Case Commands und Conditional Expressions für IML

extended IML (eIML) Compiler

SB

Compilerbau, HS 2015, Team HM

Lukas Hubschmid und Roger Müller

1 Abstract

Im Rahmen des Compilerbau-Unterrichts werden wir einen Compiler für die Sprache IML entwickeln. Mit einer Erweiterung der Sprache IML zu eIML (extended IML) soll schlussendlich der Compiler auch „Case Commands“ und „Conditional Expressions“ kompilieren können. Der Schlussbericht ist eine Erweiterung unseres Zwischenberichts. Nun werden wir genauer auf die Implementierung eingehen und unsere Gedanken hinter der Erweiterung genauer erläutern.

2 Aufbau des Compilers

Der Scanner und der Parser wurden grundsätzlich gemäss den Folien aus dem Unterricht umgesetzt. Mit FixFoxi wurde sichergestellt, dass die Grammatik mit unserer Erweiterung LL1-konform ist.

2.1 Static Analysis

Die Static Analysis teilt sich auf folgende Schritte auf dem AST auf:

1. Namespace checking
2. LVal und RVal checking
3. Param checking
4. Init checking
5. Const checking
6. Type checking

1

Routinen (procDecl und funDecl) sind global definiert. Variablen können entweder global oder lokal, d.h. innerhalb einer Routine, deklariert werden. Da die Namespaces von Variablen und Routinen getrennt behandelt werden, darf der Name einer Routine identisch sein mit dem Namen einer Variable. Da global imports aus Zeitgründen vom Dozenten nicht verlangt und somit von uns nicht spezifiziert sowie implementiert wurden, haben wir definiert, dass auf die globalen Variablen überall direkt zugegriffen werden kann. Damit immer klar ist, auf welcher Variable gearbeitet wird, ist die Deklaration einer lokalen Variable mit demselben Namen wie eine globale Variable nicht zugelassen. Bei den Routinen wird überprüft, ob eine Routine mit gleichem Namen bereits existiert. Die Überladung von Routinen (gleicher Name, andere Parameter) ist nicht implementiert, wäre aber aus unserer Sicht eine sinnvolle Erweiterung, welche relativ einfach umzusetzen wäre.

2

Bei FunCall sowie ProcCall muss überprüft werden, ob die Parameter den Richtig LRVal Wert besitzen, wobei beim Aufruf einer Routine folgende Kombinationen möglich sind:

Übergebener Param (Caller)	Erwarteter Param (Callee)	Beschreibung
LVal	LVal	Valid
RVal	LVal	Nicht erlaubt
LVal	RVal	LVal muss dereferenziert werden
RVal	RVal	Valid

Bei einem DebugIn muss sichergestellt werden, dass es sich beim Ausdruck um einen LVal handelt, damit der eingegebene Wert dieser Variable zugewiesen werden kann.

Bei einem AssignCmd muss sichergestellt werden, dass es sich beim Ausdruck auf der linken Seite (leftExpr) um einen LVal handelt.

3

Beim Routinenaufruf wird überprüft, ob die Anzahl der Caller-Parameter mit der Anzahl der Callee-Parameter übereinstimmt.

4

Damit klar definiert ist, welcher Wert eine Variable hat, muss die Variable bei der ersten Verwendung mit dem Schlüsselwort `init` initialisiert werden. Die Initialisierung kann entweder auf der linken Seite eines `AssignCmd` oder als Ausdruck eines `DebugInCmd` geschehen:

```
g1 init := 0;
debugin g2 init;
```

Nach der Initialisierung besitzt die Variable immer einen Wert.

Da zur Compiletime nicht klar ist, ob ein Abschnitt eines IF-ELSE-Konstruktes oder eine While-Schleife ausgeführt werden wird, haben die Commands IF-ELSE, WHILE und SWITCH-CASE einen eigenen Init-Scope. Eine lokale Variable muss vor der Verwendung im gleichen oder in einem übergeordneten Scope initialisiert worden sein. Die Initialisierung einer globalen Variable ist innerhalb dieser Scopes nicht gestattet, da nicht sichergestellt werden kann, dass der entsprechende Abschnitt zur Laufzeit wirklich ausgeführt und die Variable initialisiert wird.

```
if 1 < y then
  x init := 2
else
  x init := 5
endif;
debugout x // nicht erlaubt, da in untergeordnetem Scope initialisiert
```

Laut Grammatik besteht die Möglichkeit innerhalb einer Routine eine globale Variable zu initialisieren. Dies würde dazu führen, dass die globale Variable bei einem mehrfachen Aufruf der Routine mehrmals initialisiert würde. Daher wird das InitChecking für deklarierte Routinen nur durchgeführt, wenn die Routine auch aufgerufen wird. Wird eine Routine, welche eine globale Variable initialisiert, mehrfach aufgerufen, so gibt der Compiler einen Fehler aus.

5

Beim Changemode `const` ist nach der Initialisierung kein weiterer Schreibzugriff mehr erlaubt. Bei `AssignCmd` sowie `DebugIn` wird überprüft, ob eine Konstante bereits initialisiert ist. Falls ja, wird ein Fehler ausgegeben. Wird eine Konstante via ref-Parameter an eine Routine übergeben, so muss sichergestellt werden, dass auch innerhalb der Routine der Wert der Konstante nicht bearbeitet werden kann. Dies ist jedoch noch nicht implementiert.

