Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Prozeduren im Parser und in der Kontextanalyse

– Bezeichner-Identifikation: Prozeduren / Parameter
– Typ-Prüfungen: Parameter / Parameterübergaben

Mehrere Sorten von Symbolen

Symbole

Durch die Einführung von Prozeduren werden neue Symbole eingeführt:

- **neben den Symbolen für Variablen brauchen wir noch**
- **Prozedur-Symbole: sie repräsentieren definierte Prozeduren**

  Mit Prozedur-Symbolen werden die lokalen Definitionen einer Prozedur verbunden:
  - **lokale Variablen und**
  - **Parameter**
- **Parameter-Symbole: sie repräsentieren die Parameter, die im Kopf einer Prozedur definiert werden.**

  Parameter-Symbole werden mit den Eigenschaften eines Parameters verbunden:
  - **statischer Typ**
  - **Übergabe-Methode (Wert-Parameter, Referenz-Parameter, ...)**

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*

Beispiel: Mini-Sprache mit Prozeduren: Symbole

Symbole enthalten Leerstellen (None) für Info die nicht vom Parser berechnet wird:

- **Info zum statischen Typ**
- **Info für die Laufzeit-Organisation**

```
// all entities defined in a program (i.e. names with a compile time value) are symbols
sealed abstract class ProgSymbol {
  val name: String
}

// entities defined in a program that represent a storage location which exists at runtime
sealed abstract class LocSymbol extends ProgSymbol {
  var staticType: Option[TypeInfo] = None  // will be set by the typifier
  var rtLocInfo: Option[RTLocInfo] = None  // will be set by the runtimeLocator
}

// Variables denote runtime locations with a direct value
case class VarSymbol(override val name: String) extends LocSymbol

// Parameters denote runtime locations with a direct or an indirect value
sealed abstract class ParamSymbol extends LocSymbol

// Value Parameters denote runtime locations with a direct value
case class ValParamSymbol(override val name: String) extends ParamSymbol

// Reference Parameters denote runtime locations with an indirect value
case class RefParamSymbol(override val name: String) extends ParamSymbol

//Procedures
case class ProcSymbol(
  override val name: String,
  var params:      Option[List[ParamSymbol]] = None, // will be set by the typifier
  var locals:      Option[List[VarSymbol]] = None,   // will be set by the typifier
) extends ProgSymbol
```

Beispiel: Mini-Sprache mit Prozeduren: AST

Der Ast ist die Datenstruktur, die vom Parser aufgebaut wird.

```
// Expression that denote boolean values
sealed abstract class BoolExp extends Positional
...


// Expression that denote int values
sealed abstract class Exp extends Positional {
  var staticType: Option[TypeInfo] = None // will be set by the typifier
}
case class Number(d: Int)        extends Exp
...
case class Ref(ref: RefExp)      extends Exp


// Expression that denote references to storage locations
sealed abstract class RefExp extends Positional {
  var staticType: Option[TypeInfo] = None // will be set by typifier
}
case class LocRef(symb: LocSymbol) extends RefExp

// Expressions that denote type values
sealed abstract class TypeExp  extends Positional
case object IntTypeExp extends TypeExp
```

Beispiel: Mini-Sprache mit Prozeduren: AST

Bei den Anweisungen kommt ein Prozeduraufruf hinzu

```
// Commands
sealed abstract class Cmd extends Positional
case class Assign(left: RefExp, right: Exp) extends Cmd
case class If(e: BoolExp, thenCmds: List[Cmd], elseCmds: List[Cmd]) extends Cmd
case class While(e: BoolExp, cmds:  List[Cmd]) extends Cmd
case class Write(e:Exp) extends Cmd
case class Call(symb: ProcSymbol, args: List[Arg]) extends Cmd
```

```
case class Arg(exp: Exp, var method: Option[ParamPassMethod] = None )
sealed abstract class ParamPassMethod
case object ByValue extends ParamPassMethod
case object ByRef extends ParamPassMethod
```

Die Argumente werden per Wert- oder per Referenz-Übergabe übergeben. Die Kontextanalyse muss die Info auf Basis des formalen Parameters setzen.

