# Verschiedene ganzzahlige Datentypen und Routinenüberladung für IML

## Schlussbericht

Aude Kauffmann & Janik Meyer
Compilerbau, HS 2018, Team 06

Dieser Bericht beschreibt unsere Erweiterungen von IML und dem dazugehörigen Compiler. Im Folgenden, wird die grundsätzliche Idee dieser Erweiterung sowie die benötigte lexikalische und grammatikalische Syntax vorgestellt. Dann werden Einschränkungen beschrieben und es wird ein Vergleich mit anderen Programmiersprachen gemacht. Zum Schluss wird im Anhang Beispielcode in IML beschrieben.

# Übersicht

Mit unserem Projekt soll die Möglichkeit geschaffen werden, dass Routinen in IML überladen werden können. Mit Routinen sind Prozeduren wie auch Funktionen gemeint. Um das Überladen sinnvoll nutzen zu können, soll ein weiterer ganzzahliger Datentyp implementiert werden. Unsere Idee ist es, die Typen int32 und int64 umzusetzen. Die Zahl hinter int steht dabei für die Anzahl Bits, die gespeichert werden können.

Ganzzahlige Variablen und Konstanten sind entweder vom Typ int32 oder int64. Analog sind ganzzahlige Literale vom Typ int32 oder int64. Die Unterscheidung soll zur Kompilierzeit gemacht werden. Soweit möglich, sollen alle Operationen für den Typ int32 auch für int64 verfügbar sein. Zusätzlich soll man implizit von int32 zu int64 umwandeln können, da diese Umwandlung verlustfrei ist. Die Umwandlung von int64 zu int32 soll durch Funktionen realisiert werden. Diese Funktionen sind in den Beispielen ersichtlich. Als Argument nehmen die Funktion einen int64 mit den Modi “in ref const” und geben einen int32 zurück.

Der Programmierer soll mehrere Routinen mit gleichem Namen erstellen können, bei denen sich lediglich die Parameter unterscheiden. Am Ort des Aufrufs soll der Compiler dann die Routine identifizieren, die aufgerufen werden soll. Diese Identifikation geschieht über die Anzahl sowie die Typen der Parameter. Im Fall, dass keine Überladung exakt passt, können bei “in” Parametern mit direktem Zugriffsmodus implizite Typumwandlungen verwendet werden. Sollte keine Überladung oder mehrere gleichwertige Überladungen in Frage kommen, muss der Kompiliervorgang mit einem Fehler abbrechen. Die Beispiele 1 bis 4 im Anhang illustriert die eben beschrieben Fälle.

# Lexikalische Syntax

Es wird unterschieden zwischen int32 und int64. Dementsprechend gibt es für ganzzahlige Zahlen die Typen int32 und int64, welche als Keywords der Sprache hinzugefügt werden. Ein Typ int ist nicht vorgesehen.

Damit einhergehend werden ganzzahlige Literale zwischen 32 und 64 Bit unterschieden. Die Unterscheidung wird dabei zur Kompilierzeit anhand des Wertes gemacht. Falls ein Literal einen Wert hat, den man mit 32 Bit darstellen kann ist es ein int32-Literal. Fall der Wert eines Literals nicht mit 32 Bit gespeichert werden kann, es ist ein int64-Literal. Sollte es ein Literal geben, dessen Wert mehr als 64 Bit zum Speichern benötigt, wird beim Kompiliervorgang ein Fehler zurückgegeben, da kein entsprechender Datentyp vorhanden ist.

# Grammatikalische Syntax

Implizite Typumwandlungen von int32 zu int64 benötigen keine Änderungen an der Grammatik. Wie oben schon beschrieben, werden Typumwandlungen von int64 zu int32 mit Funktionen gelöst, deshalb braucht es auch hier keine Anpassungen.

