
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

Betreuung:
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

0.0.1	Umsetzung von Funktionen	8
0.0.1.1	Mehrere Funktionen	8
0.0.1.1.1	Sprung zur Main Funktion	11
0.0.1.2	Funktionsdeklaration und -definition und Umsetzung von Scopes	13
0.0.1.3	Funktionsaufruf	16
0.0.1.3.1	Ohne Rückgabewert	16
0.0.1.3.2	Mit Rückgabewert	20
0.0.1.3.3	Umsetzung von Call by Sharing für Arrays	22
0.0.1.3.4	Umsetzung von Call by Value für Structs	25
0.1	Fehlermeldungen	26
0.1.1	Error Handler	26
0.1.2	Arten von Fehlermeldungen	26
0.1.2.1	Syntaxfehler	26
0.1.2.2	Laufzeitfehler	26

Abbildungsverzeichnis

Codeverzeichnis

0.1	PicoC-Code für 3 Funktionen	8
0.2	Abstract Syntax Tree für 3 Funktionen	9
0.3	PicoC-Blocks Pass für 3 Funktionen	10
0.4	PicoC-Mon Pass für 3 Funktionen	10
0.5	RETI-Blocks Pass für 3 Funktionen	11
0.6	PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist	11
0.7	RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	12
0.8	RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	13
0.9	RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	13
0.10	PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss	14
0.11	Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss	16
0.12	PicoC-Code für Funktionsaufruf ohne Rückgabewert	16
0.13	Abstract Syntax Tree für Funktionsaufruf ohne Rückgabewert	17
0.14	PicoC-Mon Pass für Funktionsaufruf ohne Rückgabewert	18
0.15	RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert	19
0.16	RETI-Pass für Funktionsaufruf ohne Rückgabewert	20
0.17	PicoC-Code für Funktionsaufruf mit Rückgabewert	20
0.18	PicoC-Mon Pass für Funktionsaufruf mit Rückgabewert	20
0.19	RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert	21
0.20	RETI-Pass für Funktionsaufruf mit Rückgabewert	22
0.21	PicoC-Code für Call by Sharing für Arrays	22
0.22	PicoC-Mon Pass für Call by Sharing für Arrays	23
0.23	Symboltabelle für Call by Sharing für Arrays	24
0.24	RETI-Block Pass für Call by Sharing für Arrays	25
0.25	PicoC-Code für Call by Value für Structs	25
0.26	PicoC-Mon Pass für Call by Value für Structs	25
0.27	RETI-Block Pass für Call by Value für Structs	26

Tabellenverzeichnis

Definitionsverzeichnis

0.1	Funktionsprototyp	14
0.2	Scope (bzw. Sichtbarkeitsbereich)	15

Grammatikverzeichnis

0.0.1 Umsetzung von Funktionen

0.0.1.1 Mehrere Funktionen

Die Umsetzung **mehrerer Funktionen** wird im Folgenden mithilfe des Beispiels in Code 0.1 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 0.1: PicoC-Code für 3 Funktionen

Im **Abstract Syntax Tree** in Code 0.2 wird eine **Funktion**, wie z.B. `void fun(int param;){ return param; }` 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 0.2: Abstract Syntax Tree für 3 Funktionen

Im **PicoC-Blocks Pass** in Code 0.3 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>.<number>`².

Es werden parallel dazu neue Zuordnungen im **Dictionary** `fun_name_to_block_name` hinzugefügt. Das **Dictionary** ordnet einem **Funktionsnamen** den **Blocknamen** des Blockes, der das erste **Statement** der Funktion enthält und dessen **Bezeichner** `<name>.<number>` bis auf die angehängte **Nummer** identisch zu dem der Funktion ist zu³. Diese Zuordnung ist nötig, da **Blöcke** noch eine **Nummer** an ihren Bezeichner `<name>.<number>` angehängt haben.

```

1 File
2   Name './verbose_3_funs.picoc_blocks',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Block
10          Name 'main.2',
11          [
12            Return(Empty())
13          ]
14      ],
15     FunDef
16       VoidType 'void',
17       Name 'fun1',
18       [],
19       [
20         Block
21          Name 'fun1.1',
22          []
23      ],
24     FunDef
25       IntType 'int',
26       Name 'fun2',
27       [],
28       [
29         Block
30          Name 'fun2.0',
31          [
32            Return(Num('1'))
33          ]
34      ]
35 ]

```

¹Eine Erklärung dazu ist in Unterkapitel ?? zu finden.

