Einführung von switch/case und elseif als komplexe Kontrollstrukturen für IML - Schlussbericht

Schlussbericht, Tom Ohme und Simon Wächter, Compilerbau HS 2017 Team 02

Problematik

Im Gegensatz zu Hochsprachen wie C++ oder Java, verfügt IML als Lehrsprache über keine komplexeren Kontrollstrukturen wie switch/case oder elseif Bedingungen (nur if/else). Leider gibt es auch keine Alternativen wie beispielsweise das aus Haskell bekannte und mächtige Pattern Matching.

Bekommt man die Aufgabe, eine «schnelle» Fakultätsfunktion zu schreiben, welche

- die oft genutzten Fakultäten von 1 bis 4 (entsprechend 1, 2, 6, und 24) statisch ausgibt und
- höhere Werte, d.h. 120, 720, …, dynamisch & rekursiv ausrechnet,

führt dies durch das Fehlen von elseif und switch/case zu sehr schlecht lesbarem Code, der sich nur durch das Verschachteln von if/else umsetzen lässt. Alternativ wären zwar mehrere if mit einem Flag als Flankenschutz möglich, was aber alle Abfragen verkompliziert und aufbläht – wir verzichten also darauf):

```
fun fast_factorial_if(value:int32) returns result:int32
local
   tempresult:int32
do
   if value = 1 then
      result := 1
   else
      if value = 2 then
         result := 2
      else
         if value = 3 then
            result := 6
         else
            if value = 3 then
               result := 24
            else
               tempresult := fast_factorial_if(value - 1);
               result :=  value * tempresult
             endif
          endif
       endif
    endif
endfun
```

Es sollte sofort ersichtlich sein, warum komplexere Kontrollstrukturen notwendig sind. Hinzu kommt, dass je nach Implementierung der Grammatik, sogar die Gefahr eines «Dangling else» besteht (Siehe https://en.wikipedia.org/wiki/Dangling_else).

Idee / Codebeispiele

Wie eingangs erwähnt ist es deshalb sinnvoll, komplexere Kontrollstrukturen wie elseif und switch/case einzuführen. Eine Fakultätsfunktion mit einem elseif sieht wie folgt aus (Beispiele im Anhang):

```
fun fast_factorial_elseif(value:int32) returns result:int32
local
   tempresult:int32
do
   if value = 1 then
      result := 1
   elseif value = 2 then
      result := 2
   elseif value = 3 then
      result := 6
   elseif value = 4 then
      result := 24
   else
      tempresult := fast_factorial_elseif(value - 1);
      result :=  value * tempresult
   endif
endfun
```

Analog dazu ein switch/case:

```
fun fast_factorial_switch(value:int32) returns result:int32
local
   tempresult:int32
do
   switch value
      case 1 then
         result := 1
      case 2 then
         result := 2
      case 3 then
         result := 6
      case 4 then
         result := 24
      default then
         tempresult := fast_factorial_switch(value - 1);
         result :=  value * tempresult
   endswitch
endfun
```

Zu beachten ist, dass ein case Block generell abgeschlossen ist und nicht wie in C zum nächsten case durchfällt. Diese sehr schlechte Designentscheidung aus C möchten wir nicht weiterverfolgen. Folglich gibt es in einem switch/case keine break Anweisung.

Hinzu kommt, dass ein erzwungenes break in C/C++/Java das switch/case länger als nötig macht. Das if/elseif ist zwar kürzer, doch muss die Expression wiederholt werden (Beispiele für Java):

```
switch (value) {
   case 1:
      // Code
      break;
   case 2:
      // Code
      break;
   default:
      // Code
      break;
}
```
Code sehr lang & verbos

```
if(value == 1) {
   //  Code
} else if (value == 2) {
   // Code
} else {
   // Code
}
```
Kurz, arith. Ausdruck, muss aber wiederholt werden

```
switch (value) {
   case 1:
      // Code
   case 2:
      // Code
   default:
      // Code
}
```
Hypothetisches Beispiel in Java ohne erzwungenes break & ohne Durchfallen!

