Albert Ludwigs Universität Freiburg

TECHNISCHE FAKULTÄT

PicoC-Compiler

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

BACHELORARBEIT

Abgabedatum: 13. September 2022

Autor: 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.

Danksagungen

Bevor der Inhalt dieser Schrifftlichen Ausarbeitung der Bachelorarbeit anfängt, will ich einigen Personen noch meinen Dank aussprechen.

Ich schreibe die folgenden Danksagungen nicht auf eine bestimmte Weise, wie es sich vielleicht etabliert haben sollte Danksagungen zu schreiben und verwende auch keine künstlichen Floskeln, wie "mein aufrichtigster Dank" oder "aus tiefstem Herzen", sondern drücke im Folgenden die Dinge nur so aus, wie ich sie auch wirklich meine.

Estmal, ich hatte selten im Studium das Gefühl irgendwo Kunde zu sein, aber bei dieser Bachelorarbeit und dem vorangegangenen Bachelorprojekt hatte ich genau diese Gefühl, obwohl die Verhältnisse eigentlich genau umgekehrt sein sollten. Die Umgang mit mir wahr echt unglaublich nett und unbürokratisch, was ich als keine Selbverständlichkeit ansehe und sehr wertgeschätzt habe.

An erster Stelle will ich zu meinem Betreuer M.Sc. Tobias Seufert kommen, der netterweise auch bereits die Betreuung meines Bachelorprojektes übernommen hatte. Wie auch während des Bachelorprojektes, haben wir uns auch bei den Meetings während der Bachelorarbeit hervorragend verstanden. Dabei ging die Freundlichkeit und das Engagement seitens Tobias weit über das heraus, was man bereits als eine gute Betreuung bezeichnen würde.

Es gibt verschiedene Typen von Menschen, es gibt Leute, die nur genauso viel tun, wie es die Anforderungen verlangen und nichts darüberhinaus tun, wenn es nicht einen eigenen Vorteil für sie hat und es gibt Personen, die sich für nichts zu Schade sind und dies aus einer Philanthropie oder Leidenschafft heraus tun, auch wenn es für sie keine Vorteile hat. Tobias¹ konnte ich während der langen Zeit, die er mein Bachelorprojekt und dann meine Bachelorarbeit betreut hat eindeutig als letzteren Typ Mensch einordnen.

Er war sich nie zu Schade für meine vielen Fragen während der Meetings, auch wenn ich meine Zeit ziemlich oft überzogen habe², er hat sich bei der Korrektur dieser Schrifftlichen Ausarbeitung sogar die Mühe gemacht bei den einzelnen Problemstellen längere, wirklich hilfreiche Textkommentare zu verfassen und obendrauf auch noch Tippfehler usw. angemerkt und war sich nicht zu Schade die Rolle des Nachrichtenübermittlers zwischen mir und Prof. Dr. Scholl zu übernehmen. All dies war absolut keine Selbverständlichkeit, vor allem wenn ich die Betreuung anderer Studenten, die ich kenne mit der vergleiche, die mir zu Teil wurde.

An den Kommentar zu meinem Betreuer Tobias will ich einen Kommentar zu meinem Gutachter Prof. Dr. Scholl anschließen. Wofür ich meinem Gutachter Prof. Dr. Scholl sehr dankbar bin, ist, dass er meine damals sehr ambitionierten Ideen für mögliche Funktionalitäten, die ich in den PicoC-Compiler für die Bachelorarbeit implementierten wollte runtergeschraubt hat. Man erlebt es äußerst selten im Studium, dass Studenten freiwillig weniger Arbeit gegeben wird.

Bei den für die Bachelorarbeit zu implementierenden Funktionalitäten gab es bei der Implementierung viele unerwartete kleine Details, die ich vorher garnicht bedacht hatte, die in ihrer Masse unerwartet viel Zeit zum Implementieren gebraucht haben. Mit den von Prof. Dr. Scholl festgelegten Funktionalitäten für die Bachelorarbeit ist der Zeitplan jedoch ziemlich perfekt aufgegangen. Mit meinen ambitionierten Plänen wäre es bei der Bachelorarbeit dagegeben wohl mit der Zeit äußerst kritisch geworden. Das Prof. Dr. Scholl mir zu

¹Wie auch Prof. Dr. Scholl. Hier geht es aber erstmal um Tobias.

²Wofür ich mich auch nochmal Entschuldigen will.

seinem eigenen Nachteil 3 weniger Arbeit aufgebrummt hat empfand ich als ich eine äußerst nette Geste, die ich sehr geschätzt habe.

Wie mein Betreuer M.Sc. Tobias Seufert und wahrscheinlich auch mein Gutachter Prof. Dr. Scholl im Verlauf dieser Bachelorarbeit und des vorangegangenen Bachelorprojektes gemerkt haben, kann ich schon manchmal ziemlich eigensinnigen sein, bei der Weise, wie ich bestimmte Dinge umsetzen will. Ich habe es sehr geschätzt, dass mir das durchgehen gelassen wurde. Es ist, wie ich die Universitätswelt als Student erlebe bei Arbeitsvorgaben keine Selbverständlichkeit, dass dem Studenten überhaupt die Freiheit und das Vertrauen gegeben wird diese auf seine eigenen Weise umzusetzen.

Vor allem, da mein eigenes Vorgehen größtenteils Vorteile für mich hatte, da ich auf diese Weise am meisten über Compilerbau gelernt hab und eher Nachteile für Prof. Dr. Scholl, da mein eigenes Vorgehen entsprechend mehr Zeit brauchte und ich daher als Bachelorarbeit keinen dazu passenden RETI-Emulator mit Graphischer Anzeige implementieren konnte, da die restlichen Funktionalitäten des PicoC-Compilers noch implementiert werden mussten.

Glücklicherweise gibt es aber doch noch einen passenden RETI-Emulator, der den PicoC-Compiler über seine Kommandozeilenargumente aufruft, um ein PicoC-Programm visuell auf einer RETI-CPU auszuführen. Für dessen Implementierung hat sich Michel Giehl netterweise zur Verfügung gestellt. Daher Danke auch an Michel Giehl, dass er sich mit meinem PicoC-Compiler ausgeinandergesetzt hat und diesen in seinen RETI-Emulator integriert hat, sodass am Ende durch unsere beiden Arbeiten ein anschauliches Lerntool für die kommenden Studentengenerationen entstehen konnte. Vor allem da er auch mir darin vertrauen musste, dass ich mit meinem PicoC-Compiler nicht irgendeinen Misst baue. Der RETI-Emulator von Michel Giehl ist unter Link⁵ zu finden.

Mir hat die Implementierung des PicoC-Compilers tatsächlich ziemlich viel Spaß gemacht, da Compilerbau auch in mein perönliches Interessengebiet fällt⁶. Das Aufschreiben dieser Schrifftlichen Ausarbeitung hat mir dagegen eher weniger Spaß gemacht⁷. Wobei ich allerdings sagen muss, dass ich eine große Erleichterung verspüre das ganze Wissen über Compilerbau mal aufgeschrieben zu haben, damit ich mir keine Sorgen machen muss dieses ziemlich nützliche Wissen irgendwann wieder zu vergessen. Es hilft einem auch als Programmierer ungemein weiter zu wissen, wie ein Compiler unter der Haube funktioniert, da man sich so viel besser merken, wie eine bestimmte Funktionalität einer Programmiersprache zu verwenden ist. Manch eine Funktionalität einer Programmiesprache kann in der Verwendung ziemlich wilkürlich erscheinen, wenn man die technische Umsetzung dahinter im Compiler nicht kennt.

Ich wollte mich daher auch noch dafür Bedanken, dass mir ein so ergiebiges und interessantes Thema als Bachelorarbeit vorgeschlagen wurde und vor allem, dass auch das Vertrauen in mich gesteckt wurde, dass ich am Ende auch einen funktionsfähigen, sauber programmierten und gut durchdachten Compiler implementiere.

Zum Schluss nochmal ein abschließendes Danke an meinen Betreuer M.Sc Seufert und meinen Gutachter Prof. Dr. Scholl für die Betreuung und Bereitstellung dieser interessanten Bachelorarbeit und des vorangegangenen Bachelorprojektes und Michel Giehl für das Integrieren des PicoC-Compilers in seinen RETI-Emulator.

³Der PicoC-Compiler hätte schließlich mehr Funktionalitäten haben können.

