Albert Ludwigs Universität Freiburg

Technische Fakultät

PicoC-Compiler

Übersetzung einer Untermenge von C in den Befehlssatz der RETI-CPU

Bachelorarbeit

Abgabedatum: 28th April 2022

Author:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Inhaltsverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

Grammatikverzeichnis

1 Motivation

1.1 RETI

... basiert auf ... der Vorlesung C. Scholl, „Betriebssysteme“.

Definition 1.1: Caller-save Register

a

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 1.2: Callee-save Register

a

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

1.2 PicoC

1.3 Aufgabenstellung

1.4 Eigenheiten der Sprache C

Definition 1.3: Deklaration

a

[a] P. Scholl, „Einführung in Embedded Systems“.

Definition 1.4: Definition

a

[a] P. Scholl, „Einführung in Embedded Systems“.

Definition 1.5: Allokation

a

[a] Thiemann, „Einführung in die Programmierung“.

Definition 1.6: Initialisierung

[a]

[a]Thiemann, „Einführung in die Programmierung“.

Definition 1.7: Scope

[a]

[a]Thiemann, „Einführung in die Programmierung“.

Definition 1.8: Call by value

[a]

[a]Bast, „Programmieren in C“.

Definition 1.9: Call by reference

[a]

[a]Bast, „Programmieren in C“.

1.5 Richtlinien

1.5.1 Umsetzung von Funktionen

1.5.1.1 Mehrere Funktionen

Die Umsetzung **mehrerer Funktionen** wird im Folgenden mithilfe des Beispiels in Code 1.1 erklärt. Dieses Beispiel soll nur zeigen, wie Funktionen in verschiedenen, für die Kompilierung von Funktionen relevanten **Passes** kompiliert werden. Das Beispiel ist so gewählt, dass es möglichst **isoliert** von weiterem möglicherweise störendem Code ist.

```
1  void main() {
2    return;
3  }
4
5  void fun1() {
6  }
7
8  int fun2() {
9    return 1;
10 }
```

Code 1.1: PicoC-Code für 3 Funktionen

Im **Abstract Syntax Tree** in Code 1.2 wird eine **Funktion**, wie z.B. `voidfun(intparam;){ returnparam ; }` mit der Komposition `FunDef(IntType(), Name('fun'), [Alloc(Writeable(), IntType(), Name('fun'))], [Return(Exp(Name('param')))])` dargestellt. Die einzelnen **Attribute** dieses Container-Knoten sind in Tabelle **??** erklärt.

```
1  File
2    Name './verbose_3_funs.ast',
3    [
4      FunDef
5        VoidType 'void',
6        Name 'main',
7        [],
8        [
9          Return
10           Empty
11       ],
12     FunDef
13       VoidType 'void',
14       Name 'fun1',
15       [],
16       [],
17     FunDef
18       IntType 'int',
19       Name 'fun2',
20       [],
21       [
22         Return
23           Num '1'
24       ]
25   ]
```

Code 1.2: Abstract Syntax Tree für 3 Funktionen

Im **PicoC-Blocks Pass** in Code 1.3 werden die **Statements** der Funktion in **Blöcke** `Block(name, stmts_instrs)` aufgeteilt. Dabei bekommt ein Block `Block(name, stmts_instrs)`, der die Statements der Funktion vom **Anfang** bis zum **Ende** oder bis zum Auftauchen eines `If(exp, stmts)`, `IfElse(exp, stmts1, stmts2)`, `While(exp, stmts)` oder `DoWhile(exp, stmts)`[1] beinhaltet den **Bezeichner** bzw. den `Name(str)`-Token-Knoten der Funktion an sein **Label** bzw. an sein `name`-Attribut zugewiesen. Dem **Bezeichner** wird vor der Zuweisung allerdings noch eine **Nummer** angehängt `<name>.<nummer>`[2].

Es werden parallel dazu neue Zuordnungen im **Dictionary** `fun_name_to_block_name` hinzugefügt. Das **Dicionary** ordnet einem **Funktionsnamen** den **Blocknamen** des Blockes, der das erste **Statement** der Funktion enthält und dessen **Bezeichner** `<name>.<nummer>` bis auf die angehängte **Nummer** identisch zu dem der Funktion ist zu[3]. Diese Zuordnung ist nötig, da **Blöcke** noch eine **Nummer** an ihren Bezeichner `<name>.<nummer>` angehängt haben.

```
File
 Name './verbose_3_funs.picoc_blocks',
 [
   FunDef
     VoidType 'void',
     Name 'main',
     [],
     [
       Block
         Name 'main.2',
         [
           Return(Empty())
         ]
     ],
   FunDef
     VoidType 'void',
     Name 'fun1',
     [],
     [
       Block
         Name 'fun1.1',
         []
     ],
   FunDef
     IntType 'int',
     Name 'fun2',
     [],
     [
       Block
         Name 'fun2.0',
         [
           Return(Num('1'))
         ]
     ]
 ]
```

[1] Eine Erklärung dazu ist in Unterkapitel **??** zu finden.
[2] Der **Grund** dafür kann im Unterkapitel **??** nachgelesen werden.
[3] Das ist der **Block**, über den im **obigen letzten Paragraph** gesprochen wurde.

Code 1.3: PicoC-Blocks Pass für 3 Funktionen

Im **PicoC-Mon Pass** in Code 1.4 werden die `FunDef(datatype, name, allocs, stmts)`-Container-Knoten komplett aufgelöst, sodass sich im `File(name, decls_defs_blocks)`-Container-Knoten nur noch Blöcke befinden.

```
File
  Name './verbose_3_funs.picoc_mon',
  [
    Block
      Name 'main.2',
      [
        Return(Empty())
      ],
    Block
      Name 'fun1.1',
      [
        Return(Empty())
      ],
    Block
      Name 'fun2.0',
      [
        // Return(Num('1'))
        Exp(Num('1'))
        Return(Stack(Num('1')))
      ]
  ]
```

Code 1.4: PicoC-Mon Pass für 3 Funktionen

Nach dem **RETI Pass** in Code 1.5 gibt es nur noch **RETI-Instructions**, die Blöcke wurden entfernt und die **RETI-Instructions** in diesen Blöcken wurden genauso zusammengefügt, wie die Blöcke angeordnet waren. Ohne die **Kommentare** könnte man die Funktionen nicht mehr direkt ausmachen, denn die **Kommentare** enthalten die **Labelbezeichner** `<name>.<nummer>` der Blöcke, die in diesem Beispiel immer zugleich bis auf die Nummer, dem **Namen** der jeweiligen **Funktion** entsprechen.

