Semantik von Prozeduraufrufen

Kopierregelsemantik und call-by-Techniken

Gebundenes Programm

Damit klar ist, wieviel Speicherplatz mit dem Namen einer Variable referenziert wird, und mit welchen Operationen der dort abgelegte Wert manipuliert werden darf, werden Varicblen normalerweise vor ihrer ersten Benutzung *deklariert* (ggf. implizit bei der ersten Wertzuweisung).

Für einen Identifikator nennt man die Deklaration auch das ***deklarierende Auftreten*****,** und die übrigen Vorkommen ***angewandte Auftreten***.

Gebundenes Programm

In Sprachen wie der MMS2 können Deklarationen an zwei Stellen auftreten:

Im Deklarationsteil, z.B.

var x: int;

Dann nennt man x eine *lokale Variable*.

In der Deklaration einer Funktion, z.B.

func f(..., x: int, ...)

Dann nennt man x einen *formalen Parameter*.

Gebundenes Programm

Die Gültigkeitsbereiche (*scopes*) von
lokale Variablen ist der sie umgebenden Block,

von formalen Parametern der Prozedurrumpf,
also der Block der zur Funktionsdeklaration gehört.

Tritt eine Variable in einem Block auf, in dem sie nicht deklariert wurde, nennt man sie (bzgl. des aktuellen Blocks) ***globale Variable***.

Gebundenes Programm

Ein Programm heisst ***gebunden***, wenn es zu jedem angewandten Auftreten eines Identifikators ein zugehöriges deklarierendes Auftreten gibt.

Ein gebundenes Programm enthält also bzgl . des äussersten Blocks keine globalen Variablen, alle Auftreten von Variablen sind an eine Deklaration gebunden.

Kopierregelsemantik

Ein gebundenes Programm heisst *partiell übersetzbar*, falls es übersetzbar ist, wenn man alle Prozedurrümpfe durch leere Blöcke ersetzt. (also das "Hauptprogramm" übersetzbar ist)

Sei P ein partiell übersetzbares Programm, und sei $p(x_1,...,x_n)$ ein angewandtes Auftreten von p im Anweisungsteil des Programms,
Dann entsteht das Programm P' aus P durch Anwenden einer Kopierregel, die das Auftreten $p(x_1,...,x_n)$ durch eine modifizierte Kopie des Prozedurrumpfes von p ersetzt.

Kopierregelsemantik

Je nach Kopierregel unterscheidet man drei Semantiken:

- call by name
- call by reference
- call by value

Das Verfahren operiert rein syntaktisch auf dem Programmtext, nach der ersten Anwendung stehen ggf. wieder Prozeduraufrufe im Anweisungsteil, die durch einen nochmalige Anwendung der Kopierregel ersetzt werden, etc.

Beispiel

Kopierregelsemantik
static / dynamic scoping

Kopiert man Programmcode ohne Umbenennen der lokal gebundenen Variablen, kann es vorkommen, das der Wert eines formalen Parameters (also ein Identifikator) durch eine vorhandenen lokale Bindung neu gebunden wird, die Bindungsrelation sich also dynamisch zur Ausführungszeit ändert. Man spricht dann von *dynamic scoping*. Das Verhalten ist allerdings normalerweise unerwünscht.

Besonders leicht vorkommen kann dieser Fall beim der Technik *call by name* ohne Umbenennung, ggf. aber auch bei den anderern Techniken.

Beispiel

Kopierregelsemantik

Im Folgenden gehen wir von einer Deklaration

func p(a_1, ... a_n) {... rumpf ...}

und einem Aufruf

p(x_1, ..., x_n)

aus.

Kopierregelsemantik
call by value

Hier werden auf jeden Fall **Werte** übergeben, für jeden Funktionsaufruf wird eine **lokale Kopie** der übergebenen Werte angelegt.

Zuweisungen im Rumpf wirken sich nicht aus, es sei denn, der Wert war eine Speicherreferenz (ein *Pointer*).

