

# Float und Overloading für IML

Schlussbericht, Compilerbau, HS 2016, Team 06, Elias Henz, Michael Roth

## Abstract

Wir haben die Sprache IML um Float, Overloading, Castings und die Unterscheidung zwischen Int32 und Int64 erweitert. Ausserdem sind If-Statements ohne Else möglich.

Der Float-Datentyp verhält sich wie in Java. Die Syntax ist an die IML angepasst. Dasselbe gilt für Int32 (Java-Equivalent int) und Int64 (Java-Equivalent long).

Das Overloading ermöglicht verschiedene Routinendefinitionen mit demselben Namen. Die Implementation ist ebenfalls stark an Java angelehnt. Syntaxmässig bleiben die bestehenden Regeln der IML erhalten.

Beim Casting ermöglichen wir Castings zwischen den Zahlendatentypen, also Int32, Int64 und Float. Die Regeln und Syntax für das Casting haben wir der IML angepasst, das Verhalten des Castings selber ist wie in Java.

## Grundsätzliche Idee der Erweiterung

Wir wollten Floats und Overloading implementieren. Floats, weil es sehr nützlich ist in einer Programmiersprache Gleitkommazahlen verwenden zu können und Overloading weil dies ein von uns sehr häufig genutztes Feature in üblichen Programmiersprachen ist.

Zu Beginn herrschte bei uns Unklarheit, welchen Int-Datentyp wir zu implementieren haben. Im Endeffekt unterstützen wir nun beide. Somit unterstützen wir die Zahlendatentypen Float, Int32 und Int64.

Bei der Implementation ist uns aufgefallen, dass es schön wäre, wenn man zwischen den verschiedenen Datentypen casten könnte. Wir haben uns deshalb entschieden, zusätzlich noch Casting zu implementieren.

Ausserdem ist uns aufgefallen, dass IF-Bedingungen zwingend einen Else-Block benötigen. Wir finden dies suboptimal und haben deshalb die Grammatik leicht angepasst und den Else-Block optional gemacht.

## Zahlendatentypen Float, Int32 und Int64

Wir werden die Zahlentypen Int32, Int64 und Float unterscheiden. Die Equivalenten Datentypen in Java sind für Int32 int, für Int64 long und für Float float. Das Verhalten dieser Zahlentypen ist equivalent zur Java Implementation dieser Datentypen. Die Syntax haben wir für alle der IML angepasst.

## Casting

Wir wollen zwischen unseren Zahlentypen int32, int64 und Float ein Casting anbieten. Sämtliche Castings sind explizit. Das Verhalten der Castings, zum Beispiel der mögliche Genauigkeitsverlust beim Casting von Int32 zu Float regeln wir so, wie das Java macht.

## Overloading

Bei unserem Overloading ist es möglich, mit dem gleichen Routinennamen und Rückgabewert (nur Funktionen, Prozeduren haben keinen Rückgabewert), verschiedene Parameterlisten zu verwenden. Die Unterscheidung erfolgt anhand der Parametertypen und deren Reihenfolge.

### Beispiele:

```
fun f(paramA:int32, paramB:int32) returns var res:int32 ...
fun f(paramA:int32, paramB:float) returns var res:int32 ...
fun f(paramA:int64) returns var res:int32 ...
```

Dies sind alles gültige und korrekte Beispiele. Nicht erlaubt, wäre beispielsweise ein anderer Rückgabetypp, eine Prozedur mit dem Namen `f` oder eine weitere Überladung von `f` mit denselben Parametern wie bereits definiert wurde.

## Lexikalische Syntax

### Float

Um Floats verwenden zu können, müssen wir ein neues Literal einführen. Es sind gewöhnliche Gleitkommazahlen erlaubt, die Wissenschaftliche Darstellung sowie einige Konstanten. Ausserdem benötigen wir ein Token, welches den Datentyp Float darstellt.