Da es in der `main`-Funktion keinen **Funktionsaufruf** gab, wird der Code, der nach der **Instruction** in der **markierten Zeile** kommt nicht mehr betreten. Funktionen sind im **RETI-Code** nur dadurch existent, dass im RETI-Code **Sprünge** (z.B. `JUMP<rel> <im>`) zu den jeweils richtigen Positionen gemacht werden, nämlich dorthin, wo die **RETI-Instructions**, die aus den **Statemtens** einer **Funktion** kompiliert wurden anfangen.

```
# // Block(Name('start.3'), [])
# // Exp(GoTo(Name('main.2')))
# // not included Exp(GoTo(Name('main.2')))
# // Block(Name('main.2'), [])
# Return(Empty())
LOADIN BAF PC -1;
# // Block(Name('fun1.1'), [])
# Return(Empty())
```

```
LOADIN BAF PC -1;
# // Block(Name('fun2.0'), [])
# // Return(Num('1'))
# Exp(Num('1'))
SUBI SP 1;
LOADI ACC 1;
STOREIN SP ACC 1;
# Return(Stack(Num('1')))
LOADIN SP ACC 1;
ADDI SP 1;
LOADIN BAF PC -1;
```

Code 1.5: RETI-Blocks Pass für 3 Funktionen

1.5.1.1.1 Sprung zur Main Funktion

Im vorherigen Beispiel in Code 1.1 war die `main`-Funktion die **erste** Funktion, die im Code vorkam. Dadurch konnte die `main`-Funktion direkt betreten werden, da die **Ausführung** des Programmes immer ganz vorne im **RETI-Code** beginnt. Man musste sich daher keine Gedanken darum machen, wie man die **Ausführung**, die von der `main`-Funktion ausgeht überhaupt startet.

Im Beispiel in Code 1.6 ist die `main`-Funktion allerdings **nicht** die **erste** Funktion. Daher muss dafür gesorgt werden, dass die `main`-Funktion die erste Funktion ist, die ausgeführt wird.

```
void fun1() {
}

int fun2() {
  return 1;
}

void main() {
  return;
}
```

Code 1.6: PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist

Im **RETI-Blocks Pass** in Code 1.7 sind die **Funktionen** nur noch durch **Blöcke** umgesetzt.

```
File
  Name './verbose_3_funs_main.reti_blocks',
  [
    Block
      Name 'fun1.2',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun2.1',
```

```
12      [
13        # // Return(Num('1'))
14        # Exp(Num('1'))
15        SUBI SP 1;
16        LOADI ACC 1;
17        STOREIN SP ACC 1;
18        # Return(Stack(Num('1')))
19        LOADIN SP ACC 1;
20        ADDI SP 1;
21        LOADIN BAF PC -1;
22      ],
23    Block
24      Name 'main.0',
25      [
26        # Return(Empty())
27        LOADIN BAF PC -1;
28      ]
29  ]
```

Code 1.7: RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

Eine simple Möglichkeit ist es, die `main-Funktion` einfach nach **vorne** zu schieben, damit diese als **erstes** ausgeführt wird. Im `File(name, decls_defs)`-Container-Knoten muss dazu im `decls_defs`-Attribut, welches eine **Liste von Funktionen** ist, die `main`-Funktion an Index 0 geschoben werden.

Eine andere Möglichkeit und die Möglichkeit für die sich in der **Implementierung** des **PicoC-Compilers** entschieden wurde, ist es, wenn die `main`-Funktion nicht die erste auftauchende Funktion ist, einen `start.<nummer>`-Block als ersten Block einzufügen, der einen `GoTo(Name('main.<nummer>'))`-Container-Knoten enthält, der im **RETI Pass** 1.9 in einen Sprung zur `main`-Funktion übersetzt wird.

In der Implementierung des **PicoC-Compilers** wurde sich für diese Möglichkeit entschieden, da es für **Studenten**, welche die Verwender des **PiocC-Compilers** sein werden vermutlich am **intuitivsten** ist, wenn der **RETI-Code** für die Funktionen an denselben Stellen relativ zueinander verortet ist, wie die Funktionsdefinitionen im **PicoC-Code**.

Das **Einsetzen** des `start.<nummer>`-Blockes erfolgt im **RETI-Patch Pass** in Code 1.8, da der **RETI-Patch**-Pass der Pass ist, der für das **Ausbessern** (engl. to patch) zuständig ist, wenn z.B. in manchen Fällen die `main`-Funktion nicht die erste Funktion ist.

```
1 File
2   Name './verbose_3_funs_main.reti_patch',
3   [
4     Block
5       Name 'start.3',
6       [
7         # // Exp(GoTo(Name('main.0')))
8         Exp(GoTo(Name('main.0')))
9       ],
10    Block
11      Name 'fun1.2',
12      [
13        # Return(Empty())
14        LOADIN BAF PC -1;
```

```
      ],
    Block
      Name 'fun2.1',
      [
        # // Return(Num('1'))
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Return(Stack(Num('1')))
        LOADIN SP ACC 1;
        ADDI SP 1;
        LOADIN BAF PC -1;
      ],
    Block
      Name 'main.0',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

Code 1.8: RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

Im **RETI Pass** in Code 1.9 wird das `GoTo(Name('main.<nummer>'))` durch den entsprechenden `Sprung JUMP <distanz_zur_main_funktion>` ersetzt und die Blöcke entfernt.

```
# // Block(Name('start.3'), [])
# // Exp(GoTo(Name('main.0')))
JUMP 8;
# // Block(Name('fun1.2'), [])
# Return(Empty())
LOADIN BAF PC -1;
# // Block(Name('fun2.1'), [])
# // Return(Num('1'))
# Exp(Num('1'))
SUBI SP 1;
LOADI ACC 1;
STOREIN SP ACC 1;
# Return(Stack(Num('1')))
LOADIN SP ACC 1;
ADDI SP 1;
LOADIN BAF PC -1;
# // Block(Name('main.0'), [])
# Return(Empty())
LOADIN BAF PC -1;
```

Code 1.9: RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

1.5.1.2 Funktionsdeklaration und -definition und Umsetzung von Scopes

In der Programmiersprache L_C und somit auch L_{PicoC} ist es notwendig, dass eine Funktion **deklariert** ist, bevor man einen **Funktionsaufruf** zu dieser Funktion machen kann. Das ist notwendig, damit **Fehler-**

meldungen ausgegeben werden können, wenn der **Prototyp** (Definition 1.10) der Funktion nicht mit den **Datentypen** der **Argumente** oder der **Anzahl Argumente** übereinstimmt, die beim **Funktionsaufruf** an die Funktion in einer **festen** Reihenfolge übergeben werden.

Die Dekleration einer Funktion kann explizit erfolgen (z.B. `int fun2(int var);`), wie in der im Beispiel in Code 1.10 **markierten Zeile** `1` oder zusammen mit der **Funktionsdefinition** (z.B. `void fun1(){}`), wie in den **markierten Zeilen** `3-4`.

In dem Beispiel in Code 1.10 erfolgt ein **Funktionsaufruf** zur Funktion `fun2`, die allerdings erst nach der `main`-Funktion definiert ist. Daher ist eine **Funktionsdekleration**, wie in der **markierten Zeile** `1` notwendig. Beim **Funktionsaufruf** zur Funktion `fun1` ist das **nicht** notwendig, da die Funktion vorher **definiert** wurde, wie in den **markierten Zeilen** `3-4` zu sehen ist.

Definition 1.10: Funktionsprototyp

***Deklaration** einer Funktion, welche den **Funktionsbezeichner**, die **Datentypen** der einzelnen **Funktionsparameter**, die **Parametereihenfolge** und den **Rückgabewert** einer Funktion spezifiziert. Es ist **nicht** möglich zwei Funktiondeklarationen mit dem **gleichen** Funktionsbezeichner zu haben.*[a] [b]

[a] Der **Funktionsprototyp** ist von der **Funktionsignatur** zu unterschieden, die in Programmiersprache wie `C++` und `Java` für die **Auflösung** von **Überladung** bei z.B. **Methoden** verwendet wird und sich in manchen Sprachen für den **Rückgabewert** interessiert und in manchen nicht, je nach Umsetzung. In solchen Sprachen ist es möglich mehrere **Methoden** oder **Funktionen** mit dem **gleichen** Bezeichner zu haben, solange sie sich durch die **Datentpyen** von **Parametern**, die **Parameterreihenfolge**, manchmal auch **Scopes** und **Klassentpyen** usw. unterschieden.

[b] *What is the difference between function prototype and function signature?*

