

Nat64 und Casting für IML

Marco Romanutti^{1,2} und Benjamin Meyer^{1,2}

¹Fachhochschule Nordwestschweiz FHNW, Brugg

²Zwischenbericht

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 int64 in den neuen Datentypen zu casten und umgekehrt.

1 Erweiterung

1.1 Einleitung

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

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

Die IML soll um einen neuen Datentyp nat64 erweitert werden. Der neue Datentyp soll solche positiven, ganzen Zahlen bis Länge 64 in Binärdarstellung 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.

1.2 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 werden rechteckige Klammern (nachfolgend Brackets genannt) verwendet. Innerhalb der Brackets befindet sich der Zieldatentyp ¹.

¹zum Beispiel [int64]

1.3 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 := [nat64] 50;
natIdent2 init := [nat64] 10;
natIdent3 init := natIdent1 + natIdent2;

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

call functionWithNatParam([nat64] intIdent1);

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

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

Literale werden standardmässig als int64 interpretiert - ein nat64-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 : nat64;
// Throws type checking error:
natIdent init := [nat64] boolIdent
```

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

1.4 Änderungen an der Grammatik

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

```
castOpr := LBRACKET ATOMTYPE 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 `ATOMTYPE` 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.

Tabelle 1 zeigt die unterstützten Typumwandlungen der verschiedenen Datentypen. Typumwandlungen, welche zu potentiellen Laufzeitfehler führen, sind mit * gekennzeichnet. Der Datentyp `int64` umfasst einen Wertebereich von -2147483648 bis 2147483647 , wobei das Most Significant Bit (MSB) für das Vorzeichen verwendet wird. Weil der Datentyp `nat64` 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 `nat64` nach `int64` kann ein solcher Laufzeitfehler beispielsweise auftreten, falls es sich um einen Wert > 2147483647 handelt. Falls negative Werte von `int64` nach `nat64` umgewandelt werden, resultiert ebenfalls ein Laufzeitfehler.

Table 1: Casting zwischen Datentypen

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

2 Aufbau Compiler

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

2.1 Scanner

Literale werden standardmässig als `int64` interpretiert - ein `nat64`-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 `int64` und `nat64` ab. Die Überprüfung, ob der Wert innerhalb des gültigen Wertebereichs liegt, erfolgt zum Zeitpunkt der Code-Generierung.

2.2 Parser

2.3 Statische Analyse

Scope checking

Für Routinen und Variablen liegen unterschiedliche Namespaces vor. Namen für Routinen und Variablen 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 Variablen dürfen nicht denselben Namen haben wie lokalen Variablen. Überladene Signaturen für Routinen² wurde in dieser Implementation nicht umgesetzt.

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:

²selber Name, unterschiedliche Parameterlisten

Table 3: Erlaubte Typen

Klasse	Types
AddExpr, MultExpr, RelExpr	int64, nat64 *
BoolExpr, IfCmd	bool
AssignCmd	int64, nat64, bool *
FunCallFactor, ProcCallFactor	Typ von Caller muss Typ von Callee entsprechen
MonadicFactor	NOTOPR: bool ADDOPR: int64, nat64
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.

Initialization checking

2.4 Virtuelle Maschine

Grundlage für die Code-Generierung ist der Abstract Syntax Tree (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 IntData³ und NatData⁴ verwendet. Abbildung 1 zeigt die Klassenhierarchie dieser Typen.

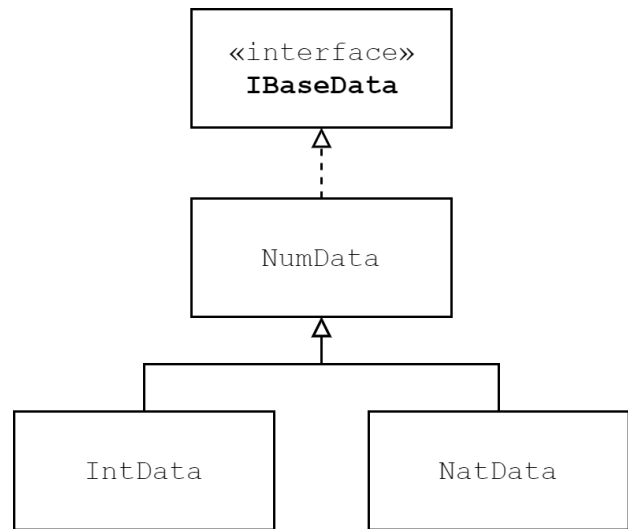


Figure 1: Daten in VM

2.5 Code Generierung

3 Vergleich mit anderen Programmiersprachen (am Beispiel von Java)

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 64-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 2 zeigt den Überlauf bei ganzzahligen, vorzeichenbehafteten Datentypen (am Beispiel von Bitlänge 3 + 1).