6

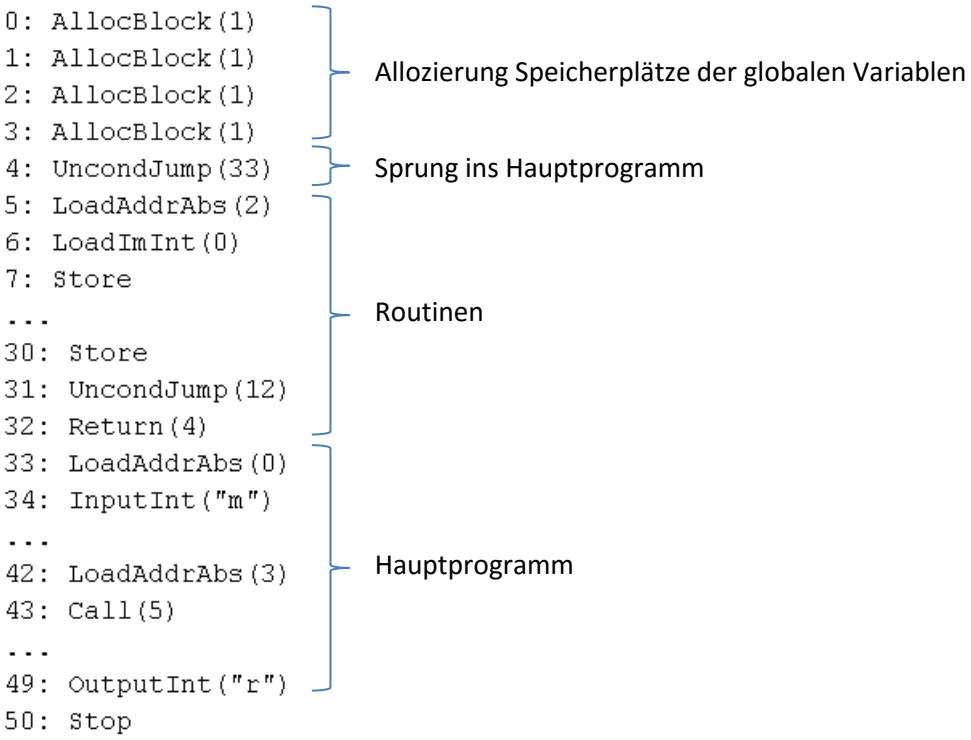
An folgenden Orten muss der Typ überprüft werden. Falls die Typen gemäss folgender Tabelle nicht übereinstimmen, wird ein Error geworfen.

Ort	Beschreibung
AddExpr, MultExpr, RelExpr	**Nur int64 für die beiden Ausdrücke erlaubt,**
BoolExpr	Nur bool für die beiden Ausdrücke erlaubt
AssignCmd	**Typ des Ausdrucks auf linker Seite muss mit Typ des Ausdrucks auf rechter Seite übereinstimmen**
CondExpr	Siehe 4.4 Kontext- und Typeneinschränkungen
FunCall, ProcCall	**Parameter des Aufrufs (Caller) müssen den gleichen Typ haben wie die Parameter der Deklaration (Callee)**
IfCmd	Bedingung muss bool sein
MonadicOpr	**NOTOPR -> erwartet bool** **ADDOPR -> erwartet int64**
SwitchCmd	Siehe 3.4 Kontext- und Typeneinschränkungen
WhileCmd	**Schleifenbedingung muss bool sein**

2.2 Virtual Machine

Für die Code-Generierung bietet der AST eine rekursive Methode an, in welcher jeder Node den für ihn relevanten Code dem CodeArray hinzufügt.

Im ersten Bereich des generierten Codes werden die Speicherplätze für die globalen Variablen alloziert. Im zweiten Bereich sind die Instruktionen der deklarierten Routinen gespeichert. Im dritten Bereich befindet sich das Hauptprogramm.



Damit auch in Routinen auf die globalen Variablen zugegriffen werden kann (direkter Zugriff), haben wir die VM um die Instruktion LoadAddrAbs ergänzt. Mit LoadAddrAbs kann auf die absoluten Stack-Adressen der globalen Variablen zugegriffen werden, unabhängig vom frame pointer.

Um bool'sche Ausdrücke verarbeiten zu können, haben wir die virtuelle Maschine um die Methoden boolAnd, boolOr, boolCAnd, boolCor und boolInv erweitert.

Weiter haben wir die VM sowie eIML (zusätzlich zu divT und modT) um divE, modE, divF und modF erweitert.

Mit den beiden existierenden Jump-Instruktionen CondJump und UncondJump muss bei der Codegenerierung bereits die Jump-Adresse bekannt sein. Bei der Implementierung der Jump-Tabelle für Switch-Case ist jedoch die genaue Jump-Adresse abhängig von einem Wert auf dem Stack. Dazu haben wir die VM um die Instruktion RelJump erweitert. Die der RelJump-Instruktion übergebene Adresse stellt den Beginn des Jump-Bereiches dar, die genaue Adresse innerhalb dieses Bereiches wird mit dem Wert auf dem Stack berechnet.

```
int index = Data.intGet(store[sp - 1]);
sp= sp - 1;
pc= jumpAddr + index;
```

3 Case Constructs für IML

3.1 Idee

Mit dem IF-ELSE-Konstrukt können in IML unterschiedliche Code-Blöcke abhängig vom Wert eines booleschen Ausdruckes ausgeführt werden. Somit kann der eine oder andere Code-Block ausgeführt werden, je nachdem ob z.B. der Ausdruck „x = 5“ zutrifft oder nicht. Soll aber mehr als eine Unterscheidung möglich sein und jeweils den zutreffen Code-Block (und nur dieser) ausgeführt werden (z.B. x = 1 → Ausführung Code-Block A, x = 2 → Ausführung Code-Block B, x = 3 → Ausführung Code Block C, usw.), so ist dies nur umständlich mit mehreren IF- oder verschachtelten IF-ELSE-Anweisungen möglich.

