CpiB: Schlussbericht Strings (UTF32)

Patrick Walther / Claude Martin

Abstract

Unsere Erweiterung führt Strings in IML ein. Dies sind Zeichenketten mit beliebigen Unicode-Zeichen. Einzelne Zeichen werden gleich wie Integer behandelt. Diese werden zur Laufzeit im Heap als Array abgelegt und per Anfangs-Adresse referenziert. Im Source-Code sind Strings zwischen zwei "-Zeichen (x22) definiert. Als Escape-Zeichen wird ein Backslash verwendet.

Scanner

Der Scanner kann bei Fehlern die genaue Position eines Token angeben. So wird auch bei nie schliessenden Strings der Anfang angegeben. Auch bei fehlerhaften Escape-Sequenzen wird genau angegeben wo sich der Fehler befindet. Dazu wird die übliche Positionsangabe *Row*:*Column* verwendet, wobei *Row* die Zeilennummer und *Column* die Zeichennummer angibt. Beide Angaben beginnen bei 1. Dabei wird auch bei einer Durchmischung von Unix- und Windows-Umbrüchen immer korrekt gezählt.

Die genaue Funktionsweise wird durch ein Diagramm im Appendix illustriert.

Lexikalische Syntax

Ein neues Attribut für String-Literale wird eingeführt. Ausserdem sind Brackets nötig und es werden zwei neue Terminale für die Stringlänge eingeführt.

```
Literal:    "abc"  →     (LITERAL, "abc")
Brackets:   [ ]    →     LBRACKET RBRACKET
Maximum:    .maxlen      →     MAXLEN
Anzahl:     .strlen      →     STRLEN

// Beispiel: Erstes Zeichen gross und Punkt (ASCII 46) anfügen.
IF (MyStr[1] >= 97) {
  MyStr[1] := MyStr[1] - 32
} ELSE { SKIP };
MyStr[MyStr.STRLEN + 1] := 46;
```

Die vom Scanner erzeugten String-Token können aus dem String bereits eine Darstellung als UTF-32 erzeugen.

Die Brackets werden verwendet um auf einzelne Zeichen zuzugreifen (siehe Beispiel oben) oder um eine leere Zeichenkennte zu initialisieren. Dabei wird die maximale Anzahl Zeichen in den Brackets angegeben.

Grammatikalische Syntax

Deklaration:

```
var strIdent1 : string;
var strIdent2 : string;
var strIdent3 : string;
```

Initialisierung (Leerer String mit maximal 6 Zeichen, Literal „bla“, Konkatenation):

```
strIdent1 init := [6];
strIdent2 init := "bla";
strIdent3 init := strIdent1 + strIdent2;
Oder (zwei mal leere Strings mit maximal 6 Zeichen):
call testFunction(strIdent1 init [6]) init strIdent2 [6];
```

Zugriff auf einzelne Zeichen:

```
strIdent2[2] := strIdent2[3]; // strIdent2 ist nun „baa“
strIdent2[1] := 97; // strIdent2 ist nun „aaa“
```

Länge auslesen:

```
strIdent1.maxlen; // Maximal 6 Zeichen möglich
strIdent1.strlen; // 0, da keine Zeichen eingefügt wurden.
strIdent2.strlen; // 3 Zeichen
```

Änderungen an der Grammatik

Zusätzlich zum normalen „factor“ wurde ein neuer „arrFactor“ erstellt und die oben erwähnten Terminale ergänzt.

```
arrFactor := LBRACKET expr RBRACKET
```

Durch diese Änderung ergibt sich der folgende neue „factor“.

```
factor := LITERAL
                  | arrFactor
                  | IDENT [INIT [arrFactor] | exprList | arrFactor | MAXLEN | STRLEN]
                  | monadicOpr factor
                  | LPAREN expr RPAREN
```

Dasselbe wurde auch für die GlobalInitListe gemacht, da auch dort die Initialisierung eines Strings mit Grösse möglich sein soll. Somit ergibt sich die globInitList wie folgt.

```
globInitList := IDENT [arrFactor] {COMMA IDENT [arrFactor]}
```

Kontext-Einschränkungen

Das Resultat der „expr“ zwischen „LBRACKET“ und „RBRACKET“ muss ein IntVal sein. Dieser wird erst zur RunTime auf > 0 und <= MAXLEN geprüft.