Bei der Routinenüberladung kann es mehrere gleichnamige Routinen haben, die sich lediglich in der Signatur unterscheiden. Für das Überladen wird kein spezielles Schlüsselwort gebraucht. Alles in allem ist also die vorgegebene Grammatik ausreichend.

# Kontext- und Typeinschränkungen

Es darf nicht mehrere Funktionen oder Prozeduren mit derselben Signatur geben. Sie dürfen denselben Namen haben, aber müssen sich durch die Anzahl Parametern oder ihre Typen unterscheiden. Bei den Aufrufen, wird dann die passende Routine wie folgt gesucht. Zuerst wird der Name der Routine geprüft. Dann wird geprüft, ob die Anzahl Parameter mit der Anzahl Argumente übereinstimmt. Als nächstes werden die Parameter genauer geprüft. Sollte ein L-Value verlangt sein, müssen die Typen exakt übereinstimmen und das Argument muss auch ein L-Value sein. Sollte nur ein R-Value verlangt sein, muss geprüft werden, ob der Typ des Arguments kompatibel ist, da noch implizite Casts von int32 zu int64 angewendet werden können. Sollten nach diesem Schritt noch mehrere Routinen in Frage kommen, wird geprüft, ob es eine Überladung gibt, bei der die Parameter exakt übereinstimmen.

Um die vordefinierten Cast-Funktionen verwenden zu können, müssen diese am Anfang der statischen Analyse zum Kontext hinzugefügt werden. Das führt dazu, dass die Funktionen dann wie normale Funktionen behandelt werden.

Bei Zuweisungen lockern sich die Typeinschränkungen ein wenig. Es gibt einen Fall, wo die Typen links und rechts nicht exakt übereinstimmen müssen. Das ist der Fall, wenn ein int32 einem int64 zugewiesen wird. Denn der int32 kann da implizit zu einem int64 umgewandelt werden.

Bei dyadischen Ausdrücken gibt es neu einen Spezialfall, bei dem eine Seite vom Typ int32 und die andere Seite int64 ist. Es spielt dabei keine Rolle, welcher der beiden Seiten von welchem Typ ist. Der int32 wird wiederum implizit zu int64 gecastet.

# Codeerzeugung

Bei der Codeerzeugung muss bei den Routinenaufrufen analog der statischen Analyse die richtige Funktion gesucht werden. Das heisst, die Call-Instruktion muss so den richtigen Sprungpunkt erhalten. Zusätzlich müssen wo nötig implizite Typumwandlungen von int32 zu int64 eingefügt werden. Diese Umwandlung geschieht über eine Instruktion, die den obersten Wert auf dem Stack durch den selben Wert vom Type int64 austauscht. Wie im nächsten Abschnitt beschrieben, werden die speziellen Cast-Funktionen wie normale Funktionen aufgerufen und brauchen keine Sonderbehandlung.

Es soll verschiedene Funktionen für die Typumwandlung von int64 zu int32 geben. Diese werden als Instruktionen umgesetzt und erhalten nur einen minimalen Code für die eigentliche Funktionsdeklaration. Die erste Variante macht ein “Clamping”. Die zweite Variante schneidet überzählige Bits wie in Java ab. Die dritte Variante konvertiert den Integer nur, wenn keinen Verlust entsteht. Das heisst, wenn der Wert von dem int64 ausserhalb des Wertebereichs von int32 ist, wird zur Laufzeit eine Exception geworfen. Diese drei Cast-Funktionen haben jeweils eine Ableitung der normalen Routinendeklaration. Dabei muss lediglich ein eigener Identifier gesetzt und die Codeerzeugung angepasst werden. Der erzeugte Code ist der einer normalen Funktionsdeklaration, wobei die Befehle nur bewirken, dass das Argument mit der entsprechenden Instruktion zu int32 umgewandelt wird und dieser Wert dann in den Rückgabewert gespeichert wird. Dieser Mechanismus hat den Vorteil, dass die Funktionsaufrufe überall den gleichen Code erzeugen können.