Mit einem Case Construct kann das o.g. Szenario im Programm-Code kompakt geschrieben werden. Zusätzlich kann die Ausführung eines Case Construct, je nach Implementierung, performanter sein als mehrere IF- oder verschachtele IF-ELSE-Anweisungen. Ein Case Construct in eIML kann wie folgt aussehen:

```
switch c
      case 5:
            ...
      endcase
      defaultcase:
            ...
      endcase
endswitch
```

3.2 Lexikalische Syntax

Für ein Case Construct werden die folgenden Tokens benötigt (Tabelle anhand o.g. Beispiel):

Keyword	Token	Beschreibung
switch	SWITCH	Start eines Case Constructs
case	CASE	Start eines Cases
:	COLON	Ende des Kriteriums des Cases
endcase	ENDCASE	Ende eines Cases oder des Default Cases
defaultcase	DEFAULTCASE	Start des Default Cases
endswitch	ENDSWITCH	Ende eines Case Constructs

3.3 Grammatikalische Syntax

Das Case Construct wird durch die folgende Grammatik definiert:

cmd ::= SWITCH expr caseNTS defaultCaseNTS ENDSWITCH
caseNTS::= CASE LITERAL COLON cpsCmd ENDCASE caseNTS
caseNTS::= ε
defaultCaseNTS ::= DEFAULTCASE COLON cpsCmd ENDCASE
defaultCaseNTS ::= ε

Ein Case Construct kann also aus 0..n Cases sowie optional einem Default Case bestehen.

Das Case Construct ist eine Alternative des Nichtterminal-Symbols (NTS) „cmd“. Es kann also analog zu den Konstrukten IF-ELSE oder WHILE-DO verwendet werden. In den Cases können Produktionen vom Typ „cpsCmd“ verwendet werden, also beispielsweise auch wieder ein Case Construct.

Als Kontroll-Variable resp. Ausdruck sind Produktionen vom Typ expr handeln, also beispielsweise eine einfache Zahl (LITERAL) oder Variable (IDENT), einen bool'schen Term (Term mit BOOLOPR) oder einen anderen Ausdruck.

Als Kriterien der Cases dürfen nur LITERALs verwendet werden, da sich diese zur Laufzeit nicht verändern dürfen. Ansonsten kann zur Laufzeit die Logik resp. der Ablauf des Programms verändert werden.

3.4 Kontext- und Typeneinschränkungen

Ist der Typ der expr ein Boolean (z.B. (LITERAL, BoolVal true) oder bool'scher Term), so dürfen als Kriterien für die Cases nur Boolean-LITERALs verwendet werden. Dies entspricht einem IF-ELSE-Konstrukt.

Ist der Typ der expr ein Int64Val, so dürfen als Kriterien für die Cases nur Int64Val-LITERALs verwendet werden.

Dieser Ansatz kann auch auf weitere LITERAL-Typen (z.B. Character) ausgeweitet werden, was aber nicht Bestandteil von unserer Erweiterung ist. Die o.g. Bedingungen müssen durch das Type-Checking überprüft werden.

3.5 Vergleich mit anderen Programmiersprachen

Die Syntax unseres Case Constructs ist von den verwendeten Schlüsselwörtern an dem Switch-Case-Konstrukt von Java angelehnt. Die Syntax ist jedoch gleich gehalten wie die restlichen Kontrollstrukturen von IML (z.B. Abschluss von Blöcken mit ENDIF oder ENDWHILE).

Case Construct (eIML)	Switch Case (Java) [1]	While Do (IML)

`switch c`
`  case 5:`
`  ...`
`  endcase`
`  defaultcase:`
`  ...`
`  endcase`
`endswitch`
	`switch (c) {`
	`  case 5:`
	`  ...`
	`  break;`
	`  default:`
	`  ...`
	`  break;`
	`}`
		`while d >= e do`
		`  ...`
		`endwhile`

3.6 Code-Generierung

Für das Case Construct haben wir schlussendlich zwei Versionen implementiert. Bei beiden Versionen wird zu Beginn analysiert, wieviel Platz die Expression, die Cases sowie das Default Case im Code Array benötigen werden, damit anschliessend die Adressen für die verschiedenen Jump-Befehle korrekt übergeben werden können.