#### Float-Literal:

- $[(0-9)^+((0-9)^+)^*(E-\{0,1\}(\backslash.)(0-9)^*(E-\{0,1\})\{0,1\})[0-9][0-9]^*)$   
| NaN | POSITIVE\_INFINITY | NEGATIVE\_INFINITY

#### Token:

- Pattern  $\rightarrow$  float
- Token  $\rightarrow$  (TYPE, FLOAT)

### Beispiele:

```
var f:float;
f init := 12.4;
f := 3E2;
f := 4.65E-9;
f := NaN;
```

### Int64

Int64 ist in der IML bereits vorgesehen, weshalb wir nichts ändern müssen.

### Int32

Für Int32 müssen wir nicht viel ergänzen, da Int32 sehr ähnlich wie Int64 ist. Das Literal ist dasselbe. Der Scanner liest die ganze Zahl und entscheidet anhand des Wertes der Zahl, ob es sich beim Literal um ein Int32 oder Int64 Literal handelt.

Folgend die Definition, wie entschieden wird, ob es sich bei einem Literal um ein Int32 oder Int64 Literal handelt. Die verwendeten Konstanten sind diejenigen der Java Language Specification.

```
Ungültig < Long.MIN_VALUE <= Int64-Literal < Int.MIN_VALUE
<= Int32-Literal <= Int.MAX_VALUE < Int64-Literal <= Long.MAX_VALUE
< Ungültig
```

Ungültig bedeutet hier, dass der Scanner mit einem Fehler abbricht.

### Overloading

Da die Routinen bereits definiert sind, müssen wir hier nichts anpassen oder erweitern.

## Casting

Für das Casting müssen wir zwei neue Symbole einfügen, welche ein Casting markieren.

- LCAST           [
- RCAST           ]

## Beispiele

```
var I : int32;
var l : int64;
var f : float;
//...
i := [int32] l;
l := [int64] f;
f := [float] i;
l := [long][float] i;
```

## grammatikalische Syntax

### Zahlendatentypen

Die Grammatik unterstützt bereits Zahlen und Rechenoperationen, also müssen wir für die Zahlentypen daran nichts anpassen.

### Overloading

Da Funktionen bereits definiert sind, müssen wir hier nichts anpassen oder erweitern.

### Casting

Casting ist in der IML bisher nicht vorgesehen. Deshalb müssen wir für das Casting unsere Grammatik erweitern. Eine Casting-Operation ist eine Expression, weshalb wir das NTS (nicht terminal Symbol)

`factor` um eine Möglichkeit erweitert haben (der neue Eintrag ist mit gelb hinterlegt):

```
(factor,
  [
    [T LITERAL],
    [T IDENT, N optionalIdent],
    [N monadicOperator, N factor],
    [T LPAREN, N expression, T RPAREN],
    [T LCAST, T TYPE, T RCAST, N factor]
  ])

```

### If ohne Else

Damit wir ein IF-Konstrukt ohne Else-Block verwenden können, haben wir die Grammatik ein wenig angepasst. Die Anpassungen sind gelb hinterlegt.

```
(cmd,
  [
    [T SKIP],
    [N expression, T BECOMES, N expression],
    [T IF, N expression, T THEN, N blockCmd, N optionalElse, T
ENDIF],
    [T WHILE, N expression, T DO, N blockCmd, T ENDWHILE],
    [T CALL, T IDENT, N expressionList, N optionalGlobalInits],

```

```

    [T DEBUGIN, N expression],
    [T DEBUGOUT, N expression]
  ),
  (optionalElse,
    [],
    [T ELSE, N blockCmd]
  )

```

## Kontext- und Typeinschränkungen

- Float als Datentyp ist an denselben Orten wie bereits bestehende Datentypen zugelassen.
- Die Overload-Routinen sind überall da zugelassen, wo Routinen bereits zugelassen sind. Es sind keine Routinen in Routinen zugelassen.
- Gecastete Werte sind RValues, sie dürfen also nicht als Zuweisungsziele oder inout/out/ref Parameter verwendet werden.
- Es gibt keine impliziten Castings. Ein Int32-Literal kann nicht einer int64 Variable zugewiesen werden, sondern muss zuerst gecastet werden.
  - o Dies bedeutet, dass wenn ein Int64 Wert um eins erhöht werden soll, muss gecastet werden. → `int64var := int64var + [int64] i;`
- Für Bool-Variablen gibt es keine definierten Castings.
- Falls ein Float jemals zu NaN werden sollte, so kann dies nicht mehr überprüft werden. Denn gemäss der Java Spezifikation ist NaN != NaN. Die in Java dafür bereitstehende Methode `Float.isNaN(float f)` gibt es in unserer Erweiterung nicht.

