NAT32 und Casting für IML

Marco Romanutti[1,2] und Benjamin Meyer[1,2]

[1] *Fachhochschule Nordwestschweiz FHNW, Brugg*
[2] Schlussbericht

Im Modul Compilerbau wird eine Erweiterung für die bestehende Sprache IML spezifiziert und implementiert. Die Implementierung beinhaltet einen neuen Datentyp für natürliche Zahlen, sowie eine Möglichkeit den Datentyp INT32 in den neuen Datentypen zu casten und umgekehrt.

1 Erweiterung

1.1 Einleitung

Unter natürlichen Zahlen werden die positiven, ganzen Zahlen und 0 verstanden:

$$\mathbb{N} = \{0; 1; 2; 3; \ldots\}$$

Die IML soll um einen neuen Datentyp `NAT32` erweitert werden. Der neue Datentyp soll solche positiven, ganzen Zahlen bis Länge 32 in Binärdarstellung abbilden können. Es sollen die bestehenden Operationen unterstützt werden. Ausserdem soll ein explizites Casting zwischen dem bestehenden Datentyp `INT32` und dem neuen Datentyp `NAT32` möglich sein.

1.2 Lexikalische Syntax

Für den neuen Datentyp wird das Keyword `(TYPE, NAT32)` und ein Castingoperator hinzugefügt.

```
Datentyp:   NAT32    (TYPE, NAT32)
Brackets:   [ ]      LBRACKET, RBRACKET
```

Casting ist nur von `(TYPE, INT32)` zu `(TYPE, NAT32)` und umgekehrt möglich. Als Castingoperator werden rechteckige Klammern (nachfolgend Brackets gennant) verwendet. Innerhalb der Brackets befindet sich der Zieldatentyp [1].

[1] zum Beispiel `[INT32]`

1.3 Grammatikalische Syntax

Das nachfolgende Code-Listing zeigt, wie der neue Datentyp `NAT32` eingesetzt werden kann.

```
// Deklaration
var natIdent1 : nat32;
var natIdent2 : nat32;
var natIdent3 : nat32;

// Initialisierung
natIdent1 init := [nat32] 50;
natIdent2 init := [nat32] 10;
natIdent3 init := natIdent1 + natIdent2;

// Casting von INT32 nach NAT32
var intIdent1 : int32;
intIdent1 init := 30;
natIdent3 := [nat32] intIdent1;

call functionWithNatParam([nat32] intIdent1);

// Casting von NAT32 nach INT32
var intIdent2 : int32;
intIdent2 init := [int32] natIdent3;

call functionWithIntParam([int32] natIdent3);
```

Literale werden standardmässig als `INT32` interpretiert - ein `NAT32`-Literal bedingt vorab deshalb den Castingoperator. Falls zwei Datentypen nicht gecastet werden können, wird ein Kompilierungsfehler geworfen. Folgendes Code-Listing zeigt ein solches Beispiel mit dem bestehenden Datentyp `bool`:

```
// Deklaration
var boolIdent : bool;
boolIdent init := false;
var natIdent : nat32;
// Throws type checking error:
natIdent init := [nat32] boolIdent
```

Unsere Erweiterung unterstützt keine impliziten Castings. Weitere Code-Beispiele sind im Anhang zu finden.

Ebenfalls haben wir in unserer IML Syntax das logische Symbol für `AND: /\` sowie `OR: \/` durch `AND: &&` resp. `OR: ||` ersetzt. Nach unserem Erachten, ist die Bedeutung von `&&` und `||` den meisten Programmierern besser bekannt, als `/\` oder `\/`.

1.4 Änderungen an der Grammatik

Zusätzlich zu den bestehende Operatoren wurde ein neuer `castOpr` erstellt, welcher anstelle des Nichtterminal-Symbol `factor` verwendet werden kann.

```
castOpr := LBRACKET TYPE RBRACKET
```

Das bestehende Nichtterminal-Symbol `factor` wird um diese neue Produktion ergänzt:

```
factor := LITERAL
| IDENT [INIT | exprList]
| castOpr factor
| monadicOpr factor
| LPAREN expr RPAREN
```

1.5 Kontext- und Typen-Einschränkungen

Der `TYPE` zwischen `LBRACKET` und `RBRACKET` muss vom Datentyp `INT32` oder `NAT32` sein. Ein Casting zum Typ `bool` oder vom Typ `bool` zu `INT32` resp. `NAT32` führt zu einem Kompilierungsfehler.