```
1  int fun2(int var);
2
3  void fun1() {
4  }
5
6  void main() {
7    int var = fun2(42);
8    fun1();
9    return;
10 }
11
12 int fun2(int var) {
13   return var;
14 }
```

Code 1.10: PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss

Die **Deklaration** einer **Funktion** erfolgt mithilfe der **Symboltabelle**, die in Code 1.11 für das Beispiel in Code 1.10 dargestellt ist. Die **Attribute** des **Symbols** `Symbols(type_qual, datatype, name, val_addr, pos, size)` werden wie üblich gesetzt. Dem `datatype`-Attribut wird dabei einfach die komplette Komposition der **Funktionsdeklaration** `FunDecl(IntType('int'), Name('fun2'), [Alloc(Writeable(), IntType('int'), Name('var'))])` zugewiesen.

Die Varaiblen `var@main` und `var@fun2` der `main`-Funktion und der Funktion `fun2` haben unterschiedliche **Scopes** (Definition 1.11). Die **Scopes** der **Funktionen** werden mittels eines **Suffix** `"@<fun_name>"` umgesetzt, der an den **Bezeichner** `var` drangehängt wird: `var@<fun_name>`. Dieser **Suffix** wird geändert sobald beim **Top-Down**[4] Durchiterieren über den **Abstract Syntax Tree** des aktuellen **Passes** ein **Funktionswechsel** eintritt und

[4] D.h. von der **Wurzel** zu den **Blättern** eines Baumes.

über die Statements der nächsten Funktion iteriert wird, für die der **Suffix** der neuen Funktion `FunDef(name, datatype, params, stmts)` angehängt wird, der aus dem `name`-Attribut entnommen wird.

Ein Grund, warum **Scopes** über das Anhängen eines **Suffix** an den **Bezeichner** gelöst sind, ist, dass auf diese Weise die **Schlüssel**, die aus dem **Bezeichner** einer Variable und einem angehängten **Suffix** bestehen, in der als **Dictionary** umgesetzten **Symboltabelle** eindeutig sind. Damit man einer Variable direkt den **Scope** ablesen kann in dem sie definiert wurde, ist der **Suffix** ebenfalls im `Name(str)`-Token-Knoten des `name`-Attribubtes eines **Symbols** der Symboltabelle angehängt. Zur beseren Vorstellung ist dies ist in Code 1.11 **markiert**.

Die Variable `var@main`, bei der es sich um eine **Lokale Variable** der `main`-Funktion handelt, ist nur innerhalb des **Codeblocks** {} der `main`-Funktion **sichtbar** und die Variable `var@fun2` bei der es sich im einen **Parameter** handelt, ist nur innerhalb des **Codeblocks** {} der Funktion `fun2` **sichtbar**. Das ist dadurch umgesetzt, dass der **Suffix**, der bei jedem **Funktionswechsel** angepasst wird, auch beim Nachschlagen eines **Symbols** in der **Symboltabelle** an den **Bezeichner** der Variablen, die man nachschlagen will angehängt wird. Und da die Zuordnungen im **Dictionary eindeutig** sind, kann eine Variable nur in genau der Funktion nachgeschlagen werden, in der sie **definiert** wurde.

Das Zeichen `'@'` wurde aus einem bestimmten Grund als **Trennzeichen** verwendet, nämlich, weil kein Bezeichner das Zeichen `'@'` jemals selbst enthalten kann. Damit ist ausgeschlossen, dass falls ein **Benutzer** des **PicoC-Compilers** zufällig auf die Idee kommt seine Funktion genauso zu nennen (z.B. `var@fun2` als Funktionsname), es zu Problemen kommt, weil bei einem Nachschlagen der **Variable** die **Funktion** nachgeschlagen wird.

Definition 1.11: Scope (bzw. Sichtbarkeitsbereich)

***Bereich** in einem Programm, in dem eine Variable **sichtbar** ist und **verwendet** werden kann.*[a]

[a] Thiemann, „Einführung in die Programmierung“.

```
1  SymbolTable
2    [
3      Symbol
4        {
5          type qualifier:         Empty()
6          datatype:               FunDecl(IntType('int'), Name('fun2'), [Alloc(Writeable(),
           ↪  IntType('int'), Name('var'))])
7          name:                   Name('fun2')
8          value or address:       Empty()
9          position:               Pos(Num('1'), Num('4'))
10         size:                   Empty()
11       },
12     Symbol
13       {
14         type qualifier:         Empty()
15         datatype:               FunDecl(VoidType('void'), Name('fun1'), [])
16         name:                   Name('fun1')
17         value or address:       Empty()
18         position:               Pos(Num('3'), Num('5'))
19         size:                   Empty()
20       },
21     Symbol
22       {
23         type qualifier:         Empty()
24         datatype:               FunDecl(VoidType('void'), Name('main'), [])
```

```
25        name:                   Name('main')
26        value or address:       Empty()
27        position:               Pos(Num('6'), Num('5'))
28        size:                   Empty()
29      },
30    Symbol
31      {
32        type qualifier:         Writeable()
33        datatype:               IntType('int')
34        name:                   Name('var@main')
35        value or address:       Num('0')
36        position:               Pos(Num('7'), Num('6'))
37        size:                   Num('1')
38      },
39    Symbol
40      {
41        type qualifier:         Writeable()
42        datatype:               IntType('int')
43        name:                   Name('var@fun2')
44        value or address:       Num('0')
45        position:               Pos(Num('12'), Num('13'))
46        size:                   Num('1')
47      }
48  ]
```

Code 1.11: Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss

1.5.1.3 Funktionsaufruf

Ein **Funktionsaufruf** (z.B. `stack_fun(local_var)`) wird im Folgenden mithilfe des Beispiels in Code 1.12 erklärt. Das Beispiel ist so gewählt, dass alleinig der **Funktionsaufruf** im **Vordergrund** steht und dieses Kapitel nicht auch noch mit z.B. Aspekten wie der Umsetzung eines **Rückgabewertes** überladen ist. Der Aspekt der Umsetzung eines **Rückgabewertes** wird erst im nächsten Unterkapitel 1.5.1.3.1 erklärt.

