Arrays und Array-Slicing in IML

Compilerbau, HS2014 (Gruppe 10)

Autoren: Stark, Philip und Zeman, Mark

Abstract

IML hat ursprünglich keine Datenstrukturen wie Tupel, Listen und Arrays. Wir wollten uns dem Thema Arrays annehmen. Uns war es vorallem 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:int32;
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 wollten .

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

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

`TYPE` würde hier `int32` 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:array (array_decl) LENGTH;
var b:array (array (array (array int32 8) 5) 3) 10;
var b:array int32 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:array LENGTH (array_decl);
const c:array 10 (array 5 (array 6 (array 2 int32)));
var c:array 3 int32;
```

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:array (4,10,5) int32;
const d:array (3) int32;
```

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

Die meisten unserer Beispiele verwenden var, da wir dies für den häufigsten Anwendungsfall halten, aber natürlich kann man auch konstante Arrays nutzen, um zum Beispiel eine fixe Wahrheitstabelle abzubilden.

```
const e:array (4, 2) bool;
e init := [[true, true], [true, false], [false, true], [false, false]];
```

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 mit dem Wert 0 zu initialisieren. 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: array (7) int32;
var b: array (3) int32;
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: array (6) int32;
c init := fill 5*3+1;
```

Beim Zugriff haben wir uns für eine Schreibweise entschieden, wie sie in verschiedenen anderen Sprachen verwendet wir, 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. Ein weiterer Vorteil des 0-basierten Index zeigt sich dann auch in der Codegeneration, beim Berechnen des Speicher-Offsets.

```
var d: array (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: array (20) int32;
var b: array (4) int32;
> 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: array (20) int32;
var d: array (10) int32;
> snip
d[5..8] := c[16..20];
```

Lexikalisch

Lexikalisch erweiterten wir die Syntax um die folgenden Tokens:

Pattern	Token
`[`	LBRACKET
`]`	RBRACKET
`fill`	FILL
`array`	ARRAY
`..`	RANGE

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 ATOMTYPE)

factor      ::= LITERAL
              | arrayLiteral
              | IDENT [INIT | exprList | arrayIndex]
              | monadicOpr factor
              | LPAREN expr RPAREN

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

arrayLiteral ::= LBRACKET arrayContent RBRACKET

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

Kontext- und Typeinschränkungen

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 letzen 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.

Bei als const deklarierten Arrays muss überprüft werden, dass keine Schreibzugriffe passieren. Dies kann durch die starke Typisierung von IML bereits zur Compile-Zeit geschehen.

Codeerzeugung

AST

Bei der Struktur des AST haben wir alle degenerierten Bäume, die durch die Repetitions-NTS entstehen in Listen aufgelöst und haben dadurch einen Baum, der sinnvoll traversiert werden kann.

Wir haben jedem Knoten des CST beigebracht wie er zu einem Knoten des AST wird und dadurch konnten wir bei vielen Knoten des CST das generierte Objekt seines Kindes nach oben reichen. Dies hat zu einer relativ übersichtlichen und logischen Struktur des AST geführt, was uns die Codeerzeugung erleichtert hat.

Allgemeine Codeerzeugung

Die Codeerzeugung geschieht bei uns über den Abstract Syntax Tree, indem der Wurzel, dem ASTProgram, der Befehl zur Codeerzeugung erteilt wird.
Danach generieren die Knoten des AST mit Hilfe der Kontextinformation des Checkers ihren jeweiligen Code, welcher zusammen mit einer String-Repräsentation abgespeichert wird. Die Kontextinformationen werden dazu genutzt, beispielsweise Adressen von Variablen abzuspeichern oder Jumps zu berechnen.
Ein Integer-Literal erzeugt also seinen eigenen `intLoad(value)` Befehl.

Speicherlayout

Im ersten Ansatz, welcher in der Version der Gruppe 06 weiter verwendet wird, wird bei der Allokation von Speicherplatz und Adressen die Grösse einer Variable nicht beachtet. Das macht den Code sehr schön, und da ein Integer, Boolean oder Decimal jeweils nur einen Platz auf dem Stack benötigt, funktioniert dieser Ansatz fehlerfrei.
Bei Arrays, welche sich über mehrere Stackplätze hinziehen, führt aber dies zu Speicherplatzverletzungen. Daher mussten wir noch eine Size() Funktion für Deklarationen einführen, mit welcher die benötigten Stackplätze ermittelt und respektiert werden konnten.
Die Size-Funktion ist auch für mehrdimensionale Arrays nutzbar, da sie mittels FoldLeft alle Dimensionen miteinander multipliziert und so die gesamte Kapazität des Arrays findet.

Ein Array wird bei uns so im Speicher abgelegt, dass das erste Element an der Stelle mit dem niedrigsten Stackindex liegt. Die Adresse des Arrays zeigt so ebenfalls auf dieses erste Element. Das erleichterte uns die Überlegungen beim Schreiben des Arrayzugriffs, da so der Index einfach als Offset dienen kann.
Wird ein bestehendes Array auf den Stack geladen, so wird es wieder in dieser Reihenfolge geladen, da es ja eventuell in eine lokale Kopie geladen wird. Positiver Nebeneffekt davon ist, dass die Umkehr eines Arrays eine triviale Operation geworden ist, bei der ein Array geladen wird, und dann einfach alle Elemente vom Stack gelesen werden können.

Erweiterung der VM

Um unsere Arrays zu implementieren, mussten wir auch die VM erweitern, was dank Herrn Peyers Effort, sie nach C# zu portieren, zügig machbar war.

Wir mussten zwei neue Methoden einführen:
ArrayOutput, um Arrays sauber ausgeben zu können, und ArrayAccess, um Array Slicing umzusetzen.
ArrayOutput nimmt als zusätzliches Argument im Vergleich zu allen anderen Outputmethoden die Länge des Arrays an, um so das ganze Array ausgeben zu können.
ArrayAccess haben wir so implementiert, dass wir zunächst drei Werte auf den Stack laden:

- Die Adresse des Arrays, auf welches wir zugreifen wollen.
- Der Startindex
- Der Endindex
  ArrayAccess() liest diese drei Werte und übernimmt gleich noch die Prüfung, dass der Startindex vor dem Endindex liegt und lädt dann alle Werte auf den Stack, wie unter Speicherlayout beschrieben.

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 1, Array Initialisierung und Zugriff auf Elemente der Arrays.