Beispiel: Mini-Sprache mit Prozeduren: AST

Die Definitionen werden um Prozedur- und Parameter-Definitionen erweitert

```
// Definitions
sealed abstract class Definition extends Positional {
  type SymbType <: ProgSymbol
  val symb: SymbType
}

case class VarDef(
  override val symb: VarSymbol,
  t: TypeExp,  // the declared type
  e: Exp       // the initiaizing expression
) extends Definition {
  type SymbType = VarSymbol
}

case class ProcDef(
  override val symb: ProcSymbol,
  fparams: List[ParamDef],
  locals:  List[VarDef],
  cmds:    List[Cmd]
) extends Definition {
  type SymbType = ProcSymbol
}

sealed abstract class ParamDef extends Definition {
  type SymbType = ParamSymbol
}
```

```
case class ValueParamDef(
   override val symb: ParamSymbol,
   t: TypeExp
) extends ParamDef

case class RefParamDef (
   override val symb: ParamSymbol,
   t: TypeExp
) extends ParamDef
```

Zu jeder Definition gehört ein Symbol vom passenden Typ

Der Parser mit Bezeichner-Identifikation

muss

bei einer Definition das passende Symbol generieren

bei einer Verwendung prüfen, ob

- **ein entsprechendes Symbol definiert wurde**
- **ob das Symbol von der passenden Art ist (z.B. Prozedur-Aufrufe nicht mit Variablen)**

```
// name look-up is performed in two stages:
// - first check whether it is defined at all (there is a symbol with this name in the actual environment)
// - the whether it is of the expected kind

// look-up name, assert that it is defined
private def definedName: Parser[ProgSymbol] =
  ident ^? (
    env.lookup,
    name => s"Name '$name' is undefined"
  )

// assert that the symbol associated with a name denotes a location
private def definedLoc: Parser[LocSymbol] =
  definedName ^? (
    { case symb: LocSymbol => symb },
    name => s"Name '$name' is not a location"
  )

// assert that the symbol associated with a name denotes a procedure
private def definedProc: Parser[ProcSymbol] =
  definedName ^? (
    { case symb: ProcSymbol => symb },
    name => s"Name '$name' is not a procedure"
  )
```

Der Parser mit Bezeichner-Identifikation

```
// parse commands
private def cmd: Parser[Cmd] = positioned {
  (KwToken("IF") ~> boolExp <~ KwToken("THEN")) ~ rep(cmd) ~ (KwToken("ELSE") ~> rep(cmd) <~ KwToken("FI")) ^^ {
    case e ~ cthen ~ cElse => If(e, cthen, cElse)
  } |
  ...
  definedProc ~ (LeftPToken("(") ~> repsep(arithExp, CommaToken(",")) <~ RightPToken(")")) <~
    SemicolonToken(";") ^^ {
      case ps ~ args => Call(ps, args.map(Arg(_, None))) // method of parameter passing not yet known
    } |
  (lExp <~ AssignToken(":=")) ~ arithExp <~ SemicolonToken(";") ^^ {
      case ref ~ e => Assign(ref, e)
    }
}
```