Die Terminale „MAXLEN“ und „STRLEN“ dürfen nur auf eine Variable des Typs string folgen.

Das Nichtterminal „arrFactor“ darf nur auf eine Variable des Typs string oder auf die Initialisierung einer solchen folgen.

Die verschiedenen „FLOWMODE“, „MECHMODE“ und „CHANGEMODE“ werden von unseren strings vollständig unterstützt. Speziell ist der Wert bei Index 0. Dieser ist nicht überschreibbar.

UTF-32

Da Unicode vollständig unterstützt werden soll, muss eine Codierung verwendet werden, die alle diese Zeichen speichern kann. Dazu verwenden wir UTF-32.

Nachteil:

- Enormer Speicherverbrauch. 11 Bits pro Zeichen werden nicht verwendet.

Vorteil:

- Sehr einfaches Abzählen der einzelnen Zeichen:
  Anzahl Zeichen = Anzahl Integer = Anzahl Bytes * 4

String-Länge

Wir unterscheiden zwei Längen:

- MAXLEN: Länge des Arrays — maximale Anzahl möglicher Zeichen.
- STRLEN: Zeichenlänge des Strings — Anzahl tatsächlicher Zeichen.

Die maximale Länge zu speichern hat diverse Vorteile. Es ist also zur Laufzeit immer bekannt wie viele Zeichen im String sein können. Der String kann aber kürzer sein und die übrigen Speicherzellen sind mit 0 gefüllt. Zum einen kann zur Laufzeit die Länge einfach ausgelesen werden, ausserdem kann ein unnötiger Speicherverbraucht reduziert werden, da Strings von passender Grösser erzeugt werden können. Trotzdem sind sehr lange Strings möglich. Die maximale Grösse von 2^{32}-1 Zeichen würde 16 GiB benötigen. Somit entscheiden wir uns für diese Lösung. Abfrage ist möglich durch das Suffix „`.maxlen`“.

Die Anzahl der tatsächlichen Zeichen muss abgezählt werden. Zur Laufzeit kann nach dem ersten Zeichen welches 0 ist gesucht werden, aber maximal bis „`.maxlen`“. Abfrage durch Suffix „`.strlen`“. Es wird garantiert, dass „`.strlen`“ nie länger ist als „`.maxlen`“.

Da Unicode auch die Möglichkeit bietet diakritische Zeichen (z.B. Umlaute) einzeln zu speichern, ist der Vorteil in diesem Falle nicht mehr so gross. Lediglich die Anzahl Codepoints kann direkt mit „`.strlen`“ ausgelesen werden.

Indexierung und Iteration

Das erste Zeichen im String wird per Index 1 ausgelesen, das letzte ist auf Position „.maxlen“. Index 0 ist die Speicherposition der maximalen Länge. Somit ist „`.maxlen`“ lediglich ein Alias für `[0]`.

```
program strlen
global
      var idx : integer;
      var str : string;
{
      str init := "bla";
      idx init := 1;
      while idx <= str.strlen {
            ! str[idx];
      }
}
```

Escape-Sequenzen

Als Escape-Zeichen wird ein Backslash verwendet. Unterstützt werden folgende Zeichen: `\t`, `\n`, `\r`, `\b`, `\f` und `\R`.

Die Bedeutungen sind gleich wie in Java. Hinzu kommt noch `\R`, welches mit `System.getProperty("line.separator")` ersetzt wird. Das heisst, dass zur Compile-Zeit ein für das System üblicher Zeilenumbruch eingesetzt wird.

Speicherbelegung

Aufbau eines Strings

Bei einer Initialisierung durch ein String-Literal ist der Speicherverbrauch:

- Speicherzellen: 1 + Anzahl Code-Points
- Bytes: 4 × (1 + Anzahl Code-Points)

Die Byte-Reihenfolge ist Big-Endian und gleich wie beim 32-Bit-Integer.

