Nat64 und Casting für IML

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

[1] *Fachhochschule Nordwestschweiz FHNW, Brugg*
[2] `Zwischenbericht`

Im Modul Compilerbau wird eine Erweiterung für die bestehende IML spezifiziert und implementiert. Neu soll ein Datentyp für natürliche Zahlen unterstützt werden. Werte vom bestehenden Datentyp int64 können in den neuen Datentypen gecastet werden und umgekehrt.

1 Compiler

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

2 Erweiterung

2.1 Einleitung

Unter natürlichen Zahlen werden die positiven Zahlen und 0 verstanden. Die IML soll um einen neuen Datentyp `nat64` erweitert werden. Der neue Datentyp soll solche Zahlen mit bis zu 64 Ziffern abbilden können. Es sollen die bestehenden Operationen unterstützt werden. Ausserdem soll ein explizites Casting zwischen dem bestehenden Datentyp `int64` und dem neuen Datentyp `nat64` möglich sein.

2.2 Lexikalische Syntax

Für den neuen Datentyp wird das Keyword `(TYPE, NAT64)` und ein Castingoperator hinzugefügt.

Datentyp:	nat64	(TYPE, NAT64)
Brackets:	[ ]	LBRACKET, RBRACKET

Casting ist nur von `(TYPE, INT64)` zu `(TYPE, NAT64)` und umgekehrt möglich. Als Castingoperator wird die rechtekige Klammern (nachfolgend Brackets gennant) verwendet. Innerhalb der Brakets befindet sich der Zieldatentyp [1].

2.3 Grammatikalische Syntax

Das nachfolgende Code-Listing zeigt, wie der neue Datentyp `nat64` eingesetzt werden kann.

```
// Deklaration
var natIdent1 : nat64;
var natIdent2 : nat64;
var natIdent3 : nat64;

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

// Casting von int64 nach nat64
var intIdent1 : int;
intIdent1 init := 30
natIdent3 := [nat64] intIdent1;

call functionWithNatParam([nat64] intIdent1);

// Casting von nat64 nach int64
var intIdent2 : int;
intIdent2 init := [int64] natIdent3;

call functionWithIntParam([int64] natIdent3);
```

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 : nat64;
// Throws error:
natIdent init := [nat64] boolIdent
```

[1] zum Beispiel `[int64]`

Unsere Erweiterung unterstützt keine impliziten Castings. Weitere Code-Beispiele sind in Kapitel 4 zu finden.

2.4 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 LITERAL RBRACKET
```

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

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

2.5 Kontext- und Typen-Einschränkungen

Der Literal zwischen `LBRACKET` und `RBRACKET` muss vom Datentyp `int64` oder `nat64` sein. Ein Casting zum Typ `bool` oder vom Typ `bool` zu `int64` resp. `nat64` führt zu einem Kompilierungsfehler.

Der neue Datentyp `nat64` unterstützt die bestehenden Operationen aus IML[2]. Sofern sich die einzelnen Operanden und auch das Resulat im Wertebereich ($\in \mathbb{N}$) befinden, entspricht das Verhalten vom Datentyp `nat64` jenem vom Datentyp `int64`. Andernfalls wird folgendes Verhalten festgelegt:

- **Werteüberlauf**: Bei einem Überlauf wird jeweils mit dem maximalen Wert weitergerechnet. Dieser entspricht dem maximalen Wert von `int64`[3].
- **Negative Werte**: Werte werten jeweils als absolute Werte Betrachtet. Ein negativer Wert -5 entspricht beispielsweise dem Betrag, also $|-5| = 5$.
- **Rest bei Division**: Wird analog `int64` behandelt und Nachkommastellen werden abgeschnitten.

Tabelle 1 zeigt die unterstützen Typumwandlungen der verschiedenen Datentypen. Typumwandlungen, welche zu potentiellem Informationsverlust führen, sind mit mit $*$ gekennzeichnet.

[2] Aktuell sind dies

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

[3] 9,223,372,036,854,775,807

Table 1: *Casting zwischen Datentypen*

Quell- \ Zieldatentyp	int64	nat64	bool
int64	✓	✓*	✗
nat64	✓	✓	✗
bool	✗	✗	✗

3 Vergleich mit anderen Programmiersprachen

3.1 Java

In Java werden Wertebereichsunterschreibungen und -überschreibungen für Initialisierung und fortlaufende Berechnungen unterschiedliche gehandhabt: Bei der Initialisierung wird beispielsweise beim Datentyp `int` geprüft, ob die Werte innerhalb der Wertebereiche liegen. Falls dies nicht der Fall ist, wird ein Error geworfen. Falls bei fortlaufenden Berechnungen Wertebereiche unter- resp. überschritten werden, wird einfach auf dem Zahlenkreis weitergegangen und der entsprechende Wert verwendet. Dies kann dazu führen, dass mit „falschen“ Werten gerechnet wird, ohne dass der Entwickler dies bemerkt.

4 Beispielprogramme

Operation:

```
program progAddition
global
    var x:nat64;
    var y:nat64;
    var r:nat64;
    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:nat64;
    var y:int64;
    var r:nat64;
    var b:bool
do
    x init := 4;
    y init := 3;
    r init := x + [nat64] 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