[1] https://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html

Das folgende Programm dient zur Veranschaulichung dieses Kapitels:

```
program programSwitchCase
global
  x:int64;
  y:int64

do
  debugin x init;
  y init := 0;
  switch x
    case 5:
      y := x * 5
    endcase
    case 7:
      y := x * (((7)))
    endcase
    defaultcase:
      y := x divE 2
    endcase
  endswitch;
  debugout y
endprogram
```

In der ersten Version wird jeweils pro Case der Wert von x auf den Stack geladen und anschliessend überprüft, ob dieser Wert mit dem Literal des Cases übereinstimmt. Bei Übereinstimmung wird mit dem nachfolgenden Code weitergefahren, ansonsten wird zum Überprüfungs-Code des nächsten Cases gesprungen (CondJump). Am Ende jedes Cases wird mit einem UncondJump aus dem gesamten Case Construct herausgesprungen. Der generierte Code ähnelt einem analogen IF-ELSE-Konstrukt – im schlimmsten Fall (Wert von x trifft auf keinen der n Cases zu) müssen n Vergleiche durchgeführt werden, bevor der Code des Default Cases zum Zug kommt.

```
8: LoadAddrAbs(0)
9: Deref
10: LoadImInt(5)
11: EqInt
12: CondJump(20)
13: LoadAddrAbs(1)
14: LoadAddrAbs(0)
15: Deref
16: LoadImInt(5)
17: MultInt
18: Store
19: UncondJump(38)
20: LoadAddrAbs(0)
21: Deref
22: LoadImInt(7)
23: EqInt
24: CondJump(32)
25: LoadAddrAbs(1)
26: LoadAddrAbs(0)
27: Deref
28: LoadImInt(7)
29: MultInt
30: Store
31: UncondJump(38)
32: LoadAddrAbs(1)
33: LoadAddrAbs(0)
34: Deref
35: LoadImInt(2)
36: DivEuclInt
37: Store
```

Überprüfungs-Code für 1. Case (Case 5)

Code von 1. Case (Case 5)

Sprung aus Case Construct

Überprüfungs-Code für 2. Case (Case 7)

Code von 2. Case (Case 7)

Sprung aus Case Construct

Code von Default Case

Da die erste Version des Case Contructs aus Sicht der VM keinen Vorteil gegenüber verschachtelten IF-ELSE-Konstrukten bietet, haben wir eine zweite Version implementiert, welche eine Jump-Tabelle verwendet. [2] In einem ersten Schritt wird hierbei überprüft, ob der Wert von x kleiner resp. grösser als der Wert des kleinsten resp. grössten Literals der Cases ist. Falls ja, so kann direkt ins Default Case gesprungen werden. Ansonsten wird der Wert von x „normiert“ indem der kleinste Literal subtrahiert wird. Dadurch kann für den Case mit dem tiefsten Literal den Index 0 in der Jump-Tabelle verwendet werden. Für das o.g. Code-Beispiel ist der Offset 5 (kleinster Literal: 5). So erhält man für Case 5 die Position 5-5=0 sowie für Case 7 die Position 7-5=2 in der Jump-Tabelle. Mit dem normierten Wert von x wird nun in den entsprechenden Index der Jump-Tabelle gesprungen. Dazu musste die VM um die Instruktion RelJump erweitert werden (siehe 2.2 Virtual Machine). Von der Jump-Tabelle wird einem UncondJump weiter zum eigentlichen Code des entsprechenden Cases gesprungen. Bei Indizes, welche keinem Case entsprechen, wird ins Default Case gesprungen, in diesem Beispiel bei Jump-Tabelle-Index 1 resp. wenn der Wert von x 6 ist. Der grosse Vorteil gegenüber Version 1 (O(n)) resp. dem IF-ELSE-Konstrukt ist die konstante Laufzeit (O(1)). Jedoch kann die Jump-Tabelle bei sehr unterschiedlichen Literals sehr gross sein (Beispiel: min. Literal 0, max. Literal 10‘000 → Jump-Tabelle-Grösse: 10‘001). Dies könnte mit mehreren Jump-Tabellen und einem binären Suchbaum weiter optimiert werden.

[2] http://www.codeproject.com/Articles/100473/Something-You-May-Not-Know-About-the-Switch-Statem

```
8: LoadAddrAbs(0)
9: Deref
10: Dup
11: LoadImInt(5)
12: LtInt
13: NegBool
14: CondJump(40)
15: Dup
16: LoadImInt(7)
17: GtInt
18: NegBool
19: CondJump(40)
20: LoadImInt(5)
21: SubInt
22: RelJump(23) + index
23: UncondJump(26)
24: UncondJump(40)
25: UncondJump(33)
26: LoadAddrAbs(1)
27: LoadAddrAbs(0)
28: Deref
29: LoadImInt(5)
30: MultInt
31: Store
32: UncondJump(46)
33: LoadAddrAbs(1)
34: LoadAddrAbs(0)
35: Deref
36: LoadImInt(7)
37: MultInt
38: Store
39: UncondJump(46)
40: LoadAddrAbs(1)
41: LoadAddrAbs(0)
42: Deref
43: LoadImInt(2)
44: DivEuclInt
45: Store
```

Wert von x auf Stack legen

Überprüfung ob Wert von x kleiner als kleinster Literal

Überprüfung ob Wert von x grösser als grösster Literal

Wert auf Stack normieren

Sprung in Jump-Tabelle abhängig von normierten Wert auf Stack

Jump-Tabelle

Code von 1. Case (Case 5)

Sprung aus Case Construct

Code von 2. Case (Case 7)

Sprung aus Case Construct

Code von Default Case

3.7 Warum wurde die Erweiterung so entworfen und nicht anders?

Ziel der Erweiterung ist eine möglichst grosse Flexibilität und doch Einfachheit. So ist als Kontroll-Ausdruck jede mögliche Produktion vom Typ expr erlaubt, was beispielsweise auch einen Funktionsaufruf erlaubt. Gleichzeitig wurden unnötige Zeichen (z.B. Klammern um Kontroll-Variable / Ausdruck) weggelassen. Im Gegensatz zu anderen Programmiersprachen (z.B. Pascal mit „Begin“ resp. „End“ oder Java mit „case“ resp. „break“) ist mit unseren Schlüsselwörtern klar ersichtlich wo ein Case beginnt (CASE oder DEFAULTCASE) und wo es endet (ENDCASE).

