

Compilerbau 2014 - Gruppe 10 - Arrays und Array-Slicing

Idee

IML hat ursprünglich keine Datenstrukturen wie Tupel, Listen und Arrays. Wir wollten uns dem Thema Arrays annehmen. Uns war es vor allem wichtig, einige Gedanken zu konsistentem, intuitivem und lesbarem Syntax festzuhalten. Zu finden ist dies im nächsten Kapitel “Gedanken zum Syntax”.

Array Slicing ist das andere Thema, das uns interessiert hat. Es ist eine Spracherweiterung, die es ermöglicht mit wenig Quellcode einen Teil eines Arrays zu extrahieren und weiterzuverwenden. Beispielsweise so:

```
a := [3,1,4,1,5,9]
b := a[1..4]
```

In diesem Beispiel wird zuerst der Variable **a** ein Arrayliteral zugewiesen. Danach wird der Variable **b** eine Teilkopie von **a** zugewiesen. Spezifisch wäre es hier `[1,4,1,5]`. Diese syntaktische Abkürzung ermöglicht es einem, seinen Quellcode sehr konzis und intuitiv zu gestalten.

Wir haben noch ein paar weitere syntaktische Abkürzungen einbezogen, die es einem vereinfachen mit Arrays zu arbeiten. Zum Beispiel das auffüllen von Arrays mit einem spezifizierten Wert. Die lexikalische und syntaktische Spezifikation sind in den respektiven Kapiteln zu finden.

Gedanken zur Syntax

Warum so und nicht anders?

Zur Syntax haben wir eine ausführliche Diskussion geführt, und dabei verschiedene Varianten ent- und wieder verworfen.

Deklaration

Zum Vergleich zunächst die Deklaration einer Variablen, wie sie bereits in IML existiert:

```
var m:int;
```

Zunächst wird deklariert, dass es eine Variable (und nicht eine Konstante) ist, dann wird sie benannt und dann der Typ angegeben. Wir wollten uns möglichst nahe an diese Vorgabe halten, da wir nicht einfach Teile einer anderen Sprache in IML einbauen wollten, sondern IML erweitern.

Unser erster Versuch, dies zu erreichen, sah so aus:

```
var a:TYPE[LENGTH][DIMENSION];
```

TYPE würde hier INT oder BOOL sein, und so bestimmen, was der Typ der Elemente im Array ist. Obwohl diese Schreibweise durchaus elegant ist, macht sie es unmöglich, mehrdimensionale Arrays zu deklarieren, welche nicht in allen Dimensionen gleich lang sind. Zudem ist diese Schreibweise noch sehr deutlich von Sprachen wie Java beeinflusst, und führt die eckigen Klammern neu ein.

Als nächste Schreibweise überlegten wir uns, die folgende zu Verwenden:

```
var b:arr (array_decl) LENGTH;  
var b:arr (arr (arr (arr int 8) 5) 3) 10;  
var b:arr int 3;
```

Während dies für ein eindimensionales Array uns sehr intuitiv erscheint, wird es für mehrdimensionale Arrays eher verwirrend, da die Längenangaben dann in der “verkehrten” Reihenfolge dastehen.

Daher modifizierten wir diese Schreibweise und verlegten die Länge nach vorne:

```
var c:arr LENGTH (array_decl);  
var c:arr 10 (arr 5 (arr 6 (arr 2 int)));  
var c:arr 3 int;
```

Statt einem “Array von ints mit Länge 3” würde man also ein “Array, Länge 3, von ints” deklarieren. Dies schien uns nur wenig unintuitiver, und bei mehrdimensionalen Arrays deutlich besser. Ganz zufrieden waren wir aber noch nicht. Klammern zur Verschachtelung werden nämlich ab einigen Levels von Verschachtelung immer schlechter lesbar.

Da alle nicht-Array Elemente eines Arrays vom gleichen Typ sind, und das einzige verschachtelbare Arrays sind, kamen wir darauf, die Redundanzen zu entfernen:

```
var d:arr (4,10,5) int;  
var d:arr (3) int;
```

Die Länge der einzelnen Dimensionen ist das einzige, was sie unterscheidet und mit dieser Deklaration nutzen wir dies.