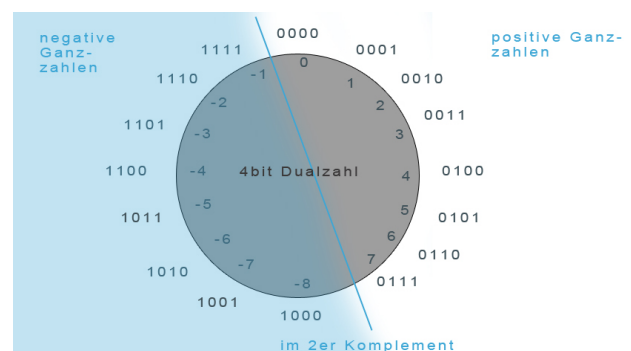


Figure 2: Überlauf mit Integerzahlen

³für den Datentyp int64

⁴für den Datentyp nat64

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, welche für Integer-Typen in Java verwendet werden, werden Fließkommazahlen intern nach IEEE Standard dargestellt. Anders als bei der Zweierkomplement-Darstellung sieht dieses Format spezielle Werte für `POSITIVE_INFINITY` und `NEGATIVE_INFINITY` vor.

4 Designentscheidungen

4.1 Spezifiziertes Verhalten

Der neue Datentyp `nat64` unterstützt die bestehenden Operationen aus IML⁵. Sofern sich die einzelnen Operanden und auch das Resultat im Wertebereich ($\in \mathbb{N}$) befinden, entspricht das Verhalten vom Datentyp `nat64` jenem vom Datentyp `int64`. Andernfalls wird folgendes Verhalten festgelegt:

- **Wertebereich:** Bei einem Überlauf wird jeweils mit dem maximalen Wert weitergerechnet. Dieser entspricht dem maximalen Wert von `int64`⁶.
- **Negative Werte:** Werte werden 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.

4.2 Alternative Ansätze

5 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;

```

⁵Aktuell sind dies

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

⁶9,223,372,036,854,775,807

```

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

A Vollständige Grammatik

B Beispiel doppeltes Casting

IML:

```

program progDouble
global
var value:int64
do
value init := [int64] [nat64] ((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:

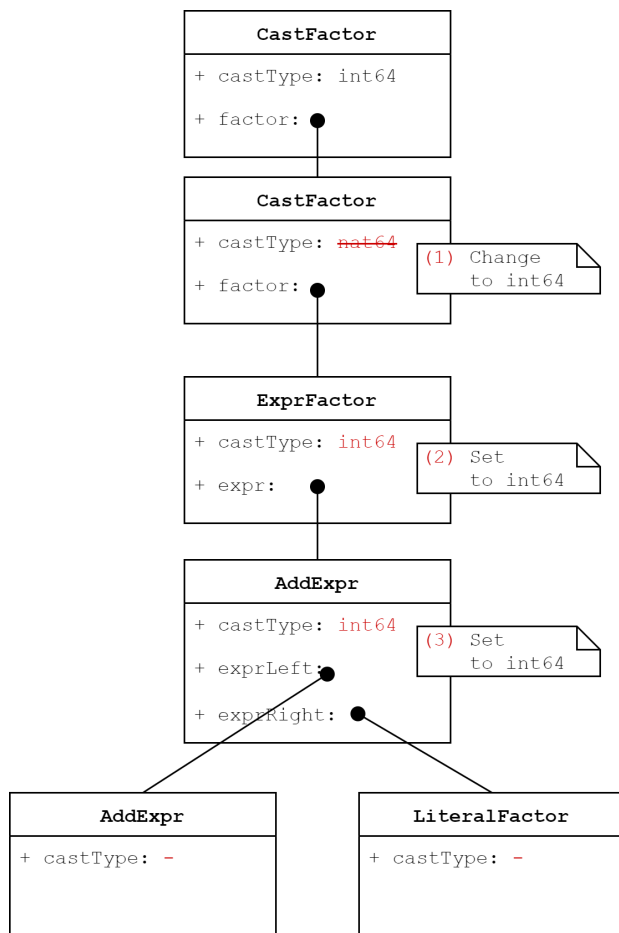


Figure 3: Auszug aus AST