⁴Vielleicht finde ich ja noch im nächsten Semester während des Betriebssysteme Tutorats noch etwas Zeit einige weitere Features einzubauen oder möglicherweise im Rahmen eines Masterprojektes ³.

 $^{^5}$ https://github.com/michel-giehl/Reti-Emulator.

⁶Womit nicht alle Studenten so viel Glück haben.

⁷Dieses ständige überlegen, wo man möglicherweise eine Erklärlücke hat, ob man nicht was wichtiges ausgelassen hat usw.

Inhaltsverzeichnis

Abbildungsv	erzeichn	is	Ι
Codeverzeich	nnis		II
Tabellenverz	eichnis		IV
Definitionsve	erzeichni	S	\mathbf{V}
Grammatikv	erzeichn	is	\mathbf{VI}
0.0.1	Umsetz	ung von Zeigern	1
	0.0.1.1	Referenzierung	
	0.0.1.2	Dereferenzierung durch Zugriff auf Feldindex ersetzen	
0.0.2	Umsetz	ung von Feldern	
	0.0.2.1	Initialisierung eines Feldes	
	0.0.2.2	Zugriff auf einen Feldindex	10
	0.0.2.3	Zuweisung an Feldindex	
0.0.3		ung von Verbunden	
	0.0.3.1	Deklaration von Verbundstypen und Definition von Verbunden	
	0.0.3.2	Initialisierung von Verbunden	
	0.0.3.3	Zugriff auf Verbundsattribut	
	0.0.3.4	Zuweisung an Verbundsattribut	
0.0.4		ung des Zugriffs auf Zusammengesetzte Datentypen im Allgemeinen	
	0.0.4.1	Anfangsteil	
	0.0.4.2	Mittelteil	
	0.0.4.3	Schlussteil	37
Literatur			Α

Abbildungsverzeichnis

1	Allgemeine	Veranschaulichung	des Zugriffs	auf Zusammengesetzte	Datentypen.	 29

Codeverzeichnis

0.1	PicoU-Code für Zeigerreierenzierung
0.2	Abstrakter Syntaxbaum für Zeigerreferenzierung
0.3	Symboltabelle für Zeigerreferenzierung
0.4	PicoC-ANF Pass für Zeigerreferenzierung
0.5	RETI-Blocks Pass für Zeigerreferenzierung.
0.6	PicoC-Code für Zeigerdereferenzierung.
0.7	Abstrakter Syntaxbaum für Zeigerdereferenzierung
0.8	PicoC-Shrink Pass für Zeigerdereferenzierung
0.9	PicoC-Code für die Initialisierung eines Feldes.
	Abstrakter Syntaxbaum für die Initialisierung eines Feldes.
0.11	Symboltabelle für die Initialisierung eines Feldes
	PicoC-ANF Pass für die Initialisierung eines Feldes
0.13	RETI-Blocks Pass für die Initialisierung eines Feldes
0.14	PicoC-Code für Zugriff auf einen Feldindex
0.15	Abstrakter Syntaxbaum für Zugriff auf einen Feldindex
0.16	PicoC-ANF Pass für Zugriff auf einen Feldindex
0.17	RETI-Blocks Pass für Zugriff auf einen Feldindex
0.18	PicoC-Code für Zuweisung an Feldindex
0.19	Abstrakter Syntaxbaum für Zuweisung an Feldindex
0.20	PicoC-ANF Pass für Zuweisung an Feldindex
0.21	RETI-Blocks Pass für Zuweisung an Feldindex
0.22	PicoC-Code für die Deklaration eines Verbundstyps
0.23	Abstrakter Syntaxbaum für die Deklaration eines Verbundstyps
0.24	Symboltabelle für die Deklaration eines Verbundstyps
	PicoC-Code für Initialisierung von Verbunden
	Abstrakter Syntaxbaum für Initialisierung von Verbunden
0.27	PicoC-ANF Pass für Initialisierung von Verbunden
0.28	RETI-Blocks Pass für Initialisierung von Verbunden
0.29	PicoC-Code für Zugriff auf Verbundsattribut
0.30	Abstrakter Syntaxbaum für Zugriff auf Verbundsattribut
0.31	PicoC-ANF Pass für Zugriff auf Verbundsattribut
0.32	RETI-Blocks Pass für Zugriff auf Verbundsattribut
0.33	PicoC-Code für Zuweisung an Verbundsattribut
	Abstrakter Syntaxbaum für Zuweisung an Verbundsattribut
0.35	PicoC-ANF Pass für Zuweisung an Verbundsattribut
0.36	RETI-Blocks Pass für Zuweisung an Verbndsattribut
0.37	PicoC-Code für den Anfangsteil
0.38	Abstrakter Syntaxbaum für den Anfangsteil
0.39	PicoC-ANF Pass für den Anfangsteil
0.40	RETI-Blocks Pass für den Anfangsteil
0.41	PicoC-Code für den Mittelteil
0.42	Abstrakter Syntaxbaum für den Mittelteil
	PicoC-ANF Pass für den Mittelteil
0.44	RETI-Blocks Pass für den Mittelteil
0.45	PicoC-Code für den Schlussteil
	Abstrakter Syntaxbaum für den Schlussteil
	PicoC-ANF Pass für den Schlussteil

Tabellenverzeichnis

1	Datensegment na	ach der I	nitialisierung	beider Felder.	 	 	 		 		

Definitionsverzeichnis

Grammatikverzeichnis

0.0.1 Umsetzung von Zeigern

Die Umsetzung von Zeigern ist in diesem Unterkapitel schnell erklärt, auch Dank eines kleinen Taschenspielertricks⁸. Hierbei sind nur die Operationen für Referenzierung und Dereferenzierung in den Unterkapiteln 0.0.1.1 und 0.0.1.2 zu erläutern. Referenzierung kann dazu genutzt werden einen Zeiger zu initialisieren und Dereferenzierung kann dazu genutzt werden, um auf diesen später zuzugreifen.

0.0.1.1 Referenzierung

Referenzierung (z.B. &var) ist eine Operation bei der ein Zeiger auf eine Location in Form der Anfangsadresse dieser Location als Ergebnis zurückgegeben wird. Die Implementierung der Referenzierung wird im Folgenden anhand des Beispiels in Code 0.1 erklärt.

```
1 void main() {
2  int var = 42;
3  int *pntr = &var;
4 }
```

Code 0.1: PicoC-Code für Zeigerreferenzierung.

Der Knoten Ref(Name('var'))) repräsentiert im Abstrakten Syntaxbaum in Code 0.2 eine Referenzierung &var und der Knoten PntrDecl(Num('1'), IntType('int')) repräsentiert einen Zeiger *pntr.

```
1
  File
2
    Name './example_pntr_ref.ast',
     {\tt FunDef}
         VoidType 'void',
6
7
8
         Name 'main',
         [],
9
           Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
10
           Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),

→ Ref(Name('var')))
11
    ]
```

Code 0.2: Abstrakter Syntaxbaum für Zeigerreferenzierung.

Bevor man einem Zeiger eine Adresse (z.B. &var) zuweisen kann, muss dieser erstmal definiert sein. Dafür braucht es einen Eintrag in der Symboltabelle in Code 0.3.

Anmerkung Q

Die Größe eines Zeigers (z.B. eines Zeigers auf ein Feld von int: pntr = int *pntr[3]), die im size-Attribut der Symboltabelle eingetragen ist, ist dabei immer: size(pntr) = 1.

⁸Später mehr dazu.

```
SymbolTable
     Γ
       Symbol
         {
                                     Empty()
           type qualifier:
                                     FunDecl(VoidType('void'), Name('main'), [])
           datatype:
           name:
                                     Name('main')
                                     Empty()
           value or address:
 9
                                     Pos(Num('1'), Num('5'))
           position:
10
                                     Empty()
           size:
11
         },
12
       Symbol
13
         {
14
           type qualifier:
                                     Writeable()
15
                                     IntType('int')
           datatype:
16
                                     Name('var@main')
           name:
17
                                     Num('0')
           value or address:
18
                                     Pos(Num('2'), Num('6'))
           position:
19
                                     Num('1')
           size:
20
         },
21
       Symbol
22
         {
23
                                     Writeable()
           type qualifier:
24
                                     PntrDecl(Num('1'), IntType('int'))
           datatype:
25
           name:
                                     Name('pntr@main')
26
           value or address:
                                     Num('1')
27
           position:
                                     Pos(Num('3'), Num('7'))
28
                                     Num('1')
           size:
29
         }
30
    ]
```

Code 0.3: Symboltabelle für Zeigerreferenzierung.