Array Initialisierung und Zugriff

Beim Initialisieren der Arrays haben wir uns sofort dafür entschieden, dass ein Array immer vollständig initialisiert werden muss, d.h. nicht teilweise unbestimmte Werte haben darf. Wir haben uns aber überlegt, dass es nützlich

sein könnte, Arrays leicht initializu machen. Dazu haben wir die folgende Syntax entworfen und das Keyword ‘fill’ eingeführt. Hier haben wir die eckigen Klammern dann doch eingeführt, da uns die Syntax mit ihnen genug deutlich besser lesbar vorkam.

```
var a: arr 7 int;  
var b: arr 3 int;  
a init := [0, 1, 2, 3, 4, 5, 6];  
b init := fill 0;
```

Bei erneuter Betrachtung haben wir keinen Grund gefunden, warum unsere Array nur “nullbar” sein sollten, und erlauben nun, ein Array komplett mit einem beliebigen Wert aufzufüllen, welcher auch durch eine Expression repräsentiert sein könnte.

```
var c: arr 6 int;  
c init := fill 5*3+1;
```

Beim Zugriff haben wir uns für eine Schreibweise entschieden, wie sie in verschiedenen anderen Sprachen verwendet wird, da sie uns gefällt, und so möglichst vielen Benutzern bekannt sein sollte. Wir indizieren aus demselben Grund auch von 0 aus und erlauben keine negativen Indices. Auch bei den Slices kommen negative Werte nicht vor.

```
var d: arr (4,2) bool;  
d init := [[true, true],[true, false],[false, true],[false,false]];
```

Dann hat `a[0]` den Inhalt `[true, true]`, und `a[0][1]` den Wert `true`.

Bevor wir nun zu den Array Slices kommen, sollte noch erwähnt werden, dass wir auch Array-Literale einführen, wie bei der Array-Initialisierung bereits gezeigt.

Array Slice Notation

Ein Array Slice ist ein erweiterter Zugriff, und soll auf möglichst simple Art erlauben, mehrere Elemente gleichzeitig zu adressieren. Daher übernahmen wir grundsätzlich die Syntax eines Array-Zugriffs, und erweiterten sie um ein Trennsymbol, ‘..’, und einen zweiten Index. Wir beschreiben Slices also über einen Start- und einen End-Index. In unserem Fall sind beide *inklusive* zu verstehen.

```
var a: arr 20 int;  
var b: arr 4 int;  
> snip  
b := a[0..3];
```

In diesem Beispiel nehmen wir daher die ersten vier Elemente eines 20-Element Arrays, und füllen sie in einen 4-Element Array. Das bedingt auch, dass Arrays nicht nur Array-Literale zugewiesen werden können, sondern auch Array-Slices, wobei die Länge und der Typ natürlich korrekt sein müssen.

Zudem sollten Slice zu Slice Zuweisungen möglich sein, also z.B:

```
var c: arr 20 int;
var d: arr 10 int;
> snip
d[5..8] := c[16..20];
```

Lexikalisch

Lexikalisch erweiterten wir die Syntax um die folgenden Tokens:

Pattern	Token
[LBRACKET
]	RBRACKET
fill	FILL
arr (int) type	(ARRAY,Length,Type)
..	DOTDOT

Grammatikalisch

Dies sind die Produktionen, die von der ursprünglichen IML Definition abweichen:

```
cmd      ::= SKIP
          | expr BECOMES [FILL] expr
          | IF expr THEN cpsCmd ELSE cpsCmd ENDIF
          | WHILE expr DO cpsCmd ENDWHILE
          | CALL IDENT exprList [globInits]
          | DEBUGIN expr
          | DEBUGOUT expr

typedIdent ::= IDENT COLON (ATOMTYPE | ARRAY LPAREN expr {COMMA expr} RPAREN TYPE)

factor    ::= LITERAL
          | arrayLiteral
          | IDENT [INIT | exprList | arrayIndex]
```

```

| monadicOpr factor
| LPAREN expr RPAREN

arrayIndex ::= LBRACKET expr [DOTDOT expr] RBRACKET {arrayIndex}

arrayLiteral ::= LBRACKET arrayContent RBRACKET

arrayContent ::= LITERAL {COMMA LITERAL}
               | arrayLiteral {COMMA arrayLiteral}

```

Kontext- und Typeinschränkung

Bei Arrays wie wir sie hier definiert haben, kommen keine neuartigen Typ- oder Kontexteinschränkungen dazu. Soll ein Element eines Arrays gesetzt werden, so muss der Typ des neuen Wertes zum Typ des Arrays passen und umgekehrt kann ein int-Literal nie durch ein Array ersetzt werden, sondern höchstens durch ein Element eines Arrays. Zudem könne in einem Array immer nur entweder weitere Arrays oder Literale enthalten sein. Bei einem multi-dimensionalen Array sind also alle Dimensionen bis zur letzten nur mit Arrays befüllt.