²Der **Grund** dafür kann im Unterkapitel ?? nachgelesen werden.

³Das ist der **Block**, über den im **obigen letzten Paragraph** gesprochen wurde.

Code 0.3: PicoC-Blocks Pass für 3 Funktionen

Im **PicoC-Mon Pass** in Code 0.4 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.

```

1 File
2   Name './verbose_3_funs.picoc_mon',
3   [
4     Block
5       Name 'main.2',
6       [
7         Return(Empty())
8       ],
9     Block
10      Name 'fun1.1',
11      [
12        Return(Empty())
13      ],
14     Block
15      Name 'fun2.0',
16      [
17        // Return(Num('1'))
18        Exp(Num('1'))
19        Return(Stack(Num('1')))
20      ]
21   ]

```

Code 0.4: PicoC-Mon Pass für 3 Funktionen

Nach dem **RETI Pass** in Code 0.5 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 **Statemens** einer **Funktion** kompiliert wurden anfangen.

```

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

```

```

9 LOADIN BAF PC -1;
10 # // Block(Name('fun2.0'), [])
11 # // Return(Num('1'))
12 # Exp(Num('1'))
13 SUBI SP 1;
14 LOADI ACC 1;
15 STOREIN SP ACC 1;
16 # Return(Stack(Num('1'))))
17 LOADIN SP ACC 1;
18 ADDI SP 1;
19 LOADIN BAF PC -1;

```

Code 0.5: RETI-Blocks Pass für 3 Funktionen

0.0.1.1.1 Sprung zur Main Funktion

Im vorherigen Beispiel in Code 0.1 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 0.6 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.

```

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

```

Code 0.6: PicoC-Code für Funktionen, wobei die `main` Funktion nicht die erste Funktion ist

Im **RETI-Blocks Pass** in Code 0.7 sind die **Funktionen** nur noch durch **Blöcke** umgesetzt.

```

1 File
2   Name './verbose_3_funs_main.reti_blocks',
3   [
4     Block
5       Name 'fun1.2',
6       [
7         # Return(Empty())
8         LOADIN BAF PC -1;
9       ],
10    Block
11      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 0.7: 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.<number>-Block als ersten Block einzufügen, der einen GoTo(Name('main.<number>'))-Container-Knoten enthält, der im **RETI Pass 0.9** 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.<number>-Blockes erfolgt im **RETI-Patch Pass** in Code 0.8, 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;

```

```

15     ],
16     Block
17     Name 'fun2.1',
18     [
19         # // Return(Num('1'))
20         # Exp(Num('1'))
21         SUBI SP 1;
22         LOADI ACC 1;
23         STOREIN SP ACC 1;
24         # Return(Stack(Num('1')))
25         LOADIN SP ACC 1;
26         ADDI SP 1;
27         LOADIN BAF PC -1;
28     ],
29     Block
30     Name 'main.0',
31     [
32         # Return(Empty())
33         LOADIN BAF PC -1;
34     ]
35 ]

```

Code 0.8: RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

Im **RETI Pass** in Code 0.9 wird das `GoTo(Name('main.<number>'))` durch den entsprechenden Sprung `JUMP <distanz_zur_main_funktion>` ersetzt und die Blöcke entfernt.

```

1 # // Block(Name('start.3'), [])
2 # // Exp(GoTo(Name('main.0'))))
3 JUMP 8;
4 # // Block(Name('fun1.2'), [])
5 # Return(Empty())
6 LOADIN BAF PC -1;
7 # // Block(Name('fun2.1'), [])
8 # // Return(Num('1'))
9 # Exp(Num('1'))
10 SUBI SP 1;
11 LOADI ACC 1;
12 STOREIN SP ACC 1;
13 # Return(Stack(Num('1')))
14 LOADIN SP ACC 1;
15 ADDI SP 1;
16 LOADIN BAF PC -1;
17 # // Block(Name('main.0'), [])
18 # Return(Empty())
19 LOADIN BAF PC -1;

```

Code 0.9: RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

0.0.1.2 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 0.1) 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 Deklaration einer Funktion kann explizit erfolgen (z.B. `int fun2(int var);`), wie in der im Beispiel in Code 0.10 **markierten Zeile** 1 oder zusammen mit der **Funktionsdefinition** (z.B. `void fun1(){}`), wie in den **markierten Zeilen** 3-4.