## Codeerzeugung

### Übertragung der Instruktionen vom C# Compiler zur Java VM

Unser Compiler ist in C# geschrieben, weshalb wir nicht direkt auf das Code-Array der Java VM zugreifen können. Deshalb erstellt unser Compiler ein eigenes Array, welches danach in ein Textfile exportiert wird. Diese Datei geben wir der VM als Startparameter mit und lesen es dort ein. Daraus wird dann das VM-eigene Array befüllt.

Der Inhalt der Datei ist so, dass alle Befehle in der Reihenfolge 0 bis N hintereinander aufgeschrieben werden, getrennt mit einem Semikolon. Falls der Befehl einen Parameter benötigt, wird dieser zwischen den Befehlsnamen und dem zugehörigen Semikolon geschrieben. Zwischen dem Parameter und dem Befehlsnamen wird ein Leerzeichen eingefügt.

Beispiel:

```
LoadImInt 12;Dup;Dup;Deref;LoadImInt 4;AddInt;Store;
```

### Anpassungen an VM

Gewisse Datentypen sind in der VM anders benannt als in der IML.

Datentyp IML	Datentyp VM
int32	int
int64	long

Wir haben eine neue Klasse erstellt, welche die zuvor genannte kompilierte Datei einliest und daraus das Codearray befüllt und danach die Ausführung startet. In dieser Klasse werden auch die Programmparameter des IML Programms extrahiert und zur Verfügung gestellt. Diese Klasse heisst `RunVirtualMachine`.

Soweit möglich, haben wir die bestehenden Befehle verwendet. Trotzdem mussten wir einige neue hinzufügen.

Befehl	Parameter	Beschreibung
CastLongToInt	Keine	Castet den obersten Wert auf dem Stack von Int64 nach Int32. Der StackPointer verändert sich dabei nicht.
CastFloatToInt	Keine	Castet den obersten Wert auf dem Stack von Float nach Int32. Der StackPointer verändert sich dabei nicht.
CastIntToLong	Keine	Castet den obersten Wert auf dem Stack von Int32 nach Int64. Der StackPointer verändert sich dabei nicht.
CastFloatToLong	Keine	Castet den obersten Wert auf dem Stack von Float nach Int64. Der StackPointer verändert sich dabei nicht.
CastIntToFloat	Keine	Castet den obersten Wert auf dem Stack von Int32 nach Float. Der StackPointer verändert sich dabei nicht.
CastLongToFloat	keine	Castet den obersten Wert auf dem Stack von Int64 nach fFloat. Der StackPointer verändert sich dabei nicht.
AddFloat	Keine	Analog zu AddInt für die Datentypen Int64 und Float.
AddLong	Keine	
SubFloat	Keine	Analog zu SubInt für die Datentypen Int64 und Float.
SubLong	Keine	
MultFloat	Keine	Analog zu MultInt für die Datentypen Int64 und Float.
MultLong	Keine	
DivTruncFloat	Keine	Analog zu DivTruncInt für die Datentypen Int64 und Float.
DivTruncLong	Keine	
ModTruncFloat	Keine	Analog zu ModTruncInt für die Datentypen Int64 und float.
ModTruncLong	Keine	
EqFloat	Keine	Analog zu EqInt für die Datentypen Int64 und Float.
EqLong	Keine	
NeFloat	Keine	Analog zu NeInt für die Datentypen Int64 und Float.
NeLong	Keine	
LtFloat	Keine	Analog zu LtInt für die Datentypen Int64 und Float.
LtLong	Keine	
GeFloat	Keine	Analog zu GeInt für die Datentypen Int64 und Float.
GeLong	Keine	
GtFloat	Keine	Analog zu GtInt für die Datentypen Int64 und Float.
GtLong	Keine	
LeFloat	Keine	Analog zu LeInt für die Datentypen Int64 und Float.
LeLong	Keine	
InputFloat	Indicator	Analog zu InputInt für die Datentypen Int64 und Float.
InputLong	Indicator	
OutputLong	Indicator	Analog zu OutputInt für die Datentypen Int64 und Float.
OutputFloat	Indicator	
LoadImFloat	Value	Analog zu LoadImInt für die Datentypen Int64 und Float.
LoadImLong	Value	
NegFloat	Keine	Analog zu NegInt für die Datentypen Int64 und Float.
NegLong	Keine	
ProgParamInt	Keine	Laden Programparameter aus dem Java args[] Array in den Stack. Unterscheidung nach Datentyp.
ProgParamLong	Keine	
ProgParamFloat	Keine	
ProgParamBool	Keine	