Im PicoC-ANF Pass in Code 0.4 wird der Knoten Ref(Name('var'))) durch die Knoten Ref(GlobalRead(Num('0'))) und Assign(GlobalWrite(Num('1')), Tmp(Num('1'))) ersetzt, deren Bedeutung in Unterkapitel ?? erklärt wurde. Im Fall, dass in Ref(exp)) das exp vielleicht nicht direkt ein Name('var') enthält und exp z.B. ein Subscr(Attr(Name('var'), Name('attr'))) ist, sind noch weitere Anweisungen zwischen den Zeilen 11 und 12 nötig, die sich in diesem Beispiel um das Übersetzen von Subscr(exp) und Attr(exp, name) nach dem Schema in Subkapitel 0.0.4.2 kümmern.

```
Name './example_pntr_ref.picoc_mon',
4
      Block
5
        Name 'main.0',
           // Assign(Name('var'), Num('42'))
8
           Exp(Num('42'))
9
           Assign(Global(Num('0')), Stack(Num('1')))
10
           // Assign(Name('pntr'), Ref(Name('var')))
11
           Ref(Global(Num('0')))
12
           Assign(Global(Num('1')), Stack(Num('1')))
13
          Return(Empty())
```

```
14 ]
15 ]
```

Code 0.4: PicoC-ANF Pass für Zeigerreferenzierung.

Im RETI-Blocks Pass in Code 0.5 werden die PicoC-Knoten Ref(Global(Num('0'))) und Assign(Global(Num('1')), Stack(Num('1'))) durch ihre semantisch entsprechenden RETI-Knoten ersetzt, deren Bedeutungen im ?? in Unterkapitel ?? erklärt sind.

```
1
  File
 2
    Name './example_pntr_ref.reti_blocks',
       Block
         Name 'main.0',
 7
8
9
           # // Assign(Name('var'), Num('42'))
           # Exp(Num('42'))
           SUBI SP 1;
10
           LOADI ACC 42;
11
           STOREIN SP ACC 1;
12
           # Assign(Global(Num('0')), Stack(Num('1')))
13
           LOADIN SP ACC 1;
14
           STOREIN DS ACC 0;
           ADDI SP 1;
           # // Assign(Name('pntr'), Ref(Name('var')))
16
17
           # Ref(Global(Num('0')))
18
           SUBI SP 1;
19
           LOADI IN1 0;
20
           ADD IN1 DS;
21
           STOREIN SP IN1 1;
22
           # Assign(Global(Num('1')), Stack(Num('1')))
23
           LOADIN SP ACC 1;
24
           STOREIN DS ACC 1;
25
           ADDI SP 1;
26
           # Return(Empty())
27
           LOADIN BAF PC -1;
28
         ]
29
    ]
```

Code 0.5: RETI-Blocks Pass für Zeigerreferenzierung.

0.0.1.2 Dereferenzierung durch Zugriff auf Feldindex ersetzen

Dereferenzierung (z.B. *var) ist eine Operation bei der einem Zeiger zur Location hin gefolgt wird, auf welche dieser zeigt und das Ergebnis z.B. der Inhalt der ersten Speicherzelle der referenzierten Location ist. Die Implementierung von Dereferenzierung wird im Folgenden anhand des Beispiels in Code 0.6 erklärt.

```
1 void main() {
2   int var = 42;
3   int *pntr = &var;
4 *pntr;
```

5 }

Code 0.6: PicoC-Code für Zeigerdereferenzierung.

Der Knoten Deref (Name ('var'), Num ('0'))) repräsentiert im Abstrakten Syntaxbaum in Code 0.7 eine Dereferenzierung *var. Es gibt hierbei zwei Fälle. Bei der Anwendung von Zeigerarithmetik, wie z.B. *(var + 2 - 1) übersetzt sich diese zu Deref (Name ('var'), Bin Op (Num ('2'), Sub(), Num ('1'))). Bei einer normalen Dereferenzierung, wie z.B. *var, übersetzt sich diese zu Deref (Name ('var'), Num ('0'))⁹.

```
File
1
    Name './example_pntr_deref.ast',
2
    Γ
4
      FunDef
        VoidType 'void',
        Name 'main',
7
8
         [],
         Γ
9
           Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
10
           Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),

→ Ref(Name('var')))
           Exp(Deref(Name('pntr'), Num('0')))
12
13
    ]
```

Code 0.7: Abstrakter Syntaxbaum für Zeigerdereferenzierung.

Im PicoC-Shrink Pass in Code 0.8 wird ein Trick angewandet, bei dem jeder Knoten Deref(Name('pntr'), Num('0')) einfach durch den Knoten Subscr(Name('pntr'), Num('0')) ersetzt wird. Die Bedeutung des letzteren Knoten wurde in Unterkapitel ?? erklärt. Der Trick besteht darin, dass der Dereferenzierungsoperator (z.B. *(var + 1)) sich identisch zum Operator für den Zugriff auf einen Feldindex (z.B. var[1]) verhält, wie es bereits im Unterkapitel ?? erläutert wurde. Damit spart man sich viele vermeidbare Fallunterscheidungen und doppelten Code und kann die Derefenzierung (z.B. *(var + 1)) einfach von den Routinen für einen Zugriff auf einen Feldindex (z.B. var[1]) übernehmen lassen.

```
Name './example_pntr_deref.picoc_shrink',
     Γ
      FunDef
         VoidType 'void',
        Name 'main',
         [],
8
9
           Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
10
           Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),

→ Ref(Name('var')))
           Exp(Subscr(Name('pntr'), Num('0')))
11
12
         ]
```

⁹Das Num('0') steht dafür, dass dem Zeiger gefolgt wird, aber danach nicht noch mit einem Versatz von der Größe des Datentyps in diesem Kontext auf eine nebenliegende Location zugegriffen wird.

3]

_

Code 0.8: PicoC-Shrink Pass für Zeigerdereferenzierung.

0.0.2 Umsetzung von Feldern

Bei Feldern ist in diesem Unterkapitel die Umsetzung der Innitialisierung eines Feldes 0.0.2.1, des Zugriffs auf einen Feldindex 0.0.2.2 und der Zuweisung an einen Feldindex 0.0.2.3 zu klären.

0.0.2.1 Initialisierung eines Feldes

Die Umsetzung der Initialisierung eines Feldes (z.B. int ar[2][1] = {{3+1}, {5}}) wird im Folgenden anhand des Beispiels in Code 0.9 erklärt.

```
void fun() {
  int ar[2][2] = {{3, 4}, {5, 6}};
}

void main() {
  int ar[2][1] = {{3+1}, {5}};
}
```

Code 0.9: PicoC-Code für die Initialisierung eines Feldes.

Die Initialisierung eines Feldes intar[2][1]={{3+1},{5}} wird im Abstrakten Syntaxbaum in Code 0.10 mithilfe der Komposition Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')), Name('ar')), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]), Array([Num('5')])])) dargestellt.

```
File
    Name './example_array_init.ast',
     Γ
       FunDef
         VoidType 'void',
         Name 'fun',
 7
8
         [],
         Γ
 9
           Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('2')], IntType('int')),
               Name('ar')), Array([Array([Num('3'), Num('4')]), Array([Num('5'), Num('6')])]))
10
         ],
11
       FunDef
12
         VoidType 'void',
13
         Name 'main',
14
         [],
15
           Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')),
16
           \rightarrow Name('ar')), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
               Array([Num('5')])))
         ]
18
    ]
```

Code 0.10: Abstrakter Syntaxbaum für die Initialisierung eines Feldes.

Bei der Initialisierung eines Feldes wird zuerst Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int'))) ausgewertet, da eine Variable zuerst definiert sein muss, bevor man sie verwenden kann¹⁰. Das Definieren der Variable ar erfolgt mittels der Symboltabelle, die in Code 0.11 dargestellt ist.

Auf dem Stackframe wird ein Feld verglichen zur Wachstumrichtung des Stacks rückwärts in den Stackframe geschrieben und die relative Adresse des ersten Elements als Adresse des Feldes in der Symboltabelle in Code 0.11 genommen. Dies ist in Tabelle 1 für ein Datensegment der Größe 8 dargstellt. Der Stack wächst zwar verglichen zu den Globalen Statischen Daten in die entgegengesetzte Richtung, aber Felder in den Globalen Statischen Daten und in einem Stackframe haben die gleiche Ausrichtung. Das macht den Zugriff auf einen Feldindex in Subkapitel 0.0.2.2 deutlich unkomplizierter. Auf diese Weise muss beim Zugriff auf einen Feldindex nicht zwischen Stackframe und Globalen Statischen Daten unterschieden werden.