Bei Zuweisungen und dyadischen Ausdrücken können wie bei den Routinenaufrufen implizite Typumwandungen von int32 zu int64 zum Einsatz kommen.

Es wurden neue Instruktionen für die Ein- und Ausgabe von int64 Werten implementiert. Dabei wurde der Datentyp long von Java verwendet, welcher 64 Bit gross ist.

# Vergleich mit anderen Programmiersprachen

In diesem Abschnitt, wird erklärt wie die Routinenüberladung und Typumwandlung in anderen Programmiersprachen funktioniert. Die Grundidee ist bei mehreren Programmiersprachen ähnlich, nur variieren die Details. Wir haben uns mit der Funktionsweise von C++ und Pascal beschäftigt und konnten auf diese Weise auch selber Anregungen für unsere Umsetzung finden.

## C++

Bei überladenen Funktionen in C++ wird der Funktionsrückgabetyp einer Funktion nicht berücksichtigt, sondern nur die Parameter. Zuerst, versucht der Compiler einen exakten Match zu finden. Das heisst, alle Argumente haben den genau gleichen Typ, wie in der Signatur einer überladenen Funktion.

Wenn kein exakter Match gefunden wird, versucht der Compiler durch Typumwandlung automatisch und implizit die Argumente anzugleichen (zum Beispiel float zu double oder unsigned short to int). Das kann aber zu mehreren Matches führen, weil alle diese Konversion als gleichwertig angesehen werden. Das heisst, wenn durch Typumwandlung mehrere Methoden passen können, entsteht eine Mehrdeutigkeit.

Wenn es mehrere Argumente gibt, versucht der Compiler für jedes Argument einen Match zu finden. Die ausgewählte Funktion ist dann diejenige, die einen besseren Match als alle anderen Möglichkeiten für mindestens ein Argument mehr ist. Wenn keine solche Funktion gefunden werden kann oder eine Mehrdeutigkeit entsteht, dann erzeugt der Compiler eine Fehlermeldung.

Bezüglich Typumwandlung gibt es in C++ mehrere Möglichkeiten. Wie bei unserer Erweiterung auch, können gewisse Typumwandlungen implizit vorgenommen werden. Bei den expliziten Typumwandlungen können einerseits die sogenannten C-Style-Casts verwendet werden. Zum anderen gibt es in C++ spezifische Typumwandlungen, dazu gehören static_cast, dynamic_cast, reinterpret_cast und const_cast. Im Vergleich mit unserer Erweiterung bietet C++ deutlich mehr Möglichkeiten, da wir uns auf Typumwandlungen zwischen ganzzahligen Datentypen beschränken.

## Pascal

In Pascal können Funktionen wie auch Prozeduren überladen werden. Dies wird wie bei unserer Erweiterung ohne irgendwelche Schlüsselwörter gemacht, sofern die überladenen Funktionen in derselben Einheit sind. Das Finden der richtigen Überladung scheint in Pascal ähnlich wie in C++ zu funktionieren. Zu dieser Erkenntnis sind wir mithilfe von Tests mit einem Online Pascal Compiler gekommen. Leider haben wir zum Matching keine genaue Dokumentation gefunden.

Beim Umwandeln von Typen orientieren wir uns stark an Pascal. In Pascal können Werte von einem Typen mit einem kleineren Wertebereich auch implizit in einen Wert von einem Datentyp mit grösserem Wertebereich umgewandelt werden. Als Beispiel sei hier die Umwandlung von Integer in Real genannt. Wichtig ist hier jedoch zu bemerken, dass Real bei Ganzzahlen nicht dieselbe Präzision wie Integer hat und somit dennoch Informationsverlust entstehen kann. Für verlustbehaftete Typumwandlungen werden wie bei unserer Erweiterung Funktionen verwendet. Ein Beispiel wäre die Umwandlung von Real zu Integer, wo entweder Round oder Trunc verwendet werden kann.

