Schlussbericht Compilerbau
Gruppe 7
IML mit Listen

Christof Weibel Benjamin Neukom

15. November 2013
Version 1.0

Abstract

Dieses Dokument beschreibt, wie die Sprache IML um Listen erweitert wird, welche Änderungen hierfür am Compiler nötig sind und wie das IML Programm nach Java Bytecode übersetzt wird. 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 :: (Cons) und *tail* neue Listen konstruiert werden.

2 Operationen mit Listen

2.1 Cons

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

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. Falls versucht wird, den Tail einer leeren Liste abzufragen, wirft die VM eine Exception.

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 List Comprehension

Mit List Comprehensions können Listen erzeugt werden. Sie folgt der mathematischen Set-Builder Notation.

$$\{\ \underbrace{3 * x}_{\textit{Output Function}}\ |\ \underbrace{x}_{\textit{Counter}}\ \underbrace{from\ 0\ to\ 100}_{\textit{Range}}\ when\ \underbrace{x\ mod\ 2\ ==\ 0}_{\textit{Predicate}}\ \}$$

Output Function
: Funktion welche auf die vom Predicate akzeptieren Elemente angewendet wird. Die Output Function muss einen Wert vom Typ *int* zurückgeben.

Counter
: Der Zähler, welche die angegebene Range durchläuft. Er ist immer vom Typ *int*, deshalb muss dieser nicht explizit angegeben werden. Der zu dem Zähler gehörenden Store ist anonym, dass heisst, er kann nur innerhalb dieser List Comprehension verwendet werden.

Range
: Die Inputmenge, angegeben durch zwei Expressions welche einen *int* zurückgeben. Im Beispiel von 0 bis und mit 100. Der Schritt ist immer 1. Es ist auch möglich die Range in umgekehrter Reihenfolge durchzulaufen. Also mit einem From Wert von 100 und To Wert von 0.

Predicate
: Filter, welcher auf jedes Element der Range angewendet wird, um zu entscheiden ob es in der resultierenden Liste enthalten ist. Das Predicate muss einen Wert vom Typ *bool* zurückgeben.

Nachdem der Context-Checker das Programm überprüft hat, wird der AST der List Comprehensions in Commands umgewandelt. Beispiel:

$$x := \{\ 3\ *\ x \mid x\ from\ 0\ to\ 100\ when\ x\ mod\ 2\ ==\ 0\ \}$$

IML Code zu dem transformierten AST:

```
var $i:int;
var $l:[int]

...

// To-Value
i init := 100;

if 100 > 0 do
       while $i >= 0 do
              // predicate
              if $i mod 2 == 0 do
                     // output function
                     $l := (3 *$i) :: $l
              else
                     skip
              endif;
              $i := $i - 1
       endwhile

else
       while $i <= 0 do
              // predicate
              if $i mod 2 == 0 do
                     // output function
                     $l := (3 *$i) :: $l
              else
                     skip
              endif;
              $i := $i + 1
       endwhile

endif

x := $l
```

Listing 1: Transformation

Da für die Range zwei Expressions erwartet werden, kann nicht zur Kompilierzeit entschieden werden, ob der From Wert grösser ist als der To Wert, somit müssen beide Fälle behandelt werden. Bei der Code Generierung müssen List Comprehensions nicht mehr speziell behandelt werden.

4 Vergleich mit Haskell und Scala

4.1 Haskell

Die Operatoren *head*, *tail* und *length* verhalten sich identisch zu den Haskell Varianten. Einen Unterschied zu Haskell ist die Präzedenz des Cons-Operators. 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:

```
// Type Error
1 > 3 : True : []

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

Listing 2: Ungültige Cons Operation in Haskell

Gleiches Beispiel in IML:

```
// Keinen Type Error
1 > 3 :: true : []