Relativ- adresse	Wert
0	4
1	5
3	3
2	4
1	5
0	6

Tabelle 1: Datensegment nach der Initialisierung beider Felder.

Anmerkung Q

Das Größe des Feldes datatype $ar[dim_1]\dots[dim_k]$ berechnet sich aus der Mächtigkeit der einzelnen Dimensionen des Feldes multipliziert mit der Größe des grundlegenden Datentyps der einzelnen Feldelemente: $size(datatype(ar)) = \left(\prod_{j=1}^n dim_j\right) \cdot size(datatype)$.

^aDie im size-Attribut des Symboltabelleneintrags eingetragen ist.

```
SymbolTable
    Γ
      Symbol
           type qualifier:
           datatype:
                                    FunDecl(VoidType('void'), Name('fun'), [])
           name:
                                    Name('fun')
           value or address:
                                    Empty()
                                    Pos(Num('1'), Num('5'))
           position:
10
                                    Empty()
           size:
         },
```

 $[^]b$ Die Funktion type ordnet einer Variable ihren Datentyp zu. Das ist notwendig, weil die Funktion size nur bei einem Datentyp als Funktionsargument die Größe dieses Datentyps als Zielwert liefert

¹⁰Das widerspricht der üblichen Auswertungsreihenfolge beim Zuweisungsoperator =, der rechtsassoziativ ist. Der Zuweisungsoperator = tritt allerdings erst später in Aktion.

```
Symbol
13
         {
14
            type qualifier:
                                     Writeable()
15
           datatype:
                                     ArrayDecl([Num('2'), Num('2')], IntType('int'))
16
           name:
                                     Name('ar@fun')
17
                                     Num('3')
           value or address:
18
                                     Pos(Num('2'), Num('6'))
           position:
19
                                     Num('4')
           size:
20
         },
21
       Symbol
22
         {
23
           type qualifier:
                                     Empty()
24
                                     FunDecl(VoidType('void'), Name('main'), [])
           datatype:
25
                                     Name('main')
           name:
26
                                     Empty()
           value or address:
27
           position:
                                     Pos(Num('5'), Num('5'))
28
           size:
                                     Empty()
29
         },
30
       Symbol
31
         {
32
            type qualifier:
                                     Writeable()
33
                                     ArrayDecl([Num('2'), Num('1')], IntType('int'))
           datatype:
34
           name:
                                     Name('ar@main')
35
                                     Num('0')
           value or address:
36
                                     Pos(Num('6'), Num('6'))
           position:
37
           size:
                                     Num('2')
38
         }
39
     ]
```

Code 0.11: Symboltabelle für die Initialisierung eines Feldes.

Im PiocC-ANF Pass in Code 0.12 werden zuerst die Logischen Ausdrücke in den Blättern des Teilbaumes, der beim Feld-Initializers Knoten Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]), Array([Num('5')])]) anfängt nach dem Prinzip der Tiefensuche, von links-nach-rechts ausgewertet und auf den Stack geschrieben¹¹.

Im finalen Schritt muss zwischen Globalen Statischen Daten bei der main-Funktion und Stackframe bei der Funktion fun unterschieden werden. Die auf den Stack ausgewerteten Expressions werden mittels der Komposition Assign(Global(Num('0')), Stack(Num('2'))) bzw. Assign(Stackframe(Num('3')), Stack(Num('5'))), die in Tabelle ?? genauer beschrieben ist, versetzt in der selben Reihenfolge zu den Globalen Statischen Daten bzw. auf den Stackframe geschrieben.

Der Trick ist hier, dass egal wieviele Dimensionen und was für einen grundlegenden Datentyp¹² das Feld hat, man letztendlich immer das gesamte Feld erwischt, wenn man einfach die Größe des Feldes viele Speicherzellen mit z.B. der Komposition Assign(Global(Num('0')), Stack(Num('2'))) verschiebt.

In die Knoten Global ('0') und Stackframe ('3') wurde hierbei die Startadresse des jeweiligen Feldes geschrieben, sodass man nach dem PicoC-ANF Pass nie mehr Variablen in der Symboltabelle nachsehen muss und gleich weiß, ob sie in Bezug zu den Globalen Statischen Daten oder dem Stackframe stehen.

¹¹Da der Zuweisungsoperator = rechtsassoziativ ist und auch rein logisch, weil man nichts zuweisen kann, was man noch nicht berechnet hat.

¹²Z.B. ein Verbund, sodass es ein "Feld von Verbunden" ist.

```
1 File
    Name './example_array_init.picoc_mon',
     Ε
 4
       Block
         Name 'fun.1',
 6
 7
           // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
           → Num('6')])))
           Exp(Num('3'))
 9
           Exp(Num('4'))
10
           Exp(Num('5'))
11
           Exp(Num('6'))
12
           Assign(Stackframe(Num('3')), Stack(Num('4')))
           Return(Empty())
14
         ],
15
       Block
16
         Name 'main.0',
17
         Γ
           // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
18

    Array([Num('5')]))))

19
           Exp(Num('3'))
20
           Exp(Num('1'))
           Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
22
           Exp(Num('5'))
23
           Assign(Global(Num('0')), Stack(Num('2')))
24
           Return(Empty())
25
         ]
26
    ]
```

Code 0.12: PicoC-ANF Pass für die Initialisierung eines Feldes.

Im RETI-Blocks Pass in Code 0.13 werden die Kompositionen Exp(exp) und Assign(Global(Num('0')), Stack(Num('2'))) bzw. Assign(Stackframe(Num('3')), Stack(Num('5'))) durch ihre entsprechenden RETI-Knoten ersetzt.

```
1 File
    Name './example_array_init.reti_blocks',
    Ε
4
      Block
        Name 'fun.1',
6
           # // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
           → Num('6')])))
           # Exp(Num('3'))
9
           SUBI SP 1;
10
          LOADI ACC 3;
11
           STOREIN SP ACC 1;
12
           # Exp(Num('4'))
13
           SUBI SP 1;
14
          LOADI ACC 4;
           STOREIN SP ACC 1;
16
           # Exp(Num('5'))
           SUBI SP 1;
           LOADI ACC 5;
```

```
STOREIN SP ACC 1;
          # Exp(Num('6'))
20
21
          SUBI SP 1;
22
          LOADI ACC 6;
23
          STOREIN SP ACC 1;
24
          # Assign(Stackframe(Num('3')), Stack(Num('4')))
25
          LOADIN SP ACC 1;
26
          STOREIN BAF ACC -2;
27
          LOADIN SP ACC 2;
28
          STOREIN BAF ACC -3;
29
          LOADIN SP ACC 3;
30
          STOREIN BAF ACC -4;
          LOADIN SP ACC 4;
32
          STOREIN BAF ACC -5;
33
          ADDI SP 4;
34
          # Return(Empty())
35
          LOADIN BAF PC -1;
36
        ],
37
      Block
38
        Name 'main.0',
39
          # // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
40
           41
          # Exp(Num('3'))
42
          SUBI SP 1;
43
          LOADI ACC 3;
          STOREIN SP ACC 1;
44
45
          # Exp(Num('1'))
          SUBI SP 1;
46
47
          LOADI ACC 1;
48
          STOREIN SP ACC 1;
49
          # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
50
          LOADIN SP ACC 2;
51
          LOADIN SP IN2 1;
52
          ADD ACC IN2;
53
          STOREIN SP ACC 2;
54
          ADDI SP 1;
55
          # Exp(Num('5'))
56
          SUBI SP 1;
57
          LOADI ACC 5;
58
          STOREIN SP ACC 1;
59
          # Assign(Global(Num('0')), Stack(Num('2')))
60
          LOADIN SP ACC 1;
61
          STOREIN DS ACC 1;
62
          LOADIN SP ACC 2;
63
          STOREIN DS ACC 0;
64
          ADDI SP 2;
65
          # Return(Empty())
66
          LOADIN BAF PC -1;
67
    ]
```

Code 0.13: RETI-Blocks Pass für die Initialisierung eines Feldes.