Grammatik

Neue Keywords

Pattern	Token
switch	SWITCH
case	CASE
default	DEFAULT
endswitch	ENDSWITCH
elseif	ELSEIF

Ergänzung / Änderung an der Grammatik EBNF

```
cmd ::= SKIP
        | expr BECOMES expr
        | IF expr THEN cpsCmd {ELSEIF expr THEN cpsCmd} [ELSE cpsCmd] ENDIF
        | SWITCH expr CASE LITERAL THEN cpsCmd {CASE LITERAL THEN cpsCmd} [DE-
        FAULT THEN cpsCmd] ENDSWITCH
        | WHILE expr DO cpsCmd ENDWHILE
        | CALL IDENT exprList [globInits]
        | DEBUGIN expr
        | DEBUGOUT expr
```

SML

```
datatype term
  =  …
    | SWITCH |
    | CASE
    | DEFAULT
    | ENDSWITCH
    | ELSEIF

val string_of_term =
  fn …
    |SWITCH => "SWITCH" |
    | CASE => "CASE"
    | DEFAULT => "DEFAULT"
    | ENDSWITCH => "ENDSWITCH"
    | ELSEIF => "ELSEIF"

val productions =
[
  (cmd, [
    [T SKIP],
    [N expr, N repExprList, T BECOMES, N expr, N repExprList],
    [T IF, N expr, T THEN, N cpsCmd, N repElseif, N optElse, T ENDIF],
    [T SWITCH, N expr, T CASE, T LITERAL, T THEN, N cpsCmd, N repCase, N optDefault, T
ENDSWITCH],
    [T WHILE, N expr, T DO, N cpsCmd, T ENDWHILE],
    [T CALL, T IDENT, N exprList, N optGlobInits],
    [T DEBUGIN, N expr],
    [T DEBUGOUT, N expr]
  ]),

  (repCase, [
    [T CASE, T LITERAL, T THEN, N cpsCmd, N repCase],
    []
  ]),

  (optDefault, [
    [T DEFAULT, T THEN, N cpsCmd],
    []
```

```
  ]),

  (optElse, [
    [T ELSE, N cpsCmd],
    []
  ]),

  (repElseif, [
    [T ELSEIF, N expr, T THEN, N cpsCmd, N repElseif],
    []
  ])
]
```

Syntax

Die Kontrollstruktur elseif kann 0…* Mal vorkommen:

```
if <Expression> then
   <Code>
elseif <Expression> then
   <Code>
else
   <Code>
endif
```

Die Kontrollstruktur case kann 1…* Mal vorkommen. Hinzu kommt ein optionales default mit 0|1 Vorkommnissen, falls keine case Bedingung erfüllt wird:

```
switch <Expression>
   case <Literal> then
      <Code>
   case <Literal> then
      <Code>
   default then
      <Code>
endswitch
```

Parsertabelle

Die Parsertabelle wurde mit dem Fix & Foxi Programm generiert und dann für das Schreiben des konkreten und abstrakten Parsers verwendet.

Statische Analyse und Contextbeschränkungen

Für die Erweiterung gelten folgende Restriktionen (Beispiele im Anhang):

- Alle Expressions in einem elseif (und auch if) müssen zu einem boolschen Ausdruck evaluiert werden können
- Die Expression des switch und dessen cases müssen vom gleichen Datentyp sein. Für zukünftige Implementierungen könnten die case Werte auch eine Teilmenge des Datentyps der switch Expression sein (damit int64 und int32 parallel verwendbar wären)
- Alle case Werte in einem switch müssen zu einem literalen Ausdruck evaluiert werde können (Die Gründe dafür sollten im nachfolgenden Kapitel Codegenerierung ersichtlich werden)
- Alle case Werte müssen über eine pro switch einzigartige Expression verfügen, heisst es darf keine duplizierten Case Expression geben (Die Gründe dafür sollten auch im nachfolgenden Kapitel ersichtlich werden).