Tabelle 1 zeigt die unterstützen Typumwandlungen der verschiedenen Datentypen. Typumwandlungen, welche zu potentiellen Laufzeitfehler führen, sind mit mit * gekennzeichnet. Der Datentyp `INT32` umfasst einen Wertebereich von -2147483648 bis 2147483647, wobei das Most Significant Bit (MSB) für das Vorzeichen verwendet wird. Weil der Datentyp `NAT32` nur positive, ganze Zahlen und die Zahl 0 darstellt, wird kein Vorzeichenbit benötigt. Der Wertebereich verschiebt sich dadurch auf 0 bis 4294967295. Falls bei Typumwandlungen der Wert ausserhalb des Wertebereichs des Zieldatentyps liegt, führt dies zu einem Laufzeitfehler. Bei der Umwandlung von `NAT32` nach `INT32` kann ein solcher Laufzeitfehler beispielsweise auftreten, falls es sich um einen Wert > 2147483647 handelt. Falls negative Werte von `INT32` nach `NAT32` umgewandelt werden, resultiert ebenfalls ein Laufzeitfehler.

Table 1: *Casting zwischen Datentypen*

Quell- \ Zieldatentyp	int32	nat32	bool
int32	✓	✓*	✗
nat32	✓*	✓	✗
bool	✗	✗	✗

2 Aufbau Compiler

Der Compiler basiert auf der IML (V2) und ist in Java geschrieben.

2.1 Scanner

Literale werden standardmässig als `INT32` interpretiert - ein `NAT32`-Literal bedingt vorab deshalb den Castingoperator. Vom Scanner werden Literale als `long` in Java eingelesen. Dieser kann Werte von -9223372036854775808 bis 9223372036854775808 annehmen und deckt somit den gesamten Wertebereich der beiden Datentypen `INT32` und `NAT32` ab. Die Überprüfung, ob der Wert innerhalb des gültigen Wertebereichs des jeweiligen Datentyps liegt, erfolgt zum Zeitpunkt der Code-Generierung.

2.2 Parser

Der neu eingeführte `castOpr` und die neue Produktion `factor := castOpr factor` sind im Abstrakten Syntax Tree (AST) abgebildet. Dabei wird der Typ `CastOpr` zum Typ `CastFactor` welcher Teil des AST ist. Abbildung 1 zeigt die Umwandlung von der konkreten in die abstrakte Syntax.

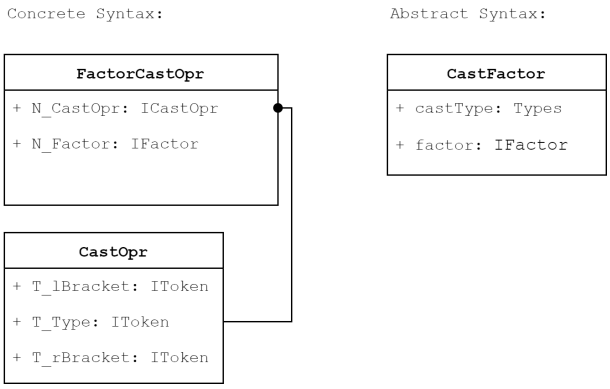

Figure 1: *Umwandlung von der konkreten in die abstrakte Syntax*

2.3 Statische Analyse

Scope checking

Für Routinen und Variablen liegen unterschiedliche Namespaces vor. Namen für Routinen und Variabeln können deshalb identisch sein. Es wird zwischen globalen und lokalen Namespaces unterschieden: Bei lokalen Namespaces werden die Variable innerhalb einer Routine definiert und haben dort ihre Gültigkeit. Globale Variabeln dürfen nicht denselben Namen haben wie lokalen Variabeln. Überladene Signaturen für Routinen[2] wurde in dieser Implementation nicht umgesetzt.

[2] selber Name, unterschiedliche Parameterlisten

Bei *FunCallFactor, ProcCallCmd, DebugIn* und *AssignCmd* muss überprüft werden, ob die Parameter den richtigen `LValue`, resp. `RValue` besitzen. Folgende Kombinationen sind dabei allgemein erlaubt:

Table 2: *LRValue-Kombinationen*

Callee	Caller	Resultat
LValue	LValue	Valid
RValue	LValue	Valid (LValue dereferenzieren)
RValue	RValue	Valid
LValue	RValue	LRValueError ↯

Bei einem *AssignCmd* muss der Ausdruck links zudem zwingend ein `LValue` sein. Bei einem *DebugIn* muss es sich ebenfalls um einen `LValue` handeln, damit der Input-Wert dieser Variable zugewiesen werden kann.