0.0.2.2 Zugriff auf einen Feldindex

Der Zugriff auf einen Feldindex (z.B. ar[0]) wird im Folgenden anhand des Beispiels in Code 0.14 erklärt.

```
1 void main() {
2   int ar[1] = {42};
3   ar[0];
4 }
5
6 void fun() {
7   int ar[3] = {1, 2, 3};
8   ar[1+1];
9 }
```

Code 0.14: PicoC-Code für Zugriff auf einen Feldindex.

Der Zugriff auf einen Feldindex ar[0] wird im Abstrakten Syntaxbaum in Code 0.15 mithilfe des Knotens Subscr(Name('ar'), Num('0')) dargestellt.

```
File
 2
    Name './example_array_access.ast',
       FunDef
 5
6
7
8
         VoidType 'void',
         Name 'main',
         [],
 9
           Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('ar')),
           → Array([Num('42')]))
10
           Exp(Subscr(Name('ar'), Num('0')))
11
         ],
12
       FunDef
13
         VoidType 'void',
14
         Name 'fun',
15
         [],
16
17
           Assign(Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('ar')),

    Array([Num('1'), Num('2'), Num('3')]))

18
           Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
19
    ]
```

Code 0.15: Abstrakter Syntaxbaum für Zugriff auf einen Feldindex.

Im PicoC-ANF Pass in Code 0.16 wird vom Knoten Subscr(Name('ar'), Num('0')) zuerst im Anfangsteil 0.0.4.1 die Adresse der Variable Name('ar') auf den Stack geschrieben. Bei den Globalen Statischen Daten der main-Funktion wird das durch die Komposition Ref(Global(Num('0'))) dargestellt und beim Stackframe der Funktionm fun wird das durch die Komposition Ref(Stackframe(Num('2'))) dargestellt.

In nächsten Schritt, dem Mittelteil 0.0.4.2 wird die Adresse ab der das Feldelement des Feldes auf das Zugegriffen werden soll anfängt berechnet. Dabei wurde im Anfangsteil bereits die Anfangsadresse des

Feldes, in dem dieses Feldelement liegt auf den Stack gelegt. Da ein Index auf den Zugegriffen werden soll auch durch das Ergebnis eines komplexeren Ausdrucks, z.B. ar[1 + var] bestimmt sein kann, indem auch Variablen vorkommen können, kann dieser nicht während des Kompilierens berechnet werden, sondern muss zur Laufzeit berechnet werden.

Daher muss zuerst der Wert des Index, dessen Adresse berechnet werden soll bestimmt werden, z.B. im einfachen Fall durch Exp(Num('0')) und dann muss die Adresse des Index berechnet werden, was durch die Komposition Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) dargestellt wird. Die Bedeutung der Komposition Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) ist in Tabelle ?? dokumentiert.

Zur Adressberechnung ist es notwendig auf die Dimensionen (z.B. [Num('3')]) des Feldes, auf dessen Feldelement zugegriffen wird, zugreifen zu können. Daher ist der Felddatentyp (z.B. ArrayDecl([Num('3')], IntType('int'))) dem Knoten Ref(exp, datatype) als verstecktes Attribut datatype angehängt. Das versteckte Attribut wird während des Kompiliervorgangs im PiocC-ANF Pass dem Knoten Ref(exp, datatype) angehängt.

Je nachdem, ob mehrere Subscr(exp, exp) eine Komposition bilden (z.B. Subscr(Subscr(Name('var'), Num('1')), Num('1'))) ist es notwendig mehrere Adressberechnungsschritte für den Index Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) einzuleiten und es muss auch möglich sein, z.B. einen Attributzugriff var.attr und eine Zugriff auf einen Arryindex var[1] miteinander zu kombinieren, was in Subkapitel 0.0.4.2 allgemein erklärt ist.

Im letzten Schritt, dem Schlussteil 0.0.4.3 wird der Inhalt des Index, dessen Adresse in den vorherigen Schritten berechnet wurde, nun auf den Stack geschrieben, wobei dieser die Adresse auf dem Stack ersetzt, die es zum Finden des Index brauchte. Dies wird durch den Knoten Exp(Stack(Num('1'))) dargestellt. Je nachdem, welchen Datentyp die Variable ar hat und auf welchen Unterdatentyp folglich im Kontext zuletzt zugegriffen wird, abhängig davon wird der Schlussteil Exp(Stack(Num('1'))) auf eine andere Weise verarbeitet (siehe Subkapitel 0.0.4.3). Der Unterdatentyp ist dabei ein verstecktes Attribut des Exp(Stack(Num('1')))-Knoten.

Der einzige Unterschied, je nachdem, ob der Zugriff auf einen Feldindex (z.B. ar[1]) in der main-Funktion oder der Funktion fun erfolgt, ist eigentlich nur beim Anfangsteil beim Schreiben der Adresse der Variable ar auf den Stack zu finden, bei dem unterschiedliche RETI-Befehle für eine Variable, die in den Globalen Statischen Daten liegt und eine Variable, die auf dem Stackframe liegt erzeugt werden müssen.

Anmerkung Q

Die Berechnung der Adresse, ab der ein Feldelement eines Feldes datatype $ar[dim_1]...[dim_n]$ abgespeichert ist, kann mittels der Formel 0.0.1:

$$\texttt{ref}(\texttt{ar}[\texttt{idx}_1] \dots [\texttt{idx}_n]) = \texttt{ref}(\texttt{ar}) + \left(\sum_{\texttt{i}=1}^n \left(\prod_{\texttt{j}=\texttt{i}+1}^n \texttt{dim}_\texttt{j}\right) \cdot \texttt{idx}_\texttt{i}\right) \cdot \texttt{size}(\texttt{datatype}) \tag{0.0.1}$$

aus der Betriebssysteme Vorlesung Scholl, "Betriebssysteme" berechnet werden^a.

Die Komposition Ref(Global(num)) bzw. Ref(Stackframe(num)) repräsentiert dabei den Summanden ref(ar) in der Formel.

Die Komposition Exp(num) repräsentiert dabei einen Subindex (z.B. i in a[i][j][k]) beim Zugriff auf ein Feldelement, der als Faktor idx_i in der Formel auftaucht.

Der Komposition Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) repräsentiert dabei einen ausmultiplizierten Summanden $\left(\prod_{j=i+1}^n \dim_j\right) \cdot \mathrm{idx}_i \cdot \mathrm{size}(\mathrm{datatpye})$ in der Formel.

```
Die Komposition \operatorname{Exp}(\operatorname{Stack}(\operatorname{Num}('1'))) repräsentiert dabei das Lesen des Inhalts \operatorname{M}[\operatorname{ref}(\operatorname{ar}[\operatorname{idx}_1]\dots[\operatorname{idx}_n])] der Speicherzelle an der finalen Adresse \operatorname{ref}(\operatorname{ar}[\operatorname{idx}_1]\dots[\operatorname{idx}_n]).

\operatorname{aref}(\operatorname{exp}) steht dabei für die Berechnung der Adresse von \operatorname{exp}, wobei \operatorname{exp} z.B. \operatorname{ar}[3][2] sein könnte.
```

```
File
    Name './example_array_access.picoc_mon',
       Block
         Name 'main.1',
 7
8
9
           // Assign(Name('ar'), Array([Num('42')]))
           Exp(Num('42'))
           Assign(Global(Num('0')), Stack(Num('1')))
10
           // Exp(Subscr(Name('ar'), Num('0')))
11
           Ref(Global(Num('0')))
12
           Exp(Num('0'))
13
           Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
14
           Exp(Stack(Num('1')))
15
           Return(Empty())
16
         ],
       Block
18
         Name 'fun.0',
19
20
           // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
           Exp(Num('1'))
22
           Exp(Num('2'))
23
           Exp(Num('3'))
24
           Assign(Stackframe(Num('2')), Stack(Num('3')))
25
           // Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
26
           Ref(Stackframe(Num('2')))
27
           Exp(Num('1'))
28
           Exp(Num('1'))
29
           Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
30
           Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
31
           Exp(Stack(Num('1')))
32
           Return(Empty())
33
         ]
```

Code 0.16: PicoC-ANF Pass für Zugriff auf einen Feldindex.

Im RETI-Blocks Pass in Code 0.17 werden die Kompositionen Ref(Global(Num('0'))), Ref(Subscr(Stack(Num('2')) und Stack(Num('1')))) durch ihre entsprechenden RETI-Knoten ersetzt.

