{(l1)}\,\langle w|s|t1.t2.op'.k|e|a\rangle = \langle w'|v1.s|t2.op'.k|e'|a'\rangle$ äquivalent zu einer Reduktion $(t1,(w,e,a)) \Longrightarrow (v1,(w',e',a'))$ ist. Da auch $t2$ maximal $n-1$ tief ist, existiert ebenso eine Funktionfolge $\Delta^{(l2)}\,\langle w'|v1.s|t2.op'.k|e'|a'\rangle = \langle w''|v2.v1.s|op'.k|e''|a''\rangle$, die äquivalent zu einer Reduktion $(t2,(w',e',a')) \Longrightarrow (v2,(w'',e'',a''))$ ist. Nun können wir auf der Seite von Δ noch einen Schritt ausführen und wir erhalten $\Delta^{(l1+l2+2)}\,\langle w|s|(t1opt2).k|e|a\rangle = \langle w|(t1\underline{op}t2).s|k|e|a\rangle$. Auf der Seite der Reduktion haben wir aus der Reduktion der Teilschritte die nötige Vorraussetzung, um $t1opt2$ reduzieren zu können. $(t1opt2,(w,e,a)) \Longrightarrow (v1\underline{op}v2,(w'',e'',a''))$. Aller Vorraussetzungen sind erfüllt. Wir wissen nun, dass wir nach $l1+l2+2$ Schritten der Δ Funktion den selben Wert, wie die Reduktion erreichen.

$\Delta\,\langle w|s|(NOTt).k|e|a\rangle = \langle w|s|t.NOT.k|e|a\rangle$. In diesem Ausdruck, ist $t1$ wiederum ein Syntaxbaum, mit maximaler Tiefe $n-1$. Daher wissen wir, dass es eine Reduktion $(t,(w,e,a)) \Longrightarrow (v,(w',e',a')$ gibt, die äquivalent zu $\Delta^{(l)}\,\langle w|s|t.k|e|a\rangle = \langle w'|v|NOT.k|e'|a'\rangle$ ist mit $l \geq 0$. Nach Δ Funktion, können wir es noch einen Schritt ableiten zu $\Delta^{(l+1)}\,\langle w|s|(Nott).k|e|a\rangle = \langle w'|(Notv).s|k|e'|a'\rangle$. Bei der Reduktion haben wir die Vorraussetzungen erfüllt und können nun nach der Regel folgern, dass folgende Reduktion gilt $(Notv,(w',e',a'))$.

Nach Struktureller Induktion kann das eine auf das andere Abgebildet werden. Wenn wir eine Semantikfunktion hätten, könnten wir an dieser Stelle auch sehen, dass die Ausgaben von beiden Spezifikationen identisch ist. Allderdings bilden beide ohnehin auf das leere Wort ab.

a) Vereinbaren Sie einen geeigneten Datentyp LA zur Darstellung von logischen Ausrücken.
Lösung:

Wie immer ist die Definition der Datentypen in Haskell reine Abschreibe arbeit.

```
type W      = Bool
data LOP    = AND | OR | NOT deriving (Show,
   Eq)
data LA     = Value W | App LA LOP LA | Not
   LA  deriving Eq
data M      = MLA LA | FUN LOP deriving (Eq,
   Show)
```

Damit wir später nichts zu tun haben, ist NOT schon Teil von LOP. Bei der AUswertung in Termen wird es später einen Fehler werfen, aber so sparen wir uns extra Kontrollwörter für den Stack der WSKEA Maschine.

b) Implementieren Sie die operationelle und die Reduktionssemantik gemäß Ihrer Lösungen zu 3 a) und b).
Lösung:

Schauen wir uns zuerst die delta Funktion an:

```
delta :: ([W],[M],[String]) -> ([W],[M],[String])
delta (s,((MLA (Value b)):k), a)          = ((b:s),k,a)
delta (s,((MLA (App t1 op t2)):k),a)      = (s, ((MLA t1):(MLA
   t2):(FUN op):k),a)
delta (s,((MLA (Not t)):k),a)             = (s, ((MLA t):(FUN
   NOT):k),a)
delta ((b:s),((FUN NOT):k),a)
    = (((not b):s),k,(("not␣("++(show b)++")␣=␣"++(show.not $
       b)):a))
delta ((b2:b1:s),((FUN AND):k),a)
    = (((b1 && b2):s),k,(((show b1)++"␣&&␣"++(show b2)++"␣=␣"
      ++(show (b1&&b2))):a))
delta ((b2:b1:s),((FUN OR):k),a)
    = (((b1 || b2):s),k,(((show b1)++"␣||␣"++(show b2)++"␣=␣"
      ++(show (b1||b2))):a))
delta k                                   = k
```

Prinzipiell ist es wie in Aufgabe 3a) gezeigt. Wir schreiben blos bei konkreten Operationen, die Operation samt Ergebniss zur Kontrolle für später in die Ausgabe.

