Zwischenbericht Compilerbau Gruppe 7
Listen

Christof Weibel Benjamin Neukom

15. November 2013 Version 1.0

Dieses Dokument beschreibt, wie die Sprache IML um Listen erweitert wird und welche Änderungen hierfür am Compiler und an der Virtual Machine gemacht werden müssen. Die Spracherweiterung wird mit den Sprachen Scala und Haskell verglichen und die Designentscheide werden erläutert.

1 Listen als Spracherweiterung

Listen für IML sind wie folgt definiert

1. Listen sind geordnete Ansammlungen von Objekten mit dem gleichen Datentyp T. Listen bestehen aus einem Head vom Typ T und einem Tail einer Liste vom Typ T. Beispiel: $[1,2,3]$ ist eine Liste vom Typ *int* mit den Werten 1, 2 und 3 (Head ist 1 und Tail ist $[2,3]$). Oder mit Verschachtelung $[[true, false], [true]]$ ist eine Liste vom Typ Liste von bools mit den Werten $[true, false]$ und $[true]$.

2. Stores vom Typ Liste können mit Brackets definiert werden. Beispiel: *var l* : [*int*] ist ein Store vom Typ Liste von ints.

3. Der Typ einer Liste ist definiert durch das erste Element (falls das Element eine Liste ist, das erste nicht leere Element) der Liste. Beispiel: $[[1,2,3],[3,4,5]]$ ist eine Liste vom Typ Liste von *int*s oder $[[], [true]]$ ist eine Liste vom Typ Liste von bools.

4. Die leere Liste [] ist vom Typ *Any* (welcher in IML nicht verwendet werden kann). Der Typ *Any* ist kompatibel mit allen anderen Typen. Beispiel: für die Deklaration des Stores *var l* : [[*int*]] ist $l := []$ (Typ von [] ist [*Any*] und *Any* ist kompatibel mit [*int*]) so wie auch $l := [[]]$ (Typ von [[]] ist [[*Any*]] und *Any* ist kompatibel mit *int*) eine gültige Zuweisung.

5. Listen sind immutable, d.h. die Werte einer Liste können nicht verändert werden. Es können nur mit den Operationen :: (Konkatenation) und *tail* neue Listen konstruiert werden.

2 Operationen auf Listen

2.1 Konkatenation

$$Type :: [Type] \rightarrow [Type]$$

Der :: Operator erstellt eine neue Liste mit dem Operand auf der Linken Seite als Head und dem Operand der rechten Seite als Tail. Beispiel: die Expression $1 :: [2, 3, 4]$ gibt die Liste $[1, 2, 3, 4]$ zurück. Der :: Operator ist rechtsassoziativ. Beispiel: die Expression $1 :: 2 :: 3 :: []$ gibt die Liste $[1, 2, 3]$ zurück.

2.2 Head

$$head\ [Type] \rightarrow Type$$

Der *head* Operator gibt den Head einer Liste zurück. Beispiel: die Expression *head* $[2, 3, 4]$ gibt den Wert 2 zurück. Falls versucht wird, den Head einer leeren Liste abzufragen, wirft die VM eine Exception.

2.3 Tail

$$tail\ [Type] \rightarrow [Type]$$

Der *tail* Operator gibt den Tail einer Liste zurück. Beispiel: die Expression *tail* $[1, 2, 3, 4]$ gibt die Liste [2,3,4] zurück. Die Tail Operation kann auch eine leere Liste zurückgeben. Beispiel: die Expression *tail* [1] gibt die leere Liste [] zurück.

2.4 Length

$$length\ [Type] \rightarrow Int$$

Der *length* Operator gibt die Länge einer Liste zurück. Beispiel: *length* $[1, 2, 3, 4, 5]$ gibt den Wert 5 zurück.

3 Vergleich mit Haskell und Scala

3.1 Haskell

Die Operatoren *head*, *tail* und *length* verhalten sich identisch zu den Haskell Varianten. Einen Unterschied zu Haskell ist die Prezedenz des konkatenations Operator. Bei der IML Erweiterung hat der Operator :: die tiefste Priorität und bei Haskell liegt die Priorität zwischen den boolschen, logischen und den arithmetischen Operatoren. Dies hat folgende Konsequenzen:

Beispiel für Haskell:

```
1 > 3 : True : [] // Type Error
(1 > ((3 : True) : [])) // Mit Klammerung um Operator Prezedenz zu zeigen
```

Listing 1: Ungültige Listen Konkatenation in Haskell

Gleiches Beispiel in IML:

```
1 > 3 :: true : [] // Keinen Type Error
((1 > 3) :: (true :: [])) // Mit Klammerung um Operator Prezedenz zu zeigen
```

Listing 2: Gültige Listen Konkatenation in IML

Wir haben uns für diese Prezedenz entschieden, da es unserer Meinung nach natürlicher ist, dass der :: Operator die tiefste Priorität hat. Ausserdem haben wir noch kein Beispiel gefunden, wo die Operator Prezedenz wie sie in Haskell implementiert ist, einen Vorteil gegenüber unserer Implementation bietet.

3.2 Scala

In Scala verhält sich die Operator Prezedenz des :: Operator gleich wie bei Haskell, also unterschiedlich zu IML.

