Records in IML - Schlussbericht, Compilerbau HS 2014, Team 01

Grundsätzliche Idee der Erweiterung

Als Erweiterung sollen Records implementiert werden. Ein Record soll dabei als neuer Datentyp zur Verfügung stehen, welcher beliebig viele, zusammengesetzte Felder beinhalten kann. Die Felder können vom Datentyp Integer oder Boolean sein. Einmal definierte Records stehen im ganzen Programm als Datentyp bereit. Er muss zu Beginn definiert werden und kann dann beliebig oft verwendet werden. Eine Definition eines Records sieht so aus:

$record\ r1(var\ x : int32,\ var\ y : int32,\ var\ b : boolean);$

Die Felder eines Records(z.B. $x,\ y,\ b$) verhalten sich wie „normale“ Variablen. Um sie verwenden zu können, müssen sie daher zuvor initialisiert werden. Zusätzlich muss jedes Feld innerhalb eines Records über einen eindeutigen Namen verfügen. Ein Beispiel soll nun verdeutlichen, wie Feldern eines Records verwendet werden können:

```
1 record vector(var x:int32, y:int32);
  var v1:vector;

  v1.x init := 1;
  v1.y init := 2;
```

Ein Record soll auch mit konstanten Werten definiert werden können. Dazu muss lediglich das Keyword var weggelassen werden:

```
  record vector(var x:int32, y:int32);
  var v1:vector;
  v1.x init := 2;
4 v1.y init := 2;
  v1.x := 3;
  v1.y := 3; // Fehler, da v1.y konstant ist.
```

Überprüfungen zur Compilezeit

Die Erweiterung soll das IML-Programm auf folgende Fehler prüfen:

- Sind die Namen der Felder innerhalb eines Records eindeutig?
- Sind die Namen der Records im gesamten Programm eindeutig?
- Existieren Felder innerhalb eines Records, die nicht als Bool oder Int definiert sind?
- Wird versucht ein Feld eines Typs mit einem Wert eines andern Typs zu überschreiben?
  (z.B. <boolean>:= <integer>)
- Wird versucht auf undefinierte Record-Felder zuzugreifen?

Lexikalische Syntax

Um die von uns vorgesehenen Operationen auf die Felder von Records zu gewährleisten, muss neben dem Token RECORD auch ein Token DOT eingeführt werden. Dabei gilt: ”.”$\longrightarrow$ DOT und ”record”$\longrightarrow$ RECORD.
Hier ein Auszug aus der Tokenliste bei der Verwendung des DOT-Operators:

```
v1.x init := 5;
```

```
(IDENT, "v1"), (DOTOPR, DOT), (IDENT, "x"), INIT, BECOMES, (LITERAL, IntVal 5)
```

Grammatikalische Syntax

Um die Erweiterung umzusetzen, werden folgende neue bzw. geänderte Produktionen benötigt:

```
declaration                 := storageDeclaration
                             | functionDeclaration
                             | prodDeclaration
                             | recordDeclaration

recordDeclaration           := RECORD IDENT recordFieldList
recordFieldList             := LPAREN storageDeclaration
    repeatingOptionalRecordFieldDeclarations RPAREN

repeatingOptionalRecordFieldDeclarations := COMMA recordFieldDeclaration
    repeatingOptionalRecordFieldDeclarations
                             | ϵ

term4                       := factor repDOTOPRfactor
repDOTOPRfactor             := DOTOPR factor repDOTOPRfactor
```

Kontext- und Typeinschränkung

Da die Grösse und der Aufwand der Erweiterung noch nicht genau abschätzbar war, wurde beschlossen, dass die nun folgenden Funktionen wenn möglich im Design der Erweiterung berücksichtigt werden. Implementiert wurden sie jedoch nicht.

