Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Code für Prozeduren und Objekte

– Zwischencode für Prozeduren
– Zwischencode für Objekte

Zwischencode für Prozeduren

Prozeduren und ihr Aufruf im Zwischencode

Die Übersetzung von Prozeduren und ihre Aufrufe hängt von der Zielmaschine ab.
Bei der Generierung von Zwischencode ist eine hypothetische Maschine die Zielmaschine
Diese Zielmaschine kann beliebig dicht an der realen Maschine sein.

Maschinenmodell im Zwischencode

- **Variante A: Die Hypothetische Maschine des Zwischencodes beherrscht Prozeduren**
  Der Codegenerator für den Zwischencode erzeugt
  aus einem Prozeduraufruf der Quellsprache
    - **einen Prozeduraufruf**
  des Zwischencodes

- **Variante B: Die Hypothetische Maschine des Zwischencodes kennt keine Prozeduren**
  Der Codegenerator für den Zwischencode erzeugt
  aus einem Prozeduraufruf der Quellsprache
    - **Sprunganweisungen und**
    - **Stack-Operationen**
  des Zwischencodes

Zwischencode für Prozeduren

Prozeduren und ihr Aufruf im Zwischencode

Variante B: Hypothetische Maschine kennt keine Prozeduren

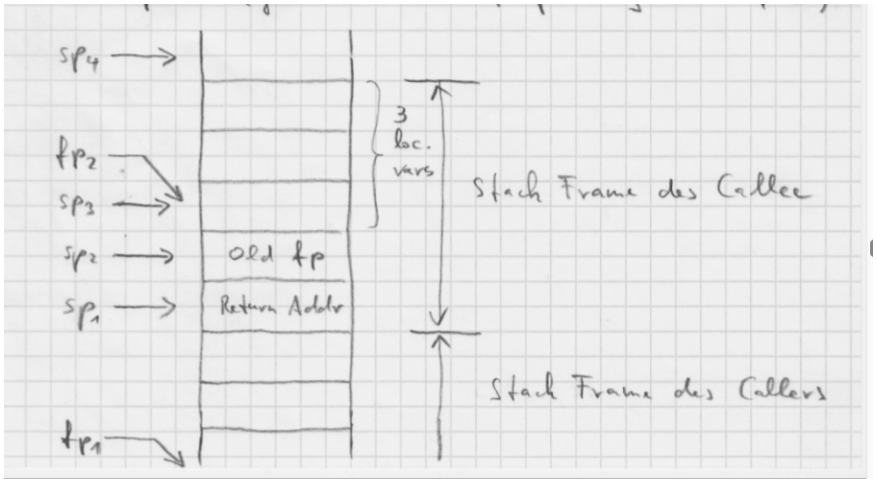
Der Zwischencode für Prozedur-Definition und -Aufruf ist komplexer,

Der Zwischencode ist näher an der Zielmaschine, wenn diese Prozeduren nicht direkt unterstützt.

Implementierung von Prozeduren ohne direkte Unterstützung von Prozeduren:

Call-Stack mit Frames / Activation-Records

Siehe Foliensatz 12 sowie Unterlagen zu KSP:



Graphik aus
https://homepages.thm.de/~hg53/ksp-ws1718/manuskript.pdf (Seite 28)

Prozeduren und ihr Aufruf im Zwischencode

Zwischencode für hypothetische (Call-) Stack-Maschine

Prozeduren werden mit einem Call-Stack implementiert

Der Stack wird mit unterschiedlichen Maschinen-abhängigen Befehlen manipuliert

Die prinzipiellen Operationen sind aber unabhängig von der konkreten Maschine

Sie können mit generischen Stack-Operationen ausgedrückt werden

Hypothetische Call-Stack-Maschine

Die hypothetische Maschine des Zwischencodes hat

- einen Stack
- zwei ausgezeichnete Register
  - SP: Stack-Pointer — zeigt auf das Stack-Ende
  - FP: Frame-Pointer — zeigt auf einen Rahmen auf den Stack (Anker-Position im Rahmen)

Sie unterstützt folgende Operationen:

- Manipulation von SP, unf FP
  - Push und Pop Operationen auf dem Stack (mit indirekter Manipulation des SP)
  - Direkte Manipulationen von SP und FP
- Sprünge zu gespeicherten Codeadressen
  Für den Rücksprung zur Aufrufstelle einer Prozedur muss ein Zugriff auf Code-Adressen möglich sein.

Zwischencode für Prozeduren

Für die Übersetzung einer Sprache mit Prozeduren in Stack-orientierten Zwischencode müssen dessen Möglichkeiten erweitert werden:

- **Code-Adressen als maschinennaher Datentyp**
- **Stack als Datenstruktur**

Zwischencode für Prozeduren

Code-Adressen als maschinennaher Datentyp

Bei der Übersetzung von Kontrollstrukturen werden Code-Adressen durch Labels repräsentiert. Das Ziel jedes Sprungs kann zur Compilezeit festgelegt werden.