Innerhalb des Scope checkings wird zudem überprüft, ob die Anzahl der erwarteten Parameter mit der Anzahl übergebener Parameter übereinstimmt.

Type checking

Das Casten zwischen zwei Datentypen ist nur für bestimmte Typen erlaubt (vgl. Tabelle 1). Zusätzlich sind bei der Abarbeitung des AST nur die folgenden Typen erlaubt:

Table 3: *Erlaubte Typen*

Klasse	Types
AddExpr, MultExpr, RelExpr	`int32, nat32` *
BoolExpr, IfCmd	`bool`
AssignCmd	`int32, nat32, bool` *
FunCallFactor, ProcCallFactor	Typ von Caller muss Typ von Callee entsprechen
MonadicFactor	NOTOPR: `bool` ADDOPR: `int32, nat32`
CastFactor	Typ von Factor und Typ von CastFactor müssen castable sein

Bei Einträgen, die mit * gekennzeichnet sind, müssen `LValue` und `RValue` vom selben Typ sein. Beim der Typenüberprüfung innerhalb vom `CastFactor` wird überprüft, ob der Typ vom `CastFactor` und jener des zugehörigen `factors` gecasted werden können (vgl. Tabelle 1). Die effektive Typ-Konversion wird erst bei der Code-Generierung durchgeführt.

Initalization checking

Über das Interface `IAstNode` müssen alle Knotenpunkte vom AST, die Methode `executeInitCheck` implementieren. Der AST wird dabei von unten nach oben überprüft.

Anhand nachfolgendem Besipielprogramm wird erklärt, wann welcher Fehler auftreten kann. Die Fehler werden unterhalb näher beschrieben:

```
program exampleInitErrors
global
    var x:int32;
    const y:int32
do
    x := 7; // throws NotInitializedError
    x init := 4;
    x init := 3; // throws
        AlreadyInitializedError
    y init := 1;
    y := 2 // throws CannotAssignToConstError
endprogram
```

Folgende Fehler können dabei auftreten:

- `NotInitializedError`: Die Variable wurde zuvor nicht initialisiert.
- `AlreadyInitializedError`: Die Variable wurde bereits zuvor initialisiert.
- `CannotAssignToConstError`: Die Variable ist eine Konstante und wurde bereits einmal inizialisiert.

Falls einer dieser Fehler auftritt, wird die Überprüfung sofort beendet und der Initialisierungscheck ist fehlgeschlagen.

2.4 Virtuelle Maschine

Grundlage für die Code-Generierung ist der AST. Vom Root-Knoten ausgehend fügt jeder Knoten seinen Code zum Code-Array.

Im Falle eines Castings zwischen zwei Datentypen befindet sich an mindestens einer Stelle in der AST Struktur ein `CastFactor`-Element. Von diesem Element aus wird der Datentyp des zugehörigen `factor` geändert, indem dessen Attribut `castFactor` geändert wird. Dieses Attribut übersteuert den eigentlichen Datentyp des Elements innerhalb des AST. Weil der `factor` gemäss Grammatik unterschiedliche Produktionen besitzt (vgl. Änderungen an der Grammatik), muss die Typanpassung rekursiv weitergegeben werden. Die Rekursion wird durch Literale oder Expressions unterbrochen, wie am Beispiel in Anhang B aufgezeigt.

In der virtuellen Maschine wurde ein neuer generischer Typ `NumData` eingeführt. Dieser wird für die Konversion zwischen Daten vom Typ `IntData`[3] und `NatData`[4] verwendet. Abbildung 2 zeigt die Klassenhierarchie dieser Typen.

