Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Ergänzung Kontextanalyse

- Organisation der Kontextanalyse:
  Baumtransformation oder Aufbau eines erweiterten AST
- Beispiel AST-Erweiterung:
  Virtuelle / Reale Referenz-Typen
- Beispiel Baum-Traversierung:
  Modifiziertes Besuchermuster

Beispiel: Mini-Sprache mit Prozeduren

Programmbeispiel: Objekt mit Prozeduren

```
OBJECT
  VAR y : INT := 12;

  PROC p(p: INT, REF q: INT)
  VAR z: INT := 0;
  BEGIN
    q := y + z + p;
  END

BEGIN
   y := y+1;
   p(y+1, y);
END
```

In einem Objekt können Variablen und Prozeduren definiert werden.
Prozeduren
- *haben Referenz- und Wert-Parameter*
- *Lokale Variablen-Definitionen*
- *einen Anweisungsblock*

Organisation der Kontextanalyse: Beispiel Referenzen

Variablen und Wert-Parameter vs Referenz-Parameter

Variablen und Wert-Parameter unterscheiden sich von Referenz-Parametern:

- **Variablen und Wert-Parameter werden zur Laufzeit durch Speicherstellen repräsentiert, die den aktuellen Wert enthalten**
- **Referenz-Parameter werden zur Laufzeit durch Speicherstellen repräsentiert, die die Adresse einer Speicherstelle enthalten.**

Der Unterschied spielt eine Rolle:

- **Bei der Übergabe eines Arguments an einen Parameter (Adresse oder Wert)**
- **Bei jeden lesenden oder schreibenden Zugriff auf auf eine Speicherstelle:**
  direkt bei Variablen und Wert-Parametern
  indirekt (via Referenz / Adresse / Pointer) bei Referenz-Parametern
- **Referenz-Übergabe in (*plain KR*-) C**
  Angenommen Prozedur p hat einen Referenz-Parameter:
  p(x) ~> p(&x)

Organisation der Kontextanalyse: Beispiel Referenzen

Referenzen sind implizite Zeiger-Typen

Referenzen sind implizite („versteckte“) Zeiger-Typen

So wie die beiden Vorkommen von `'x'` in

```
x := x;
```

zu unterschiedlichem Code führen müssen (l-Wert / r-Wert)

So muss `'x'` / `'a'` in

```
PROC p(REF x: INT)          und     PROC p(x: INT)
    … x …                               … x …

p(a)                        und     p(a)
```

zu unterschiedlichem Code führen (hier C als „Maschinensprache“):

```
void p(int *x)              und     void p(int x)
    … *x …                              … x …

p(&a)                       und     p(a)
```

Organisation der Kontextanalyse: Beispiel Referenzen

Referenzen sind implizite Zeiger-Typen

Wer führt die versteckten Zeiger-Typen ein.

Der Parser

könnte – falls ihm die entsprechenden Informationen vorliegen – die die Referenzen in den erzeugten AST einbauen (als „*“- und „&“-Operatoren).

Die Kontextanalyse

könnte die Semantik der Referenzen

- **in Form weiterer Attribute in den AST einbauen**
- **den AST transformieren, z.B.:**

  x ~> *x als implizite Konversion vom Typ Ref-Int ~> Int

Die Codegenerierung

Der AST wird nicht modifiziert, die Codegenerierung muss die notwendigen Analysen nachholen

Letztlich ist es eine Frage des Geschmacks und der SW-Technik wie die Referenzen behandelt werden:

- **In dem vom Parser erzeugten AST sollte nur die Information stecken, die vom Programmierer „gemeint“ war, nichts, was Zielsprachen-spezifisch ist – waren Referenzen in der Quellsprache „gemeint“, oder sind sie Zielsprachen-spezifisch?**
- **Bei welchen Verfahren wird man am ehesten die Übersicht behalten**

Organisation der Kontextanalyse: Beispiel Referenzen

Referenzen sind ~~implizite~~ Zeiger-Typen

Referenzen können explizit als Typen eingeführt werden

Typen für die es keine Typ-Ausdrücke in der Quellsprache gibt:

```
object StaticTypes {

  sealed abstract class TypeInfo

  object IntTypeInfo extends TypeInfo

  // virtual reference type introduced by the compiler
  case class RefTypeInfo(baseType: TypeInfo) extends TypeInfo

}
```

Philosophie: Referenz-Typen sind „eigentlich gemeint“, auch wenn die Programmierer sie nicht explizit definieren müssen / können.