### Code für Float, Int64 und Int32

Für die Zahlendatentypen musste nichts Neues erarbeitet werden, aber an einigen Stellen musste eine Fallunterscheidung gemacht werden, um welchen Datentyp es sich handelt und die entsprechenden Befehle aufrufen. Also z.B. `GtLong` bei Vergleichen zwischen zwei Int64 und `GtFloat` bei Vergleichen zweier Float Werte.

### Code für Overloading

Hierfür musste am Code selbst gar nichts angepasst werden. Es musste lediglich die korrekte Routine ausgesucht werden und schlussendlich die richtigen Adressen zugewiesen werden. Beim `Call` haben wir also als Platzhalter nicht nur den Namen der Routine gespeichert, sondern eine Kombination von Name und Parametertypen. Aus `func f(i:int32, f:float) returns ...` wird `f_int32_float`. Somit konnten wir sicherstellen, dass die richtige Überladung aufgerufen wird (ob eine und nur eine passende Überladung existiert, haben wir bereits im Checker geprüft, wodurch sichergestellt ist, dass alle Routinen korrekt ersetzt werden).

### Code für Casting

Für die Castings haben wir die neuen Befehle in der VM, welche das Casting durchführen. Zur Erinnerung: Ein Casting in unserem abstrakten Baum sieht so aus, dass wir einen Zieldatentyp und eine Expression haben. Der Code funktioniert nun so, dass zuerst die Expression gecoded wird und danach der entsprechende Cast-Befehl in die Instruktionsliste geschrieben wird.

Aus

```
intVar := 4;
floatVar := [float] intVar;
```

wird

```
LoadImInt 0;LoadImInt 4;Store;LoadImInt 1;LoadImInt 0;Deref;
CastIntToFloat;Store;
```

## Input und Output

Unser Compiler und VM unterstützen zwei Arten von Ein- und Ausgaben. Zum einen werden die beiden Commands debugin und debugout, sowie Programparameter unterstützt. Die Debug-Commands sind nach Definition der IML implementiert und benötigen somit keiner weiteren Erklärung.

Das Verwenden der Programparameter funktioniert so, dass beim Aufruf des Programms die *in* und *inout* Parameter mitgegeben werden müssen, in der Reihenfolge wie sie im Code stehen. Am Ende des Codes fügt der Compiler debugout-Commands hinzu, welche sämtliche *inout* und *out* Parameter ausgeben.

### Beispiel:

- Prog.cpid → kompiliertes Programm
- ProgramParameter-Definition im Code → `program prog(in i:int32, inout io:int32, out o:int32)`
- Aufruf → `java ch.fhnw.lederer.virtualmachineFS2015.RunVirtualMachine Prog.cpid 1 2`
- Ausgabe am Ende → `2 3` (Darstellung abweichend)

## Vergleich mit anderen Programmiersprachen

Wir werden unsere Erweiterungen im Folgenden mit Java vergleichen. Sämtliche Erweiterungen wurden von uns so implementiert, dass sie syntaktisch an die IML angepasst sind. Diesen Aspekt werden wir deshalb im Weiteren nicht mehr beachten.