Prozeduraufruf und Zuweisung

Der Parser mit Bezeichner-Identifikation

```
// parse definitions
// when parsing a definition, the defined name is entered into the static environment,
// which creates a symbol for the name

private def varDef: Parser[VarDef] = positioned {
  (varDefHeader <~ ColonToken(":")) ~ typeExp ~ (AssignToken(":=") ~> arithExp <~ SemicolonToken(";")) ^^ {
    case varsymb ~ t ~ e => VarDef(varsymb, t, e)
  }
}

private def procDef: Parser[ProcDef] = positioned {
  procDefHeader ~ (LeftPToken("(") ~> repsep(paramDef, CommaToken(",")) <~ RightPToken(")")) ~ rep(varDef) ~
  (KwToken("BEGIN") ~> rep(cmd) <~ KwToken("END"))  ^^ {
    case procsymb ~  paramList ~ vardefs ~ cmds =>
      println(s"Proc parsed: $procsymb, $paramList, $vardefs, $cmds")
      env.leaveScope() // leave scope of procedure (scope was entered when parsing the procedure name)
      ProcDef(procsymb, paramList, vardefs, cmds)
  }
}

private def definition: Parser[Definition] = positioned {
  varDef |
  procDef
}
```

Prozedur-Definition: Scopes nicht vergessen!

Der Parser mit Bezeichner-Identifikation

```
// parse first part (essentially the name) of a procedure definition
// name of procedure belongs to outer scope
// define proc in outer scope, then enter proc-scope
// parameters and local definitions belong to inner scope

private def procDefHeader: Parser[ProcSymbol] =
  KwToken("PROC") ~> ident ^? (
    env.defineProcedure.andThen( procSymbol => {env.enterScope(); procSymbol} ),
    { case name => s"$name is already defined" }
  )

private def varDefHeader: Parser[VarSymbol] =
  KwToken("VAR") ~> ident ^? (
    env.defineVariable,
    { case name => s"$name is already defined" }
  )

private def refParamDefHeader: Parser[RefParamSymbol] =
  KwToken("REF") ~> ident ^? (
    env.defineRefParam,
    { case name => s"$name is already defined" }
  )

private def valParamDefHeader: Parser[ParamSymbol] =
  ident ^? (
    env.defineValParam,
    { case name => s"$name is already defined" }
  )
```

Typisierung

Die Typisierung muss auf die neuen Möglichkeiten erweitert werden

- **Es gibt Prozeduren: diese müssen typisiert werden**

```
private def typifyDefs(defs: List[Definition]) : Unit = {
  defs.foreach {
    case VarDef(symb@VarSymbol(name), typeExp, e) =>
      symb.staticType = Some(evalTypeExp(typeExp))

    case ProcDef(procSymb, fparams, locals, cmds) =>
      fparams.foreach {
        case RefParamDef(paramSymb, typeExp) =>
          paramSymb.staticType = Some(evalTypeExp(typeExp))
        case ValueParamDef(varSymb, typeExp) =>
          varSymb.staticType = Some(evalTypeExp(typeExp))
      }
      locals.foreach {
        case VarDef(symb@VarSymbol(name), typeExp, e) =>
          symb.staticType = Some(evalTypeExp(typeExp))
      }
      // put static info about procedure into symbol
      procSymb.params = Some(fparams.map( _.symb ))
      procSymb.locals = Some(locals.map( _.symb) )

    case ValueParamDef(_, _) =>  /* should never match: not a toplevel definition */
    case RefParamDef(_, _) =>    /* should never match: not a toplevel definition */
  }
}
```

Den formalen Parametern wird ein Typ zugeordnet

Den lokalen Variablen wird ein Typ zugeordnet

Die Info in den Procedure-Symbolen wird vervollständigt

Typisierung

Die Typisierung muss auf die neuen Möglichkeiten erweitert werden

- **Prozeduraufrufe müssen geprüft werden**
- **Wert und Referenz-Übergaben werden unterschieden (auch an der Aufrufstelle !)**

```
private def typifyCmds(cmds: List[Cmd]): Unit = {
  cmds.foreach {
    ...
    case call@Call(symb, args) =>
      if (symb.params.get.length != args.length) {
        throw new Throwable(s"argent list does not match parameter list at ${call.pos}")
      }
      symb.params.get.zip(args).foreach {
        case (ValParamSymbol(name), arg@Arg(e, None) ) =>
          arg.method = Some(ByValue)
        case (RefParamSymbol(name), arg@Arg(e, None)) =>
          e match {
            case Ref(refExp) =>
              arg.method = Some(ByRef)
            case _ => throw new Throwable(s"Only variables may be passed by reference at ${call.pos}")
          }
      }

  }
}
```

formale Parameter und Argumente „zippen“

```
case class Arg(exp: Exp, var method: Option[ParamPassMethod] = None )
sealed abstract class ParamPassMethod
case object ByValue extends ParamPassMethod
case object ByRef extends ParamPassMethod
```

