Grundlagen des Übersetzerbaus: Verfahren

Marco Ammon (my04mivo)

27. Juni 2020

Inhaltsverzeichnis

1 AST-Transformationen **2**
1.1 Innere Klassen . . . 2
1.2 Generics . . . 2

2 Transformation zu Zwischensprache **2**

3 Funktionsaufrufe **3**

4 Geschachtelte Funktionen **3**

5 Objekt-orientierte Sprachen **4**
5.1 Methodenauswahl in Java . . . 4
5.2 Einfachvererbung . . . 4
5.3 Vererbung mit Interfaces . . . 5
5.4 Mehrfachvererbung . . . 6

6 Code-Selektion **6**
6.1 Mit Registerzuteilung . . . 6
6.1.1 Naiver Code-Generator . . . 6
6.1.2 Einfacher Code-Generator mit `getreg` . . . 6
6.1.3 Sethi-Ullman-Algorithmus . . . 7
6.1.4 Dynamische Programmierung . . . 7
6.2 Ohne Registerzuteilung . . . 8
6.2.1 Baumtransformationen . . . 8
6.2.2 Verfahren von Graham/Glanville . . . 8

7 Registerzuteilung **8**
7.1 Grad-$<$ R-Regel . . . 9

8 Instruktionsanordnung **9**

1 AST-Transformationen

1.1 Innere Klassen

innere Klasse: in `Outer` enthaltene, nicht statische Klasse `Inner`

1. flache Hierarchie durch Verschieben der inneren Klasse außerhalb der umgebenden Klasse(n): `Outer.Inner` → `Outer$Inner`
2. Konstruktor der inneren Klasse um Parameter ggf. erzeugen und um Parameter `Outer this$i` ergänzen (mit i als Schachtelungstiefe von `Outer`), zusätzlich gleichnamige Instanzvariable einfügen
3. Zugriffen auf Instanzvariablen von `Outer` ein `this$i.` voranstellen
4. Hilfsmethoden für Zugriff auf private Instanzvariablen von `Outer` in `Outer` einfügen (mit aktueller Java-Version durch spezielles Attribut in Klassendatei nicht mehr notwendig)
5. Alle Auftreten von `Inner` durch `Outer$Inner` ersetzen
6. Bei von `Inner` erbenden Klassen `(new Outer()).super();` im Konstruktor ergänzen, damit Outer-Instanz erzeugt wird
7. Bei in Blöcken deklarierten inneren Klassen wird der Zugriff auf finale (oder „effectively-final“) Variablen durch Ergänzen des Konstruktors um diese Variablen ermöglicht

1.2 Generics

1. „Ausradieren“ der Typen („type erasure“):
   - `GenericClass<TypeParameter>` → `GenericClass`
   - Typ `TypeParameter` bleibt gleich
   - Typparameter `TypeParameter` → `Object`
2. Brückenmethoden einfügen, die `Object` zu `A` casten und dann eigentliche Implementierung aufrufen
3. Wenn Typparameter `A` einer Methode nicht aus den Argumenten ableitbar ist, Verwendung des abgeleiteten Typs `*`, der Untertyp aller Typen ist

2 Transformation zu Zwischensprache

- mehrdimensionale Arrays meistens zu eindimensionalen Array linearisiert
- Operatorenabbildung in „Post-Order“-Reihenfolge
- Kurzschlusssemantik:
  - code(`a && b`, $L_{\texttt{true}}$, $L_{\texttt{false}}$) → code(`a`, `L1`, $L_{\texttt{false}}$); `L1:` code(`b`, $L_{\texttt{true}}$, $L_{\texttt{false}}$)
  - code(`a || b`, $L_{\texttt{true}}$, $L_{\texttt{false}}$) → code(`a`, $L_{\texttt{true}}$, `L1`); `L1:` code(`b`, $L_{\texttt{true}}$, $L_{\texttt{false}}$)
  - code(`!a`, $L_{\texttt{true}}$, $L_{\texttt{false}}$) → code(`a`, $L_{\texttt{false}}$, $L_{\texttt{true}}$
- code(`while e do st od`) → `jmp` $L_{\texttt{cond}}$; $L_{\texttt{true}}$: code(`st`); $L_{\texttt{cond}}$: code(`e`, $L_{\texttt{true}}$, $L_{\texttt{false}}$); $L_{\texttt{false}}$:
- `switch-case`:
  - `if`-Kaskade

- `lookupswitch`: Tabelle aus $(c_i, \texttt{L}_i)$-Tupel von Konstante c_i und Sprungziel $\texttt{L}_i$ wird durchsucht
- `tableswitch`: Konstante wird als Index in Tabelle mit Sprungzielen („jump table“) gewählt