Ein Beispiel wäre hier die Definition von Records durch andere Records, was beispielsweise bei einer Matrix sinnvoll wäre. Dies bedingt jedoch, dass als Datentyp nicht nur primitive Datentypen erlaubt wären. Aktuell sind jedoch nur primitive Datentypen erlaubt, was zur Folge hat, dass das folgende Beispiel nicht verwendet werden kann:

```
record vector(var x:int32, var y:int32, var z:int32);
record matrix(var v1:vector, var v2:vector);
var v1:vector;
var v2:vector;
var m1:matrix;

m1.v1.x init := 0; // nicht implementiert
m1.v2.y init := 2; // nicht implementiert
```

Es wäre schön, wenn es eine kurze Schreibweise zur Initialisierung der Felder gäbe. Dies wird vorerst jedoch nicht unterstützt. Ein Beispiel könnte so aussehen:

```
record r1(var x:int32, var y:int32, var active:boolean);
var test:r1;

test init := (22, 13, true); //nicht implementiert
```

Um arithmetische Operationen auf Records zu zulassen, müsste überprüft werden, ob die beiden Variablen als Typ den gleichen Record haben. Dies wurde allerdings nicht implementiert.

```
record vector (var x:int32, var y:int32, var z:int32);
var vl:vector;
var v2:vector;

//Arithmetische Operationen auf Felder von Records wurden nicht implementiert
v1 := v1 + v2;
v1 := v1 - v2;
v1 := v1 * v2;
v1 := v1 div v2;
```

Weiter besitzt ein Record keinen Zustand und somit auch keinen bool'schen Wert. Deshalb kann er nicht als Bedingung für Conditionals verwendet werden:

```
record r1 (var x:int32, var y:int32, var b:boolean);
var test:r1;

if (test) ..

while test do ..
```

Die Übergabe eines gesamten Records an eine Funktion oder Prozedur wäre sehr sinnvoll, aufgrund des Aufwands war es noch nicht möglich dies zu implementieren. Allerdings können die Felder eines Records übergeben werden:

```
proc divide(in copy vec:vector, out ref q:vector){}
proc divide2(in copy x:int32, out ref q:int32) {}
record vector(var x:int32, var y:int32, var z:int32);
var q:int32;
var v1:vector;
v1.x init := 5;
v1.y init := 6;
v1.z init := 7;
call divide(v1, q init); //nicht implementiert
call divide2(v1.x,q init); //erlaubt
```

Codeerzeugung

Verarbeitung der Records

Die Felder einer Record Deklaration werden im konkreten Syntaxbaum zu einer RecordFieldList zusammengefasst. Bei der Überführung in den abstrakten Syntaxbaum werden die Records mit Ident und den dazugeöhrigen Feldern in einer Liste gespeichert. Speicher wird dabei keiner alloziert, da die Definition eines Records nicht direkt verwendet werden kann. Erst die Definition einer Variable mit dem Typ eines gespeicherten Records führt zur Allozierung des Speichers. Dabei werden die Informationen über die Felder aus der zuvor erstellten Liste geladen und neue Variablen erstellt.

Verarbeitung der Zugriffe auf Felder

Der Zugriff auf ein Feld entspricht einer dyadischen Operation. In der Codeerzeugung wird der Aufruf des DOT-Operators gesondert behandelt, d.h. der Aufruf der Funktion, um die dyadischen Operationen aufzulösen überprüft, ob es sich beim Operator um den DOT-Operator handelt. Ist dies der Fall werden über den linken Ausdruck alle zugehörigen Felder geladen. Über den rechten Ausdruck wird dann das entsprechende Feld ausgewählt.
Auswirkungen auf andere Teile der Codeerzeugung gibt es dabei kaum. Es muss lediglich überprüft werden, ob der Ausdruck ein Dyadischer ist. Ist dies der Fall, muss zuerst der dyadische Ausdruck aufgelöst werden. Da diese Auflösung aber gleich den Code für die VM generiert, ist beispielsweise bei einer Zuweisung bereits die korrekte Adresse des entsprechenden Feldes im generierten Code festgehalten.