```
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

endprogram
```

Beispiel 2, Bubble Sort

```
program bubble()

global
  var tosort : array (10) int32;

proc bubbleSort(inout copy A : array (10) int32)
local var n : int32;
      var i : int32;
      var tmp : int32;
```

```
      var swapped : bool
do
  n init := 10;
  i init := 1;
  tmp init := 0;
  swapped init := false;
  while not swapped do
    while i < n do
      if A[i-1] > A[i] then
        tmp := A[i];
        A[i] := A[i-1];
        A[i-1] := tmp;
        swapped := true
      else
        skip
      endif;
      i := i + 1
    endwhile;
    n := n - 1
  endwhile
endproc

do
  tosort init := [4,6,5,2,8,7,3,1,9,0];
  call bubblesort(tosort);
  debugout tosort
endprogram
```

Beispiel 3, 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
      skip
    endif;
    i := i+1
  endwhile
endprogram
```

Appendix: Ehrlichkeitserklärung

Wir, Philip Stark und Mark Zeman, haben diesen Bericht eigenhändig für den Kurs Compilerbau verfasst. Der Code für den Compiler entstand in Zusammenarbeit mit Janis Peyer und Ralf Grubenmann (Gruppe 06), wobei der Code für Arrays gänzlich von uns stammt und der Code für Decimal gänzlich von ihnen. Weitere Ideen stammen aus dem Austausch mit anderen Gruppen, speziell Manuel Jenny und Yannick Augstburger. (Gruppe 01)

Philip Stark (Grammatik, CST, AST, VM Erweiterung, Bericht, Codebeispiele, und mehr)

Mark Zeman (Scanner, Grammatik, VM Erweiterung, Codeerzeugung, Bericht, und mehr)