- Leere Zeichenkette:
  `00000000`
- „A“:
  `00000001 00000041`
- „日本語“:
  `00000003 000065E5 0000672C 00008A9E`
- „å“:
  `00000001 000000E5 (å)`
  `00000002 00000061 0000030A (a + ˚)`

Sowohl auf dem Stack als auch auf dem Heap ist MAXLEN immer an der Speicherzelle mit der kleinsten Adresse.

Memory Management

Da die VM selbst keine Verwaltung des Speichers anbietet wird dieser die freigegeben. Es bestehen auch keine Optimierungen um Speicher zu vermeiden. So wird bei einfachen String-Operationen sehr viel Speicher verbraucht.

Durch Optimierungen und eine gute Speicherverwaltung könnte dies aber verhindert werden.

Beispiel:

```
VAR MyString : STRING;
…
MyString INIT := "Hallo "+"IML";
```

Hier werden für Hall und IML bereits je 7 und 4 Speicherzellen auf dem Heap belegt. Beim Konkatenieren (+) wird ein neuer String angelegt für die 9 Zeichen, also 10 Speicherzellen. Das Assignment (`:=`) kopiert nun diese Zeichen und legt für MyString nochmals 10 Speicherzellen an. Somit sind 31 Speicherzellen auf dem Heap verbraucht.

String-Operationen

Operation	Syntax	Beschreibung
Maximal-Länge	`str.maxlen`	Max. mögliche Zeichen.
String-Länge	`str.strlen`	Anzahl Zeichen bis 0-Terminierung.
Index-Zugriff	`str[i]`	1-basierter Array-Zugriff.
Konkatenation	`s3 := s1 + s2;`	Zusammenführen zweier Strings. *
Vergleich	`s1 = s2`	True, wenn beide äquivalent.
Ein-/Ausgabe	`? str;` `! str;`	Eingabe oder Ausgabe per Terminal.

* Dabei wird ein neuer String auf dem Heap angelegt. Es gilt: `s3.maxlen = s1.maxlen + s2.maxlen`

Vergleich zu anderen Programmiersprachen

Die sehr weit verbreitete Syntax für Strings mit Anführungszeichen und Backslash als Escape-Zeichen ist bei vielen Sprachen gleich. Als Datentyp für einzelne Zeichen (Character) wird einfach ein 32-Bit-Integer verwendet. Das Vorzeichen spielt sowieso keine Rolle, so dass hier nicht unterschieden wird.

Vergleich zu Java

Die Deklaration von Strings im Source-Code ist identisch. In Java werden Strings jedoch nicht als Array behandelt sondern als Objekt (*String* oder andere Implementation vom *CharSequence*-Interface).

Java verwendet hierbei durchwegs UTF-16, mit dem Nachteil, dass die Anzahl Codepoints nicht der Länge des Arrays entspricht. Die möglichen Zeichen sind jedoch die gleichen.

In Java steht durch die String-Klasse eine grosse Auswahl an Methoden für Strings zur Verfügung. Dies bieten wir nicht an. Die Grundoperationen würden jedoch ausreichen solche Methoden zu schreiben.

Vergleich zu C

Die Verwendung von Strings im Source-Code ist ähnlich. Da es bei C üblich ist Pointer zu verwenden ist dies anders.

In C kann grundsätzlich jede Zeichenkodierung verwendet werden. So entspricht unsere Lösung der Verwendung von char32_t. Die Terminierung von Strings mit 0 ist sehr ähnlich. Da wir jedoch die maximale Länge speichern wird verhindert, dass zu weit gelesen wird.

Vergleich zu Delphi (Pascal)

Delphi verwendet eine andere Syntax für Strings. Es wird das einfache Anführungszeichen verwendet und zur Escape-Sequenz wird einfach das Zeichen doppelt angegeben.

Delphi kennt bei kurzen Strings auch den Ansatz die Länge am Anfang der Kette zu speichern. Delphi verhindert, dass dieser Wert manipuliert wird, so dass dieser immer mit der tatsächlichen Länge übereinstimmt. So beginnt ein String bei Delphi auch bei Index 1. Unsere Umsetzung ist im Ansatz ähnlich. Doch wir speichern die maximale Länge, während diese bei Delphi diese immer 255 ist und das erste Byte die String-Länge enthält.