```
1  struct st {int attr1; int attr2[2];};
2
3  void stack_fun(struct st param[2][3]);
4
5  void main() {
6    struct st local_var[2][3];
7    stack_fun(local_var);
8    return;
9  }
10
11 void stack_fun(struct st param[2][3]) {
12   int local_var;
13 }
```

Code 1.12: PicoC-Code für Funktionsaufruf ohne Rückgabewert

Im **Abstract Syntax Tree** in Code 1.13 wird ein **Funktionsaufruf** `stack_fun(local_var)` durch die **Komposition** `Exp(Call(Name('stack_fun'), [Name('local_var')]))` dargestellt.

```
File
  Name './example_fun_call_no_return_value.ast',
  [
    StructDecl
      Name 'st',
      [
        Alloc(Writeable(), IntType('int'), Name('attr1'))
        Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr2'))
      ],
    FunDecl
      VoidType 'void',
      Name 'stack_fun',
      [
        Alloc
          Writeable,
          ArrayDecl
            [
              Num '2',
              Num '3'
            ],
            StructSpec
              Name 'st',
          Name 'param'
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Exp(Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
        ↪  Name('local_var')))
        Exp(Call(Name('stack_fun'), [Name('local_var')]))
        Return(Empty())
      ],
    FunDef
      VoidType 'void',
      Name 'stack_fun',
      [
        Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
        ↪  Name('param'))
      ],
      [
        Exp(Alloc(Writeable(), IntType('int'), Name('local_var')))
      ]
  ]
```

Code 1.13: Abstract Syntax Tree für Funktionsaufruf ohne Rückgabewert

Im **PicoC-Mon Pass** in Code 1.14 wird die Komposition und Container-Knoten `Exp(Call(Name('stack_fun'), [Name('local_var')]))` durch die Kompositionen `StackMalloc(Num('2'))`, `Ref(Global(Num('0')))`, `NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))`, `Exp(GoTo(Name('stack_fun.0')))` und `RemoveStackframe()` ersetzt, welche in den Tabellen **??** und **??** genauer erklärt sind.

```
File
  Name './example_fun_call_no_return_value.picoc_mon',
  [
    Block
      Name 'main.1',
      [
        StackMalloc(Num('2'))
        Ref(Global(Num('0')))
        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('stack_fun.0')))
        RemoveStackframe()
        Return(Empty())
      ],
    Block
      Name 'stack_fun.0',
      [
        Return(Empty())
      ]
  ]
```

Code 1.14: PicoC-Mon Pass für Funktionsaufruf ohne Rückgabewert

Im **RETI-Blocks Pass** in Code 1.15 werden die Kompositionen `StackMalloc(Num('2'))`, `Ref(Global(Num('0')))`, `NewStackframe(Name('stack_fun')`, `GoTo(Name('addr@next_instr')))`, `Exp(GoTo(Name('stack_fun.0')))` und `RemoveStackframe()` durch ihre entsprechenden **RETI-Knoten** ersetzt.

Unter den **RETI-Knoten** entsprechen die **Kompostionen** `LOADI ACC GoTo(Name('addr@next_instr'))` und `Exp(GoTo(Name('stack_fun.0')))` noch keine fertigen **RETI-Instructions** und werden später in dem für sie vorgesehenen **RETI-Pass** passend ergänzt bzw. ersetzt.

Für den **Bezeichner des Blocks** `stack_fun.0` in der Komposition `Exp(GoTo(Name('stack_fun.0')))` wird im **Dictionary** `fun_name_to_block_name`[5] mit dem Schlüssel `stack_fun`, dem **Bezeichner der Funktion**, der im Container-Knoten `NewStackframe(Name('stack_fun'))` gespeichert ist nachgeschlagen.

```
File
  Name './example_fun_call_no_return_value.reti_blocks',
  [
    Block
      Name 'main.1',
      [
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
        MOVE BAF ACC;
        ADDI SP 3;
        MOVE SP BAF;
        SUBI SP 4;
```

[5] Dieses Dictionary wurde in Unterkapitel 1.5.1.1 eingeführt.

```
19        STOREIN BAF ACC 0;
20        LOADI ACC GoTo(Name('addr@next_instr'));
21        ADD ACC CS;
22        STOREIN BAF ACC -1;
23        # Exp(GoTo(Name('stack_fun.0')))
24        Exp(GoTo(Name('stack_fun.0')))
25        # RemoveStackframe()
26        MOVE BAF IN1;
27        LOADIN IN1 BAF 0;
28        MOVE IN1 SP;
29        # Return(Empty())
30        LOADIN BAF PC -1;
31      ],
32    Block
33      Name 'stack_fun.0',
34      [
35        # Return(Empty())
36        LOADIN BAF PC -1;
37      ]
38  ]
```

Code 1.15: RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert

Im **RETI Pass** in Code 1.15 wird nun der finale **RETI-Code** erstellt. Eine Änderung, die direkt auffällt, ist, dass die **RETI-Befehle** aus den **Blöcken** nun zusammengefügt sind und es keine **Blöcke** mehr gibt. Des Weiteren wird das `GoTo(Name('addr@next_instr'))` in der Komposition `LOADI ACC GoTo(Name('addr@next_instr'))` nun durch die **Adresse** des nächsten Befehls direkt nach dem dem Befehl `JUMP 5`, der für den **Sprung zur gewünschten Funktion** verantwortlich ist[6] ersetzt: `LOADI ACC 14`. Und auch der **Container-Knoten**, der den Sprung `Exp(GoTo(Name('stack_fun.0')))` darstellt wird durch den **Container-Knoten** `JUMP 5` ersetzt.

Die **Distanz** `5` im **RETI-Knoten** `JUMP 5` wird mithilfe des `instrs_before`-Attribute des **Zielblocks**, der den ersten Befehl der gewünschten Funktion enthält und des **aktuellen Blocks**, in dem der **RETI-Knoten** `JUMP 5` enthalten ist berechnet.

Die **relative Adresse** `14` direkt nach dem Befehl `JUMP 5` wird ebenfalls mithilfe des `instrs_before`-Attributs des **aktuellen Blocks** berechnet. Es handelt sich bei bei 14 um eine **relative Adresse**, die **relativ** zum `CS`-Register berechnet wird, welches im **RETI-Interpreter** von einem **Startprogramm** im **EPROM** immer so gesetzt wird, dass es die **Adresse** enthält, an der das **Codesegment** anfängt.

Die Berechnung der **Adresse** `'<addr@next_instr>'` (bzw. in der Formel adr_{danach}) des Befehls nach dem **Sprung** `JUMP <distanz>` für den Befehl `LOADI ACC <addr@next_instr>` erfolgt dabei mithilfe der folgenden Formel:

$$adr_{danach} = \#Bef_{vor\,akt.\,Bl.} + idx + 4 \tag{1.5.1}$$

wobei:

- es sich bei bei adr_{danach} um eine **relative Adresse** handelt, die **relativ** zum `CS`-Register berechnet wird.
- $\#Bef_{vor\,akt.\,Bl.} \hat{=}$ **Anzahl** Befehle vor dem momentanen Block. Es handelt sich hierbei um ein versteckes Attribut `instrs_before` eines jeden **Blockes** `Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)`, welches im **RETI-Patch**-Pass gesetzt wird. Der

[6] Also der Befehl, der bisher durch die Komposition `Exp(GoTo(Name('stack_fun.0')))` dargestellt wurde.

Grund dafür, dass das Zuweisen dieses versteckten Attributes `instrs_before` im **RETI-Patch** Pass erfolgt ist, weil erst im **RETI-Patch** Pass die **finale Anzahl** an Befehlen in einem Block feststeht, da im **RETI-Patch** Pass `GoTo()`'s entfernt werden, deren Sprung nur **eine** Adresse weiterspringen würde. Die **finale Anzahl** an Befehlen kann sich in diesem **Pass** also noch ändern und steht erst nach diesem **Pass** fest.

- $idx \hat{=}$ relativer Index des Befehls `LOADI ACC <addr@next_instr>` selbst im Block.
- $4 \hat{=}$ **Distanz**, die zwischen den in Code 1.16 markierten Befehlen `LOADI ACC <im>` und `JUMP <im>` liegt und noch **eins** mehr, weil man ja zum nächsten Befehl will.

Die Berechnug der **Distanz** `<distanz>` für den Sprung `JUMP <distanz>` zum **ersten** Befehl eines im **Pass** zuvor **existenten Blockes** erfolgt dabei nach der folgenden Formel:

$$distanz = \begin{cases} -\#Bef_{vor\ akt.\ Bl.} + \#Bef_{vor\ Zielbl.} - idx & \#Bef_{vor\ Zielbl.} < \#Bef_{vor\ akt.\ Bl.} \\ -idx & \#Bef_{vor\ Zielbl.} = \#Bef_{vor\ akt.\ Bl.} \\ \#Bef_{vor\ Zielbl.} - \#Bef_{vor\ akt.\ Bl.} - idx & \#Bef_{vor\ Zielbl.} > \#Bef_{vor\ akt.\ Bl.} \end{cases} \quad (1.5.2)$$

wobei:

- $\#Bef_{vor\ Zielbl.} \hat{=}$ **Anzahl** Befehle vor dem **Zielblock**, der den **ersten** Befehl einer Funktion enthält und zu dem gesprungen werden soll. Es handelt sich hierbei um ein verstecktes Attribut `instrs_before` eines jeden **Blockes** `Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)`.
- $\#Bef_{vor\ akt.\ Bl.}$ und idx haben die **gleiche Bedeutung** wie in der Formel 1.5.1.