Star-Conv: Implizite Dereferenzierungen explizit machen

Eine Referenz kann überall dort verwendet werden, wo ein Wert gebraucht wird:

Notwendig ist eine implizite Konversion *Star-Conv* : Ref-T ~> T (Derefernzierung x ~> *x)

Diese Konversion kann in den AST eingebaut werden:

- **sofort bei der AST-Generierung vom Parser, oder**
- **in einer Transformation des AST**

Wenn der Parser den „korrekten“ AST mit impliziten Konversionen generieren soll, dann muss die Typisierung – zumindest teilweise – mit dem Parser verschränkt ablaufen.

In unserer Beispiel-Sprache sind beide Optionen möglich

Organisation der Kontextanalyse: Beispiel Referenzen

Dereferenzierungen explizit machen

Variante 1: Ein Flag wird im AST gesetzt

Da die Beispiel-Sprache keine allgemeine Pointer-Arithmetik unterstützt, sondern nur Referenz-Parameter erlaubt, können nur Speicherstellen als Adressen verwendet werden.

Es reicht darum ein Flag aus, das kenntlich macht, ob eine Speicherstelle vor dem Zugriff dereferenziert werden muss:

```
// Expression that denote int values
sealed abstract class Exp extends Positional {
  var staticType: Option[TypeInfo] = None // will be set by typifier
}
case class Number(d: Int)        extends Exp
case class Add(e1: Exp, e2: Exp) extends Exp
case class Sub(e1: Exp, e2: Exp) extends Exp
case class Div(e1: Exp, e2: Exp) extends Exp
case class Mul(e1: Exp, e2: Exp) extends Exp
case class LocAccess(var locExp: LocExp)   extends Exp

// Expressions that denote storage locations
sealed abstract class LocExp extends Positional {
  var staticType: Option[TypeInfo] = None // will be set by typifier
}
case class DirectLoc( // elementary storage location
   symb: LocSymbol,
   var starOp: Option[Boolean] = None // apply *-Conv to the value (will be set by typifier)
) extends LocExp
```

Organisation der Kontextanalyse: Beispiel Referenzen

Dereferenzierungen explizit machen

Variante 2: Ein *-Konversions-Knoten wird in den AST eingefügt:

```
// Expressions that denote int values
sealed abstract class Exp extends Positional {
  var staticType: Option[TypeInfo] = None
}
case class Number(d: Int)                  extends Exp
case class Add(e1: Exp, e2: Exp)           extends Exp
...
case class LocAccess(var locExp: LocExp)   extends Exp



// Expressions that denote storage locations
sealed abstract class LocExp extends Positional {
  var staticType: Option[TypeInfo] = None // will be set by typifier
}

// elementary storage location
case class DirectLoc(symb: LocSymbol) extends LocExp

// virtual dereferencing operation (*-operation)
case class StarConv(locExp: LocExp) extends LocExp
```

Organisation der Kontextanalyse: Beispiel Referenzen

Dereferenzierungen explizit machen

Variante 2: Ein *-Konversions-Knoten wird in den AST eingefügt

Das kann der Parser problemlos erledigen:

```
private def lExp: Parser[LocExp] = positioned {
  definedLoc ^^  {
    case symb@RefParamSymbol(_) => StarConv(DirectLoc(symb)) // Ref-Parameters always need de-referencing
    case symb => DirectLoc(symb)
  }
}
```

Organisation der Kontextanalyse: Beispiel Referenzen

Dereferenzierungen explizit machen

Die Typisierung muss dann natürlich die neuen Konversions-Knoten im Betracht ziehen:

```
private def analyseLocExp(le: LocExp): Unit = le match {
  case DirectLoc(v@VarSymbol(_)) =>
    le.staticType = v.staticType
  case DirectLoc(pv@ValParamSymbol(_)) =>
    le.staticType = pv.staticType
  case DirectLoc(pr@RefParamSymbol(_)) =>
    le.staticType = pr.staticType
  case StarConv(sub_le) =>
    analyseLocExp(sub_le)
    sub_le.staticType match {
      case Some(RefTypeInfo(baseType)) =>
        le.staticType = Some(baseType)
      case _ =>
        throw new Exception(s"internal error: $le with type ${le.staticType} may not be de-referenced")
    }
}
```

Generische AST-Traversierung

Wenn der AST komplexer wird und mehrfach durchlaufen werden muss, dann empfiehlt es sich eine (eventuell mehrere) generische Traversierungsroutine zu formulieren.