In dem Beispiel in Code 0.10 erfolgt ein **Funktionsaufruf** zur Funktion `fun2`, die allerdings erst nach der `main`-Funktion definiert ist. Daher ist eine **Funktionsdeklaration**, 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 0.1: Funktionsprototyp

*Deklaration einer Funktion, welche den **Funktionsbezeichner**, die **Datentypen** der einzelnen **Funktionsparameter**, die **Parameterreihenfolge** und den **Rückgabewert** einer Funktion spezifiziert. Es ist **nicht** möglich zwei Funktionendeklarationen mit dem **gleichen** Funktionsbezeichner zu haben.^{a,b}*

^aDer **Funktionsprototyp** ist von der **FunktionsSignatur** zu unterscheiden, 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 **Datentypen** von **Parametern**, die **Parameterreihenfolge**, manchmal auch **Scopes** und **Klassentypen** usw. unterscheiden.

^bWhat 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 0.10: PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss

Die **Deklaration** einer **Funktion** erfolgt mithilfe der **Symboltabelle**, die in Code 0.11 für das Beispiel in Code 0.10 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(Writable(), IntType('int'), Name('var'))])` zugewiesen.

Die Variablen `var@main` und `var@fun2` der `main`-Funktion und der Funktion `fun2` haben unterschiedliche **Scopes** (Definition 0.2). 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

⁴D.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`-Attributtes eines **Symbols** der Symboltabelle angehängt. Zur besseren Vorstellung ist dies in Code 0.11 **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 um 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 0.2: Scope (bzw. Sichtbarkeitsbereich)

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

^aThiemann, „Einführung in die Programmierung“.

```

1 SymbolTable
2 [
3     Symbol
4     {
5         type qualifier:      Empty()
6         datatype:            FunDecl(IntType('int'), Name('fun2'), [Alloc(Writable(),
7                               ↪ IntType('int'), Name('var'))])
8         name:                Name('fun2')
9         value or address:     Empty()
10        position:            Pos(Num('1'), Num('4'))
11        size:                Empty()
12    },
13    Symbol
14    {
15        type qualifier:      Empty()
16        datatype:            FunDecl(VoidType('void'), Name('fun1'), [])
17        name:                Name('fun1')
18        value or address:     Empty()
19        position:            Pos(Num('3'), Num('5'))
20        size:                Empty()
21    },
22    Symbol
23    {
24        type qualifier:      Empty()
25        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 0.11: Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss

0.0.1.3 Funktionsaufruf

0.0.1.3.1 Ohne Rückgabewert

Ein **Funktionsaufruf** (z.B. `stack_fun(local_var)`), wird im Folgenden mithilfe des Beispiels in Code 0.12 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 0.0.1.3.2 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 0.12: PicoC-Code für Funktionsaufruf ohne Rückgabewert

Im **Abstract Syntax Tree** in Code 0.13 wird ein **Funktionsaufruf** `stack_fun(local_var)` durch die **Kom-**

position `Exp(Call(Name('stack_fun'), [Name('local_var')]))` dargestellt.

```

1 File
2   Name './example_fun_call_no_return_value.ast',
3   [
4     StructDecl
5       Name 'st',
6       [
7         Alloc(Writeable(), IntType('int'), Name('attr1'))
8         Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr2'))
9       ],
10    FunDecl
11      VoidType 'void',
12      Name 'stack_fun',
13      [
14        Alloc
15          Writeable,
16          ArrayDecl
17            [
18              Num '2',
19              Num '3'
20            ],
21          StructSpec
22            Name 'st',
23            Name 'param'
24        ],
25      FunDef
26        VoidType 'void',
27        Name 'main',
28        [],
29        [
30          Exp(Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
31            ↪ Name('local_var'))
32          Exp(Call(Name('stack_fun'), [Name('local_var')]))
33          Return(Empty())
34        ],
35      FunDef
36        VoidType 'void',
37        Name 'stack_fun',
38        [
39          Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
40            ↪ Name('param'))
41        ],
42        [
43          Exp(Alloc(Writeable(), IntType('int'), Name('local_var'))))
44        ]
45    ]
46  ]

```

Code 0.13: Abstract Syntax Tree für Funktionsaufruf ohne Rückgabewert

Im **PicoC-Mon Pass** in Code 0.14 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.