```
1  # // Exp(GoTo(Name('main.1')))
2  # // not included Exp(GoTo(Name('main.1')))
3  # StackMalloc(Num('2'))
4  SUBI SP 2;
5  # Ref(Global(Num('0')))
6  SUBI SP 1;
7  LOADI IN1 0;
8  ADD IN1 DS;
9  STOREIN SP IN1 1;
10 # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
11 MOVE BAF ACC;
12 ADDI SP 3;
13 MOVE SP BAF;
14 SUBI SP 4;
15 STOREIN BAF ACC 0;
16 LOADI ACC 14;
17 ADD ACC CS;
18 STOREIN BAF ACC -1;
19 # Exp(GoTo(Name('stack_fun.0')))
20 JUMP 5;
21 # RemoveStackframe()
22 MOVE BAF IN1;
23 LOADIN IN1 BAF 0;
24 MOVE IN1 SP;
25 # Return(Empty())
26 LOADIN BAF PC -1;
27 # Return(Empty())
```

```
28 LOADIN BAF PC -1;
```

Code 1.16: RETI-Pass für Funktionsaufruf ohne Rückgabewert

1.5.1.3.1 Rückgabewert

Ein **Funktionsaufruf inklusive Zuweisung eines Rückgabewertes** (z.B. `int var = fun_with_return_value()`) wird im Folgenden mithilfe des Beispiels in Code 1.17 erklärt.

Um den Unterschied zwischen einem `return` ohne **Rückgabewert** und einem `return 21 * 2` mit **Rückgabewert** hervorzuheben, wurde ist auch eine Funktion `fun_no_return_value`, die **keinen** Rückgabewert hat in das Beispiel integriert.

```
1  int fun_with_return_value() {
2    return 21 * 2;
3  }
4
5  void fun_no_return_value() {
6    return;
7  }
8
9  void main() {
10   int var = fun_with_return_value();
11   fun_no_return_value();
12 }
```

Code 1.17: PicoC-Code für Funktionsaufruf mit Rückgabewert

Im **Abstract Syntax Tree** in Code 1.18 wird ein **Return-Statement mit Rückgabewert** `return 21 * 2` mit der Komposition `Return(BinOp(Num('21'), Mul('*'), Num('2')))` dargestellt, ein **Return-Statement ohne Rückgabewert** `return` mit der Komposition `Return(Empty())` und ein **Funktionsaufruf** inklusive **Zuweisung des Rückgabewertes** `int var = fun_with_return_value()` durch die Komposition `Assign(Alloc(Writeable(),IntType('int'),Name('var')),Call(Name('fun_with_return_value'),[]))`.

```
1  File
2    Name './example_fun_call_with_return_value.ast',
3    [
4      FunDef
5        IntType 'int',
6        Name 'fun_with_return_value',
7        [],
8        [
9          Return(BinOp(Num('21'), Mul('*'), Num('2')))
10       ],
11     FunDef
12       VoidType 'void',
13       Name 'fun_no_return_value',
14       [],
15       [
16         Return(Empty())
```