4 Conditional Expressions für IML

4.1 Idee

Mit der Conditional Expression kann, im Gegensatz zum IF-ELSE-Konstrukt, möglichst kurz und übersichtlich in einer Zeile eine Fallunterscheidung durchgeführt werden. Eine Conditional Expression kann z.B. direkt bei einer Zuweisung oder in einem Funktionsaufruf verwendet werden. Weiter können Conditional Expressions auch verschachtelt werden. Im Anhang ist noch ein weiteres Conditional Expression Programm vorhanden, welches Rekursiv die Fibonacci Zahl berechnet. [6.2.1 Conditional Expression]

Conditional Expressions in eIML sehen wie folgt aus:

```
// Conditional Expression bei Zuweisung
max1 := a > b ? a : b;
// Nicht sehr sinnvoll, aber möglich
max2 := 8 > 4 ? 8 : 4;
// Mit Methoden als expr
a := (f(a) > f(b) ? f(c) : f(d)) AND (f(e) = f(f) ? f(g) : f(h))
// Conditional Expression in Funktionsaufruf
f(a > b ? c : d)
// Verschachtelung mit zusätzlichem Funktionsaufruf
max3 = a > (b = c ? k : f(l)) ? t : z
```

4.2 Lexikalische Syntax

Für ein Conditional Expression werden die folgenden Tokens benötigt (Tabelle anhand erstem o.g. Beispiel):

Keyword	Token	Beschreibung
?	QUESTIONMARK	Trennung Kontroll-Ausdruck und True-Operand
:	COLON	Trennung True-Operand und False-Operand

4.3 Grammatikalische Syntax

Die Conditional Expression wird durch die folgende Grammatik definiert:

expr ::= term0 condExprNTS
condExprNTS ::= QUESTIONMARK expr COLON expr
condExprNTS ::= ε
term0 ::= term1 term0NTS
term0NTS ::= BOOLOPR term1 term0NTS
term0NTS ::= ε

Die Conditional Expression ist eine Produktion vom Typ expr, wodurch sie an vielen Orten verwendet werden kann (= Flexibilität), z.B. als Kontroll-Ausdruck eines If-Else-, While-Do-, oder Case-Konstruktes. Beim Kontroll-Ausdruck kann es sich um einen der verschiedenen Terme (z.B. BOOLOPR, RELOPR, LITERAL, Verschachtelter Ausdruck (LPAREN expr RPAREN) handeln.

Der True- und der False-Operand der Conditional Expression ist eine Produktion vom Typ expr, wodurch wiederum z.B. eine weitere Conditional Expression oder eine andere Alternative vom Typ expr (z.B. Terme) verwendet werden kann.

4.4 Kontext- und Typeneinschränkungen

Handelt es sich bei der expr um eine Conditional Expression (condExprNTS != ε), so muss es sich bei term0 um einen bool'schen Wert handeln.

Die beiden Ausdrücke vom Typ expr (True-Operand und False-Operand) müssen den gleichen Typ (z.B. (LITERAL, Int64Val) ergeben.

4.5 Vergleich mit anderen Programmiersprachen

Die Syntax der Conditional Expression ist an den gängigen Programmiersprachen Java oder C++ orientiert.

`max := a > b ? c : d`	`max = i > j ? i : j`	`max = (a > b) ? a : b;`
Conditional Expression (eIML)	Conditional Expression (C++) [3]	Conditional Expression (Java) [4]

4.6 Code-Generierung

Zu Beginn wird analysiert, wieviel Platz die drei Ausdrücke im Code Array benötigen werden, damit anschliessend die Adressen für die verschiedenen Jump-Befehle korrekt übergeben werden können.

Trifft die Bedingung der Conditional Expression zu, so wird der Code des „True-Ausdrucks" ausgeführt und anschliessend aus der Conditional Expression herausgesprungen. Trifft die Bedingung nicht zu, so erfolgt ein Sprung zum Code des „False-Ausdrucks".

IML Code: `n < m ? n : m`

Generierter Code:

```
42: LoadAddrAbs(0)
43: Deref
44: LoadAddrAbs(1)
45: Deref
46: LtInt
47: CondJump(51)
48: LoadAddrAbs(0)
49: Deref
50: UncondJump(53)
51: LoadAddrAbs(1)
52: Deref
```

Bedingung mit Sprung zu False-Ausdruck

True-Ausdruck

Sprung aus Conditional Expression

False-Ausdruck

4.7 Warum wurde die Erweiterung so entworfen und nicht anders?

Ziel der Erweiterung ist eine möglichst grosse Flexibilität und doch Einfachheit. Durch Realisierung als Produktion vom Typ expr kann die Conditional Expression an vielen Orten eingesetzt werden. Mit der gewählten Syntax (analog zu Java und C++) kann die Conditional Expression sehr übersichtlich und Platz sparend geschrieben werden.