Kopierregelsemantik
call by value

Kopierregel für call by value:

a. **Substitution:** Für jeden formalen Parameter a_i wird eine neue Hilfsvariable a_i' eingeführt, und zu Beginn des Anweisungsteils des Rumpfes wird $a_i' := x_i$ eingefügt. Dann werden alle Auftreten von a_i im Rumpf durch a_i' ersetzt.
b. **Umbenennung:** Alle lokalen Variablen im Prozedurrumpf werden unter Verwendung neuer Identifikatoren umbenannt.

BSP

Kopierregelsemantik
call by name

Kopierregel für call by name:

a. **Substitution:** Ersetze alle formalen Parameter a_i, die im Prozedurrumpf auftreten, durch die aktuellen Parameter x_i (genau so, namentlich)
b. **Umbenennung:** Alle lokalen Variablen im Prozedurrumpf werden unter Verwendung neuer Identifikatoren umbenannt.

Beispiel

Kopierregelsemantik
call by reference

Kopierregel für call by reference:

a. **Substitution:** Jeder aktuelle Parameter x_i wird so lange ausgewertet, bis man eine Speicherreferenz Stor(x_i) hat (üblicherweise ein Variablename). Dann wird jedes Auftreten von a_i im Rumpf durch Stor(x_i) ersetzt.

b. **Umbenennung:** Alle lokalen Variablen im Prozedurrumpf werden unter Verwendung neuer Identifikatoren umbenannt.

Beispiel

Kopierregelsemantik
call by ...

HIer spielt auch noch die Auswertungsreihenfolge von Parametern eine Rolle, und die Auswertungsstrategie.

Insbesondere funktionale Sprachen realisieren auch oft

call by need

Aktuelle Parameter(ausdrücke) werden hier erst dann ausgewertet, wenn sie benötigt werden.

Man spricht dann auch von

lazy evaluation

In hochparallelen Sprachen ist dagegegen ggf. eine parallele Auswertung der Parameter vorgesehen.

Kopierregelsemantik

- Mit dieser formalen Semantik kann man weitere Eigenschaften von Programmiersprachen untersuchen, z. B. formal korrekte Parameterübergaben, formale Erreichbarkeit, Makro-Eigenschaft. Wir gehen hier aber nicht weiter darauf ein.
- Diese formale Semantik wird durch die Speicherorganisation mit Activation Records und ein Laufzeitsystem, das diese Technik unterstützt, entsprechend umsetzt

Kopierregelsemantik

- Kann man das Verhalten von f() im go-Beispiel für die Fibonacci-Rekursion damit erklären?

Kopierregelsemantik

- Kann man das Verhalten von f() im go-Beispiel für die Fibonacci-Rekursion damit erklären?

Mit etwas grosszügiger Interpretation, insbesondere bei der Rückgabe von Werten, die wir ja nicht vorgesehen haben, ja. (Wir mogeln aber bei den scopes)

Kopierregelsemantik

```
package main

// fib returns a function that returns
// successive Fibonacci numbers.
func fib() func() int {
    a, b := 0, 1
    return func() int {
        a, b = b, a+b
        return a
    }
}

func main() {
    f := fib()
    // Function calls are evaluated left-to-right.
    f()
    f()
    f()
}
```

Kopierregelsemantik

```
package main

// fib returns a function that returns
// successive Fibonacci numbers.
func fib() func() int
}

func main() {
      a, b := 0, 1
      f := func() int {
          a, b = b, a+b
          return a
     // Function calls are evaluated left-to-right.
     f()
     f()
     f()
}
```

Kopierregelsemantik

```
package main

...
func main() {
      a, b := 0, 1
      f := func() int {
          a, b = b, a+b
          return a
     // Function calls are evaluated left-to-right.
     a, b = b, a+b
     a
     f()
     f()
}
```

Kopierregelsemantik

```
package main

...
func main() {
      a, b := 0, 1
      f := func() int {
          a, b = b, a+b
          return a
     // Function calls are evaluated left-to-right.
     a, b = b, a+b
     a
     a, b = b, a+b
     a
     a, b = b, a+b
     a

}
```

Lernziele

- die Aufrufttechniken call by value / reference / name verstanden haben, und Programme entsprechend der Techniken auswerten können.
- Ein Beispiel für den Unterschied zwischen dynamic scoping und static scoping angeben können.
- Das Prinzip einer Kopierregelsemantik verstanden haben.