```
17      ],
18    FunDef
19      VoidType 'void',
20      Name 'main',
21      [],
22      [
23        Assign(Alloc(Writeable(), IntType('int'), Name('var')),
          ↪  Call(Name('fun_with_return_value'), []))
24        Exp(Call(Name('fun_no_return_value'), []))
25      ]
26  ]
```

Code 1.18: Abstract Syntax Tree für Funktionsaufruf mit Rückgabewert

Im **PicoC-Mon Pass** in Code 1.19 wird bei der **Komposition** `Return(BinOp(Num('21'), Mul('*'), Num('2')))` erst die **Expression** `BinOp(Num('21'), Mul('*'), Num('2'))` ausgewertet. Die hierführ erstellten Kompositionen `Exp(Num('21'))`, `Exp(Num('2'))` und `Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))` berechnen das Ergebnis des Ausdrucks `21*2` auf dem **Stack**. Dieses Ergebnis wird dann von der **Komposition** `Return(Stack(Num('1')))` vom **Stack** gelesen und in das **Register** `ACC` geschrieben und als letztes wird die **Rücksprungadresse** in das `PC`-Register geladen, die durch den `NewStackframe()`-Token-Knoten eine Speicherzelle nach dem Wert des `BAF`-Registers der aufrufenden Funktion im **Stackframe** gespeichert ist.

Ein wichtiges Detail bei der **Funktion** `fun_with_return_value` ist, dass der **Funktionsaufruf** `Call(Name('fun_with_return_value'), [])` anders übersetzt wird, da die **Funktion** einen Rückgabewert vom **Datentyp** `IntType()` und nicht `VoidType()` hat. Um den **Rückgabewert**, der durch die Komposition `Return(BinOp(Num('21'), Mul('*'), Num('2')))` in das `ACC`-Register geschrieben wurde für die aufrufende Funktion, deren Stackframe nun wieder das aktuelle ist auf den **Stack** zu schreiben, muss ein neue **Komposition** `Exp(ACC)` definiert werden. In Tabelle **??** ist die **Komposition** `Exp(ACC)` genauer erklärt.

Dieser Trick mit dem Speichern des **Rückgabewertes** im `ACC`-Register ist notwendidg, weil durch das **Entfernen** des **Stackframes** der **aufgerufenen Funktion** das `SP`-Register nicht mehr an der gleichen Stelle steht. Daher sind alle **temporären** Werte, die in der **aufgerufenen Funktion** auf den **Stack** geschrieben wurden unzugänglich, weil man nicht wissen kann, um wieviel die Adresse im `SP`-Register verglichen zu vorher verschoben ist, weil der **Stackframe** von unterschiedlichen **aufgerufenen Funktionen** unterschiedlich groß sein kann.

Die **Komposition** `Assign(Alloc(Writeable(),IntType('int'),Name('var')),Call(Name('fun_with_return_value'),[]))` wird nach dem **allokieren** der Variable `Name('var')` durch die Komposition `Assign(Global(Num('0')), Stack(Num('1')))` ersetzt, welche den **Rückgabewert** der Funktion `Name('fun_with_return_value')`, welcher durch die **Komposition** `Exp(Acc)` aus dem `ACC`-Register auf den **Stack** geschrieben wurde nun vom **Stack** in die Speicherzelle der Variable `Name('var')` speichert. Hierzu muss die **Adresse** der Variable `Name('var')` in der **Symboltabelle** nachgeschlagen werden.

Die **Komposition** `Return(Empty())` für ein `return` **ohne Rückgabewert** bleibt unverändert und stellt nur das Laden der **Rücksprungsadresse** in das `PC`-Register dar.

Des Weiteren ist zu beobachten, dass wenn bei einer Funktion mit dem **Rückgabedatentyp** `void` kein `return`-Statement explizit ans Ende geschrieben wird, im **PicoC-Mon Pass** eines hinzufügt wird in Form der `Komposition Return(Empty())`. Beim Nicht-Angeben im Falle eines Dantentyps, der **nicht** `void` ist, wird allerdings eine `MissingReturn`-**Fehlermeldung** ausgelöst.

```
File
  Name './example_fun_call_with_return_value.picoc_mon',
  [
    Block
      Name 'fun_with_return_value.2',
      [
        // Return(BinOp(Num('21'), Mul('*'), Num('2')))
        Exp(Num('21'))
        Exp(Num('2'))
        Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
        Return(Stack(Num('1')))
      ],
    Block
      Name 'fun_no_return_value.1',
      [
        Return(Empty())
      ],
    Block
      Name 'main.0',
      [
        // Assign(Name('var'), Call(Name('fun_with_return_value'), []))
        StackMalloc(Num('2'))
        NewStackframe(Name('fun_with_return_value'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('fun_with_return_value.2')))
        RemoveStackframe()
        Exp(ACC)
        Assign(Global(Num('0')), Stack(Num('1')))
        StackMalloc(Num('2'))
        NewStackframe(Name('fun_no_return_value'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('fun_no_return_value.1')))
        RemoveStackframe()
        Return(Empty())
      ]
  ]
```

Code 1.19: PicoC-Mon Pass für Funktionsaufruf mit Rückgabewert

Im **RETI-Blocks Pass** in Code 1.20 werden die Kompositionen `Exp(Num('21'))`, `Exp(Num('2'))`, `Exp(BinOp(Stack(Num('2')),Mul('*'),Stack(Num('1'))))`, `Return(Stack(Num('1')))` und `Assign(Global(Num('0')),Stack(Num('1')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_fun_call_with_return_value.reti_blocks',
  [
    Block
      Name 'fun_with_return_value.2',
      [
        # // Return(BinOp(Num('21'), Mul('*'), Num('2')))
        # Exp(Num('21'))
        SUBI SP 1;
        LOADI ACC 21;
        STOREIN SP ACC 1;
        # Exp(Num('2'))
        SUBI SP 1;
```

```
14        LOADI ACC 2;
15        STOREIN SP ACC 1;
16        # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
17        LOADIN SP ACC 2;
18        LOADIN SP IN2 1;
19        MULT ACC IN2;
20        STOREIN SP ACC 2;
21        ADDI SP 1;
22        # Return(Stack(Num('1')))
23        LOADIN SP ACC 1;
24        ADDI SP 1;
25        LOADIN BAF PC -1;
26      ],
27    Block
28      Name 'fun_no_return_value.1',
29      [
30        # Return(Empty())
31        LOADIN BAF PC -1;
32      ],
33    Block
34      Name 'main.0',
35      [
36        # // Assign(Name('var'), Call(Name('fun_with_return_value'), []))
37        # StackMalloc(Num('2'))
38        SUBI SP 2;
39        # NewStackframe(Name('fun_with_return_value'), GoTo(Name('addr@next_instr')))
40        MOVE BAF ACC;
41        ADDI SP 2;
42        MOVE SP BAF;
43        SUBI SP 2;
44        STOREIN BAF ACC 0;
45        LOADI ACC GoTo(Name('addr@next_instr'));
46        ADD ACC CS;
47        STOREIN BAF ACC -1;
48        # Exp(GoTo(Name('fun_with_return_value.2')))
49        Exp(GoTo(Name('fun_with_return_value.2')))
50        # RemoveStackframe()
51        MOVE BAF IN1;
52        LOADIN IN1 BAF 0;
53        MOVE IN1 SP;
54        SUBI SP 1;
55        STOREIN SP ACC 1;
56        # Assign(Global(Num('0')), Stack(Num('1')))
57        LOADIN SP ACC 1;
58        STOREIN DS ACC 0;
59        ADDI SP 1;
60        # StackMalloc(Num('2'))
61        SUBI SP 2;
62        # NewStackframe(Name('fun_no_return_value'), GoTo(Name('addr@next_instr')))
63        MOVE BAF ACC;
64        ADDI SP 2;
65        MOVE SP BAF;
66        SUBI SP 2;
67        STOREIN BAF ACC 0;
68        LOADI ACC GoTo(Name('addr@next_instr'));
69        ADD ACC CS;
70        STOREIN BAF ACC -1;
```

```
71          # Exp(GoTo(Name('fun_no_return_value.1')))
72          Exp(GoTo(Name('fun_no_return_value.1')))
73          # RemoveStackframe()
74          MOVE BAF IN1;
75          LOADIN IN1 BAF 0;
76          MOVE IN1 SP;
77          # Return(Empty())
78          LOADIN BAF PC -1;
79        ]
80    ]
```