[3] https://msdn.microsoft.com/en-us/library/e4213hs1.aspx
[4] http://www.cafeaulait.org/course/week2/43.html

5 Schlusswort

Das ganze Projekt hat uns sehr viel Spass und Freude bereitet. Wir konnten uns in Gebiete einarbeiten, an denen wir während des Studiums teils nur oberflächlich gekratzt haben. Da wir uns sehr tief in die Materie einarbeiten mussten, haben wir Probleme und Tücken entdeckt, welche uns sonst garantiert verborgen geblieben wären. Auch wenn einige Teile des Compilers sehr viel Fleissarbeit war, sind wir nun froh und stolz einen praktisch vollständig funktionierenden Compiler erarbeitet zu haben.

5.1 Arbeitsteilung

Für den Zwischenbericht wurden die Zwischenschritte „Scanner“ und „Parser“ des Compilers analysiert und teils umgesetzt. Beim Scanner hat sich Roger Müller auf die Generierung der Tokens (Properties-File), Lukas Hubschmid auf die State-Machine konzentriert. Die restlichen Arbeiten (Grammatik, Grammatik-Datei für FixFoxi, Zwischenbericht, CST, AST, Code-Generierung, usw.) wurden zusammen durchgeführt.

5.2 Ehrlichkeitserklärung

Hiermit erklären wir, den vorliegenden Schlussbericht unseres Compilers mit den Erweiterungen „Case Commands und Conditional Expressions“ selbständig und ohne Hilfe Dritter verfasst zu haben. Als Hilfsmittel haben die Materialien aus dem Unterricht gedient.

Brugg, 9.1.2016

Lukas Hubschmid

Roger Müller

6 Anhang

6.1 Grammatik

program ::= PROGRAM IDENT globalNTS DO cpsCmd ENDPROGRAM

globalNTS ::= GLOBAL cpsDecl
globalNTS ::= ε

cpsDecl ::= decl cpsDeclNTS
cpsDeclNTS ::= SEMICOLON decl cpsDeclNTS
cpsDeclNTS ::= ε

decl ::= stoDecl
decl ::= funDecl
decl ::= procDecl

stoDecl ::= typedIdent
stoDecl ::= CHANGEMODE typedIdent

typedIdent ::= IDENT COLON TYPE

funDecl ::= FUN IDENT paramList RETURNS stoDecl funDeclNTS DO cpsCmd ENDFUN
funDeclNTS ::= LOCAL cpsStoDecl
funDeclNTS ::= ε

paramList ::= LPAREN paramListNTS RPAREN
paramListNTS ::= param paramNTS
paramListNTS ::= ε

paramNTS ::= COMMA param paramNTS
paramNTS ::= ε

param ::= mechModeNTS changeModeNTS typedIdent
mechModeNTS ::= MECHMODE
mechModeNTS::= ε
changeModeNTS ::= CHANGEMODE
changeModeNTS::= ε

cpsCmd ::= cmd cpsCmdNTS
cpsCmdNTS ::= SEMICOLON cmd cpsCmdNTS
cpsCmdNTS ::= ε

cmd ::= SKIP
cmd ::= expr BECOMES expr
cmd ::= IF expr THEN cpsCmd ifelseNTS ENDIF
ifelseNTS ::= ELSE cpsCmd
ifelseNTS ::= ε
cmd ::= WHILE expr DO cpsCmd ENDWHILE

//Switch case
cmd ::= SWITCH expr caseNTS defaultCaseNTS ENDSWITCH
caseNTS::= CASE LITERAL COLON cpsCmd ENDCASE caseNTS
caseNTS::= ε
defaultCaseNTS ::= DEFAULTCASE COLON cpsCmd ENDCASE
defaultCaseNTS ::= ε
cmd ::= CALL IDENT exprList
cmd ::= DEBUGIN expr
cmd ::= DEBUGOUT expr

//Conditional Expression
expr ::= term0 condExprNTS
condExprNTS ::= QUESTIONMARK expr COLON expr
condExprNTS ::= ε

term0 ::= term1 term0NTS
term0NTS ::= BOOLOPR term1 term0NTS
term0NTS ::= ε

term1 ::= term2 term1NTS
term1NTS::= RELOPR term2
term1NTS::= ε

term2 ::= term3 term2NTS
term2NTS ::= ADDOPR term3 term2NTS
term2NTS ::= ε

term3 ::= factor term3NTS
term3NTS ::= MULTOPR factor term3NTS
term3NTS ::= ε

factor ::= LITERAL
factor ::= IDENT factorNTS
factorNTS ::= INIT
factorNTS ::= exprList
factorNTS ::= ε

factor ::= monadicOpr factor
factor ::= LPAREN expr RPAREN

exprList ::= LPAREN exprListLparenNTS RPAREN
exprListLparenNTS ::= expr exprListNTS
exprListLparenNTS ::= ε

exprListNTS ::= COMMA expr exprListNTS
exprListNTS ::= ε

monadicOpr ::= NOT
monadicOpr ::= ADDOPR

cpsStoDecl ::= stoDecl cpsStoDeclNTS
cpsStoDeclNTS ::= SEMICOLON stoDecl cpsStoDeclNTS
cpsStoDeclNTS ::= ε

procDecl ::= PROC IDENT paramList procDeclNTS DO cpsCmd ENDPROC
procDeclNTS ::= LOCAL cpsStoDecl
procDeclNTS ::= ε