Nur beim Slicing kommt ein neuer Check hinzu, es müssen nämlich die Längen der Slices überprüft werden. Dies kann z.T. zur Compile-Zeit geschehen, aber, da wir auch Expressions zulassen beim Beschreiben eines Slices, nicht ausschliesslich.

Codeerzeugung (Final Report only)

— EMPTY FOR NOW —

Vergleich mit anderen Sprachen

Wir haben nach unserer Diskussion uns auch überlegt was wir eventuell von anderen Sprachen übernehmen wollen, und nachgelesen, wie Array Slicing von anderen umgesetzt wurde.

Dabei ist uns aufgefallen, dass wir die Syntax für Array Slicing nahezu genauso definieren wie Perl. Die Deklaration ist anders, und Perl bietet an, eine Expression zu verwenden welche z.B. nur jedes dritte Element zurückliefert statt nur eine Range. Dies erweitert aber Array Slicing zu einem generischeren Filter, was wir ablehnen, da wir hier eine klare Funktion einbauen wollen und nicht ein Schweizer Taschenmesser entwerfen. Perl vertritt da einen ganz anderen Standpunkt, weshalb diese Fähigkeit auch passt.

Ebenso ist Python sehr ähnlich, allerdings ist bei Python der End-Index exklusiv, d.h. `a[0:3]` liefert 3 Werte zurück. Uns erschien dies aber unintuitiv, da `a[3]` ja nicht das dritte, sondern das vierte Element liefern würde. Python hat auch eine sehr elegante Lösung für “jedes x-te Element”, indem eine Schrittlänge als drittes

Argument mitgeliefert werden kann: `a[0:9:2]` liefert das erste, dritte, fünfte, siebte und neunte Element. Wir entschieden uns dagegen, dies einzubauen, da wir solche Schrittlängen nicht für sehr nützlich erachten, und es uns widerstrebt die Sprache mit allzu vielen neuen Konstrukten und Funktionen zu überladen. In einer weiteren Erweiterung mögen Schrittlängen ein erstrebenswertes Feature sein, aber nicht in unserer.

Schlussendlich haben wir eine Syntax erarbeitet, wie sie alle Sprachen haben, welche Array Slicing unterstützen. Natürlich bestehen immer kleine Unterschiede, aber wir haben keine Sprache gefunden, welche eine komplett andere Syntax verwendet. Dadurch fühlen wir uns in unseren Designentscheiden auch etwas bestätigt, denn so sind wir uns relativ sicher, dass wir eine brauchbare und klare Syntax entworfen haben.

Appendix: Testprogramme

Beispiel für die Array Initialisierung und den Zugriff auf Elemente der Arrays.

```
// matrixmult.iml

program matrixMult

global
  var a : array (2,3) int;
  var b : array (3,2) int;
  var c : array (2,2) int;
  var i : int;
  var j : int;
  var k : int
do
  a init := [[1,2,3],[4,5,6]];
  b init := [[1,2],[3,4],[5,6]];
  c init := fill 0;
  i init := 0;
  j init := 0;
  k init := 0;

  while i > 2 do
    while j > 2 do
      while k > 3 do
        c[i][j] := c[i][j] + a[i][k] * b[k][j];
        k := k+1
      endwhile
      j := j+1
    endwhile
    i := i+1
  endwhile
```

```

        endwhile
        i := i+1
    endwhile

endprogram

```

Beispiel für Array Slicing

Der Input ist ein Array von 300 ints. Diese kommen von einer Wetterstation und sind eigentlich 100 Datenpunkte mit je 3 Werten, nämlich Datum (Sekunden seit Unix-Epoch), Maximaltemperatur (in Grad Celsius) und Niederschlag (in mm). Die Auswertung hier soll aufsummieren an wievielen Tagen es mehr als 5mm geregnet hat und die Temperatur trotzdem über 25 Grad Celsius gestiegen ist.

```

program parseInput(in const input: array (300) int,
                   out var result: int)
global
    var datapoint : array (3) int;
    var i : int
do
    datapoint init    := fill 0;
    i init := 0
    result init := 0;

    while i < 100 do
        datapoint := input[i * 3 .. i * 3 + 2]
        if datapoint[1] > 25 && datapoint[2] > 5 then
            result := result + 1;
        else
            endif
        i := i+1
    endwhile
endprogram

```