Codegenerierung

Ursprünglicher Plan und letztendliche Umsetzung

Ursprünglich war geplant, die Codegenerierung mittels dem JVM Assembler Jasmin (Nicht zu verwechseln mit dem in Java geschriebenen x86 Assembler der Technischen Universität München http://wwwi10.lrr.in.tum.de/~jasmin/tutorials_basic.html) zu realisieren. Aufgrund unterschiedlicher Leistungsfähigkeiten in der Projektgruppe musste dieses Vorhaben aber leider abgebrochen und nach einer Alternativlösung gesucht werden. Da die Zeit verhältnismässig fortgeschritten war, entschloss man sich, aus dem AST regulären Java Code zu generieren und zu kompilieren, um daraus dann eine ausführbare JAR Datei zu bauen. Da dieses Vorhaben nicht weiter interessant ist, möchten wir darauf nicht weiter eingehen, sondern die ursprüngliche Implementierung mittels Jasmin konzeptionell beschreiben.

Konzeptionelle Umsetzung von if/elseif und else

In der JVM gibt es für die if/elseif/else Kontrollstrukturen nur Vergleichsinstruktionen mit einem Sprunglabel, welches beim Erfüllen der Instruktion angesprungen wird, beispielsweise das if_icmpne (Sprung bei ungleichen Werten: https://cs.au.dk/~mis/dOvs/jvmspec/ref--30.html). Die Parameter für die Expression müssen dabei vorgeladen werden. Ein generiertes if-elseif-elseif-else Konstrukt würde dann in Jasmin wie folgt aussehen (Zu beachten ist, dass das Laden der Operatoren für die Vergleichsinstruktion weggelassen werden):

```
; TODO: Operatoren laden
if_icmpne ELSEIF_<BranchID>_0
   ; Code für if
   goto END_<BranchID>

ELSEIF_<BranchID>_0:
   ; TODO: Operatoren laden
   if_icmpne ELSEIF_<BranchID>_1
      ; Code für 1. elseif
      goto END_<BranchID>

ELSEIF_<BranchID>_1:
   ; TODO: Operatoren laden
   if_icmpne ELSE_<BranchID>
      ; Code für 2. elseif
      goto END_<BranchID>

   ; Code für 2. elseif
   goto END_<BranchID>

ELSE_<BranchID>:
   ; Code für else
   goto END_<BranchID>

END_<BranchID>:

; Ende
```

Da Sprunglabels ganzheitlich einzigartig sein müssen, wird eine sogenannte Branchnummer pro Befehl eingeführt, welche verschachtelte Kontrollstrukturen erlaubt und somit die Einzigartigkeit garantiert.

Konzeptionelle Umsetzung von switch/case

Beim switch/case hingegen gibt es zwei unterschiedliche Instruktion in der JVM:

1. tableswitch: Alle cases werden mit ihrem Sprunglabel in einer Tabellen-, sprich Listenstruktur abgespeichert.
2. lookupswitch: Alle cases werden mit ihrem Sprunglabel und Wert des cases in einer Lookup, sprich Hashmapstruktur abgespeichert

Ein kombiniertes Beispiel beider Instruktionen (Zuerst tableswitch, dann lookupswitch) könnte wie folgt aussehen:

```
; TODO: Operator laden
tableswitch 0
   CASE_<BranchID>_0
   CASE_<BranchID>_1
default: DEFAULT_<BranchID>

; TODO: Operator laden
lookupswitch
   1: CASE_<BranchID>_0
   2: CASE_<BranchID>_1
default: DEFAULT_<BranchID>

CASE_<BranchID>_0
   ; Code für 1. case
   goto END_<BranchID>

CASE_<BranchID>_1
   ; Code für 2. case
   goto END_<BranchID>

DEFAULT_<BranchID>
   ; Code für default
   goto END_<BranchID>

END_<BranchID>
```

Das Fehlen der case Werte in der Tabellenstruktur führt dazu, dass in einem tableswitch die Werte der cases nummerisch hintereinanderliegen oder aber mit einem leeren case- Platzhalter samt entsprechendem Literalwert aufgefüllt werden müssen. Bei nah zusammenliegenden case Werten stellt dieses Auffüllen noch kein grosses Problem dar und die Zugriffskomplexität ist mit O(1) optimal. Doch spätestens bei den beiden Werten 1 und 1000 werden ganze 998 Platzhalter verwendet – man stelle sich nur die prozentuelle Verschwendung vor.