6.2 IML-Testprogramme

6.2.1 Conditional Expression

Input: n = 10, m = 7

```
program programConditionalExpression
global
  const n:int64; const m:int64; var s:int64;

  fun fibonacciRek(copy const a:int64) returns r:int64
  do
    r init := a = 0 ? 0 : a = 1 ? 1 : fibonacciRek(a - 1) + fibonacciRek(a - 2);
    debugout r
  endfun

do
  debugin n init;
  debugin m init;
  debugout fibonacciRek(n < m ? n : m);

  s init := n > m ? fibonacciRek(n) : fibonacciRek(m > fibonacciRek(n) ? fibonacciRek(m) * 2 :
                                                    fibonacciRek(n) * 4)

  //s init := n ? 12 : 64          // does not work, condExpr need to be of type bool
  //s init := true ? n : false     // does not work, exprTrue and exprFalse not same type
  //s init := n > m ? true : false // does not work, cant assign bool to int64
Endprogram
```

Output: 13

```
0: AllocBlock(1)
1: AllocBlock(1)
2: AllocBlock(1)
3: UncondJump(37)
4: LoadAddrRel(-2)
5: LoadAddrRel(-1)
6: Deref
7: LoadImInt(0)
8: EqInt
9: CondJump(12)
10: LoadImInt(0)
11: UncondJump(32)
12: LoadAddrRel(-1)
13: Deref
14: LoadImInt(1)
15: EqInt
16: CondJump(19)
17: LoadImInt(1)
18: UncondJump(32)
19: AllocBlock(1)
20: LoadAddrRel(-1)
21: Deref
22: LoadImInt(1)
23: SubInt
24: Call(4)
25: AllocBlock(1)
26: LoadAddrRel(-1)
27: Deref
28: LoadImInt(2)
29: SubInt
30: Call(4)
31: AddInt
32: Store
33: LoadAddrRel(-2)
34: Deref
35: OutputInt("r")
36: Return(1)
37: LoadAddrAbs(0)
38: InputInt("n")
39: LoadAddrAbs(1)
40: InputInt("m")
41: AllocBlock(1)
42: LoadAddrAbs(0)
43: Deref
44: LoadAddrAbs(1)
45: Deref
46: LtInt
47: CondJump(51)
48: LoadAddrAbs(0)
49: Deref
50: UncondJump(53)
51: LoadAddrAbs(1)
52: Deref
53: Call(4)
54: OutputInt("<anonymous>")
55: LoadAddrAbs(2)
56: LoadAddrAbs(0)
57: Deref
58: LoadAddrAbs(1)
59: Deref
60: GtInt
61: CondJump(67)
62: AllocBlock(1)
63: LoadAddrAbs(0)
64: Deref
65: Call(4)
66: UncondJump(90)
67: AllocBlock(1)
68: LoadAddrAbs(1)
69: Deref
70: AllocBlock(1)
71: LoadAddrAbs(0)
72: Deref
73: Call(4)
74: GtInt
75: CondJump(83)
76: AllocBlock(1)
77: LoadAddrAbs(1)
78: Deref
79: Call(4)
80: LoadImInt(2)
81: MultInt
82: UncondJump(89)
83: AllocBlock(1)
84: LoadAddrAbs(0)
85: Deref
86: Call(4)
87: LoadImInt(4)
88: MultInt
89: Call(4)
90: Store
91: Stop
```

6.2.2 Case Commands 1

Input: x = 6

```
program programSwitchCase
global
  x:int64;
  y:int64;

  fun f(m:int64) returns r:int64
  do
    r := m - 1
  endfun

do
  debugin x init;
  y init := 0;
  switch f(x)
    case 5:
      y := x * 5
    endcase
    case 7:
      y := x * (((7)))
    endcase
    defaultcase:
      y := x divE 2
    endcase
  endswitch;
  debugout y
endprogram
```

Output: 30

```
0: AllocBlock(1)
1: AllocBlock(1)
2: UncondJump(10)
3: LoadAddrRel(-2)
4: LoadAddrRel(-1)
5: Deref
6: LoadImInt(1)
7: SubInt
8: Store
9: Return(1)
10: LoadAddrAbs(0)
11: InputInt("x")
12: LoadAddrAbs(1)
13: LoadImInt(0)
14: Store
15: AllocBlock(1)
16: LoadAddrAbs(0)
17: Deref
18: Call(3)
19: Dup
20: LoadImInt(5)
21: LtInt
22: NegBool
23: CondJump(49)
24: Dup
25: LoadImInt(7)
26: GtInt
27: NegBool
28: CondJump(49)
29: LoadImInt(5)
30: SubInt
31: RelJump(32) + index
32: UncondJump(35)
33: UncondJump(49)
34: UncondJump(42)
35: LoadAddrAbs(1)
36: LoadAddrAbs(0)
37: Deref
38: LoadImInt(5)
39: MultInt
40: Store
41: UncondJump(55)
42: LoadAddrAbs(1)
43: LoadAddrAbs(0)
44: Deref
45: LoadImInt(7)
46: MultInt
47: Store
48: UncondJump(55)
49: LoadAddrAbs(1)
50: LoadAddrAbs(0)
51: Deref
52: LoadImInt(2)
53: DivEuclInt
54: Store
55: LoadAddrAbs(1)
56: Deref
57: OutputInt("y")
58: Stop
```