[3] für den Datentyp `INT32`
[4] für den Datentyp `NAT32`

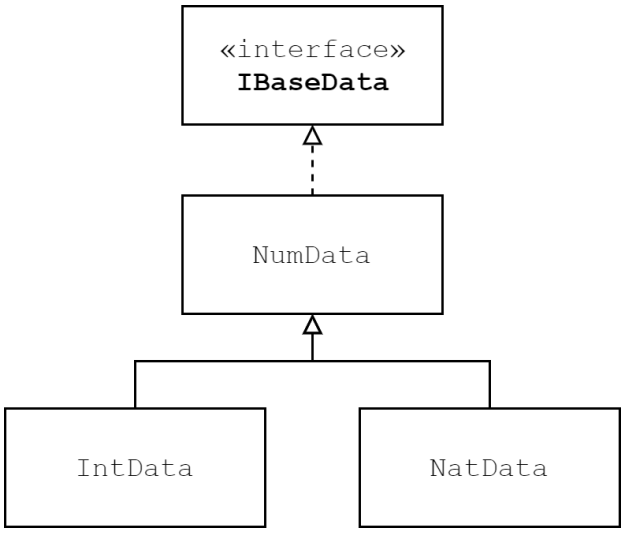

Figure 2: *Daten in VM*

Beim Dereferenzieren muss im Falle eines Castings der Datentyp von bereits typisierten Daten auf dem Stack geändert werden. Anhang C zeigt ein Beispiel, bei welchem der Datentyp aufgrund des neu propagierten Attributs `castType` umgewandelt wird.

2.5 Code Generierung

3 Vergleich mit anderen Programmiersprachen

3.1 Ganzzahlige Werte

In Java wird bei Zuweisungen die Länge einer Zahl in Bitdarstellung überprüft: Beim Datentyp `long` wird beispielsweise geprüft, ob der Wert als ganzzahliger Wert von 32-bit Länge dargestellt werden kann. Falls dies nicht der Fall ist, wird ein Fehler zur Kompilierungszeit geworfen. Das MSB wird als Vorzeichenbit verwendet, womit rund die Hälfte der vorzeichenlos darstellbaren `long`-Werte entfällt, resp. zur Darstellung von negativen Zahlen eingesetzt wird. Falls bei fortlaufenden Berechnungen Wertebereiche unter- resp. überschritten werden, führt dies zu einem arithmetischen Überlauf. Abbildung 3 zeigt den Überlauf bei ganzzahligen, vorzeichenbehafteten Datentypen (am Beispiel von Bitlänge 3 + 1).

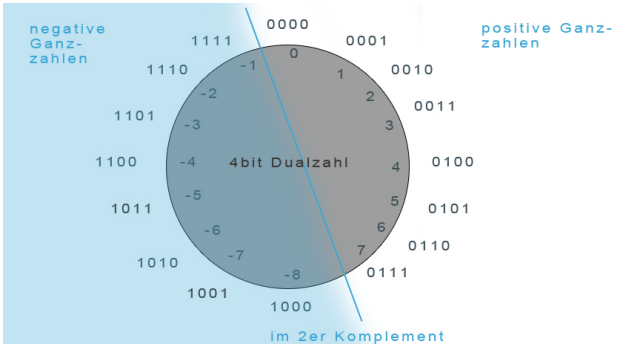

Figure 3: *Überlauf mit Integerzahlen*

Dadurch führt z.B. beim Datentyp `int` der Ausdruck `Integer.MAX_VALUE + 1` zum Wert `Integer.MIN_VALUE`. Dies kann dazu führen, dass mit „falschen“ Werten gerechnet wird, ohne dass der Entwickler dies bemerkt.

3.2 Fliesskommazahlen

Im Gegensatz zur Darstellung im Zweierkomplement, welches für Integer-Typen in Java verwendet werden, werden Fliesskommazahlen intern nach IEEE Standard dargestellt. Anders als bei der Zweierkomplement-Darstellung sieht dieses Format spezielle, konstante Werte vor. So sind in Java beispielsweise Konstanten für `Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY` und `Double.NaN` definiert.

4 Designentscheidungen

4.1 Spezifiziertes Verhalten

Der neue Datentyp `NAT32` unterstützt die bestehenden Operationen aus IML[5]. Sofern sich die einzelnen Operanden und auch das Resulat im Wertebereich[6] befinden, entspricht das Verhalten vom Datentyp `NAT32` jenem vom Datentyp `INT32`. Andernfalls wird folgendes Verhalten festgelegt:

- **Wertebereich**: Wertebereichsüber- resp. Unterschreitungen resultieren in einem Laufzeitfehler[7]. Dies erhöht die Typsicherheit beim Einsatz der verschiedenen Datentypen.
- **Nachkommastellen**: Gemäss IML-Spezifikation sind als Literale nur ganzzahlige Werte erlaubt. Falls z.B. bei einer Division ein Rest resultiert, wird ein ganzzahliges Resultat zurückgegeben. Der

[5] Aktuell sind dies

- MULTOPR(*, divE, modE)
- ADDOPR(+, -)
- RELOPR(<, <=, >, >=, =, /=)
- BOOLOPR(, ||, , |)

[6] [0, 4294967295]

[7] Negative Zahlen entsprechen Wertebereichsunterschreitungen

Wert des Resultats ist abhängig von der gewählten Operation (`DivFloor, DivTrunc, etc.`).