```

1 File
2   Name './example_fun_call_no_return_value.picoc_mon',
3   [
4     Block
5       Name 'main.1',
6       [
7         StackMalloc(Num('2'))
8         Ref(Global(Num('0')))
9         NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
10        Exp(GoTo(Name('stack_fun.0')))
11        RemoveStackframe()
12        Return(Empty())
13      ],
14    Block
15      Name 'stack_fun.0',
16      [
17        Return(Empty())
18      ]
19  ]

```

Code 0.14: PicoC-Mon Pass für Funktionsaufruf ohne Rückgabewert

Im **RETI-Blocks Pass** in Code 0.15 werden die Kompositionen `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()` dann durch ihre entsprechenden **RETI-Knoten** ersetzt.

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

```

1 File
2   Name './example_fun_call_no_return_value.reti_blocks',
3   [
4     Block
5       Name 'main.1',
6       [
7         # StackMalloc(Num('2'))
8         SUBI SP 2;
9         # Ref(Global(Num('0')))
10        SUBI SP 1;
11        LOADI IN1 0;
12        ADD IN1 DS;
13        STOREIN SP IN1 1;
14        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
15        MOVE BAF ACC;
16        ADDI SP 3;
17        MOVE SP BAF;
18        SUBI SP 4;
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 0.15: RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert

Im **RETI Pass** in Code 0.15 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 nach dem **Sprung zur gewünschten Funktion**, der bisher durch die Komposition `Exp(GoTo(Name('stack_fun.0')))` dargestellt wurde ersetzt: `LOADI ACC 14`; . Dieser Sprung `Exp(GoTo(Name('stack_fun.0')))` wird mithilfe des **Container-Knoten** `JUMP 5` ersetzt.

```

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 0.16: RETI-Pass für Funktionsaufruf ohne Rückgabewert

0.0.1.3.2 Mit Rückgabewert

```
1 void stack_fun() {
2     return 42;
3 }
4
5 void main() {
6     int var = stack_fun();
7 }
```

Code 0.17: PicoC-Code für Funktionsaufruf mit Rückgabewert

```
1 File
2   Name './example_fun_call_with_return_value.picoc_mon',
3   [
4       Block
5         Name 'stack_fun.1',
6         [
7             // Return(Num('42'))
8             Exp(Num('42'))
9             Return(Stack(Num('1')))
10        ],
11        Block
12          Name 'main.0',
13          [
14              // Assign(Name('var'), Call(Name('stack_fun'), []))
15              StackMalloc(Num('2'))
16              NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))
17              Exp(GoTo(Name('stack_fun.1')))
18              RemoveStackframe()
19              Assign(Global(Num('0')), Stack(Num('1')))
20              Return(Empty())
21          ]
22      ]
```

Code 0.18: PicoC-Mon Pass für Funktionsaufruf mit Rückgabewert

```
1 File
2   Name './example_fun_call_with_return_value.reti_blocks',
3   [
4       Block
5         Name 'stack_fun.1',
6         [
```

```

7      # // Return(Num('42'))
8      # Exp(Num('42'))
9      SUBI SP 1;
10     LOADI ACC 42;
11     STOREIN SP ACC 1;
12     # Return(Stack(Num('1')))
13     LOADIN SP ACC 1;
14     ADDI SP 1;
15     LOADIN BAF PC -1;
16 ],
17 Block
18   Name 'main.0',
19   [
20     # // Assign(Name('var'), Call(Name('stack_fun'), []))
21     # StackMalloc(Num('2'))
22     SUBI SP 2;
23     # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
24     MOVE BAF ACC;
25     ADDI SP 2;
26     MOVE SP BAF;
27     SUBI SP 2;
28     STOREIN BAF ACC 0;
29     LOADI ACC GoTo(Name('addr@next_instr'));
30     ADD ACC CS;
31     STOREIN BAF ACC -1;
32     # Exp(GoTo(Name('stack_fun.1')))
33     Exp(GoTo(Name('stack_fun.1')))
34     # RemoveStackframe()
35     MOVE BAF IN1;
36     LOADIN IN1 BAF 0;
37     MOVE IN1 SP;
38     # Assign(Global(Num('0')), Stack(Num('1')))
39     LOADIN SP ACC 1;
40     STOREIN DS ACC 0;
41     ADDI SP 1;
42     # Return(Empty())
43     LOADIN BAF PC -1;
44   ]
45 ]

```

Code 0.19: RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert

```

1 # // Exp(GoTo(Name('main.0')))
2 JUMP 7;
3 # // Return(Num('42'))
4 # Exp(Num('42'))
5 SUBI SP 1;
6 LOADI ACC 42;
7 STOREIN SP ACC 1;
8 # Return(Stack(Num('1')))
9 LOADIN SP ACC 1;
10 ADDI SP 1;
11 LOADIN BAF PC -1;
12 # // Assign(Name('var'), Call(Name('stack_fun'), []))

```

```

13 # StackMalloc(Num('2'))
14 SUBI SP 2;
15 # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))))
16 MOVE BAF ACC;
17 ADDI SP 2;
18 MOVE SP BAF;
19 SUBI SP 2;
20 STOREIN BAF ACC 0;
21 LOADI ACC 17;
22 ADD ACC CS;
23 STOREIN BAF ACC -1;
24 # Exp(GoTo(Name('stack_fun.1'))))
25 JUMP -15;
26 # RemoveStackframe()
27 MOVE BAF IN1;
28 LOADIN IN1 BAF 0;
29 MOVE IN1 SP;
30 # Assign(Global(Num('0')), Stack(Num('1'))))
31 LOADIN SP ACC 1;
32 STOREIN DS ACC 0;
33 ADDI SP 1;
34 # Return(Empty())
35 LOADIN BAF PC -1;

```

Code 0.20: RETI-Pass für Funktionsaufruf mit Rückgabewert

0.0.1.3.3 Umsetzung von Call by Sharing für Arrays

```

1 void stack_fun(int (*param1)[3], int param2[2][3]) {
2 }
3
4 void main() {
5     int local_var1[2][3];
6     int local_var2[2][3];
7     stack_fun(local_var1, local_var2);
8 }

```

Code 0.21: PicoC-Code für Call by Sharing für Arrays

```

1 File
2   Name './example_fun_call_by_sharing_array.picoc_mon',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         Return(Empty())
8       ],
9     Block
10      Name 'main.0',
11      [
12        StackMalloc(Num('2'))

```

```

13     Ref(Global(Num('0')))
14     Ref(Global(Num('6')))
15     NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
16     Exp(GoTo(Name('stack_fun.1')))
17     RemoveStackframe()
18     Return(Empty())
19 ]
20 ]

```

Code 0.22: PicoC-Mon Pass für Call by Sharing für Arrays

```

1 SymbolTable
2 [
3     Symbol
4     {
5         type qualifier:      Empty()
6         datatype:            FunDecl(VoidType('void'), Name('stack_fun'),
7                               ↪ [Alloc(Writeable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
8                               ↪ Name('param1')), Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')),
9                               ↪ Name('param2'))])
10        name:                Name('stack_fun')
11        value or address:     Empty()
12        position:             Pos(Num('1'), Num('5'))
13        size:                 Empty()
14    },
15    Symbol
16    {
17        type qualifier:      Writeable()
18        datatype:            PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
19        name:                Name('param1@stack_fun')
20        value or address:     Num('0')
21        position:             Pos(Num('1'), Num('21'))
22        size:                 Num('1')
23    },
24    Symbol
25    {
26        type qualifier:      Writeable()
27        datatype:            PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
28        name:                Name('param2@stack_fun')
29        value or address:     Num('1')
30        position:             Pos(Num('1'), Num('37'))
31        size:                 Num('1')
32    },
33    Symbol
34    {
35        type qualifier:      Empty()
36        datatype:            FunDecl(VoidType('void'), Name('main'), [])
37        name:                Name('main')
38        value or address:     Empty()
39        position:             Pos(Num('4'), Num('5'))
40        size:                 Empty()
41    },
42    Symbol
43    {

```