### Zahlendatentypen

Die Zahlendatentypen verhalten sich beinahe exakt so wie in Java. Die einzige Abweichung ist, dass wir die Namen der Konstanten für die Infinity-Werte von Float angepasst haben.

Konstante	In Java	In IML
Positive Unendlichkeit	Infinity	POSITIVE_INFINITY
Negative Unendlichkeit	-Infinity	NEGATIVE_INFINITY

Der Grund hierfür ist, dass wir unsere Variante (angelehnt an C#.Net) viel schöner finden und es für den Scanner einfacher zu implementieren war.

Obwohl sich die Datentypen gleich Verhalten wie in Java, stehen viele Funktionalitäten nicht zur Verfügung. Beispielsweise die statischen Methoden der Java-Klasse Float wie isNaN().

### Overloading

Wie in Java, muss der Rückgabotyp immer gleich sein und die Routinen werden ebenfalls aufgrund der Anzahl und Reihenfolge der Parameter unterschieden.

In Java ist die Auswahl der richtigen Routine um einiges komplexer als bei uns. Dies unter anderem, weil in Java implizite Castings existieren. Bei uns gibt es diese nicht und somit muss jeder einzelne Parameter exakt übereinstimmen, um eine Routine auswählen zu können.

### Casting

In unserer Erweiterung gibt es kein implizites Casting. Sämtliche Castings müssen explizit angegeben werden. Ansonsten folgt unser Casting denselben Regeln wie Java. Dies betrifft z.B. was passieren soll, wenn ein Float zu einem Int32 gecastet werden soll, der Wert allerdings zu gross für einen Int32 ist.

## Warum wurde die Erweiterung so entworfen und nicht anders?

Wir haben uns bei unseren Erweiterungen syntaktisch an die IML gehalten und vom Verhalten an Java. Dies, weil die syntaktischen Gegebenheiten der IML nicht verändert werden sollten und weil viele schwierige Fälle beispielsweise beim Casting in Java bereits «gelöst», resp. definiert sind.

Streng genommen müsste unser Float-Datentyp Float32 heissen. Da wir jedoch anders als bei Int32 keine 64Bit Version von Float haben, haben wir auf das 32-Appendix verzichtet.

## Wer hat was gemacht

Folgende Arbeiten wurden erledigt:

- |                                |   |
|--------------------------------|---|
| - Scanner:                     | Gemeinsam: Pair-Programming                   |
| - IML Grammatik:               | Michael Roth                                  |
| - Parser:                      | Gemeinsam: Paralleles Programmieren           |
| - 1. Bericht und Präsentation: | Hauptsächlich Elias Henz                      |
| - Syntax- und Abstrakter Baum: | Gemeinsam: Pair- und paralleles Programmieren |
| - Code-Erzeugung:              | Michael Roth                                  |
| - Anpassungen an VM:           | Elias Henz                                    |
| - 2. Bericht und Präsentation: | Gemeinsam                                     |

## Austausch mit anderen Teams

Während dem Semester tauschten wir uns mündlich mit anderen Mitstudierenden über aufgetretene Probleme und deren Lösungen aus.

Wir hatten Zugriff auf das Code-Repository von Team 11. Im Gegenzug hatten auch sie Zugriff auf unser Repository. Ausserdem haben wir von Team 11 die Grammatik der IML in einer grundlegenden Version erhalten, welche wir für unsere Zwecke überarbeitet und angepasst haben.

## Ehrlichkeitserklärung

Hiermit erklären wir, dass wir die vorliegende schriftliche Arbeit selbständig und nur unter Zuhilfenahme der in den Verzeichnissen oder in den Anmerkungen genannten Quellen angefertigt haben. Wir versichern zudem, diese Arbeit nicht bereits anderweitig als Leistungsnachweis verwendet zu haben. Eine Überprüfung der Arbeit auf Plagiate unter Einsatz entsprechender Software darf vorgenommen werden.