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

```
# // Assign(Name('ar'), Array([Num('42')]))
           # Exp(Num('42'))
9
           SUBI SP 1;
10
           LOADI ACC 42;
11
           STOREIN SP ACC 1;
12
           # Assign(Global(Num('0')), Stack(Num('1')))
13
           LOADIN SP ACC 1;
14
           STOREIN DS ACC 0;
15
           ADDI SP 1;
16
           # // Exp(Subscr(Name('ar'), Num('0')))
17
           # Ref(Global(Num('0')))
18
           SUBI SP 1;
19
           LOADI IN1 0;
           ADD IN1 DS;
           STOREIN SP IN1 1;
21
22
           # Exp(Num('0'))
23
           SUBI SP 1;
24
           LOADI ACC 0;
25
           STOREIN SP ACC 1;
26
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
27
           LOADIN SP IN1 2;
28
           LOADIN SP IN2 1;
29
           MULTI IN2 1;
30
           ADD IN1 IN2;
31
           ADDI SP 1;
32
           STOREIN SP IN1 1;
33
           # Exp(Stack(Num('1')))
34
           LOADIN SP IN1 1;
35
           LOADIN IN1 ACC O;
36
           STOREIN SP ACC 1;
37
           # Return(Empty())
38
           LOADIN BAF PC -1;
39
        ],
40
       Block
41
         Name 'fun.0',
42
43
           # // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
44
           # Exp(Num('1'))
45
           SUBI SP 1;
           LOADI ACC 1;
46
47
           STOREIN SP ACC 1;
48
           # Exp(Num('2'))
49
           SUBI SP 1;
50
           LOADI ACC 2;
51
           STOREIN SP ACC 1;
           # Exp(Num('3'))
52
53
           SUBI SP 1;
54
           LOADI ACC 3;
55
           STOREIN SP ACC 1;
56
           # Assign(Stackframe(Num('2')), Stack(Num('3')))
57
           LOADIN SP ACC 1;
58
           STOREIN BAF ACC -2;
59
           LOADIN SP ACC 2;
60
           STOREIN BAF ACC -3;
61
           LOADIN SP ACC 3;
62
           STOREIN BAF ACC -4;
           ADDI SP 3;
```

```
# // Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
           # Ref(Stackframe(Num('2')))
65
66
           SUBI SP 1;
67
           MOVE BAF IN1;
68
           SUBI IN1 4;
69
           STOREIN SP IN1 1;
70
           # Exp(Num('1'))
71
           SUBI SP 1;
72
           LOADI ACC 1;
73
           STOREIN SP ACC 1;
74
           # Exp(Num('1'))
75
           SUBI SP 1;
76
           LOADI ACC 1;
77
           STOREIN SP ACC 1;
           # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
78
79
           LOADIN SP ACC 2;
80
           LOADIN SP IN2 1;
81
           ADD ACC IN2;
82
           STOREIN SP ACC 2;
83
           ADDI SP 1;
84
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
85
           LOADIN SP IN1 2;
86
           LOADIN SP IN2 1;
87
           MULTI IN2 1;
88
           ADD IN1 IN2;
89
           ADDI SP 1;
90
           STOREIN SP IN1 1;
91
           # Exp(Stack(Num('1')))
92
           LOADIN SP IN1 1;
93
           LOADIN IN1 ACC 0;
           STOREIN SP ACC 1;
95
           # Return(Empty())
96
           LOADIN BAF PC -1;
97
         ]
98
    ]
```

Code 0.17: RETI-Blocks Pass für Zugriff auf einen Feldindex.

0.0.2.3 Zuweisung an Feldindex

Die Zuweisung eines Wertes an einen Feldindex (z.B. ar[2] = 42;) wird im Folgenden anhand des Beispiels in Code 0.18 erläutert.

```
1 void main() {
2  int ar[2];
3  ar[1] = 42;
4 }
```

Code 0.18: PicoC-Code für Zuweisung an Feldindex.

Im Abstrakten Syntaxbaum in Code 0.19 wird eine Zuweisung an einen Feldindex ar[2] = 42; durch die Komposition Assign(Subscr(Name('ar'), Num('2')), Num('42')) dargestellt.

```
File
Name './example_array_assignment.ast',

[
FunDef
VoidType 'void',
Name 'main',
[],
[],
[]
Exp(Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
Assign(Subscr(Name('ar'), Num('1')), Num('42'))

[]
]
```

Code 0.19: Abstrakter Syntaxbaum für Zuweisung an Feldindex.

Im PicoC-ANF Pass in Code 0.20 wird zuerst die rechte Seite des rechtsassoziativen Zuweisungsoperators =, bzw. des Knotens der diesen darstellt ausgewertet: Exp(Num('42')).

Danach ist das Vorgehen, bzw. sind die Kompostionen, die dieses darauffolgende Vorgehen darstellen: Ref(Global(Num('0'))), Exp(Num('2')) und Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) identisch zum Anfangsteil und Mittelteil aus dem vorherigen Subkapitel 0.0.2.2. Es wird die Adresse des Index, dem das Ergebnis der Ausdrucks auf der rechten Seite des Zuweisungsoperators = zugewiesen wird berechet, wie in Subkapitel 0.0.2.2.

Zum Schluss stellt die Komposition Assign(Stack(Num('1')), Stack(Num('2')))¹³ die Zuweisung = des Ergebnisses des Ausdrucks auf der rechten Seite der Zuweisung zum Feldindex, dessen Adresse im Schritt danach berechnet wurde dar.

```
Name './example_array_assignment.picoc_mon',
 4
5
       Block
         Name 'main.0',
           // Assign(Subscr(Name('ar'), Num('1')), Num('42'))
 8
           Exp(Num('42'))
 9
           Ref(Global(Num('0')))
10
           Exp(Num('1'))
11
           Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
12
           Assign(Stack(Num('1')), Stack(Num('2')))
13
           Return(Empty())
14
         ]
15
    ]
```

Code 0.20: PicoC-ANF Pass für Zuweisung an Feldindex.

Im RETI-Blocks Pass in Code 0.21 werden die Kompositionen Ref(Global(Num('0'))), Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) und Assign(Stack(Num('1')), Stack(Num('2'))) durch ihre entsprechenden RETI-Knoten ersetzt.

¹³Ist in Tabelle ?? genauer beschrieben ist

```
Name './example_array_assignment.reti_blocks',
       Block
         Name 'main.0',
 6
7
8
9
           # // Assign(Subscr(Name('ar'), Num('1')), Num('42'))
           # Exp(Num('42'))
           SUBI SP 1;
10
           LOADI ACC 42;
11
           STOREIN SP ACC 1;
12
           # Ref(Global(Num('0')))
13
           SUBI SP 1;
14
           LOADI IN1 0;
15
           ADD IN1 DS;
16
           STOREIN SP IN1 1;
           # Exp(Num('1'))
18
           SUBI SP 1;
19
           LOADI ACC 1;
20
           STOREIN SP ACC 1;
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
           LOADIN SP IN1 2;
23
           LOADIN SP IN2 1;
24
           MULTI IN2 1;
25
           ADD IN1 IN2;
26
           ADDI SP 1;
27
           STOREIN SP IN1 1;
28
           # Assign(Stack(Num('1')), Stack(Num('2')))
29
           LOADIN SP IN1 1;
30
           LOADIN SP ACC 2;
31
           ADDI SP 2;
32
           STOREIN IN1 ACC 0;
33
           # Return(Empty())
34
           LOADIN BAF PC -1;
         ]
36
    ]
```

Code 0.21: RETI-Blocks Pass für Zuweisung an Feldindex.

0.0.3 Umsetzung von Verbunden

Bei Verbunden wird in diesem Unterkapitel zunächst geklärt, wie die **Deklaration von Verbundstypen** umgesetzt ist. Ist ein Verbundstyp deklariert, kann damit einhergehend ein Verbund mit diesem Verbundstyp definiert werden. Die Umsetzung von beidem wird in Unterkapitel 0.0.3.1 erläutert. Des Weiteren ist die Umsetzung der Innitialisierung eines Verbundes 0.0.3.2, des **Zugriffs auf ein Verbundsattribut** 0.0.3.3, der **Zuweisung an ein Verbundsattribut** 0.0.3.4 zu klären.