Das gilt nicht mehr bei Prozeduren:

- **Der Einstieg in eine Prozedur ist zwar zur Compilezeit bekannt: Ein Aufruf geht zu einer festen Sprungmarke**
- **Der Ausstieg ist aber nicht zur Compilezeit bekannt: Der Rücksprung geht zur Aufrufstelle und die ist erst zur Laufzeit bekannt.**

Konsequenz:

- **die (hypothetische Zwischencode-) Maschine muss Code-Adressen speichern und manipulieren können. D.h.:**
- **neben den Adressen von Int-Speicherplätzen müssen Code-Adressen als Typ der Zwischensprache unterstützt werden**

Zwischencode für Prozeduren

Call-Stack und Register im Zwischencode

Prozeduren werden über den Call-Stack realisiert. Die (hypothetische Zwischencode-) Maschine unterstützt dies durch einige vordefinierte Operationen und Speicherstellen:

- **Stackpointer SP**

  Speicherstelle (Register), enthält die Adresse des Stack-Endes

  Der Stackpointer SP wird im wesentlichen implizit durch Push- und Pop-Operationen manipuliert

- **Framepointer FP**

  Speicherstelle (Register), enthält die Anker-Adresse eines Stack-Frames,

  d.h. die Adresse relativ zu der Parameter und lokale Variable adressiert werden.

- **Return Register RR**

  Speicherstelle (Register), enthält die Adresse zu der mit der Return-Anweisung gesprungen wird.

Call-Stack im Zwischencode

Struktur des Call-Stacks

Der Call-Stack kann alle Arten von Daten aufnehmen, die in der Maschinen-Sprache eine Rolle spielen. Beispielsweise:

- **Int-Werte**
- **Adressen von Speicherstellen im Stack oder im statischen Datenbereich**
- **Adressen von Maschinencode**

Der Call-Stack enthält also Werte von unterschiedlichem Typ, wobei der Typ des Wertes an einer bestimmten Stack-Position regelmäßig wechseln kann.

Der Typ eines Wertes bestimmt

- **die Interpretation des Bitmusters, das ihn repräsentiert**
- **die Zahl der Speicherstellen, die die er belegt.**

Die Interpretation des Bitmusters hängt von den verwendeten Operationen ab, diese können zur Compilezeit mit der richtigen „Buchführung“ vorherbestimmt werden.

Die Zahl der für einen Wert benötigten Speicherstellen im Stack kann ebenfalls zur Compilezeit vorherbestimmt werden.

Der Stack muss also nur die Möglichkeit bieten, dass seine Speicherstellen flexibel interpretiert werden können.

Der Call-Stack gehört zu der hypothetischen Maschine, die den Zwischencode ausführt.

Hypothetische Maschinen haben keine Implementierung. Es darum nicht notwendig, über eine konkrete Realisation nachzudenken. – Es sei denn, der Zwischencode wird interpretiert und die hypothetische wird damit zu einer virtuellen Maschine.

Erweiterungen des Zwischencodes

Stack-Operationen im Zwischencode

```
// push MInt value on stack
case class PushMIntInstr(t: MIntLocOrValue)  extends Instr

// push Address value on stack
case class PushMAddressInstr(a: MAddressLoc) extends Instr

// push code address on stack
case class PushCodeAddrInstr(returnLabel: String) extends

// push frame pointer on stack
case object PushFPInstr                      extends Instr

// pop MInt value from stack
case object PopMIntInstr                     extends Instr

// pop address value from stack
case object PopMAddressInstr                 extends Instr

// pop code address from stack and store it in register RR
case object PopCodeAddrToRRInstr             extends Instr

// pop frame pointer from stack to register FP
case object PopFPInstr                       extends Instr

// copy SP to FP
case object StoreSPasFPInstr                 extends Instr
```

Erweiterungen des Zwischencodes

Prozduraufruf und Rücksprung im Zwischencode

```
// call instruction – essentially a junp to the label
case class CallInstr(callLabel: String) extends Instr

// return instruction – essentially a jump to the address in register RR
case object ReturnInstr extends Instr
```

Prozeduren und ihr Aufruf im Zwischencode

Beispiel: Prozedur und Prozeduraufruf im Zwischencode

```
OBJECT
    VAR a: INT := 1;
    VAR b: INT := 2;

    PROC p(REF x: INT, y: INT)
    BEGIN
       x := y;
    END

    PROC q()
    BEGIN
      p(a,b);
    END
INIT
END
```

```
proc_p: NOOP
    FP = SP
    a_0 = ADDR[1, 2]
    *[a_0] = INT[1, 3]
    RR = POP_CODE_ADDR
    JUMP RR
```