# Grund der Implementierung

Bei der Routinenüberladung haben wir uns entschlossen, dem Programmierer möglichst viele Möglichkeiten zu geben und nur wo wirklich nötig Kompilationsfehler zu produzieren. Denn der Sinn und Zweck von Überladungen sollte es sein, den Programmierer die Arbeit zu erleichtern und nicht, ihn mit Einschränkungen zu plagen.

Wir haben uns einerseits für Funktionen entschieden, um von int64 zu int32 zu konvertieren, weil diese leicht erweiterbar sind. Andererseits wird der Programmierer so gezwungen, sich Gedanken über den Umgang mit dem möglichen Datenverlust zu machen. Das heisst er muss explizit wählen, welche Umwandlung am jeweiligen Ort die richtige ist. Wir haben uns entschieden, drei Cast-Funktionen umzusetzen, damit man die Möglichkeiten und Vorteile sieht.

# Featureliste des Compilers

Die Schritte vom Scanning bis zur Umwandlung in den abstrakten Syntaxbaums sind selbstverständlich in vollem Umfang umgesetzt.

Bezüglich der statischen Analyse, hat der Compiler folgende Eigenschaften:

- Scope checking
    1. Prüft, ob ein Identifier pro Namespace/Scope nur maximal einmal deklariert ist.
    2. Prüft, ob globale Stores bei Prozeduren importiert sind.
    3. Prüft, ob jeder angewendete Identifier deklariert ist.
    4. Prüft, ob bei Routinen die Argumente die L-Values sein müssen, es wirklich sind.
- Type checking
    1. Vollständige Überprüfung aller Typen.
- Const checking
    1. Prüft, ob Werte nicht an Konstanten zugewiesen werden.
- Flow checking (teilweise)
    1. Prüft, ob der Flowmode bei Funktionen “In” ist.
    2. Prüft, ob der Changemode bei Routinen Const ist, wenn der Parameter “In Ref” ist.
    3. Prüft, ob der Changemode bei Routinen Var ist, wenn es der Flowmode Inout ist.
- Aliasing analysis (teilweise)
    1. Prüft, ob bei Prozeduraufrufen nicht dieselbe Variable mehrfach für Out/Inout Parameter verwendet wird.
- Initialization (nicht realisiert)

Die Codeerzeugung wurde vollständig umgesetzt. Das heisst inklusive Linksassoziativität bei Ausdrücken, bedingte Evaluation bei &? sowie |? und den beiden Mechmodes Ref/Copy.

Die Beispielprogramme im Ordner “IML_ExamplePrograms” können von unserem Compiler alle fehlerfrei kompiliert werden. Dies wurde durch das Ausführen des kompilierten Codes getestet. Zusätzlich können natürlich unsere Programme, die im Bericht beschrieben sind, kompiliert und ausgeführt werden, sofern keine Exceptions vorgesehen sind.

# Anhang: IML Testprogramme

## Beispiel 1: Overloading Übersicht

In diesem Beispiel, wird, je nach verwendeten Typen, die entsprechende Prozedur automatisch aufgerufen.

```
program testOverloading(in a:int32, in b:int64)
global
 proc printSum(in copy const m:int32, in copy const n:int32)
 local
   var s:int32
 do
   s init := m + n;
   debugout s
 endproc ;

 proc printSum(in copy const m:int32, in copy const n:int64)
 local
   var s:int64
 do
   s init := m + n;
   debugout s
 endproc ;

 proc printSum(in copy const m:int64, in copy const n:int64)
 local
   var s:int64
 do
   s init := m + n;
   debugout s
 endproc ;

proc printSum(in copy const m:int64, in copy const n:int32)
 local
   var s:int64
 do
   s init := m + n;
   debugout s
 endproc
```

```
do
 call printSum(a, a); // calls  printSum(int32,int32)
 call printSum(b, b); // calls  printSum(int64,int64)
 call printSum(a, b); // calls  printSum(int32,int64)
 call printSum(b, a); // calls  printSum(int64,int32)
 call printSum(b, a, a) // compile time error, no matching overload found
endprogram
```

