Übung zur Vorlesung Compiler 1: Grundlagen

Prof. Dr. Andreas Koch
Jens Huthmann, Julian Oppermann

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 14/15
Aufgabenblatt 2 - Lösungsvorschlag

Abgabemodalitäten

Gruppenarbeit ist erlaubt und erwünscht. Bitte geben Sie dann nur eine Lösung pro Gruppe ab. Ihre Lösungen reichen Sie als Zip-Archiv per E-Mail bis zum 14.12.2013 um 23:59 MET an oc@esa.informatik.tu-darmstadt.de ein. Die Zip-Datei sollte ein PDF mit Ihrer Lösung der Aufgabe 2.1 und Ihre Implementierung der Klassen *SymbolTable* und *ConstantFolding* jeweils als Java-Quelldatei enthalten. Die E-Mail muss als Betreff "Compiler 1 Aufgabe 2" haben.

Aufgabe 2.1 Allgemeine Fragen

a) Warum wird für Sprachen wie Java, C++ oder Triangle eine verschachtelte Blockstruktur in der Symboltabelle benötigt? Wieso reicht nicht eine der anderen in Vorlesung vorgestellten Varianten aus?

Aufgabe 2.1 Lösung

Die genannten Sprachen unterstützen neben globalen und funktions-lokalen Deklarationen noch weitere, kleinere geschachtelte Geltungsbereiche innerhalb von Funktionen (durch `let .. in` in Triangle und `{, }` in Java, C++). Die Spezifikation der Sprachen schreibt vor, dass einem Bezeichner jeweils die speziellste, d.h. im kleinstmöglichen umgebenden Geltungsbereich gefundene Deklaration zuzuordnen ist. Für die Auflösung eines Bezeichners müssen also die Geltungsbereiche von innen nach außen durchsucht werden. Daher muss die Organisation der Symboltabelle der Schachtelung der Geltungsbereiche entsprechen.

Aufgabe 2.2 Effiziente Implementierung der Symboltabelle

In Vorlesungsblock 3 wurde die im Rahmen der Kontextanalyse stattfindende Erstellung der Symboltabelle thematisiert. Die Datenstruktur der im Triangle-Compiler per se implementierten Variante basiert auf verketteten Listen. Wie aufgezeigt sind weitaus effizientere Implementierungen denkbar (vgl. Skript: Block 3, S. 20-22). Im Rahmen dieser Aufgabe sollen Sie diese effizientere Fassung implementieren.

Laden Sie sich hierzu zunächst den auf der Website der Vorlesung verfügbaren Quellcode des Triangle-Compilers herunter. Sie werden feststellen, dass die für diese Aufgabe notwendigen Dateien im Package *Triangle.ContextualAnalyzer* zu finden sind. Die angesprochene Variante einer Symboltabelle auf Basis verketteter Listen liegt in der Klasse *IdentificationTable* vor.

Implementieren Sie nun auf Basis des bereitgestellten Grundgerüsts die Klasse *SymbolTable*, die als Ersatz der Klasse *IdentificationTable* dienen soll. Verwenden Sie bei Ihrer Implementierung das in der Vorlesung vorgestellte Konzept. Achten Sie insbesondere darauf, die bereitgestellte API nicht zu verändern, um Ihre Klasse zum Triangle-Compiler kompatibel zu halten. Hierzu zählen alle öffentlichen Methoden der bereitgestellten Klasse.

Um Ihre Lösung zu testen, können Sie Ihre Implementierung in den Triangle-Compiler einfügen. Bearbeiten Sie hierzu die Klasse *Checker*. Lokalisieren Sie das Objekt-Attribut *idTable* und ändern Sie dessen Typ von *IdentifcationTable* auf *SymbolTable*. Hiernach wird der Triangle-Compiler auf Ihre Implementierung der Symboltabelle zurückgreifen.

