Routinenüberladung für IML

Zwischenbericht

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 eine Funktion toInt32 realisiert werden. Als Argument nimmt die Funktion einen int64 mit den Modi “in ref const” und gibt einen int32 zurück.

Der Programmierer soll mehrere Routinen mit gleichem Namen erstellen könne, 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 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 einer Funktion 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

Siehe Schlussbericht

Es darf zweimal dieselbe Funktion mit derselben Signatur geben, solange sie nicht aufgerufen wird.

Codeerzeugung

Siehe Schlussbericht

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 dem 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. Im Gegensatz zu unserer Erweiterung scheint aber das Finden der richtigen Überladung bei Pascal ein wenig einfacher von statten zu gehen. Um die richtige Funktion zu finden, werden die Typen der Parameter verglichen. Falls keine Funktion gefunden wird, bei welcher die Parameter vollständig übereinstimmen, wird ein Compiler-Fehler generiert.

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

Siehe Schlussbericht

Wir haben uns für Funktionen entschieden um von int64 zu in32 konvertieren, weil diese leicht erweiterbar sind. Das heisst man könnte eine Funktion machen, die bei der Umwandlung die zusätzlichen Bits einfach ignoriert. Eine andere Option wäre, eine Funktion zu schreiben, die den maximalen Wert von int32 nimmt, wenn der numerische Wert von einem int64 zu gross für einen int32 ist.

Anhang: IML Testprogramme

Weitere siehe Schlussbericht

Beispiel 1

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

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

 proc printSum(in copy const m:int64, in copy const n:int64)
 do
   var s:int32;
   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,int64)
```

```
 call printSum(b, a, a) // compile time error, no matching overload found
endprogram
```

Quellen:

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