## Beispiel 2: Overloading ambiguous & exact

Dieses Beispiel zeigt, dass die exakte Überladung genommen wird, wenn mehrere Überladungen in Frage kommen. Zudem wird der Fall gezeigt, dass nicht eindeutig bestimmt werden kann, welche Prozedur aufgerufen werden soll.

```
program testOverloading(in a:int32, in b:int64)
global

 proc printSum(in copy const m:int32, in copy const n:int64)
 local
   var s:int64
 do
   s init := m + n;
   debugout s
 endproc ;

 proc printSum(in copy const m:int64, in copy const n:int64)
 local
   var s:int64
 do
   s init := m + n;
   debugout s
 endproc
do
 call printSum(a, b); // calls  printSum(int32,int64)
 call printSum(a, a) // compile time error, multiple possible matches
endprogram
```

## Beispiel 3: Overloading out

Dieses Beispiel demonstriert, dass out Parameter immer exakt übereinstimmen müssen und keine implizite Typumwandung gemacht wird.

```
program testOverloading(inout a:int32)
global
```

```
 proc setNull(out copy m:int64)
 do
   m init := 0
 endproc
do
 call setNull(a) // compile time error, no match found
endprogram
```

## Beispiel 4: Overloading function

Dieses Beispiel zeigt, dass auch Funktionen überladen werden können.

```
program testOverloading()
global
 fun returnNull(in copy const n:int32) returns const r:int32
 do
   r init := 0
 endfun ;

 fun returnNull(in copy const n:int64) returns const r:int64
 do
   r init := 0
 endfun;
 const a:int32;
 const b:int64
 do
   a init := 5;
   b init := 7;
   debugout returnNull(a) ; // calls setNull(int32)
   debugout returnNull(b) // calls setNull(int64)
endprogram
```

## Beispiel 5: Umwandlung von int64 zu int32 clamp

In diesem Beispiel, wird die Methode, die dem int32 den maximal möglichen Wert für einen zu grossen int64 gibt, verwendet.

```
program testToInt32Clamp()
global
  var normalInt:int64;
  var biggestInt:int64;
  var smallestInt:int64;
  var bigInt:int64;
  var smallInt:int64
do
  normalInt init := 5;
```

```
  biggestInt init := 2147483647; // int32.maxValue
  smallestInt init := -2147483648; // int32.minValue
  bigInt init := 10000000000; // value bigger than int32.maxValue
  smallInt init := -10000000000; // value smaller than int32.minValue
  debugout toInt32Clamp(normalInt); // prints 5
  debugout toInt32Clamp(biggestInt); // prints int32.maxValue which is
2,147,483,647
  debugout toInt32Clamp(smallestInt); // prints int32.minValue which is
-2,147,483,648
  debugout toInt32Clamp(bigInt); // prints int32.maxValue which is
2,147,483,647
  debugout toInt32Clamp(smallInt) // prints int32.minValue which is
-2,147,483,648
endprogram
```

## Beispiel 6: Umwandlung von int64 zu to int32 cut

In diesem Beispiel, wird der Cast, der überschüssige Bits abschneidet, verwendet.

```
program testToInt32Cut()
global
  var normalInt:int64;
  var biggestInt:int64;
  var smallestInt:int64;
  var bigInt:int64;
  var smallInt:int64
do
  normalInt init := 5;
  biggestInt init := 2147483647; // int32.maxValue
  smallestInt init := -2147483648; // int32.minValue
  bigInt init := 10000000000; // value bigger than int32.maxValue
  smallInt init := -10000000000; // value smaller than int32.minValue
  debugout toInt32Cut(normalInt); // prints 5
  debugout toInt32Cut(biggestInt); // prints int32.maxValue which is
2,147,483,647
  debugout toInt32Cut(smallestInt); // prints int32.minValue which is
-2,147,483,648
  debugout toInt32Cut(bigInt); // prints 1410065408
  debugout toInt32Cut(smallInt) // prints -1410065408
endprogram
```