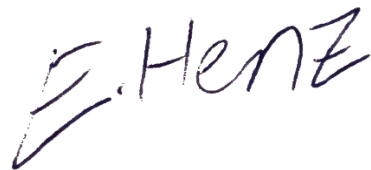
*Kopiert und angepasst von*

<http://www.rwi.uzh.ch/lehreforschung/alphabetisch/schnyder/lehrv/sem/sem1/MBZitierenPlagiate.pdf>

Michael Roth



Elias Henz





## Anhang: IML Testprogramme

### Program 1: Castings

```
program CastingDemo ()
global
    var int:int32;
    var long:int64;
    var f:float;

    fun getPI() returns var res:float
    do
        res init := 3.1415
    endfun
do
    int init := 42;
    long init := [int64]4242;

    // f = 4242 / (42+4242)
    f init := [float]long divE [float](int + [int32]long);

    // cast a function result
    int := [int32]getPI();

    debugout int; // 3
    debugout long; // 4242
    debugout f // 0.99019605
endprogram
```

### Program 2: Overloading

```
program OverloadDemo ()
global
    fun sum(a:int32, b:int32) returns res:int32
    do
        res init := a + b
    endfun;

    fun sum(a:int32, b:int32, c:int32, d:int32, e:int32) returns
var res:int32
    do
        res init := sum(a,b,c,d) + e
    endfun;

    fun sum(a:int32, b:int32, c:int32, d:int32) returns var
res:int32
    do
        res init := sum(a,b,c) + d
    endfun;

    fun sum(a:int32, b:int32, c:int32, d:int32, e:int32, f:int32)
returns var res:int32
```

```

do
    res init := sum(a,b,c,d,e) + f
endfun;

fun sum(a:int32, b:int32, c:int32) returns var res:int32
do
    res init := sum(a,b) + c
endfun;

fun sum(a:int32, b:int32, c:int32, d:int32, e:int32, f:int32,
g:int32) returns var res:int32
do
    res init := sum(a,b,c,d,e,f) + g
endfun
do
    debugout sum(3,6);           // 9
    debugout sum(3,6,9);         // 18
    debugout sum(3,6,9,12);      // 30
    debugout sum(3,6,9,12,15);   // 45
    debugout sum(3,6,9,12,15,18); // 63
    debugout sum(3,6,9,12,15,18,21) // 84
endprogram

```

Program 3: Volumenberechnung (Overloading, Casting, If ohne Else, int32/64/float, ProgramParameter)

```

// a: 1. Seitenlänge des Quaders
// b: 2. Seitenlänge des Quaders
// h: Höhe des Quaders
// volumeQube: Volumen des Quaders
// volumeInsideZylinder: Volumen des eingeschlossenen Zylinders

program VolumeCalculations (in var a:int32, in var b:int32, in var
h:int32, out var volumeQube:int64, out var
volumeInsideZylinder:float)

global
    const PI:float;
    var radius:float;

    // Berechnung des Volumens vom Quader
    proc CalcVolume(in copy var a:int32, in copy var b:int32, in
copy var h:int32, out volume:int64)
    local
        var area:int64
    do
        area init := [int64]a * [int64]b;
        volume := area * [int64]h
    endproc;

```

```
// Berechnung des Volumens vom Zylinder
proc CalcVolume(in copy var r:float, in copy var h:int32, out
volume:float)
  global in const PI
  local
    var area:float
  do
    area init := PI*r*r;
    volume := area * [float]h
  endproc
do
  PI init := 3.14159265358979323846;

  // radius definieren
  radius init := [float]a divE [float]2;

  // radius anpassen falls b < a
  if (a > b) then
    radius := [float]b divE [float]2
  endif;

  // Berechnen der Volumen
  call CalcVolume(a, b, h, volumeQube init);
  call CalcVolume(radius, h, volumeInsideZylinder)
endprogram
```