Code 1.20: RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert

1.5.1.3.2 Umsetzung von Call by Sharing für Arrays

Die **Call by Reference** (Definition 1.9) Übergabe eines Arrays an eine andere Funktion, wird im Folgenden mithilfe des Beispiels in Code 1.21 erklärt.

```
1  void fun_array_from_stackframe(int param[2][3]) {
2  }
3
4  void fun_array_from_global_data(int (*param)[3]) {
5    fun_array_from_stackframe(param);
6  }
7
8  void main() {
9    int local_var[2][3];
10   fun_array_from_global_data(local_var);
11 }
```

Code 1.21: PicoC-Code für Call by Sharing für Arrays

Im **PicoC-Mon Pass** wird im Fall dessen, dass der **oberste Container-Knoten** im Teilbaum, der den Datentyp darstellt und an die Funktion übergeben wird ein **Array** `ArrayDecl(nums, datatype)` ist, dieser zu einem **Pointer** `PntrDecl(num, datatype)` umgewandelt und der Rest des Teilbaumes, der am `datatype`-Attribut hängt, an das `datatype`-Attribut des **Pointers** `PntrDecl(num, datatype)` drangehängt. Diese **Umwandlung** des **Datentyps** kann in der **Symboltabelle** in Code 1.22 beobachtet werden. Die **lokalen Variablen** `local_var1@main` und `local_var2@main` sind beide vom Datentyp `ArrayDecl([Num('2'), Num('3')], IntType('int'))` und bei der Übergabe änder sich der Datentyp beider Variablen zu `PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))`.

```
1  SymbolTable
2    [
3      Symbol
4        {
5          type qualifier:         Empty()
6          datatype:               FunDecl(VoidType('void'), Name('fun_array_from_stackframe'),
           ↪  [Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
           ↪  Name('param'))])
7          name:                   Name('fun_array_from_stackframe')
```

```
8         value or address:       Empty()
9         position:               Pos(Num('1'), Num('5'))
10        size:                   Empty()
11      },
12    Symbol
13      {
14        type qualifier:         Writeable()
15        datatype:               PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
16        name:                   Name('param@fun_array_from_stackframe')
17        value or address:       Num('0')
18        position:               Pos(Num('1'), Num('37'))
19        size:                   Num('1')
20      },
21    Symbol
22      {
23        type qualifier:         Empty()
24        datatype:               FunDecl(VoidType('void'), Name('fun_array_from_global_data'),
          ↪  [Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('param'))])
25        name:                   Name('fun_array_from_global_data')
26        value or address:       Empty()
27        position:               Pos(Num('4'), Num('5'))
28        size:                   Empty()
29      },
30    Symbol
31      {
32        type qualifier:         Writeable()
33        datatype:               PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
34        name:                   Name('param@fun_array_from_global_data')
35        value or address:       Num('0')
36        position:               Pos(Num('4'), Num('36'))
37        size:                   Num('1')
38      },
39    Symbol
40      {
41        type qualifier:         Empty()
42        datatype:               FunDecl(VoidType('void'), Name('main'), [])
43        name:                   Name('main')
44        value or address:       Empty()
45        position:               Pos(Num('8'), Num('5'))
46        size:                   Empty()
47      },
48    Symbol
49      {
50        type qualifier:         Writeable()
51        datatype:               ArrayDecl([Num('2'), Num('3')], IntType('int'))
52        name:                   Name('local_var@main')
53        value or address:       Num('0')
54        position:               Pos(Num('9'), Num('6'))
55        size:                   Num('6')
56      }
57  ]
```

Code 1.22: Symboltabelle für Call by Sharing für Arrays

Im **PicoC-Mon Pass** in Code 1.23 ist zu sehen, dass zur Übergabe der beiden Arrays die **Adresse** der Arrays auf den **Stack** geschrieben wird. Die **Adresse** der beiden Arrays auf den **Stack** zu schreiben wird

durch die Kompositionen `Ref(Global(Num('0')))` und `Ref(Global(Num('6')))` repräsentiert. Dabei stellen die Zahlen in den **Container-Knoten** `Global(num)` die **relative Adressen** relativ zum `DS`-Register dar, die aus der **Symboltabelle** entnommen sind.

```
File
  Name './example_fun_call_by_sharing_array.picoc_mon',
  [
    Block
      Name 'fun_array_from_stackframe.2',
      [
        Return(Empty())
      ],
    Block
      Name 'fun_array_from_global_data.1',
      [
        StackMalloc(Num('2'))
        Exp(Stackframe(Num('0')))
        NewStackframe(Name('fun_array_from_stackframe'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('fun_array_from_stackframe.2')))
        RemoveStackframe()
        Return(Empty())
      ],
    Block
      Name 'main.0',
      [
        StackMalloc(Num('2'))
        Ref(Global(Num('0')))
        NewStackframe(Name('fun_array_from_global_data'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('fun_array_from_global_data.1')))
        RemoveStackframe()
        Return(Empty())
      ]
  ]
```

Code 1.23: PicoC-Mon Pass für Call by Sharing für Arrays

```
File
  Name './example_fun_call_by_sharing_array.reti_blocks',
  [
    Block
      Name 'fun_array_from_stackframe.2',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun_array_from_global_data.1',
      [
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # Exp(Stackframe(Num('0')))
        SUBI SP 1;
        LOADIN BAF ACC -2;
```

```
18        STOREIN SP ACC 1;
19        # NewStackframe(Name('fun_array_from_stackframe'), GoTo(Name('addr@next_instr')))
20        MOVE BAF ACC;
21        ADDI SP 3;
22        MOVE SP BAF;
23        SUBI SP 3;
24        STOREIN BAF ACC 0;
25        LOADI ACC GoTo(Name('addr@next_instr'));
26        ADD ACC CS;
27        STOREIN BAF ACC -1;
28        # Exp(GoTo(Name('fun_array_from_stackframe.2')))
29        Exp(GoTo(Name('fun_array_from_stackframe.2')))
30        # RemoveStackframe()
31        MOVE BAF IN1;
32        LOADIN IN1 BAF 0;
33        MOVE IN1 SP;
34        # Return(Empty())
35        LOADIN BAF PC -1;
36      ],
37    Block
38      Name 'main.0',
39      [
40        # StackMalloc(Num('2'))
41        SUBI SP 2;
42        # Ref(Global(Num('0')))
43        SUBI SP 1;
44        LOADI IN1 0;
45        ADD IN1 DS;
46        STOREIN SP IN1 1;
47        # NewStackframe(Name('fun_array_from_global_data'), GoTo(Name('addr@next_instr')))
48        MOVE BAF ACC;
49        ADDI SP 3;
50        MOVE SP BAF;
51        SUBI SP 3;
52        STOREIN BAF ACC 0;
53        LOADI ACC GoTo(Name('addr@next_instr'));
54        ADD ACC CS;
55        STOREIN BAF ACC -1;
56        # Exp(GoTo(Name('fun_array_from_global_data.1')))
57        Exp(GoTo(Name('fun_array_from_global_data.1')))
58        # RemoveStackframe()
59        MOVE BAF IN1;
60        LOADIN IN1 BAF 0;
61        MOVE IN1 SP;
62        # Return(Empty())
63        LOADIN BAF PC -1;
64      ]
65  ]
```