Beginn einer Prozedur: Setze FP auf den aktuellen SP

x := y;
x und y liegen im Parameterbereich an offset 2 und 3; x ist REF-Parameter

Return-Adress vom Stack holen und Sprung zu dieser Adresse

```
proc_q: NOOP
    FP = SP
    PUSH_FP
    a_0 = &[INT[0, 0]]
    PUSH_ADDR a_0
    t_0 = INT[0, -1]
    PUSH_INT t_0
    PUSH_CODE_ADDR L_1
    JUMP proc_p
L_1: NOOP
    POP_INT
    POP_ADDR
    FP = POP_FP
    RR = POP_CODE_ADDR
    JUMP RR
```

Beginn einer Prozedur: Setze FP auf den aktuellen SP

Aufruf von p:
- *FP im Stack speichern*
- *Argumente berechnen und auf Stack pushen*
- *Rücksprung-Adresse pushen*
- *Zur Prozedur springen*

Aufräumen nach call von q:
- *Die Argumente vom Stack entfernen*
- *FP wiederherstellen*

Rückkehr aus q

Die Instruktionen des Zwischencodes mit werden zur besseren Lesbarkeit mit geeigneten toString-Methoden versehen.

Zwischencode für Objekte

Objekte

Bestandteile eines Objekts:

- **Statische Variablen**

  In einem Objekt können Variablen definiert werden, die in einem statischen Datenbereich liegen – also nicht auf den Stack

- **Initialisierungscode**

  Objekte enthalten Initialisierungscode:

  - Die Initialisierung der globalen Variablen
  - Die Anweisungen in der Initialisierungs-Sektion den Objekts

Beispiel:

```
OBJECT
    VAR a: INT := 1;
    VAR b: INT := 2;        globale Variablen mit ihren Initialisierungen

    PROC p(REF x: INT, y: INT)
    BEGIN
       x := y;
    END

    PROC q()
       VAR a: INT := b+3;
    BEGIN
      p(a,a+b);
    END

INIT
    a := a+b;               Initialisierungs-Sektion
END
```

Code für Objekte

Der Code eines Objekts umfasst:

- **Code für jede im Objekt definierte globale Variable**
  Variablen, die nicht im Stack existieren werden im Codebereich angelegt
- **Code für jede im Objekt definierte Prozedur**
  Jede lokale Prozedur wird in eine Prozedur des Zwischen- / Maschinen-Codes übersetzt
- **Pseudo-Prozedur zur Initialisierung**
  Die Initialisungen werden in einer eigenen „künstlichen“ / „Pseudo-“ Prozedur zusammengefasst

```
OBJECT
    VAR a: INT := 1;
    VAR b: INT := 2;

    PROC p(REF x: INT, y: INT)
    BEGIN
       x := y;
    END

    PROC q()
       VAR a: INT := b+3;
    BEGIN
      p(a,a+b);
    END

INIT
    a := a+b;
END
```