3 Funktionsaufrufe

1. Vorbereitung:
   a) Argumentauswertung gemäß Übergabemechanismus
   b) Sichern von Caller-Save-Registern auf dem Stack
   c) Argumente in Registern/auf dem Stack ablegen
   d) Funktionsaufruf
2. Prolog:
   a) Sichern des alten FP und Allokation des Stackframes
   b) Sichern von Callee-Save-Registern im Stackframe
3. Funktionsrumpf
4. Epilog:
   a) Ablage des Rückgabewerts in Register/auf dem Stack
   b) Restauration von Callee-Save-Registern
   c) Freigabe des Stackframes und Restauration des FP
   d) Rückkehr
5. Nachbereitung:
   a) Abspeichern des Ergebnis an vorgesehener Stelle
   b) Entfernen der Argumente vom Stack
   c) Restauration der Caller-Save-Register

4 Geschachtelte Funktionen

- ohne Display:
  - Aufruf der geschachtelte Funktion mit Zeiger auf Aktivierungsrahmen der umschließenden Funktion (sog. statischer Vorgängerverweis SV)
  - bei Aufruf aus tieferer Schachtelungstiefe SV des Aufrufers ggf. bis zum relevanten Aktivierungsrahmen verfolgen
- mit Display (gesondertes, globales Array) zur Speicherung der SV:
  - Bei Betreten von Funktion der Schachtelungstiefe t, ihren FP an Index t im Display speichern und ggf. bereits bestehenden Wert einer Schwesterfunktion im eigenen Aktivierungsrahmen sichern
  - Durch statisch bekannte Schachtelungstiefe Größe des Displays zur Übersetzungszeit bekannt und Zugriff auf lokale Variablen aus umschließenden Kontext durch Dereferenzieren des SV aus statisch bekannter Position im Display
- Funktionszeiger: auch Argumentwerte müssen mit Zeiger gespeichert werden

5 Objekt-orientierte Sprachen

5.1 Methodenauswahl in Java

1. Bestimmung der Klasse (des Interfaces), in der nach Methode zu suchen ist
2. Bestimmung der zu Argumenttypen passenden, anwendbaren/zugreifbaren Methoden
   a) Auswahl der Methodendeklarationen, deren Parameter in Anzahl und Typ zu den statischen Argumenttypen passen
   b) Verwerfen der in Sichtbarkeit eingeschränkte Methoden
   c) Auswahl der spezifischsten Methode
3. Kontextüberprüfung (z.B. statische Funktionen, `void` Rückgabetyp, etc.)

5.2 Einfachvererbung

- Attribute aus Oberklasse `O` liegen auch in Unterklasse `U` am selben statischen Offset zum Objektanfang → einfacher, statischer Zugriff
- Objekt enthält Zeiger auf Klassendeskriptor
- Klassendeskriptor enthält:
  - V-Table mit Adressen der Funktionen(Indizes in Tabelle wie Attribute)
  - Verweis auf Klassendeskriptor der Elternklasse
- Dynamischer Methodenaufruf:
  1. Verfolgung des Zeigers zum Klassendeskriptor
  2. Index in V-Table im Klassendeskriptor ist für auszuführende Methode bereits vom Compiler bekannt
  3. Indirekter Sprung
- Casts:
  - Upcasts können vom Übersetzer verifiziert werden
  - Downcasts müssen zur Laufzeit überprüft werden:
    * ohne Display:
      1. Verfolgung des Zeigers zum Klassendeskriptor
      2. Vergleich des Klassendeskriptors des Objekts mit Klassendeskriptor der Zielklasse
      3. Solange keine Übereinstimmung, Vergleich mit Klassendeskriptor der Elternklasse
      4. Beim Erreichen von gesuchtem Klassendeskriptor ist Casts erlaubt, ansonsten Laufzeitfehler
    * mit Display:
      1. (maximale) Vererbungstiefe statisch feststellbar → pro Klasse kann Array aus Oberklassen angelegt werden
      2. Zur Überprüfung in Array vergleichen, ob an (statisch aus der Schachtelungstiefe bekannter) Position der Zielklasse wirklich Zielklasse eingetragen ist