Code 1.24: RETI-Block Pass für Call by Sharing für Arrays

1.5.1.3.3 Umsetzung von Call by Value für Structs

```
struct st {int attr1; int attr2[2];};


void fun_struct_from_stackframe(struct st param) {
}

void fun_struct_from_global_data(struct st param) {
  fun_struct_from_stackframe(param);
}


void main() {
  struct st local_var;
  fun_struct_from_global_data(local_var);
}
```

Code 1.25: PicoC-Code für Call by Value für Structs

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('fun_array_from_stackframe'),
        ↪  [Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
        ↪  Name('param'))])
        name:                   Name('fun_array_from_stackframe')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
        name:                   Name('param@fun_array_from_stackframe')
        value or address:       Num('0')
        position:               Pos(Num('1'), Num('37'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('fun_array_from_global_data'),
        ↪  [Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('param'))])
        name:                   Name('fun_array_from_global_data')
        value or address:       Empty()
        position:               Pos(Num('4'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
```

```
34        name:                   Name('param@fun_array_from_global_data')
35        value or address:       Num('0')
36        position:               Pos(Num('4'), Num('36'))
37        size:                   Num('1')
38      },
39    Symbol
40      {
41        type qualifier:         Empty()
42        datatype:               FunDecl(VoidType('void'), Name('main'), [])
43        name:                   Name('main')
44        value or address:       Empty()
45        position:               Pos(Num('8'), Num('5'))
46        size:                   Empty()
47      },
48    Symbol
49      {
50        type qualifier:         Writeable()
51        datatype:               ArrayDecl([Num('2'), Num('3')], IntType('int'))
52        name:                   Name('local_var@main')
53        value or address:       Num('0')
54        position:               Pos(Num('9'), Num('6'))
55        size:                   Num('6')
56      }
57  ]
```

Code 1.26: Symboltabelle für Call by Sharing für Arrays

```
1 File
2   Name './example_fun_call_by_value_struct.picoc_mon',
3   [
4     Block
5       Name 'fun_struct_from_stackframe.2',
6       [
7         Return(Empty())
8       ],
9     Block
10      Name 'fun_struct_from_global_data.1',
11      [
12        StackMalloc(Num('2'))
13        Assign(Stack(Num('3')), Stackframe(Num('2')))
14        NewStackframe(Name('fun_struct_from_stackframe'), GoTo(Name('addr@next_instr')))
15        Exp(GoTo(Name('fun_struct_from_stackframe.2')))
16        RemoveStackframe()
17        Return(Empty())
18      ],
19    Block
20      Name 'main.0',
21      [
22        StackMalloc(Num('2'))
23        Assign(Stack(Num('3')), Global(Num('0')))
24        NewStackframe(Name('fun_struct_from_global_data'), GoTo(Name('addr@next_instr')))
25        Exp(GoTo(Name('fun_struct_from_global_data.1')))
26        RemoveStackframe()
27        Return(Empty())
```

```
      ]
  ]
```

Code 1.27: PicoC-Mon Pass für Call by Value für Structs

```
File
  Name './example_fun_call_by_value_struct.reti_blocks',
  [
    Block
      Name 'fun_struct_from_stackframe.2',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun_struct_from_global_data.1',
      [
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # Assign(Stack(Num('3')), Stackframe(Num('2')))
        SUBI SP 3;
        LOADIN BAF ACC -4;
        STOREIN SP ACC 1;
        LOADIN BAF ACC -3;
        STOREIN SP ACC 2;
        LOADIN BAF ACC -2;
        STOREIN SP ACC 3;
        # NewStackframe(Name('fun_struct_from_stackframe'), GoTo(Name('addr@next_instr')))
        MOVE BAF ACC;
        ADDI SP 5;
        MOVE SP BAF;
        SUBI SP 5;
        STOREIN BAF ACC 0;
        LOADI ACC GoTo(Name('addr@next_instr'));
        ADD ACC CS;
        STOREIN BAF ACC -1;
        # Exp(GoTo(Name('fun_struct_from_stackframe.2')))
        Exp(GoTo(Name('fun_struct_from_stackframe.2')))
        # RemoveStackframe()
        MOVE BAF IN1;
        LOADIN IN1 BAF 0;
        MOVE IN1 SP;
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'main.0',
      [
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # Assign(Stack(Num('3')), Global(Num('0')))
        SUBI SP 3;
        LOADIN DS ACC 0;
        STOREIN SP ACC 1;
```

```
50          LOADIN DS ACC 1;
51          STOREIN SP ACC 2;
52          LOADIN DS ACC 2;
53          STOREIN SP ACC 3;
54          # NewStackframe(Name('fun_struct_from_global_data'), GoTo(Name('addr@next_instr')))
55          MOVE BAF ACC;
56          ADDI SP 5;
57          MOVE SP BAF;
58          SUBI SP 5;
59          STOREIN BAF ACC 0;
60          LOADI ACC GoTo(Name('addr@next_instr'));
61          ADD ACC CS;
62          STOREIN BAF ACC -1;
63          # Exp(GoTo(Name('fun_struct_from_global_data.1')))
64          Exp(GoTo(Name('fun_struct_from_global_data.1')))
65          # RemoveStackframe()
66          MOVE BAF IN1;
67          LOADIN IN1 BAF 0;
68          MOVE IN1 SP;
69          # Return(Empty())
70          LOADIN BAF PC -1;
71        ]
72    ]
```

Code 1.28: RETI-Block Pass für Call by Value für Structs

1.6 Fehlermeldungen

1.6.1 Error Handler

1.6.2 Arten von Fehlermeldungen

1.6.2.1 Syntaxfehler

1.6.2.2 Laufzeitfehler

Literatur

Online

- *What is the difference between function prototype and function signature?* SoloLearn. URL: https://www.sololearn.com/Discuss/171026/what-is-the-difference-between-function-prototype-and-function-signature/ (besucht am 18. 07. 2022).

Bücher

- G. Siek, Jeremy. *Course Webpage for Compilers (P423, P523, E313, and E513)*. 28. Jan. 2022. URL: https://iucompilercourse.github.io/IU-Fall-2021/ (besucht am 28. 01. 2022).

Vorlesungen

- Bast, Hannah. „Programmieren in C“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://ad-wiki.informatik.uni-freiburg.de/teaching/ProgrammierenCplusplusSS2020 (besucht am 09. 07. 2022).
- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09. 07. 2022).
- Scholl, Philipp. „Einführung in Embedded Systems“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: https://earth.informatik.uni-freiburg.de/uploads/es-2122/ (besucht am 09. 07. 2022).
- Thiemann, Peter. „Einführung in die Programmierung“. Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/ (besucht am 09. 07. 2022).