Die Argumente werden per Wert- oder per Referenz übergeben. Die Kontextanalysesetzt die Info auf Basis des formalen Parameters.

Identifikation von Bezeichnern zur Laufzeit

Die Codegenerierung benötigt

zur effizienten Adressierung von Variablen und Parametern deren

- **Verschachtelungstiefe und**
- **Distanzadresse**

Die Verschachtelungstiefe ist leicht zu berechnen, in unserem Beispiel mit flachen Prozeduren ist jeder Bezeichner auf Objekt-Ebene oder in einer Prozedur definiert.

Die Distanzadresse hängt vom Layout der Stack-Frames und dem Speicherbedarf der einzelnen Variablen ab.

Identifikation von Bezeichnern zur Laufzeit

Berechnung von Verschachtelungstiefe und Distanz-Adresse (1)

```
object RuntimeOrganisation {

  case class RTLocInfo (nesting: Int, offset: Int)

  // memory cell claim of values of of basic types
  private val addressableCellsPerInt     = 1
  private val addressableCellsPerPointer = 1


  // position of local variables and parameters within a frame:
  // parameters are located at increasing addresses starting at paramsStart
  // locals are located at decreasing addresses starting at localsStart
  private val paramsStart  = 2 // distance of parameters from Stackpointer SP
  private val localsStart  = 0 // distance of local variables from Stackpointer SP


  /**
    * Compute the number of addressable storage cells for values of a given type
    * @param typ  the type of the value
    * @return     the number of storage cells needed to store the value
    */
  private def cellClaim(typ: TypeInfo): Int = typ match {
    case IntTypeInfo    => addressableCellsPerInt
    case RefTypeInfo(_) => addressableCellsPerPointer
  }
```

Im Frame-Layout hat man sch schnell schon mal vertan.

Gut organisierte, klare Definitionen sind hilfreich.

Berechnung von Verschachtelungstiefe und Distanz-Adresse (2)

```
/**
  * assigns a frame position to all parameters and variables of the procedure.
  * @param nestingLevel  the nesting level of the procedure
  * @param symbol        the procedure's symbol
  */
def frameLayout(nestingLevel: Int, symbol: ProcSymbol): Unit = {
  var actDownOffset: Int = localsStart // keeps track of offset for local variables
  var actUpOffset: Int   = paramsStart // keeps track of offsets for parameters

  def nextUpOffset(cellClaim: Int): Int = {
    val offset = actUpOffset
    actUpOffset = actUpOffset + cellClaim
    offset
  }

  def nextDownOffset(cellClaim: Int): Int = {
    val offset = actDownOffset
    actDownOffset = actDownOffset - cellClaim
    offset
  }

  symbol.params match {
    case None => throw new Exception(s"internal errror, uninitilized parameter symbols of proc $symbol")
    case Some(paramSymbols) =>
      paramSymbols.foreach((ps: ParamSymbol) => {
        val offset = nextUpOffset(cellClaim(ps.staticType.get))
        ps.rtLocInfo = Some(RTLocInfo(nestingLevel, offset))
      })
  }

  symbol.locals match {
    case None => throw new Exception(s"internal errror, uninitilized parameter symbols of proc $symbol")
    case Some(varSymbols) =>
      varSymbols.foreach ( (vs: VarSymbol) => {
        val offset = nextDownOffset(cellClaim(vs.staticType.get))
        vs.rtLocInfo = Some(RTLocInfo(nestingLevel, offset))
      })
  }
}
```