5.3 Vererbung mit Interfaces

- Statische Bestimmung der V-Table-Indizes nicht mehr möglich
- Pro Kombination aus Klasse `A` und Interface `I` wird eine Interface-V-Table `A:I` angelegt, in der Interface-Funktionen an statisch bekannten Indizes liegen:
  - Interface-V-Table enthält Verweis auf Klassendeskriptor von `A`
  - Da von Klasse implementierte Funktionen erwarten, dass `this` auf Objektbeginn zeigt, müssen Hilfsmethoden, die `this` korrigieren, erstellt und in die Interface-V-Table eingetragen werden
- Objekt enthält „hinter“ Attributen noch Verweise auf alle assoziierten Interface-V-Tables
- Klassendeskriptor enthält Anzahl der von einer Klasse implementierten Interfaces sowie Verweis auf Interface-Tabelle der Klasse
- Interface-Tabelle enthält Tripel aus (Offset, Verweis auf Interface-Deskriptor, Verweis auf Interface-V-Table) für alle von Klasse implementierten Interfaces:
  - Offset gibt Offset von Verweis auf jeweilige Interface-V-Table zum Anfang des Objekts an
- Dynamischer Methodenaufruf falls statischer Typ Interface ist (sonst wie bei Einfachvererbung):
  1. Hilfsmethode mit statisch bekanntem Offset in der Interface-V-Table nachschlagen
  2. Hilfsmethode verschiebt `this`-Zeiger um statisch bekannten Offset auf Objektanfang und ruft statisch bekannte, in `A` implementierte Methode auf
- Casts:
  - Klasse → Interface:
    1. Objektzeiger zu Klassendeskriptor zu Interface-Tabelle verfolgen
    2. In Interface-Tabelle nach Eintrag für Interface suchen (ggf. komplettes Durchlaufen)
    3. Im Erfolgsfall ist Cast gültig, sonst Laufzeitfehler
    4. Referenz für neue Variable aus Objektanfang + zu Interface gehörigem Offset aus Interface-Tabelle bestimmen
  - Interface → Klasse:
    1. Objektzeiger zu Interface-V-Table zu Klassendeskriptor verfolgen
    2. Falls Klassendeskriptor nicht Zielklasse entspricht, Verfolgung zu Oberklasse(n)
    3. Im Erfolgsfall ist Cast gültig, sonst Laufzeitfehler
    4. Zeiger von Klassendeskriptor zu Interface-Tabelle verfolgen
    5. In Interface-Tabelle Eintrag für Interface nachschlagen (garantiert vorhanden)
    6. Referenz für neue Variable aus aktuellem Zeiger - zu Interface gehörigem Offset aus Interface-Tabelle bestimmen
- Interface mit Standardimplementierung:
  - Wird Standardimplementierung verwendet, muss der Objektzeiger beim Aufruf mit Interface als statischem Typ nicht korrigiert werden; allerdings ist Hilfsmethode anzulegen und in Klassen-V-Table einzutragen

- Interface mit veränderlichem Zustand:
  - Implementierende Klassen erhalten Instanzvariablen des Interfaces als eigene Instanzvariablen
  - Interface hat implizite Getter-/Setter-Methoden, die von Klasse implementiert werden

5.4 Mehrfachvererbung

- Objekt enthält V-Table pro Elternklasse und eine V-Table für eigene Methoden:
  - Wenn Elternmethoden nicht überschrieben werden, Verhalten wie bei Interfaces mit Standardimplementierung
  - Sonst Hilfsmethoden in V-Tables für Anteil der jeweiligen Elternklasse eintragen
- Diamantenproblem: Mehrere Elternklassen erben von selber Klasse `A` $\rightarrow$ Attribute von `A` mehrfach vorhanden
- Virtual Inheritance als Lösung für Diamentenproblem (nur ein Feld für doppeltes Attribut `a`):
  - `A`-Anteil des Objekts nicht mehr an festem Offset, sondern zusätzliches Offset-Feld in jedem „Anteil“ des Objekts

6 Code-Selektion

6.1 Mit Registerzuteilung

6.1.1 Naiver Code-Generator

- arbeitet auf minimalem Grundblöcken (einzelner Zwischencode-Befehl)
- pro Befehl:
  1. Da alle Variablen im Speicher liegen, Laden der Operanden in Register
  2. Durchführung der Operation auf Registern
  3. Rückschreiben des Ergebnisses in Speicher
- Optimierung: feste Zuteilung von Argumenten, Variablen und Zwischenergebnissen einer Funktion in Register (wenige Register müssen aber für Operationen frei bleiben)