An dieser Stelle kommt die Instruktion lookupswitch ins Spiel, welche mit ihrer Datenstruktur die case Werte lokalisieren kann und so keine Platzhalter mehr benötigt, dafür aber in der Ausführung nicht so schnell ist. Je nach Möglichkeit, die case Werte zu sortieren, kommt man mit einer binären Suche auf eine Zugriffskomplexität von O(log n), allenfalls auf eine lineare Komplexität von O(n).

Durch den Aufbau der switch Struktur sollte jetzt ersichtlich sein, warum nur literale Expression verwendet werden dürfen, da diese zur Compilezeit aufgelöst werden müssen. Die Entscheidung, ob ein tableswitch oder lookupswitch verwendet werden soll, obliegt alleine dem Compiler bei der Codegeneration. Im Falle des javac Compiler findet man den Algorithmus hier: http://hg.openjdk.java.net/jdk8/jdk8/langtools/file/30db5e0aaf83/src/share/classes/com/sun/tools/javac/jvm/Gen.java#l1153

Ausblick

Umsetzung via JVM Bytecode

Wie im Kapitel Codegenerierung erwähnt, konnte die Codegenerierung aus Zeitgründen leider nicht umgesetzt werden. Für eine zukünftige Version unseres IML Compilers wäre es also sehr interessant, Jasmin als Assembler zu verwenden.

Bei der Verwendung von Jasmin kann man dann in der Codegeneration auch darüber entscheiden, ob man ein tableswitch oder lookupswitch für die Generierung des switch/case verwenden möchte und dementsprechend einen Algorithmus implementieren.

Ranged Cases

In Hochsprachen wie C++ ist es möglich, einem case einen Wertebereich als Expression zuzuweisen, z.B als Zahlenbereich (0 .. 9) oder Alphabetbereich (a .. z). Ein solches Feature in IML wäre sehr interessant, setzt aber voraus, dass die Expression zur Kompilationszeit statisch auflösbar ist und die darin verwendeten Werte alle über eine «Wertung» besitzen.

Da IML nur über numerische Werte und Booleans verfügt, ist dies recht einfach, doch was würde passieren, wenn ein Datentyp «char» eingeführt würde? Der Compiler müsste dann wissen, dass der Ausdruck «a .. z» das ganze kleingeschriebene Alphabet umfasst und den Ausdruck dementsprechend mit 26 cases ersetzen.

```
fun number_of_digits(value:int32) retuns number:int32
do
   switch value
      case 0 .. 9 then
         number := 1
      case 10 .. 99 then
         number := 2
      case 100 .. 999 then
         number := 3
      case 1000 .. 9999 then
         number := 4
      default then
         number := -1
   endswitch
endfun
```

Hat man verstanden, wie das switch/case in Java funktioniert, sollte ersichtlich werden, dass für den obigen Code 10'000 Case erstellt werden müssten, was den Code doch schon sehr aufbläht.

Man müsste sich also überlegen, ob man ab einem bestimmten Grad das ranged switch/case, sprich deren einzelne Wertebereiche, nicht durch ein if/elseif/else mit grösser/kleiner Expressions in der Codegenerierung ersetzt.