Aufgabe 2.2 Lösung

Den Lösungsvorschlag zu dieser Aufgabe finden Sie in Form der Datei *SymbolTable.java* in der Anlage.

Aufgabe 2.3 Vereinfachung von konstanten Ausdrücken und algebraische Vereinfachungen

Einige einfache Optimierungen können bereits auf dem abstrakten Syntaxbaum realisiert werden. In dieser Aufgabe sollen Sie die Vereinfachung von konstanten Ausdrücken durch ein Visitor Pattern kennenlernen und implementieren.

Hierbei werden Ausdrücke zusammengefasst, deren Ergebnis bereits zur Kompilierungszeit berechnet werden kann. Dadurch kann ein Programm wie in Listing 1 zu dem Programm in Listing 2 vereinfacht werden.

Listing 1: Beispielprogramm

```
let
  var   a : Boolean;
  var   b : Boolean;
  var   x : Integer;
  var   y : Integer;
  var   z : Integer;
  const n ~ 42 + 1;
  const m ~ n + 1
in
begin
  a := true /\ false;
  b := true /\ (1 < 2);
  x := n * 1;
  y := (4 + 7) * 5;
  z := y + 1;
end
```

Listing 2: Optimiertes Beispielprogramm

```
let
  var   a : Boolean;
  var   b : Boolean;
  var   x : Integer;
  var   y : Integer;
  var   z : Integer;
  const n ~ 43;
  const m ~ n + 1
in
begin
  a := false;
  b := true;
  x := n;
  y := 55;
  z := y + 1;
end
```

Implementieren Sie hierzu folgende Vereinfachungsregeln

- C_1 op C_2 wird ersetzt durch das Ergebnis der binären Operation op
  - Wenn C_1 und C_2 Konstanten sind, dann ersetzen Sie die Operation durch das Ergebnis der Operation.
  - Wenn op eine Operation mit neutralem Element ist und C_1 oder C_2 eben dieses neutrale Element ist, dann ersetzen Sie die Operation durch den jeweils anderen Operanden. Beachten Sie hierbe, dass es Operationen gibt, welche ein neutrales Element für C_2 haben.
  - Bei logischen Operationen beachten Sie die Fälle false /\ x = false bzw. true \/ x = true
- op C_1 wird ersetzt durch das Ergebniss der unären Operation op

mit Hilfe eines Visitor Patterns (siehe Skript S.54ff).

Die folgenden binären Operatoren müssen behandelt werden: ”+”, ”-”, ”*”, ”/”, ”//”, ”<”, ”>”, ”<=”, ”>=”, ”/\”, ”\/”

Der folgende unäre Operator muss behandelt werden: ”\”

Beachten Sie hierbei, dass das Ergebnis einer Operation einen anderen Typ als das der Operatoren haben kann. Zum Beispiel kann $1 < 2$ durch $true$ ersetzt werden.

Ihre Lösung implementieren Sie in der Klasse *ConstantFolding*, welche das Interface *Visitor* implementieren muss. Geben Sie dabei so vor, dass Ihre Optimierung den veränderten AST als Ergebnis zurückliefert. Wenn Sie dies konsequent durchführen, reicht es die unveränderten Elemente einfach zurückzugeben. Zur Implementierung können Sie die Hilfsfunktionen in der Datei *constant_folding_helper_functions.java* verwenden, welche die Erkennung und Erzeugung neuer Konstanten erleichtern soll.

Um Ihre Implementierung zu testen, bauen Sie den Aufruf Ihrer Optimierung in der Klasse *Compiler* nach dem Aufruf des Checkers in der Methode *compileProgram* ein. Vergessen Sie auch nicht zu überprüfen, ob die resultierenden Programme mit der Optimierung immer noch das gleiche Verhalten haben.

Überläufe der Zahlenbereiche dürfen Sie bei Ihrer Implementierung ignorieren.

Aufgabe 2.3 Lösung

Die Musterlösung dieser Aufgabe finden Sie in Form der Datei *ConstantFolding.java* in der Anlage.