0.0.3.1 Deklaration von Verbundstypen und Definition von Verbunden

Die Deklaration (Definition ??) eines neuen Verbundstyps (z.B. struct st {int len; int ar[2];}) und die Definition (Definition ??) eines Verbundes mit diesem Verbundstyp (z.B. struct st st_var;) wird im Folgenden anhand des Beispiels in Code 0.22 erläutert.

```
1 struct st {int len; int ar[2];};
2
3 void main() {
4    struct st st_var;
5 }
```

Code 0.22: PicoC-Code für die Deklaration eines Verbundstyps.

Bevor irgendwas definiert werden kann, muss erstmal ein Verbundstyp deklariert werden. Im Abstrakten Syntaxbaum in Code 0.24 wird die Deklaration eines Verbundstyps struct st {int len; int ar[2];} durch die Komposition StructDecl(Name('st'), [Alloc(Writeable(), IntType('int'), Name('len')) Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))]) dargestellt.

Die **Definition** einer Variable mit diesem **Verbundstyp** struct st st_var; wird durch die Komposition Alloc(Writeable(), StructSpec(Name('st')), Name('st_var')) dargestellt.

```
1 File
    Name './example_struct_decl_def.ast',
       StructDecl
         Name 'st',
           Alloc(Writeable(), IntType('int'), Name('len'))
           Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))
 9
         ],
10
       {\tt FunDef}
11
         VoidType 'void',
12
         Name 'main',
13
         [],
14
         Γ
           Exp(Alloc(Writeable(), StructSpec(Name('st')), Name('st_var')))
16
         ]
    ]
```

Code 0.23: Abstrakter Syntaxbaum für die Deklaration eines Verbundstyps.

Für den Verbundstyp selbst wird in der Symboltabelle, die in Code 0.24 dargestellt ist ein Eintrag mit dem Schlüssel st erstellt. Die Attribute dieses Symbols type_qualifier, datatype, name, position und size sind

wie üblich belegt, allerdings sind in dem value_address-Attribut des Symbols die Attribute des Verbundstyps [Name('len@st'), Name('ar@st')] aufgelistet, sodass man über den Verbundstyp st die Attribute des Verbundstyps in der Symboltabelle nachschlagen kann. Die Schlüssel der Attribute haben einen Suffix @st angehängt, der eine Art Sichtbarkeitsbereich innerhalb des Verbundtyps für seine Attribut darstellt. Es gilt foglich, dass innerhalb eines Verbundstyps zwei Attribute nicht gleich benannt werden können, aber dafür zwei unterschiedliche Verbundstypen ihre Attribute gleich benennen können.

Jedes der Attribute [Name('len@st'), Name('ar@st')] erhält auch einen eigenen Eintrag in der Symboltabelle, wobei die Attribute type_qualifier, datatype, name, value_address, position und size wie üblich belegt werden. Die Attribute type_qualifier, datatype und name werden z.B. bei Name('ar@st') mithilfe der Attribute von Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))]) belegt.

Für die Definition einer Variable st_var@main mit diesem Verbundstyp st wird ein Eintrag in der Symboltabelle angelegt. Das datatyp-Attribut enthält dabei den Namen des Verbundstyps als Komposition StructSpec(Name('st')), wodurch jederzeit alle wichtigen Informationen zu diesem Verbundstyp¹⁴ und seinen Attributen in der Symboltabelle nachgeschlagen werden können.

Anmerkung Q

Die Größe einer Variable st_var, die ihm size-Attribut des Symboltabelleneintrags eingetragen ist und mit dem Verbundstyp struct st {datatype₁ attr₁; ... datatype_n attr_n; }; definiert ist (struct st st_var;), berechnet sich dabei aus der Summe der Größen der einzelnen Datentypen datatype₁ ... datatype_n der Attribute attr₁, ... attr_n des Verbundstyps: size(st) = $\sum_{i=1}^{n}$ size(datatype_i).

^aHier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache L_{PicoC} nicht die Fragwürdige Designentscheidung, auch die eckigen Klammern [] für die Definition eines Feldes vor die Variable zu schreiben von Lc übernommen. Es wird so getann, als würde der komplette Datentyp immer hinter der Variable stehen: datatype var.

```
SymbolTable
 2
     Ε
       Symbol
         {
           type qualifier:
                                      Empty()
 6
7
8
9
           datatype:
                                      IntType('int')
           name:
                                      Name('len@st')
                                      Empty()
           value or address:
                                      Pos(Num('1'), Num('15'))
           position:
10
                                      Num('1')
           size:
11
         },
12
       Symbol
13
         {
14
           type qualifier:
15
                                      ArrayDecl([Num('2')], IntType('int'))
           datatype:
                                      Name('ar@st')
16
           name:
17
                                      Empty()
           value or address:
18
           position:
                                      Pos(Num('1'), Num('24'))
19
           size:
                                      Num('2')
20
         },
21
       Symbol
22
         {
            type qualifier:
                                      Empty()
```

 $^{^{14}}$ Wie z.B. vor allem die Größe bzw. Anzahl an Speicherzellen, die dieser Verbundstyp einnimmt.

```
StructDecl(Name('st'), [Alloc(Writeable(), IntType('int'),
           datatype:
               Name('len'))Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')),
               Name('ar'))])
                                    Name('st')
                                    [Name('len@st'), Name('ar@st')]
           value or address:
27
           position:
                                    Pos(Num('1'), Num('7'))
28
           size:
                                    Num('3')
29
         },
30
       Symbol
31
32
           type qualifier:
                                    Empty()
33
                                    FunDecl(VoidType('void'), Name('main'), [])
           datatype:
34
                                    Name('main')
           name:
35
                                    Empty()
           value or address:
36
                                    Pos(Num('3'), Num('5'))
           position:
37
           size:
                                    Empty()
38
         },
39
       Symbol
40
         {
41
                                    Writeable()
           type qualifier:
42
                                    StructSpec(Name('st'))
           datatype:
43
                                    Name('st_var@main')
44
                                    Num('0')
           value or address:
45
                                    Pos(Num('4'), Num('12'))
           position:
46
                                    Num('3')
           size:
47
    ]
```

Code 0.24: Symboltabelle für die Deklaration eines Verbundstyps.

0.0.3.2 Initialisierung von Verbunden

Die Initialisierung eines Verbundes wird im Folgenden mithilfe des Beispiels in Code 0.25 erklärt.

```
1 struct st1 {int *attr[2];};
2
3 struct st2 {int attr1; struct st1 attr2;};
4
5 void main() {
6  int var = 42;
7  struct st2 st = {.attr1=var, .attr2={.attr={&var, &var}}};
8}
```

Code 0.25: PicoC-Code für Initialisierung von Verbunden.

Im Abstrakten Syntaxbaum in Code 0.26 wird die Initialisierung eines Verbundes struct st1 st = {.attr1=var, .attr2={.attr={{&var, &var}}}} mithilfe der Komposition Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')), Struct(...)) dargestellt.

```
1 File
2 Name './example_struct_init.ast',
```

```
4
      StructDecl
        Name 'st1',
6
           Alloc(Writeable(), ArrayDecl([Num('2')], PntrDecl(Num('1'), IntType('int'))),
           → Name('attr'))
8
        ],
9
      StructDecl
10
        Name 'st2',
11
12
           Alloc(Writeable(), IntType('int'), Name('attr1'))
13
           Alloc(Writeable(), StructSpec(Name('st1')), Name('attr2'))
14
        ],
15
      FunDef
16
         VoidType 'void',
17
         Name 'main',
18
         [],
19
20
           Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
           Assign(Alloc(Writeable(), StructSpec(Name('st2')), Name('st')),
21

    Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'),
               Struct([Assign(Name('attr'), Array([Ref(Name('var')), Ref(Name('var'))]))])))
23
    ]
```

Code 0.26: Abstrakter Syntaxbaum für Initialisierung von Verbunden.

Im PicoC-ANF Pass in Code 0.27 wird die Komposition Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')), Struct(...)) auf fast dieselbe Weise ausgewertet, wie bei der Initialisierung eines Feldes in Subkapitel 0.0.2.1 daher wird um keine Wiederholung zu betreiben auf Subkapitel 0.0.2.1 verwiesen. Um das ganze interressanter zu gestalten wurde das Beispiel in Code 0.25 so gewählt, dass sich daran eine komplexere, mehrstufige Initialisierung mit verschiedenen Datentypen erklären lässt.