// Mit Klammerung um Operator Praezedenz zu zeigen
((1 > 3) :: (true :: []))
```

Listing 3: Gültige Listen Konkatenation in IML

Wir haben uns für diese Präzedenz 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 Präzedenz wie sie in Haskell implementiert ist, einen Vorteil gegenüber unserer Implementation bietet.

4.2 Scala

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

5 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.

5.1 Expression

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
| listComprehension;
exprList ::= LPAREN [*expr* {COMMA *expr*}] RPAREN
monadicOpr::= NOT | ADDOPR | HEAD | TAIL | SIZE

5.2 List Comprehension

Neu:

listComprehension::= LCURL expr PIPE ident FROM expr TO expr WHEN expr RCURL

5.3 Type

Vorher:

atomType ::= INT | BOOL

Nachher:

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

5.4 Literal

Vorher:

literal ::= INTLITERAL | BOOLLITERAL

Nachher:

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

6 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

7 Code Generierung

Da für Listen ein Heap benötigt wird, haben wir uns entschieden für die Java Virtual Machine (JVM) zu kompilieren. Aus dem IML Programm wird Java Bytecode generiert.

7.1 Aufbau

IML Programm
Wird in eine Java Klasse mit dem gleichen Namen übersetzt

Globale Felder
Werden in private statische Variabeln übersetzt

Methoden und Prozeduren
Werden in statische Java Methoden übersetzt

IML Hauptprogramm
Wird in die Java Main-Methode übersetzt

7.2 Listen

Da die Element Anzahl von IML Listen unveränderbar ist, werden sie in Java Arrays umgewandelt. Dies hat zur Folge, dass bei einer *tail* oder *cons* Operation das Array kopiert werden muss.

7.3 Out Parameter

Da die JVM keine Out oder Ref Parameter unterstützt müssen diese speziell behandelt werden.
Für Out-Parameter wird der übergebene Store wird in ein Array der Länge Eins gespeichert. In der Prozedur werden alle Lese und Schreibe Operationen über dieses Array ausgeführt. Nach dem Aufruf der Prozedur wird der Parameter wieder aus dem Array gelesen und in den übergebenen Store geschrieben.
Ref Parameter wurden bei unserer Implementation nicht speziell behandelt. Sie funktionieren gleich wie Copy Out Parameter.

8 Resultate

Die im Zwischenbericht spezifizierten Listen wurden so umgesetzt und der Context-Checker wurde so erweitert, dass Type-Fehler von Listen erkennt werden.
Nicht implementiert wurden Programm-Paramter, Global-Imports und Ref Parameter. Auch ist der Memory Footprint von den Listen bei unserer Implementation sehr hoch, da wir Listen für die Operationen *tail* und *head* immer kopieren.

9 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 init := [1,2,3,4,5,6,7,8,9,10];
       sum init := sum(l);
       debugout sum
endprogram
```

Listing 4: Beispiel für die Berechnung der Summe der Element einer Liste in IML

List Contains:

```
program listContains()
global

fun contains(in copy l:[int], in copy i:int) returns var r:bool
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];
       debugout l;
       debugin i init;
       c init := contains(l,i);
       debugout c
endprogram
```

Listing 5: Listen Contains

Liste umkehren:

```
program 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 6: Liste reverse 1

Liste von Primzahlen mit List Comprehensions

```
program primesList()
global
	fun isPrime(in copy const p:int) returns var b:bool
	global
	local
		var c:int
	do
		c init := 2;

		if p > 1 do
			b init := true;
			while c < p do
				if p mod c == 0 do
					b := false
				else
					skip
				endif;
				c := c + 1
			endwhile
		else
			b init := false
		endif
	endfun;

	fun sum(in copy const l:[int]) returns var r:int
	local
	var x:int
	do
		if length l == 0 do
			r init := 0
		else
			r init := head l + sum(tail l)
		endif
	endfun;

	var l:[int];
	var max:int
do
	debugin max init;
	l init := { x | x from 0 to max when isPrime(x)};
	debugout l;
	debugout sum(l)
endprogram
```

Listing 7: Primzahlen Liste

```
program listReverse2()
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 l, 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 8: Liste reverse 2

```
program divisibility()
global
	var l:[[int]];
	var max:int;
	var counter:int
do
	debugin max init;

	l init := [];
	counter init := max;
	while counter > 0 do
		l := { x | x from 1 to max when x mod counter == 0} :: l;
		counter := counter - 1
	endwhile;

	debugout l
endprogram
```

Listing 9: Teilbarkeit

10 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:

Progranmming in Scala	Odersky et. al.