Vergleich mit anderen Programmiersprachen

Programmiersprache	Deklaration	Instanziierung

Pascal	`type`
	`  TVector = record`
	`  x : Integer;`
	`  y : Integer;`
	`  z : Integer;`
	`end;`
		`var`
		`  v1 : TVector`
		`begin`
		`  v1.x := 5;`
		`  v1.y := 6;`
		`  v1.z := 7;`
		`end;`

Haskell	`data Vector = Vector {`
	`  x :: Int,`
	`  y :: Int,`
	`  z :: Int`
	`}`
	`deriving (Show)`
		`v1 = vector 5 6 7`

C/C++	`struct vector {`
	`  int x;`
	`  int y;`
	`  int z;`
	`} ;`
		`vector v1;`
		`v1.x = 5;`
		`v1.x = 6;`
		`v1.z = 7;`

IML mit Record Spracherweiterung	`record vector (`
	`  var x:int32,`
	`  var y:int32,`
	`  var z:int32`
	`);`
		`var v1:vector;`
		`v1.x := 5;`
		`v1.y := 6;`
		`v1.z := 7;`

Gründe für diesen Aufbau der Erweiterung

Datentypen sind wichtig für Programmiersprachen. Man kann Daten zusammengefasst speichern, um den Überblick über die Daten besser behalten zu können.

Syntax

Damit die Syntax bestmöglich zur IML-Syntax passt, wurde eine Mischung zwischen Pascal und C/C++ gewählt. Dabei wird die eigentliche Definition des Records ähnlich wie ein Struct aus C/C++ definiert. Die Definition der Felder entspricht dann eher der Umsetzung in Pascal, da dies durch die IML-Syntax quasi gegeben ist.

Grammatik

Ein Record kann beliebig viele Variablen enthalten, dazu wurde ein neues Nichtterminalsymbol „recordFieldList“ eingeführt. Damit der Dot-Operator möglichst stark bindet wurde ein neuer „term4“ eingeführt analog zu den anderen Operatoren.

Ehrlichkeitserklärung

Dieser Schlussbericht ist das Resultat unserer Beschäftigung mit dem Modul Compilerbau. Bei der Entwicklung der Grammatik, sowie bei Fragestellungen bzgl. Codeerzeugung und Kontextüberprüfung wurde sich mit Team 10 (Zeman/-Stark) ausgetauscht. Für das Grundgerüst des Compilers haben wir die Arbeit von 2012 der Herren Fässler/Oesch und Gürber zu Hilfe genommen.
Alle Teile des Compilers wurden von beiden Teammitgliedern bearbeitet. Es kann somit nicht gesagt werden, wer wieviel an welchen Teilen gearbeitet hat.
Hiermit bestätigen wir, dass die vorliegende Arbeit und die Erweiterung ohne fremde Hilfe, ausser den obengenannten Personen, angefertigt haben.

Die Verfasser

Manuel Jenny	Yannick Augstburger

Datum: **3. Januar 2015**

Anhang

IML Beispiele

Skalarprodukt: lauffähig

```
program vecScalProd()
global
    record vector(var x:int32; var y:int32; var z:int32);var v1:vector;var v2:vector;var r:
        int32
do
    debugin v1.x init;
    debugin v1.y init;
    debugin v1.z init;

    debugin v2.x init;
    debugin v2.y init;
    debugin v2.z init;

    r init := (v1.x * v2.x) + (v1.y * v2.y) + (v1.z * v2.z);

  debugout r
endprogram
```

Übergabe an Funktionen/Prozeduren: nicht lauffähig

```
program calculatVec()
global
    record vector(x:int32;y:int32;z:int32);var vec:vector; var vec2:vector; var r:vector;
    proc addition(in copy vec:vector, in copy vec2:vector, out ref r:vector)
    do
        r.x init := vec.x + vec2.x;
        r.y init := vec.y + vec2.y;
        r.z init := vec.z + vec2.z;
    endproc
do
    call addition(vec, vec2,r)
endprogram
```

Verkettung von Vektoren: nicht lauffähig

```
program vecScalProd2()
global
    record vector(var x:int32; var y:int32; var z:int32);record matrix(var v1:vector;var v2
        :vector); var m1:matrix; var r:int32
do
    debugin m1.v1.x init;
    debugin m1.v1.y init;
    debugin m1.v1.z init;

    debugin m1.v2.x init;
    debugin m1.v2.y init;
    debugin m1.v2.z init;

    r init := (m1.v1.x * m1.v2.x) + (m1.v1.y * m1.v2.y) + (m1.v1.z * m1.v2.z);
    debugout r
endprogram
```