```

41     type qualifier:      Writeable()
42     datatype:           ArrayDecl([Num('2'), Num('3')], IntType('int'))
43     name:               Name('local_var1@main')
44     value or address:    Num('0')
45     position:           Pos(Num('5'), Num('6'))
46     size:               Num('6')
47 },
48 Symbol
49 {
50     type qualifier:      Writeable()
51     datatype:           ArrayDecl([Num('2'), Num('3')], IntType('int'))
52     name:               Name('local_var2@main')
53     value or address:    Num('6')
54     position:           Pos(Num('6'), Num('6'))
55     size:               Num('6')
56 }
57 ]

```

Code 0.23: Symboltabelle für Call by Sharing für Arrays

```

1 File
2   Name './example_fun_call_by_sharing_array.reti_blocks',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         # Return(Empty())
8         LOADIN BAF PC -1;
9       ],
10    Block
11      Name 'main.0',
12      [
13        # StackMalloc(Num('2'))
14        SUBI SP 2;
15        # Ref(Global(Num('0')))
16        SUBI SP 1;
17        LOADI IN1 0;
18        ADD IN1 DS;
19        STOREIN SP IN1 1;
20        # Ref(Global(Num('6')))
21        SUBI SP 1;
22        LOADI IN1 6;
23        ADD IN1 DS;
24        STOREIN SP IN1 1;
25        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
26        MOVE BAF ACC;
27        ADDI SP 4;
28        MOVE SP BAF;
29        SUBI SP 4;
30        STOREIN BAF ACC 0;
31        LOADI ACC GoTo(Name('addr@next_instr'));
32        ADD ACC CS;
33        STOREIN BAF ACC -1;
34        # Exp(GoTo(Name('stack_fun.1')))

```

```

35     Exp(GoTo(Name('stack_fun.1')))
36     # RemoveStackframe()
37     MOVE BAF IN1;
38     LOADIN IN1 BAF 0;
39     MOVE IN1 SP;
40     # Return(Empty())
41     LOADIN BAF PC -1;
42 ]
43 ]

```

Code 0.24: RETI-Block Pass für Call by Sharing für Arrays

0.0.1.3.4 Umsetzung von Call by Value für Structs

```

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

```

Code 0.25: PicoC-Code für Call by Value für Structs

```

1 File
2   Name './example_fun_call_by_value_struct.picoc_mon',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         Return(Empty())
8       ],
9     Block
10      Name 'main.0',
11      [
12        StackMalloc(Num('2'))
13        Assign(Stack(Num('3')), Global(Num('0')))
14        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
15        Exp(GoTo(Name('stack_fun.1')))
16        RemoveStackframe()
17        Return(Empty())
18      ]
19 ]

```

Code 0.26: PicoC-Mon Pass für Call by Value für Structs

```
1 File
2   Name './example_fun_call_by_value_struct.reti_blocks',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         # Return(Empty())
8         LOADIN BAF PC -1;
9       ],
10    Block
11      Name 'main.0',
12      [
13        # StackMalloc(Num('2'))
14        SUBI SP 2;
15        # Assign(Stack(Num('3')), Global(Num('0')))
16        SUBI SP 3;
17        LOADIN DS ACC 0;
18        STOREIN SP ACC 1;
19        LOADIN DS ACC 1;
20        STOREIN SP ACC 2;
21        LOADIN DS ACC 2;
22        STOREIN SP ACC 3;
23        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
24        MOVE BAF ACC;
25        ADDI SP 5;
26        MOVE SP BAF;
27        SUBI SP 5;
28        STOREIN BAF ACC 0;
29        LOADI ACC GoTo(Name('addr@next_instr'));
30        ADD ACC CS;
31        STOREIN BAF ACC -1;
32        # Exp(GoTo(Name('stack_fun.1')))
33        Exp(GoTo(Name('stack_fun.1')))
34        # RemoveStackframe()
35        MOVE BAF IN1;
36        LOADIN IN1 BAF 0;
37        MOVE IN1 SP;
38        # Return(Empty())
39        LOADIN BAF PC -1;
40      ]
41  ]
```

Code 0.27: RETI-Block Pass für Call by Value für Structs

0.1 Fehlermeldungen

0.1.1 Error Handler

0.1.2 Arten von Fehlermeldungen

0.1.2.1 Syntaxfehler

0.1.2.2 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).

Vorlesungen

- 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).