Aufgabe 2.4 Typprüfung

a) Erläutern Sie den Unterschied zwischen *struktureller Typäquivalenz* und *Typäquivalenz über Namen*.

b) In Triangle gilt strukturelle Typäquivalenz. Wie ist die Typprüfung dort implementiert? Warum ist der Einsatz der *TypeDenoter*-Klassen sinnvoll?

c) Angenommen, Triangle würde Typäquivalenz über Namen verwenden; wie würden Sie Äquivalenz zweier *TypeDenoter*-Objekte testen?

d) Nehmen Sie an, dass Zeigertypen existieren und durch einen Asterisk nach dem Typnamen gekennzeichnet sind. Können Sie das Typsystem (und die Typprüfung) von Triangle einfach um rekursive Typen erweitern? Beispiel:

```
type List ~ ListNode *;
type ListNode ~ record next : List; data : Integer end;
```

Aufgabe 2.4 Lösung

a) Bei struktureller Typäquivalenz sind zwei Typen gleich, genau dann wenn sie die gleiche Struktur besitzen, z.B. Integertypen mit der gleichen Bitbreite, oder Record-Typen, die aus den selben Elementtypen in der selben Reihenfolge zusammengesetzt sind.

Bei Typäquivalenz über Namen wird für jede Typkonstruktion ein einzigartiger Typ erzeugt. Beispiel:

```
let
  type Frac ~ record n: Integer, d: Integer end ;
  var  q1   ~ Frac ;
  var  q2   ~ Frac ;
  var  q3   ~ record n: Integer, d: Integer end ;
  var  q4   ~ record n: Integer, d: Integer end ;
  ...
in ...
```

Hier werden drei verschiedene Typen jeweils durch den Typkonstruktor `record n: Integer, d: Integer end` erzeugt. Der erste Typ hat den Namen “Frac”, die beiden anderen sind anonym und daher ausschließlich zu sich selbst äquivalent. Im Beispiel sind nur die Typen der Variablen w1 und w2 äquivalent; insbesondere sind w3 und w4 nicht äquivalent.

b) Um die Typen von Variablen und anderen Bezeichnern zu speichern, hat im Falle von Mini-Triangle eine Aufzählung der möglichen (primitiven) Typen ausgereicht.

Da Triangle auch zusammengesetzte Typen unterstützt, müsste hier also eine mächtigere Datenstruktur zur Beschreibung eines Typs eingesetzt werden. Einfacher ist es allerdings, eine Referenz auf den entsprechenden `TypeDenoter`-Knoten im AST zu verwenden, dessen Unterbaum die Struktur vollständig beschreibt. Man kann dann die strukturelle Äquivalenz zweier Typen durch Vergleich dieser Unterbäume prüfen. Im Triangle-Compiler ist dieser Vergleich durch entsprechende `equals`-Methoden der `TypeDenoter`-Unterklassen implementiert, die sich rekursiv aufrufen.

Da unterschiedliche Type-Bezeichner diese Prüfung verkomplizieren würden, werden sie in einem Vorverarbeitungsschritt eliminiert und durch eine Referenz auf den `TypeDenoter`, für den der Typ-Bezeichner steht, ersetzt. Dadurch wird der AST zu einem DAG (directed acyclic graph), was aber für den Rest des Compiler kein Problem darstellt.

c) Es würde ausreichen, die Gleichheit der beiden `TypeDenoter`-Objekte mit dem ==-Operator zu testen.

d) Nach der Elimination der Typ-Bezeichner erhält man im Beispiel einen *zyklischen* Graph. Bei der Implementierung der `equals`-Methode zum Testen der strukturellen Äquivalenz kann man nun also nicht mehr unbedingt die `equals`-Methode des Elementtyps aufrufen, sondern muss Vorsichtsmaßnahmen treffen, um Nichttermination zu verhindern.