Arbeitsverteilung

Vorbesprechung der Datenstrukturen	Simon Wächter, Tom Ohme
Grammatik in EBNF	Tom Ohme
Grammatik in SML	Tom Ohme
Parser Table Generierung, Excel Tabelle	Tom Ohme
Scanner Implementierung	Simon Wächter
Verfassung Zwischenbericht, Präsentation	Simon Wächter, Tom Ohme
Parser Implementierung	Simon Wächter
Konkreter Syntaxbaum Implementierung	Simon Wächter
Abstrakter Syntaxbaum Implementierung	Simon Wächter
Kontext- und Typeinschränkung Implementierung	Tom Ohme
Codeerzeugung Implementierung	Simon Wächter
Verfassung Schlussbericht, Präsentation	Simon Wächter

Hilfsobjekte und Referenzen

- Als Beispielcompiler und auch, um IML Code zu testen, wurde der Compiler von Yanick Schraner verwendet
- Herr Ohme hat beim Generieren und Anpassen der Grammatik auf die Hilfe von Yanick Schraner zurückgegriffen
- Ursprünglicher Jasim JVM Assembler: http://jasmin.sourceforge.net/
- Ursprünglich ergänzendes Buch, welches die JVM und Jasmin Codegenerierung in je einem Kapitel beschreiben: Writing compilers and interpreters : a modern software engineering approach using Java (ISBN 978-0-470-17707-5)

Unterschriften

Simon Wächter: _____

Tom Ohme: _____

Anhang: IML Fakultätsprogram

```
program Factorial(inputvalue:int32, outputvalue:int32)
global

   fun fast_factorial_if(value:int32) returns result:int32
   local
      tempresult:int32
   do
      if value = 1 then
         result := 1
      else
         if value = 2 then
            result := 2
         else
            if value = 3 then
               result := 6
            else
               if value = 3 then
                  result := 24
               else
                  tempresult := fast_factorial_if(value - 1);
                  result :=  value * tempresult
               endif
            endif
         endif
      endif
   endfun;

   fun fast_factorial_elseif(value:int32) returns result:int32
   local
      tempresult:int32
   do
      if value = 1 then
         result := 1
      elseif value = 2 then
         result := 2
      elseif value = 3 then
         result := 6
      elseif value = 4 then
         result := 24
      else
         tempresult := fast_factorial_elseif(value - 1);
         result :=  value * tempresult
      endif
   endfun;

   fun fast_factorial_switch(value:int32) returns result:int32
   local
      tempresult:int32
   do
      switch value
         case 1 then
            result := 1
         case 2 then
            result := 2
         case 3 then
            result := 6
         case 4 then
            result := 24
         default then
```

```
            tempresult := fast_factorial_switch(value - 1);
            result :=  value * tempresult
        endswitch
    endfun

do
    debugin inputvalue;

    outputvalue := fast_factorial_if(inputvalue);
    debugout outputvalue;

    outputvalue := fast_factorial_elseif(inputvalue);
    debugout outputvalue;

    outputvalue := fast_factorial_switch(inputvalue);
    debugout outputvalue;

    skip
endprogram
```

Anhang: Keine boolsche Expression in einem if/elseif

```
program Factorial(inputvalue:int32, outputvalue:int32)
do

    if false then
        skip
    elseif 2 + 2 then // ch.fhnw.cpib.platform.checker.CheckerException: IF condition needs to be BOOL.
        skip
    endif;

   skip
endprogram
```

Anhang: Keine literale Expression

```
program Factorial(inputvalue:int32, outputvalue:int32)
do

    switch 10
        case 10 then
            skip
        case 2 + 10 then // ch.fhnw.cpib.platform.parser.exception.ParserException: Parser expected the
terminal THEN, but found the terminal ADDOPR
            skip
        default then
            skip
   endswitch;

   skip
endprogram
```

Anhang: Duplizierte Case Ausdrücke

```
program Factorial(inputvalue:int32, outputvalue:int32)
do

    switch 10
        case 10 then
            skip
```

```
      case 10 then // ch.fhnw.cpib.platform.checker.CheckerException: Case literal values have the same
value.
          skip
      default then
         skip
  endswitch;

  skip
endprogram
```