Verschiedene ganzzahlige Datentypen und Routinenüberladung für IML

Schlussbericht

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

Dieser Bericht beschreibt unsere geplanten 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. Der Hauptfokus liegt bei unserer Arbeit auf der Routinenüberladung. Der hinzugefügte Datentyp inklusive der Typumwandlung ist dabei eher Nebensache und dient nur als Mittel zum Zweck.

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. Beispiel 1 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 dürfen nicht mehrere Funktionen oder Prozeduren mit derselben Signatur geben. Sie dürfen denselben Namen haben aber müssen sich durch den Anzahl Parametern oder ihren Typen unterscheiden. Bei den einzelnen Aufrufen wir

Codeerzeugung

Es soll verschiedene Funktionen für die Typumwandlung von int64 zu int32 geben. Die werden als Instruktionen umgesetzt und erhalten nur einen minimalen Code. Die erste Variante beschränkt macht ein “Clamping”. Die zweite Variante schneidet überzählige Bits ab. Die dritte Variante konvertiert den Integer nur, wenn keinen Verlust entsteht. Das heisst, wenn der Wert von dem int64 kleiner als der maximale Wert von int32 ist und wirft sonst einen Exception.

Der Vorteil davon ist, dass der Benutzer eine von den verschiedenen Möglichkeiten auswählen muss.

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. Für möglicherweise 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

Wir haben uns für Funktionen entschieden um von int64 zu in32 konvertieren, weil diese leicht erweiterbar sind. Man hätte eine einzige Funktion schreiben können aber mit drei Funktionen, verfügt der Benutzer über mehr Möglichkeiten.

Somit, ist es auch möglich das Casten durch Hinzufügen von weiteren Funktionen zu erweitern.

Der Benutzer muss selber entscheiden, welche Möglichkeit er wählen möchte. Und es wird ihm auch klar mit welchen Verlusten er umgehen muss.

Featureliste des Compilers

Bezüglich der Analyse, hat der Compiler folgende Eigenschaften:

- Scope checking
    1. Ein Identifier (in einem einzelnen namespace und scope) darf nicht mehr als einmal deklariert werden.
    2. Prüft ob globalImports eigentlich existiert
    3. Prüft ob die aufgerufene Prozedur existiert
    4. Prüft ob jeder angewendete Identifier deklariert ist
    5. Prüft ob die Parameters die L-Values sein müssen, es eigentlich sind
- Type checking
- Function matching
- Const checking
- Flow checking (teilweise)
- Aliasing analysis (teilweise)

Nicht realisiert:

- Initialization

Es ist möglich Kommentare im Code zu schreiben.

Anhang: IML Testprogramme

Beispiel 1: Overloading Übersicht

In diesem Beispiel, werden, 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

```
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(int64,int64)
 call printSum(a, a) // compile time error, multiple possible matches
endprogram
```

Beispiel 3: Overloading out

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

 proc setNull(out copy m:int64)
 do
   m init := m + 1
 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(in a:int32, in b:int64)
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:int32
 do
   r init := 0
 endfun

 do
   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 Int32 gibt, verwendet. Es werden sehr grosse und kleine Werte sowie auch Randfälle geprüft.

```
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 die ersten 32 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. Hier 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 geschrieben: 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 von dem Typ Int64 geprüft.

```
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
```

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