4 Änderungen an der Grammatik

Folgende Änderungen wurden an der Grammatik vorgenommen. Die Änderungen wurden mit einem eigens entwickelten Tool (mehr dazu später) getestet und sind *LL*(1) konform.

4.1 Expression

Vorher:

expr ::= *term1* {BOOLOPR term1}
term1 ::= *term2* [RELOPR term2]
term2 ::= *term3* {ADDOPR term3}
term3 ::= *factor* {MULTOPR factor}
factor ::= *literal*
| IDENT [INIT | *exprList*]
| monadicOpr *factor*
| LPAREN *expr* RPAREN;
exprList ::= LPAREN [*expr* {COMMA *expr*}] RPAREN
monadicOpr::= NOT | ADDOPR

Nachher:

expr ::= *term0* {CONCATOPR term0}
term0 ::= *term1* {BOOLOPR term1}
term1 ::= *term2* [RELOPR term2]
term2 ::= *term3* {ADDOPR term3}
term3 ::= *factor* {MULTOPR factor}
factor ::= *literal*
| IDENT [INIT | *exprList*]
| monadicOpr *factor*
| LPAREN *expr* RPAREN;
exprList ::= LPAREN [*expr* {COMMA *expr*}] RPAREN
monadicOpr::= NOT | ADDOPR | HEAD | TAIL | SIZE

4.2 Type

Vorher:

atomType ::= INT | BOOL

Nachher:

type ::= *atomType* | LBRACKET *type* RBRACKET
atomType ::= INT | BOOL

4.3 Literal

Vorher:

literal ::= INTLITERAL | BOOLLITERAL

Nachher:

literal ::= INTLITERAL | BOOLLITERAL | *listLiteral*
listLiteral ::= LBRACKET [*expr* {COMMA *expr*}] RBRACKET

5 Grammatik auf LL(1) Fehler überprüfen

Die Grammatik wurde, mit einem eigens entwickelten Tool, auf LL(1) Fehler überprüft. Das Tool akzeptiert eine EBNF Grammatik (wie im Unterricht besprochen) und wandelt diese in eine normale Grammatik um. Danach werden die $NULLABLE$, $FIRST$ und $FOLLOW$ Sets berechnet und damit die Parse Tabelle generiert.

Grammatik der EBNF Grammatik:

grammar ::= *production* SEMICOLON *production*;
production::= NTIDENT ASSIGN *term0*;
term0 ::= {*term1*} {PIPE {*term1*}};
term1 ::= *repTerm* | *optTerm* | *symbol*;
repTerm ::= LCURL *term0* RCURL;
optTerm ::= LBRAK *term0* RBRAK;
symbol ::= TIDENT | NTIDENT

6 Code Beispiele

Summe der Elemente einer int Liste:

```
program listSum()
global
fun sum(in copy l:[int]) returns var r:int
do
       if length l == 0 do
              r init := 0
       else
              r init := head l + sum(tail l)
       endif
endfun;

var l:[int];
var sum:int
do
       l = [1,2,3,4,5,6,7,8];
       sum = sum(l)
endprogram
```

Listing 3: Beispiel für die Berechnung der Summe der Element einer Liste in IML

Contains:

```
program listContains()
global

fun contains(in copy l:[int], in copy i) returns var r:boolean
do
       if length l == 0 do
              r init := false
       else
              r init := head l == i || contains(tail l, i)
       endif
endfun;


var l:[int];
var i:int;
var c:bool
do
       l init := [1,2,3,4,5,6,7,8];
       i init := 6;
       c init := contains(i)
endprogram
```

Listing 4: Listen Contains

Liste umkehren:

```
rogram listReverse()
global

// returns last element
fun last(in copy l:[int]) returns var r:int
do
	if length l == 1 do
		r init := head l
	else
		r init := last(tail l)
	endif
endfun;

// init for haskell (list without last)
fun initial(in copy l:[int]) returns var r:[int]
do
	if length l == 1 do
		r init := []
	else
		r init := head l :: initial(tail l)
	endif
endfun;

// would be easier with ++ operator
// reverses the given list and returns a new one
fun reverse(in copy l:[int]) returns var r:[int]
do
	if length l == 0 do
		r init := []
	else
		r init := last(l) :: reverse(initial(l))
	endif
endfun;


var l:[int];
var r:[int]
do
	l init := [1,2,3,4,5,6,7,8];
	r init := reverse(l)
endprogram
```

Listing 5: Liste reverse 1

```
rogram listReverse()
global

// reverse 2
fun reverse(in copy l:[int], in copy acc:[int]) returns var r:[int]
do
       if length l == 0 do
              r init := acc
       else
              r init := reverse(tail r, head l :: acc)
       endif
endfun;

var l:[int];
var r:[int]
do
       l init := [1,2,3,4,5,6,7,8];
       r init := reverse(l)
endprogram
```

Listing 6: Liste reverse 2

7 Quellen

Internet:

Wikipedia	en.wikipedia.org
Haskell Listen	http://andres-loeh.de/haskell/4.pdf
Haskell Language Specification	www.haskell.org/onlinereport/
Scala	www.scala.org

Bücher:

Programmming in Scala	Odersky et. al.