Diese kann einer der vielen Varianten des Besuchermusters entsprechen, oder völlig unabhängig davon sein.

Beispiel Traversierung ohne Besuchermuster:

- **Generiere alle relevante Knoten des AST als eine Folge von Knoten (Sequenz / Liste / Iterable / Stream / …)**
- **Durchlaufe die Folge mit einer Analyse-Funktion**

Generische AST-Traversierung

Beispiel: Generiere Folgen relevanter Knoten – triviale Rekursion über die Baumstruktur:

```
sealed abstract class Exp
case class Number(v: Int) extends Exp

case class Add(left: Exp, right: Exp) extends Exp


def allNumbers(e: Exp): Seq[Number] = e match {
  case n@Number(v) => Seq(n)
  case Add(l, r) => allNumbers(l) ++ allNumbers(r)
}

def allAdds(e: Exp): Seq[Add] = e match {
  case Number(v) => Seq()
  case a@Add(l, r) => Seq(a) ++ allAdds(l) ++ allAdds(r)
}
```

Anwendungsbeispiel

```
val tree = Add(Add(Number(1), Number(2)), Add(Number(3), Add(Number(4), Number(5))))

for (n <- allNumbers(tree))
  println(n)
```

Generische AST-Traversierung

Beispiel: Generiere Folgen relevanter Knoten

Fehlerbehandlung

Während der Traversierung werden eventuell Fehler festgestellt und geworfen werden

Beispiel:

```
val tree = Add(Add(Number(1), Number(2)), Add(Number(3), Add(Number(4), Number(5))))

val res = allNumbers(tree).map ( (n:Number) =>
  Try { n match {
    case Number(3) => throw new Throwable("3 is not allowed!")
    case Number(x) => println(x)
  }}
)

println(res)
```

```
List(Success(()), Success(()), Failure(java.lang.Throwable: 3 is not allowed!), Success(()), Success(()))
```

Das Ergebnis ist eine Liste von Try-s. Nicht unbedingt, das, was wir wollten.

Generische AST-Traversierung

Beispiel: Generiere Folgen relevanter Knoten

Fehlerbehandlung

Während der Traversierung werden eventuell Fehler festgestellt und geworfen werden

Wir wollten bei einem Fehler die gesamte Aktion abbrechen und ein (das erste) *Failure* als Gesamt-Ergebnis haben.

***Fold* leistet das. Beispiel:**

```
val tree = Add(Add(Number(1), Number(2)), Add(Number(3), Add(Number(4), Number(5))))

val res = allNumbers(tree).foldLeft(Success(()): Try[Unit]) {
  case (Success(()), Number(n)) => Try {
    if (n != 3) println(n) else throw new Throwable("3 is not allowed!")
  }
  case (Failure(t), _) => Failure(t)
}

println(res)
```

```
Failure(java.lang.Throwable: 3 is not allowed!)
```

Generische AST-Traversierung

Beispiel: Generiere Folgen relevanter Knoten

Fehlerbehandlung

… mit der Traversierung in einer Hilfsklasse verpackt:

```
case class Traverse[A](coll: Traversable[A]) {
  def withAction(action: A => Unit): Try[Unit] =
    coll.foldLeft(Success(()): Try[Unit]) {
      case (Success(()), x) => Try {
        action(x)
      }
      case (Failure(t), _) => Failure(t)
    }
}
```

Ob der Nutzen generischer Traversierungs-Routinen den zusätzlichen Aufwand wert ist, muss von Fall zu Fall und individuell entschieden werden.

```
val res = Traverse(allNumbers(tree)).
  withAction {
    case Number(n) => {
      if (n != 3) println(n) else throw new Throwable("3 is not allowed!")
    }
  }

println(res)
```

```
Failure(java.lang.Throwable: 3 is not allowed!)
```