Die Reduktionsfunktion ist auch nicht viel spannender:

```
red :: (LA, [String]) -> (LA, [String])   -- Programm und
   Ausgabe
red (Value b, a)    = (Value b, a)
red (App t1 AND t2, a')  =
    let (v1,a'')     = red (t1,a')
        (v2,a)       = red (t2,a'')
    in
        case (v1,v2) of
            (Value w1, Value w2)
                -> (Value (w1 && w2), (((show w1)++"␣&&␣"++(
                   show w2)++"␣=␣"++(show (w1&&w2))):a))
            _   -> error "no␣value␣returned"
red (App t1 OR t2, a')   =
    let (v1,a'')     = red (t1,a')
        (v2,a)       = red (t2,a'')
    in
        case (v1,v2) of
            (Value w1, Value w2)
                -> (Value (w1 || w2), (((show w1)++"␣||␣"++(
                   show w2)++"␣=␣"++(show (w1||w2))):a))
            _   -> error "no␣value␣returned"
red (Not t1, a')    =
    let (v1,a)  = red (t1,a') in
        case v1 of
            (Value w)   -> (Value (not w),(("not␣("++(show w)
               ++")␣=␣"++(show.not $ w)):a))
            _           -> error "no␣value␣returned"
```

Die Fehler sollten nie auftreten, da wir wie man sich überzeugen kann, immer einen Value zurückgeben.

Beide Implementierungen sind analog zur Aufgabe 3. Danach haben wir noch 2 einfache tests geschrieben, sowie die Semantikfunktionen eval und O.

```
eval :: LA -> [String]
eval z = snd.red $ (z,[])
```

```
anfang :: LA -> ([W],[M],[String])
anfang prog = ([],[(MLA prog)],[])

fix :: ([W],[M],[String]) -> ([W],[M],[String])
fix z   =
    case delta z of
        z   -> z
        z'  -> fix z'

o :: ([W],[M],[String]) -> [String]
o z =
    let (_,_,aus) = fix z in
        aus

test1, test2 :: LA
test1   = App (Value True) AND (App (Value False) OR (Value
   True))
test2   = Not (App (Value True) AND (App (Value True) AND (
   App (Value False) OR (Value False))))
```

Und zu guter letzt die beiden Tests ausgeführt.

```
LA> o.anfang $ test1
["True␣&&␣True␣=␣True","False␣||␣True␣=␣True"]
LA> eval test1
["True␣&&␣True␣=␣True","False␣||␣True␣=␣True"]
LA> fix.anfang $ test1
([True],[],["True␣&&␣True␣=␣True","False␣||␣True␣=␣True"])
LA> red (test1,[])
(True,["True␣&&␣True␣=␣True","False␣||␣True␣=␣True"])
LA> o.anfang $ test2
["not␣(False)␣=␣True","True␣&&␣False␣=␣False","True␣&&␣False␣
   =␣False","False␣||␣False␣=␣False"]
LA> eval.anfang $ test2
["not␣(False)␣=␣True","True␣&&␣False␣=␣False","True␣&&␣False␣
   =␣False","False␣||␣False␣=␣False"]
LA> fix.anfang $ test2
([True],[],["not␣(False)␣=␣True","True␣&&␣False␣=␣False","
   True␣&&␣False␣=␣False","False␣||␣False␣=␣False"])
LA> red (test2,[])
(True,["not␣(False)␣=␣True","True␣&&␣False␣=␣False","True␣&&␣
   False␣=␣False","False␣||␣False␣=␣False"])
```

Wie wir in Aufgabe 3 bewiesen haben, bestätigt sich hier, dass zumindest diese einfachen Tests, das selbe Ergebnis liefern.