globale Variablen

Initialisierungsprozedur

Zwischencode für Objekte

Zwischencode für statische Variable

Statische Variablen werden mit einer Anweisung des Zwischencodes erzeugt,

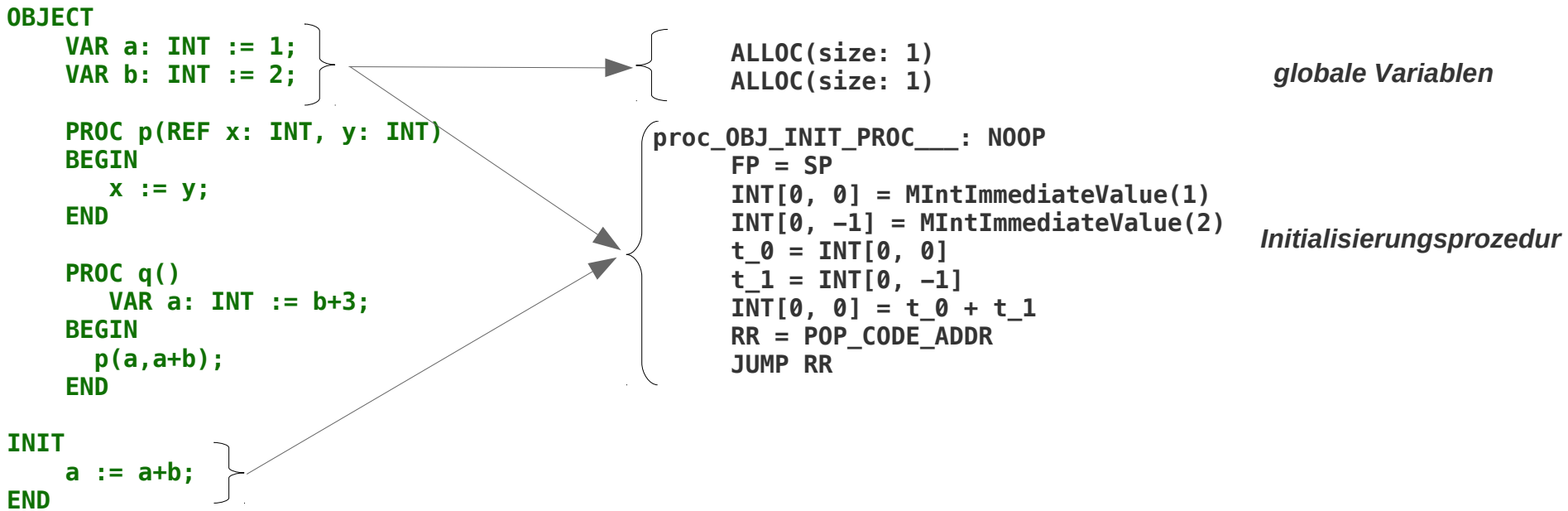
zusätzliche Erweiterungen des Zwischencodes sind nicht notwendig

```
//-----------------------------------------------------
// Instructions that allocate static data areas

// allocate satic storage with size storage cells
case class AllocStaticInstr(size: Int)  extends Instr
```

Zwischencode für die Initialisierungsprozedur

Beispiel:



Zwischencode für Objekte

Zwischencode für die Initialisierungsprozedur

```
// translate initialisation of global variables
// and init section to init procedure
def genInitProc(obj: Obj): Unit = {

  // create pseudo proc
  val pseudoProc = ProcDef(
    ProcSymbol(
      "OBJ_INIT_PROC___", // name
      Some(Nil),          // parameters
      Some(Nil)),         // locals
    Nil,                  // parameters
    obj.defs.flatMap{     // locals
      case v:VarDef => List(v)
      case _ => Nil
    },
    obj.cmds
  )

  // translate pseudo proc
  genCodeProc(pseudoProc)
}
```

Zwischencode für Objekte

Objekte insgesamt werden übersetzt beispielsweise mit:

```
// translate definition of global variables
obj.defs foreach {
  case varDef@VarDef(_, _, _) =>
    genCodeGlobVarDef(varDef)
  case _ =>
}

// translate initialisation of global variables
// and init section to init procedure
genInitProc(obj)


// translate procedure definitions
obj.defs foreach {
  case procDef@ProcDef(_, _, _, _) =>
    genCodeProc(procDef)
  case _ =>
}

codeBuf.toList
```

Zwischencode für Prozeduren

Übersetzung einer Prozedur – 1

Prozeduren können jetzt auch Pseudo-Prozeduren sein. D.h. Prozeduren mit lokalen Variablen, die globale Variablen sind. Für diese wird natürlich kein Platz auf den Stack reserviert.

```
// generate code for a procedure
private def genCodeProc(procDef: ProcDef): Unit = {
  val procLabel = procSymbToLabel(procDef.symb)
  codeBuf += LabeledInstr(procLabel)

  // store SP as new FP
  codeBuf += StoreSPasFPInstr

  // allocate non-static locals on stack
  procDef.locals.foreach {
    case VarDef(symb, _, _) =>
      if (symb.rtLocInfo.get.nesting > 0 ) { // deal with stack variables only
        symb.staticType match {
          case Some(IntTypeInfo) =>
            codeBuf += PushMIntInstr(MIntImmediateValue(0))
          case _ => throw new Exception("internal error language supports only int variables")
        }
      }
  }

  // fill locals with init values
  procDef.locals.foreach {
    case VarDef(symb, _, initExp) =>
      val varLoc = MIntFrameLoc(symb.rtLocInfo.get)
      genCodeValExp(initExp, varLoc)
    case _ => // ignore
  }
```

Zwischencode für Prozeduren

Übersetzung einer Prozedur – 2

```
  // code for body
  procDef.cmds.foreach( cmd => genCodeCmd(cmd) )

  // release locals on stack
  procDef.locals.reverse.foreach {
    case VarDef(symb, _, _) =>
      if (symb.rtLocInfo.get.nesting > 0 ) { // deal with stack variables only
        symb.staticType match {
          case Some(IntTypeInfo) =>
            codeBuf += PopMIntInstr
          case _ => throw new Exception("internal error: language supports only int variables")
        }
      }
  }

  // set RR from stack
  codeBuf += PopCodeAddrToRRInstr

  // return to caller
  codeBuf += ReturnInstr

}
```