Später wurden in Delphi Strings mit „Copy-On-Write“ und Referenzzähler für Garbage-Collection eingeführt.

Delphi-Beispiel:

```
use System;
// …
{$LongStrings Off}
var s : string; i : integer;
// …
s := 'Delphi';
s := Copy(s,2,3);
// s = 'elp';
i := Pos('hi',s);
// i = 5;
if i = 0 then
  ShowMessage('''hi'' wurde nicht gefunden.');
```

Unicode Zeichen als Alias für Standard-Symbole.

Beim Scannen werden gewisse Operatoren vom Scanner bereits erkannt und gleich wie die Standard-IML-Schreibweise eingelesen. Da die selben Token erzeugt werden, spielt es nach dem Scannen keine Rolle mehr was im Source-Code ist. Eine Durchmischung der Schreibweisen ist auch problemlos möglich.

Folgende Aliase kann der Scanner bereits erkennen:

Terminal/Attribute	Standard-IML	Alias
CAND	`CAND`	∧
COR	`COR`	∨
NOT	`NOT`	¬
BECOMES	`:=`	≔
GE; LE; NE	`>=; <=; /=`	≥; ≤; ≠
FUN	`FUN`	*ƒ*
DIV; TIMES	`DIV; *`	÷; ×
QUESTMARK; EXCLAMARK	`?; !`	→; ←

Codegenerierung

Anpassungen der VM

Es wurde immer versucht die bestehenden Befehle zu verwenden. Die neuen Befehle sind hier aufgelistet:

VM-Befehl	Beschreibung
IntArrayInitHeap	Zur Laufzeit wird vom Stack eine Längenangabe gelesen und diese Anzahl Speicherzellen auf dem Heap reserviert. Dann wird die Längenangabe auf dem Stack mit der Adresse von der „ersten“ Zelle (in Wirklichkeit die zuletzt eingefügte Zelle) überschrieben. Der Stack-Pointer ändert sich nicht, der Heap-Pointer wird um die Länge verkleinert.
StoreIntArray	Zur Laufzeit wird vom Stack eine Zieladresse und eine Längenangabe gelesen. Darauf folgt ein Array der gegebenen Länge, welches kopiert wird. Der Stack-Pointer steht danach auf dem ersten Element des Arrays, welches jedoch mit der ursprünglich angegebenen Adresse ersetzt wird (unter der Annahme, dass die Daten als Expression weiter verwendet werden). Der Heap-Pointer wird nicht verändert, denn es wird davon ausgegangen, dass IntArrayInitHeap zuvor verwendet wurde.
StringInput	Ein String wird von der Konsole eingelesen. Analog zu BoolInout und IntInput.
StringOutput	Ein String wird in der Konsole ausgegeben. Analog zu BoolOutput und IntOutput.

Code für String-Operationen

Manche der String-Operationen (wie Konkatenation oder String-Vergleich) sind relativ komplex und erfordern einiges an Code. Trotzdem wird der Code nicht als Subroutine ausgeführt sondern „inline“ kompiliert. So wird etwas mehr Code benötigt, aber die Ausführung sollte ein klein wenig schneller sein und der Stack wird weniger beansprucht.

Da solche String-Operationen gewisse Werte wie die maximale Länge zwischenspeichern müssen, werden einige Speicherzellen auf dem Heap nur dazu verwendet. Da es in unserer VM keine Mechanismen wie Multithreading gibt, welche erlauben, dass solche Operationen (quasi) gleichzeitig ausgeführt werden, können immer dieselben Zellen verwendet werden. Diese Verwendung ist auch ein Ersatz für CPU-Register, welche diese Aufgabe übernehmen könnten.

Indexbasierter Zugriff

Ein Zugriff auf Zeichen bei einem bestimmten Index ist jedoch sehr einfach, da lediglich dieser Index zur Adresse addiert werden muss.

Bei Zuweisungen von Strings wird immer kopiert. So wird jedes Zeichen einzeln kopiert und immer beide MAXLEN-Angaben berücksichtigt. Ist das Ziel kürzer wird abgeschnitten und sonst wird mit 0 gefüllt.