## Beispiel 7: Umwandlung von int64 zu int32 lossless

In diesem Beispiel wird gezeigt, wie man vorgehen kann, wenn man beim Casten keinen Verlust erwartet. In diesem Fall treten jedoch teilweise Verluste auf, welche dann zur Programmlaufzeit zu einem spezifischen ExecutionError führen.

```
program testToInt32Lossless()
global
  var normalInt:int64;
  var biggestInt:int64;
  var smallestInt:int64;
  var bigInt:int64;
  var smallInt:int64
do
  normalInt init := 5;
  biggestInt init := 2147483647; // int32.maxValue
  smallestInt init := -2147483648; // int32.minValue
  bigInt init := 10000000000; // value bigger than int32.maxValue
  smallInt init := -10000000000; // value smaller than int32.minValue
  debugout toInt32Lossless(normalInt); // prints 5
  debugout toInt32Lossless(biggestInt); // prints int32.maxValue which is
2,147,483,647
  debugout toInt32Lossless(smallestInt); // prints int32.minValue which is
-2,147,483,648
  debugout toInt32Lossless(bigInt); // throws ExecutionError
  debugout toInt32Lossless(smallInt)  // throws ExecutionError
endprogram
```

## Beispiel 8: Fakultätsfunktion - Schleife und Rekursiv

In diesem Beispiel wird die Fakultätsfunktion auf zwei Weisen ausgeführt: mit einer Schleife und rekursiv.

```
program faculty(in n:int32)
global
  fun facultyLoop(in copy const n:int32) returns var r:int32
  local
    var i:int32
  do
    i init := n-1;
    r init := n;
    while i > 1 do
      r := r * i ;
      i := i - 1
    endwhile
```

```
 endfun;

 fun facultyRecursiv(in copy const n:int32) returns const r:int32
 do
   if n>1 then
    r init := n * facultyRecursiv(n-1)
   else
    r init := 1
   endif
 endfun

 do
  debugout facultyLoop(n);
  debugout facultyRecursiv(n)
endprogram
```

## Beispiel 9: Operationen auf Int64

In diesem Beispiel, werden alle möglichen Operationen des Typs Int64 geprüft. In diesem Beispiel könnte man auch einen oder beide Typen auf int32 ändern. Die Resultate würden, danke impliziten Typumwandlungen, gleich bleiben.

```
program operationOnInt64()
global
  var a:int64;
  var b:int64
do
  a init := 10;
  b init := 20;
  debugout a + b;
  debugout a - b;
  debugout a * b;
  debugout a divE b;
  debugout a modE b;
  debugout a = b;
  debugout a < b;
  debugout a > b;
  debugout a /= b;
  debugout a >= b;
  debugout a <= b
endprogram
```

# Verweis zur Arbeit

Der gesamte Quellcode inklusive Versionsgeschichte unseres Compilers ist unter der nachfolgenden Adresse öffentlich einsehbar.
https://github.com/StrelokCH/cpib-km
Im Ordner “doc” ist ebenfalls die aktuelle Version des Berichts zu finden.

# Quellen:

https://www.learncpp.com/cpp-tutorial/76-function-overloading/
https://www.freepascal.org/docs-html/ref/refse93.html#x183-20500014.6
https://rextester.com/l/pascal_online_compiler
http://wiki.freepascal.org/Typecast

# Ehrlichkeitserklärung

Hiermit bestätigen wir, Aude Kauffmann und Janik Meyer, die vorliegende Arbeit selbständig, ohne Hilfe Dritter und nur unter Benutzung der angegebenen Quellen verfasst zu haben.

---

Aude Kauffmann Janik Meyer