6.1.2 Einfacher Code-Generator mit `getreg`

- arbeitet auf maximalen Grundblöcken
- alle Variablen vor und nach maximalem Grundblock im Speicher
- Durchlaufen aller Zwischencode-Befehle und Mitführen von Register- und Adressdeskriptoren
- $$\texttt{getreg}\ (x \leftarrow y\ \texttt{op}\ z) = \begin{cases} R & \text{falls } y \text{ alleine in } R \text{ steht und } y \text{ nicht mehr benötigt wird} \\ U & \text{falls es unbenutztes Register } U \text{ gibt} \\ R & \text{falls } x \text{ in einem Register stehen muss/soll, schreibe Inhalt von } R \text{ in Speicherplätze aller darin enthaltenen Variablen} \\ M_x & \text{sonst} \end{cases}$$

,
dazu zusätzlich Deskriptoren anpassen

- Für jeden Befehl $x \leftarrow y \,\texttt{op}\, z$:
  1. $L_x =$ `getreg` $(x \leftarrow y \,\texttt{op}\, z)$
  2. Wähle L_y aus Adressdeskriptor von y (möglichst Register)
  3. Falls y noch nicht in L_x, generiere $L_x = L_y$
  4. Wähle Platz L_z aus Adressdeskriptor von z (möglichst Register)
  5. Generiere $L_x = L_x \,\texttt{op}\, L_z$
  6. Deskriptoren aktualisieren:
     a) L_x in Adressdeskriptor von x eintragen
     b) Falls L_x Register, zugehörigen Registerdeskriptor aktualisieren
  7. Falls keine weitere Verwendung von y, z im Grundblock und in Register,
     a) ggf. Werte in Speicher zurückschreiben
     b) Register von y bzw. z freigeben
     c) Deskriptoren aktualisieren

6.1.3 Sethi-Ullman-Algorithmus

1. Baue DAG-Repräsentation von Befehlen, Variablen und Zwischenergebnissen auf
2. Wandle DAG durch Herausschneiden gemeinsamer Teilausdrücke in Wald um
3. Bestimme Ershov-Zahl r (Anzahl der für Auswertung benötigten Register) rekursiv für jeden Teilbaum (vertausche Operanden bei kommutativen Operationen so dass möglichst viele Variablen direkt aus Speicher gelesen werden können):

$$r(t_1 \,\texttt{op}\, t_2) = \begin{cases} r(t_1) + 1 & \text{falls } r(t_1) = r(t_2) \\ \max(r(t_1), r(t_2)) & \text{sonst} \end{cases}$$

$$r(v) = \begin{cases} 1 & \text{falls linker Operand} \\ 0 & \text{falls rechter Operand} \end{cases}$$

4. Code-Erzeugung und Mitführung eines Stapels unbenutzter Register:
   a) Rekursives Traversieren des Ausdrucksbaums
   b) Besuchen der Kinder nach absteigendem Registerbedarf
   c) Generierung des Befehls für aktuellen Ausdruck
5. Auftrennung des Baumes, wenn Teilbäume mehr Register als verfügbar benötigen → Teilbaum vor Auswertung des übrigen Baums im Speicher auswerten

6.1.4 Dynamische Programmierung

1. Rekursives Berechnen eines Kostenfelds $C[i]$ für jeden Knoten des Baums und $0 \leq i \leq |R|$ (R Register der Zielarchitektur):
   - $C[i]$: Kosten der Auswertung des Teilbaums in ein Register, sofern i Register zur Verfügung stehen:

a) Für alle möglichen Maschinenbefehle x alle möglichen Auswertungsreihenfolgen y der Registeroperanden als $C[x, y, i]$ berechnen: erster Operand hat i Register, zweiter $i - 1$ Register, etc. zur Verfügung

b) $C[x, y, i]$ = Summe der Kindkosten bei Auswertungsreihenfolge y + Kosten des Befehls x

c) $C[i] = \min C[x, y, i]$, dann mit x und y des Minimums annotieren

- $C[0]$: Kosten der Auswertung des Teilbaums, wenn alle Register zur Verfügung stehen, aber Ergebnis am Ende in Speicher geschrieben wird (ergo $C[j] + 1$)

2. Rekursives Auftrennen des Baums, wenn Ergebnis in Speicher abgelegt wird (aus annotiertem Befehl ersichtlich)
3. Ablegen der Teilbäume in Schlange („untere Bäume zuerst“)
4. Code-Generierung durch Abarbeitung der Schlange und Code-Generierung der jeweiligen Teilbäume

6.2 Ohne Registerzuteilung

6.2.1 Baumtransformationen

1. Top-Down-Überdeckung des Baums mit passenden Mustern, Auswahl nach Kostenmaß (etwa meiste überdeckte Knoten)
2. Bottom-Up-Transformation gibt Instruktion bei Anwendung des Musters aus