Bei der Initialisierung eines Strings in einer globInitList „`init str1 [5]`“ oder in einem param „`str1 init [5]`“, ist die Grössenangabe optional. Falls diese nicht angegeben wird, wird der String mit der Grösse 255 initialisiert.

MAXLEN und STRLEN

Die Operation `MAXLEN` gibt einfach nur den INT32 Wert in der ersten Speicherzelle des Strings auf dem Heap zurück. Wie oben beschrieben wird beim Speichern des Strings die Länge dort abgelegt. Diese kann sich danach nicht mehr ändern.

`STRLEN` hingegen zählt vom Anfang des Strings jedes Zeichen bis zum ersten 0, oder maximal bis `MAXLEN` und findet so die Länge des aktuellen Strings.

Mögliche Defekte

- Keine genaue Unterscheidung zwischen *INT32* und [*INT32*]:
  `str INIT := 6;` und `str INIT := [6];` werden gleich kompiliert.

Aufteilung der Arbeit

Die einzelnen Programmieraufträge wurden abwechslungsweise vom einen Studenten implementiert. Bei den checks sowie der Codeerzeugung haben wir die Klassen des abstrakten Syntaxbaums unter uns aufgeteilt und gleichzeitig daran gearbeitet.

Die Konzepte und Inhalte wurden immer gemeinsam erarbeitet. Mit diesen Massnahmen haben wir sichergestellt, dass beide Studenten immer auf dem gleichen Wissensstand sind.

Zusammenarbeit

Für das Handling der Arrays haben wir mit dem Team Rao/Giedenmann zusammengearbeitet, um von den Erkenntnissen gegenseitig profitieren zu können. Wir haben jedoch keinen Code untereinander ausgetauscht.

Rechtliches

Hiermit erklären wir, den kompletten Code selbst erarbeitet zu haben, sofern nicht anders deklariert.

Claude Martin

Patrick Walther

Appendix

Code-Beispiele

Hallo Welt!

```
program HalloWelt
{
	! "Hallo Welt!"
}
```

Hallo User!

```
program HalloUser
global
	var name : string
{
	! "Wie heisst du?";
	name init := [32];
	? name;
	! "Hallo " + name + "!"
}
```

Equals

```
program Equals
global
	var str1 : string;
	var str2 : string;
	ƒ checkAnswer() returns ret : string
	global in str1, in str2
	{
		ret init := [8];
		if(str1 == str2) {
			ret := "Richtig!"
		} else {
			ret := "Falsch!"
		}
	}
{
	str1 init := [4];
	str1 := "Eier";
	str2 init := [4];
	← "Der Schrein ohne Deckel, Schlüssel, Scharnier, birgt einen goldenen Schatz, glaub es
mir!\nWas ist des Rätsels Lösung?";
	→ str2;
	← call checkAnswer()
}
```

GROSSBUCHSTABEN

Standard-IML

```
program GROSSBUCHSTABEN
global
      var idx : int32;
      var len : int32;
      var chr : int32;
      var str : string
{
      ! "Geben Sie einen Text ein.";
      str init := [255];
      ? str;
      idx init := 1;
      len init := str.strlen;
      chr init :=  0;
      while (idx <= len) {
            chr := str[idx];
            if (((chr >= 97) CAND (chr <= 122)) COR
                  ((chr >= 224) CAND (chr <= 254) CAND (chr /= 247))) {
                  str[idx] := chr - 32
            } else { skip };
            idx := idx + 1
      };
      ! str
}
```

Unter Verwendung von Unicode-Symbolen:

```
program GROSSBUCHSTABEN
global
      var idx : int32;
      var len : int32;
      var chr : int32;
      var str : string
{
      ← "Geben Sie einen Text ein.";
      str init ≔ [255];
      → str;
      idx init ≔ 1;
      len init ≔ str.strlen;
      chr init ≔ 0;
      while (idx ≤ len) {
            chr ≔ str[idx];
            if (((chr ≥ 97) ∧ (chr ≤ 122)) ∨
                  ((chr ≥ 224) ∧ (chr ≤ 254) ∧ (chr ≠ 247))) {
                  str[idx] ≔ chr - 32
            } else { skip };
            idx ≔ idx + 1
      };
      ← str
}
```

Scanner State-Machine