4.2 Alternative Ansätze

Folgende weiteren Ansätze wurden für die Umsetzung der Erweiterung in Betracht gezogen:

- **Arithmetischer Überlauf:** Wertebereisüber- resp. unterschreitungen resultieren in einem arithmetischen Überlauf. Gegenüber der Darstellung in Abbildung 3 müsste kein negativer Wertebereich verwendet werden und das Addieren von $+1$ zum grössten Darstellbaren Wert des Datentyps `NAT32` führt zum Wert 0. Weil auf ein Vorzeichenbit verzichtet werden kann, verdoppelt sich der Wertebereich gegenüber dem Datentyp `INT32`. Nachteilig ist dabei, dass der Entwickler verantwortlich ist für das Einhalten der Wertebereichsgrenzen.
- **Vordefinierte Konstanten:** Ähnlich wie bei Fliesskommazahlen (vgl. Kapitel 3.2) könnten konstante Werte für z.B. `POSITIVE_INFINITY` und `NEGATIVE_INFINITY` vorgesehen werden. Das Verhalten bei Wertebereichsüberschreitungen müsste definiert werden. Nachteilig bei dieser Variante ist, dass der Wertebereich um die Anzahl solcher konstante Werte verringert wird.
- **Absolute Werte:** Bei dieser Variante wird bei negativen Werten deren Absolutwert verwendet. Von dieser initial angedachten Variante wurde abgesehen, weil das Verhalten schnell zu ungewollten Resultaten führen kann.

5 Beispielprogramme

Operation:

```
program progAddition
global
    var x:nat32;
    var y:nat32;
    var r:nat32;
    var b:bool
do
    x init := 4;
    y init := 3;
    r init := x + y;
    b init := r = 7;
    debugout r;
    debugout b
endprogram
```

Casting:

```
program progCasting
global
    var x:nat32;
    var y:int32;
    var r:nat32;
    var b:bool
do
    x init := 4;
    y init := 3;
    r init := x + [nat32] y;
    b init := r = 7;
    debugout r;
    debugout b
endprogram
```

References

[1] Wikipedia: Natürliche Zahl, `https://de.wikipedia.org/wiki/Nat\%C3\%BCrliche_Zahl`

[2] Wikipedia: Natural numbers (engl.), `https://en.wikipedia.org/wiki/Natural_number`

[3] Wikipedia: Integer (Datentyp) `https://de.wikipedia.org/wiki/Integer_(Datentyp)`

A Vollständige Grammatik

```
program ::= PROGRAM IDENT progParamList [GLOBAL
    cpsDecl] DO cpsCmd ENDPROGRAM

decl ::= stoDecl
       | funDecl
       | procDecl

stoDecl ::= [CHANGEMODE] typedIdent

funDecl ::= FUN IDENT paramList RETURNS stoDecl
    [GLOBAL globImps] [LOCAL cpsStoDecl] DO
    cpsCmd ENDFUN

procDecl ::= PROC IDENT paramList [GLOBAL
    globImps] [LOCAL cpsStoDecl] DO cpsCmd
    ENDPROC

globImps ::= globImp {COMMA globImp}

globImp ::= [FLOWMODE] [CHANGEMODE] IDENT

cpsDecl ::= decl {SEMICOLON decl}

cpsStoDecl ::= stoDecl {SEMICOLON stoDecl}

progParamList ::= LPAREN [progParam {COMMA
    progParam}] RPAREN

progParam ::= [FLOWMODE] [CHANGEMODE] typedIdent

paramList ::= LPAREN [param {COMMA param}] RPAREN

param ::= [FLOWMODE] [MECHMODE] [CHANGEMODE]
    typedIdent

typedIdent ::= IDENT COLON TYPE

cmd ::= SKIP
       | expr BECOMES expr
       | IF expr THEN cpsCmd ELSE cpsCmd ENDIF
       | WHILE expr DO cpsCmd ENDWHILE
       | CALL IDENT exprList [globInits]
       | DEBUGIN expr
       | DEBUGOUT expr

cpsCmd ::= cmd {SEMICOLON cmd}

globInits ::= INIT idents

idents ::= IDENT {COMMA IDENT}

expr ::= term1 {BOOLOPR term1}

term1 ::= term2 [RELOPR term2]

term2 ::= term3 {ADDOPR term3}

term3 ::= factor {MULTOPR factor}

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

castOpr ::= LBRACKET TYPE RBRACKET

exprList ::= LPAREN [expr {COMMA expr}] RPAREN

monadicOpr ::= NOT | ADDOPR
```