6.2.2 Verfahren von Graham/Glanville

1. Instruktionsbäume in Präfixform („polnische Notation“) kodieren
2. Definition einer Maschinengrammatik, die für jeden verfügbaren Maschinenbefehl eine Produktion enthält:
   - linke Seite: Betriebsmittel, das das Ergebnis des Befehls enthält (Speicher, Register)
   - rechte Seite: Befehl mit Operanden
3. Parsen des Baumtexts mit Maschinengrammatik emittiert den Maschinencode (dabei Mehrdeutigkeit nach Heuristik auflösen)

7 Registerzuteilung

- Lebendigkeit der Definition eines symbolischen Registers (einer Variable), wenn Pfad von Eintrittsknoten über Definition und Definition nach Verwendung noch gebraucht wird
- Lebensspanne einer Definition umfasst alle Punkte, an denen Definition lebendig ist
- Lebensspannen zweier Definitionen des selben symbolischen Registers verschmelzen, sofern sich beide überschneiden
- Generelles Vorgehen:
  1. Konstruktion des Kollisionsgraphs:
     a) Knoten für jede Lebensspanne von symbolischen Registern
     b) Kanten zwischen kollidierenden (sich überschneidende) Lebensspannen
  2. Konstruktion des Interferenzgraphs:

a) Erweiterung des Kollisionsgraphs um einen Knoten pro realem Register

b) Einfügen von Kanten zwischen allen realen Registern („Clique“)

c) Einfügen von Kanten zwischen sich (aufgrund von Limitationen der Befehle) ausschließenden realen und symbolischen Registern

3. Färben des Interferenzgraphs mit R (Anzahl der realen Register) Farben entspricht Registerzuteilung
4. Bei Nichtfärbbarkeit des Graphs ggf. symbolisches Register durchgängig im Speicher halten

7.1 Grad-$<$ R-Regel

1. Sofern Graph Knoten mit Grad $< R$ enthält, ist Graph färbbar $\rightarrow$ Knoten aus Graph (und inzidente Kanten) entfernen
2. Rekursive Anwendung auf verbleibenden Graph
3. Wird leerer Graph erreicht, ist ursprünglicher Graph färbbar $\rightarrow$ Knoten in umgekehrter Reihenfolge Farbe zuweisen, die nicht mit Nachbarn in Konflikt steht
4. Optimistische Erweiterung: Wenn kein Knoten mit Grad $< R$ verbleibt, einen Knoten als aussortiert markieren, dann bei Farbvergabe aber trotzdem versuchen, passende Farbe zu finden
5. Lebensspannenaufspaltung durch zusätzlich eingefügte Kopieroperationen
6. Vermeidung von Kopieroperationen durch Registerverschmelzung („Coalescing“):
   - Kopieroperation $s_i \leftarrow s_j$ kann gespart werden, wenn jeweilige Lebensspannen nicht kollidieren
   - R-Färbbarkeit des entstehenden Interferenzgraphs bleibt erhalten wenn eine der folgenden Bedingungen zutrifft:
     - Grad des verschmolzenen Knoten $< R$
     - verschmolzener Knoten hat weniger als R Nachbarn vom Grad $\geq R$
     - alle Nachbarn beider Knoten entweder schon mit dem anderen Knoten interferieren oder einen Grad $< R$ haben

8 Instruktionsanordnung

- Abhänigkeits-DAG:
  - Betrachtung pro Grundblock
  - Knoten: Instruktion
  - Kante: Datenabhängigkeit zwischen zwei Instruktionen:
    - Flussabhängigkeit („read after write“)
    - Ausgabeabhängigkeit („write after write“)
    - Antiabhängigkeit („write after read“)
- Ausführungszeit („ExecTime“) einer Instruktion: Zeit bis Instruktion durch alle Stufen der Pipeline gewandert ist
- Latenz („Latency“) einer Instruktion: Zeit, die Instruktion bis zur Ausführung warten muss

- Verzögerung („Delay“) einer Instruktion: Zeit bis alle davon abhängigen Instruktionen ausgeführt wurden:

$$\text{Delay}(n) = \begin{cases} \text{ExecTime}(n) & \text{falls } n \text{ Blattknoten} \\ \max_{m \in \text{Succ}(n)} \left(\text{Latency}(n, m) + \text{Delay}(m)\right) & \text{sonst} \end{cases}$$

- List-Scheduling: solange DAG Knoten enthält
  1. Wurzelknoten w mit größtem Delay wählen
  2. w an Ausgabeliste anhängen
  3. w aus DAG entfernen