6.2.3 Case Commands 2

```
program allExceptExtension
global
  const typeOfDiv:int64;
  const dividend:int64;
  const divisor:int64;
  quotientenWert:int64;
  restWert:int64;

  proc T(copy const a:int64, copy const b:int64, ref q:int64, ref r:int64)
  do
     q := a divT b;
     r := a modT b
  endproc;

  proc F(copy const a:int64, copy const b:int64, ref q:int64, ref r:int64)
  do
     q := a divF b;
     r := a modF b
  endproc;

  proc E(copy const a:int64, copy const b:int64, ref q:int64, ref r:int64)
  do
     q := a divE b;
     r := a modE b
  endproc

do
  debugin typeOfDiv init;
  debugin dividend init;
  debugin divisor init;
  switch typeOfDiv
    case 5:
      call T(dividend, divisor, quotientenWert, restWert)
    endcase
    case 10:
      call F(dividend, divisor, quotientenWert, restWert)
    endcase
    case 15:
      call E(dividend, divisor, quotientenWert, restWert)
    endcase
    defaultcase:
      //call T(dividend, divisor, quotientenWert, restWert)
      debugout 99
    endcase
  endswitch;

  debugout quotientenWert;
  debugout restWert

endprogram
```

6.2.4 While Loop (Factorial)

```
// Source: https://en.wikipedia.org/wiki/While_loop
program allExceptExtension
global
  const input:int64;
  const result:int64;

  // test basic program
  proc factorial(copy const value:int64, ref var factorial:int64)
  local
    var counter:int64

  do

    counter := value;
    factorial := 1;

    while counter > 1 do
      factorial := factorial * counter;
      counter := counter - 1
    endwhile
  endproc

do
  debugin input init;
  call factorial(input, result init);
  debugout result
endprogram
```

6.2.5 Special Cases

```
program SpecialCases
global
  g1:int64; g2:int64; // integers
  const g3:bool;      // no access after first write
  g4:bool; g5:bool;   // boolean
  g6:bool;
  //g5:int64;         // does not work, name already declared

  // function
  fun fCheckStuff(a1:int64,
                  copy var a2:bool, copy const a3:int64,
                  ref var a4:bool, ref const a5:int64) returns const r1:int64
  local
    var l1:int64;
    var l3:bool;
    //var g1:bool;          // does not work, name already globally declared
    const l2:int64
  do
    l1 init := 5;
    a1 := 20;
    //g3 := false;          // does not work, can't change global const variable

    l2 init := 0;
    //l2 := 4;              // does not work, can't change local const variable

    //a5 := 66;             // does not work, can't change const variable
    a4 := false;            // change ref param a4 from caller

    call pCheckStuff(l1);   // check change of local l1 over ref param
    //call pCheckStuff(l2); // does not work, cant change local const variable l2
                                                                  over ref param
    r1 init := 999
    //r1 := 888             // does not work, can't change const return variable
  endfun;

  // procedure
  proc pCheckStuff(ref a1:int64)
  do
    skip;
    //g5 := false        // does not work, g5 need to be initialised
    g5 init := false;    // init and change global variable g5
    a1 := ((777))         // change ref variable a1 from caller
  endproc

do
  g4 init := true;
  ////////////////////// init checking ////////////////////////////
  //debugout g1; // does not work, need to be initialised
  g1 init := 0;  // first init from global variable
  g1 := 50;
  //debugin g2;      // does not work, need to be initialised
  debugin g2 init;
  //g1 init;     // does not work, for safety if u init a variable then set
                                                              something
  //g1 init := 4;// does not work, g1 is already initialised

  ////////////////////// const checking ////////////////////////////
  g3 init := true;  // first init from global const variable
  //g3 := false;    // does not work, can't change const variable
  //debugin g3;     // does not work, can't change const variable
  debugout g3;
  //debugout fCheckStuff(10, true, 10, g3, g1); // does not work, can't change
                                     global const variable over ref param

  ////////////////////// type checking ////////////////////////////
  //g5 init := g1; // does not work, can't assign int64 to bool
  //g1 := g4;      // does not work, can't assign bool to int64
```

```
  //debugout fCheckStuff(g1, true, true, g4, g1); // does not work, expected int64
                                                                    but found bool
  //debugout fCheckStuff(g1, 10, 10, g4, g1); // does not work, expected bool but
                                                                        found int64

  ////////////////////// name checking ////////////////////////////
  //debugout g0;   // does not work, name is not globally declared

  ////////////////////// invalid param count checking/////////////////////////////
  //debugout fCheckStuff(10, true, 10, g4, g1, 10); // does not work, expected 5
                                                                    params found 6
  //call pCheckStuff(g1, 10);          // does not work, expected 1 param found 2

  ////////////////////// LRval checking ////////////////////////////
  ////// expected fCheckStuff(rval, rval, rval, lval, lval) ///////////
  debugout fCheckStuff(g1, true, 10, g4, g1);     // rval assigable to lval (will
                                                 be copied, no change of the g1)
  //debugout fCheckStuff(g1, true, 10, true, g1); // does not work, expected lval
                                                                    but found rval
  //call pCheckStuff(10);           // does not work, expected lval but found rval

  ////////////////////// routine call checking ////////////////////////////
  call pCheckStuff(g1);   // check call of procedure
  debugout g5;            // check change of global g5 in pCkeckStuff
  debugout g1;            // check change of global g1 over ref param in proc
  debugout fCheckStuff(10, true, 10, g4, g1);// check call of function
  debugout g4             // check change of global g4 over ref param in func
endprog
```