B Beispiel doppeltes Casting

IML:

```
program progDouble
global
    var value:int32
do
    value init := [int32] [nat32] ((4 + 1) + 1)
endprogram
```

Code-Array:

```
0: AllocBlock(1)
1: UncondJump(2)
2: LoadAddrAbs(0)
3: LoadImInt(4)
4: LoadImInt(1)
5: AddInt
6: LoadImInt(1)
7: AddInt
8: Store
9: Stop
```

UML:

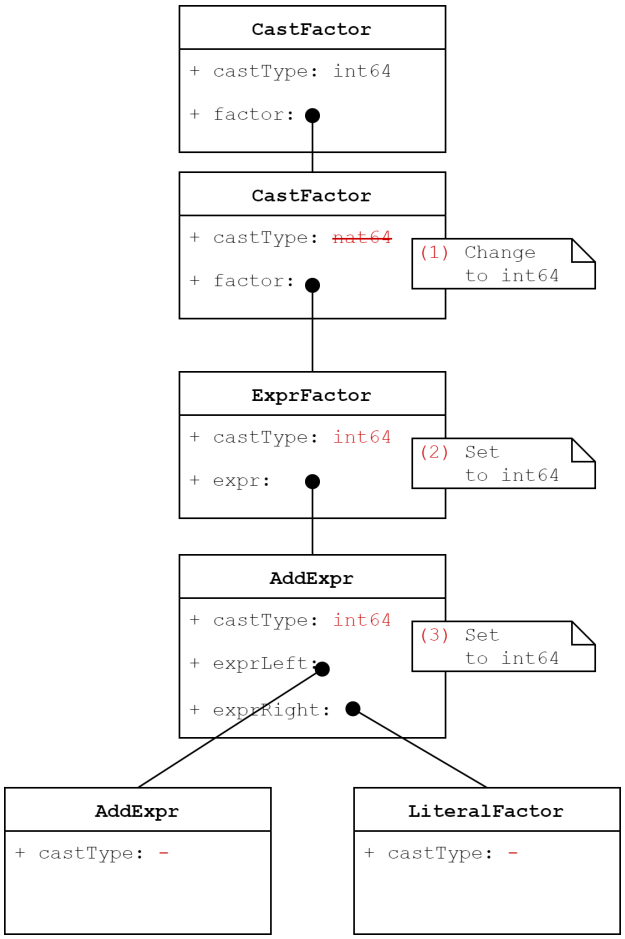

Figure 4: *Auszug aus AST*

C Beispiel `Deref`

IML:

```
program exampleCasting
global
    var x:int32;
    var y:nat32
do
    x init := [int32] 4;
    y init := [nat32] 3;
    x := [int32] y
endprogram
```

Code-Array:

```
codeArrayPoiner: 14
0: AllocBlock(1)
1: AllocBlock(1)
2: UncondJump(3)
3: LoadAddrAbs(0)
4: LoadImInt(4)
5: Store
6: LoadAddrAbs(1)
7: LoadImNat(3)
8: Store
9: LoadAddrAbs(0)
10: LoadAddrAbs(1)
11: Deref
12: Store
13: Stop
```

Stack:

```
    ...

(8) Executing instruction Store
pc: 8
sp: 4
Stack content:
0: IntData: 4
1: NatData: 3

(9) Executing instruction LoadAddrAbs(0)
pc: 9
sp: 2
Stack content:
0: IntData: 4
1: NatData: 3
2: IntData: 0

(10) Executing instruction LoadAddrAbs(1)
pc: 10
sp: 3
Stack content:
0: IntData: 4
1: NatData: 3
2: IntData: 0
3: IntData: 1

(11) Executing instruction Deref
pc: 11
sp: 4
Stack content:
0: IntData: 4
1: NatData: 3
2: IntData: 0
3: IntData: 3

(12) Executing instruction Store
pc: 12
sp: 4
Stack content:
0: IntData: 3
1: NatData: 3

    ...
```

D Beispiel casting einer expression

IML:

```
program exampleCastExpression
global
    var x:int32;
    var y:int32;
    var z:nat32;
    var b:bool;
    var c:bool
do
    x init := 4;
    y init := 2;
    z init := [nat32] (x * y + y + x);
    b init := x >= y;
    c init := x >= [int32] z
endprogram
```

Code-Array:

```
codeArrayPoiner: 40
0: AllocBlock(1)
1: AllocBlock(1)
2: AllocBlock(1)
3: AllocBlock(1)
4: AllocBlock(1)
5: UncondJump(6)
6: LoadAddrAbs(0)
7: LoadImInt(4)
8: Store
9: LoadAddrAbs(1)
10: LoadImInt(2)
11: Store
12: LoadAddrAbs(2)
13: LoadAddrAbs(0)
14: Deref
15: LoadAddrAbs(1)
16: Deref
17: MultInt
18: LoadAddrAbs(1)
19: Deref
20: AddInt
21: LoadAddrAbs(0)
22: Deref
23: AddNat
24: Store
25: LoadAddrAbs(3)
26: LoadAddrAbs(0)
27: Deref
28: LoadAddrAbs(1)
29: Deref
30: GeInt
31: Store
32: LoadAddrAbs(4)
33: LoadAddrAbs(0)
34: Deref
35: LoadAddrAbs(2)
36: Deref
37: GeInt
38: Store
39: Stop
```

Stack:

```
    ...
(12) Executing instruction LoadAddrAbs(2)
pc: 12
sp: 5
Stack content:
0: IntData: 4
1: IntData: 2
2: null
3: null
4: null
5: IntData: 2

(13) Executing instruction LoadAddrAbs(0)
pc: 13
sp: 6
Stack content:
0: IntData: 4
1: IntData: 2
2: null
3: null
4: null
5: IntData: 2
6: IntData: 0

(14) Executing instruction Deref
pc: 14
sp: 7
Stack content:
0: IntData: 4
1: IntData: 2
2: null
3: null
4: null
5: IntData: 2
6: IntData: 4

(15) Executing instruction LoadAddrAbs(1)
pc: 15
sp: 7
Stack content:
0: IntData: 4
1: IntData: 2
2: null
3: null
4: null
5: IntData: 2
6: IntData: 4
7: IntData: 1

(16) Executing instruction Deref
pc: 16
sp: 8
Stack content:
0: IntData: 4
1: IntData: 2
2: null
3: null
4: null
5: IntData: 2
6: IntData: 4
```

```
7: IntData: 2

(17) Executing instruction MultInt
pc: 17
sp: 8
Stack content:
0: IntData: 4
1: IntData: 2
2: null
3: null
4: null
5: IntData: 2
6: IntData: 8

(18) Executing instruction LoadAddrAbs(1)
pc: 18
sp: 7
Stack content:
0: IntData: 4
1: IntData: 2
2: null
3: null
4: null
5: IntData: 2
6: IntData: 8
7: IntData: 1

(19) Executing instruction Deref
pc: 19
sp: 8
Stack content:
0: IntData: 4
1: IntData: 2
2: null
3: null
4: null
5: IntData: 2
6: IntData: 8
7: IntData: 2

(20) Executing instruction AddInt
pc: 20
sp: 8
Stack content:
0: IntData: 4
1: IntData: 2
2: null
3: null
4: null
5: IntData: 2
6: IntData: 10

(21) Executing instruction LoadAddrAbs(0)
pc: 21
sp: 7
Stack content:
0: IntData: 4
1: IntData: 2
2: null
3: null
4: null
5: IntData: 2
6: IntData: 10
7: IntData: 0

(22) Executing instruction Deref
pc: 22
sp: 8
Stack content:
0: IntData: 4
1: IntData: 2
2: null
3: null
4: null
5: IntData: 2
6: IntData: 10
7: IntData: 4

(23) Executing instruction AddNat
pc: 23
sp: 8
Stack content:
0: IntData: 4
1: IntData: 2
2: null
3: null
4: null
5: IntData: 2
6: NatData: 14

(24) Executing instruction Store
pc: 24
sp: 7
Stack content:
0: IntData: 4
1: IntData: 2
2: NatData: 14
3: null
4: null

    ...
```