Der Verbund-Initializer Teilbaum Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Ref(Name('var')), Ref(Name('var'))])]))])), der beim Verbund-Initializer Knoten anfängt, wird auf dieselbe Weise nach dem Prinzip der Tiefensuche von links-nach-rechts ausgewertet, wie es bei der Initialisierung eines Feldes in Subkapitel 0.0.2.1 bereits erklärt wurde.

Beim Iterieren über den Teilbaum, muss beim Verbund-Initializer nur beachtet werden, dass bei den Assign(lhs, exp)-Knoten, über welche die Attributzuweisung dargestellt wird (z.B. Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Array([Ref(Name('var')), Ref(Name('var'))])]))))) der Teilbaum beim rechten exp Attribut weitergeht.

Im Allgemeinen gibt es beim Initialisieren eines Feldes oder Verbundes im Teilbaum auf der rechten Seite, der beim jeweiligen obersten Initializer anfängt immer nur 3 Fällte, man hat es auf der rechten Seite entweder mit einem Verbund-Initialiser, einem Feld-Initialiser oder einem Logischen Ausdruck zu tuen. Bei Feld- und Verbund-Initialisier wird einfach über diese nach dem Prinzip der Tiefensuche von links-nach-rechts iteriert und die Ergebnisse der Logischen Ausdrücken in den Blättern auf den Stack gespeichert. Der Fall, dass ein Logischer Ausdruck vorliegt erübrigt sich damit.

```
Name './example_struct_init.picoc_mon',
4
      Block
        Name 'main.0',
6
7
8
           // Assign(Name('var'), Num('42'))
           Exp(Num('42'))
9
           Assign(Global(Num('0')), Stack(Num('1')))
           // Assign(Name('st'), Struct([Assign(Name('attr1'), Name('var')),
10
           → Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Ref(Name('var')),
              Ref(Name('var'))]))])))
           Exp(Global(Num('0')))
11
12
           Ref(Global(Num('0')))
13
           Ref(Global(Num('0')))
14
           Assign(Global(Num('1')), Stack(Num('3')))
15
           Return(Empty())
16
17
    ]
```

Code 0.27: Pico C-ANF Pass für Initialisierung von Verbunden.

Im RETI-Blocks Pass in Code 0.28 werden die Kompositionen Exp(exp), Ref(exp) und Assign(Global(Num('1')), Stack(Num('3'))) durch ihre entsprechenden RETI-Knoten ersetzt.

```
1 File
    Name './example_struct_init.reti_blocks',
       Block
         Name 'main.0',
 7
8
           # // Assign(Name('var'), Num('42'))
           # Exp(Num('42'))
 9
           SUBI SP 1;
10
           LOADI ACC 42;
11
           STOREIN SP ACC 1;
12
           # Assign(Global(Num('0')), Stack(Num('1')))
13
           LOADIN SP ACC 1;
14
           STOREIN DS ACC 0;
15
           ADDI SP 1;
16
           # // Assign(Name('st'), Struct([Assign(Name('attr1'), Name('var')),
           → Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Ref(Name('var')),

→ Ref(Name('var'))]))]))))))))
17
           # Exp(Global(Num('0')))
18
           SUBI SP 1;
19
           LOADIN DS ACC 0;
20
           STOREIN SP ACC 1;
21
           # Ref(Global(Num('0')))
22
           SUBI SP 1;
23
           LOADI IN1 0;
24
           ADD IN1 DS;
25
           STOREIN SP IN1 1;
26
           # Ref(Global(Num('0')))
           SUBI SP 1;
```

```
LOADI IN1 0;
29
           ADD IN1 DS;
           STOREIN SP IN1 1;
30
31
           # Assign(Global(Num('1')), Stack(Num('3')))
32
           LOADIN SP ACC 1;
33
           STOREIN DS ACC 3;
34
           LOADIN SP ACC 2;
35
           STOREIN DS ACC 2;
36
           LOADIN SP ACC 3;
37
           STOREIN DS ACC 1;
38
           ADDI SP 3;
39
           # Return(Empty())
40
           LOADIN BAF PC -1;
41
         ]
42
    ]
```

Code 0.28: RETI-Blocks Pass für Initialisierung von Verbunden.

0.0.3.3 Zugriff auf Verbundsattribut

Der Zugriff auf ein Verbundsattribut (z.B. st.y) wird im Folgenden mithilfe des Beispiels in Code 0.29 erklärt.

```
1 struct pos {int x; int y;};
2
3 void main() {
4   struct pos st = {.x=4, .y=2};
5   st.y;
6 }
```

Code 0.29: PicoC-Code für Zugriff auf Verbundsattribut.

Im Abstrakten Syntaxbaum in Code 0.30 wird der Zugriff auf ein Verbundsattribut st.y mithilfe der Komposition Exp(Attr(Name('st'), Name('y'))) dargestellt.

```
File
    Name './example_struct_attr_access.ast',
 4
5
       StructDecl
         Name 'pos',
6
7
8
9
           Alloc(Writeable(), IntType('int'), Name('x'))
           Alloc(Writeable(), IntType('int'), Name('y'))
         ],
       FunDef
         VoidType 'void',
11
12
         Name 'main',
         [],
14
         [
15
           Assign(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),

    Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
```

```
16 Exp(Attr(Name('st'), Name('y')))
17 ]
18 ]
```

Code 0.30: Abstrakter Syntaxbaum für Zugriff auf Verbundsattribut.

Im PicoC-ANF Pass in Code 0.31 wird die Komposition Exp(Attr(Name('st'), Name('y'))) auf ähnliche Weise ausgewertet, wie die Komposition, die einen Zugriff auf ein Feldelement Exp(Subscr(Name('ar'), Num('0'))) in Subkapitel 0.0.2.2 darstellt. Daher wird hier, um Wiederholung zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsnten auf das Subkapitel 0.0.2.2 verwiesen.

Die Komposition Exp(Attr(Name('st'), Name('y'))) wird genauso, wie in Subkapitel 0.0.2.2 durch Kompositionen ersetzt, die sich in Anfangsteil 0.0.4.1, Mittelteil 0.0.4.2 und Schlussteil 0.0.4.3 aufteilen lassen. In diesem Fall sind es Ref(Global(Num('0'))) (Anfangsteil), Ref(Attr(Stack(Num('1')), Name('y'))) (Mittelteil) und Exp(Stack(Num('1'))) (Schlussteil). Der Anfangsteil und Schlussteil sind genau gleich, wie in Subkapitel 0.0.2.2.

Nur für den Mittelteil wird eine andere Komposition Ref(Attr(Stack(Num('1')), Name('y'))) gebraucht. Diese Komposition Ref(Attr(Stack(Num('1')), Name('y'))) erfüllt die Aufgabe die Adresse, ab der das Attribut auf das zugegriffen wird anfängt zu berechnen. Dabei wurde die Anfangsadresse des Verbundes indem dieses Attribut liegt bereits vorher auf den Stack gelegt.

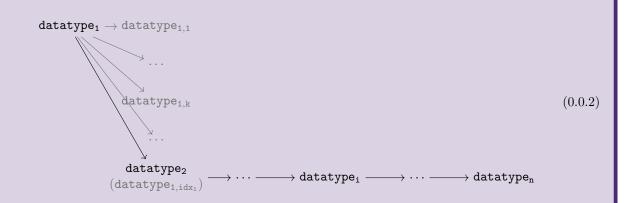
Im Gegensatz zur Komposition Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) beim Zugriff auf einen Feldindex in Subkapitel 0.0.2.2, muss hier vorher nichts anderes als die Anfangsadresse des Verbundes auf dem Stack liegen. Das Verbundsattribut auf welches zugegriffen wird steht bereits in der Komposition Ref(Attr(Stack(Num('1')), Name('y'))), nämlich Name('y'). Den Verbundstyp, dem dieses Attribut gehört, kann man aus dem versteckten Attribut datatype herauslesen. Das versteckte Attribut wird während des Kompiliervorgangs im PiocC-ANF Pass dem Knoten Ref(exp, datatype) angehängt.

Anmerkung Q

Sei datatype_i ein Knoten eines entarteten Baumes (siehe Definition 0.1 und Abbildung 0.0.2), dessen Wurzel datatype_i ist. Dabei steht i für eine Ebene des entarteten Baumes. Die Knoten des entarteten Baumes lassen sich Startadressen ref(datatype_i) von Speicherbereichen ref(datatype_i) ... ref(datatype_i) + size(datatype_i) im Hauptspeicher zuordnen, wobei gilt, dass ref(datatype_i) \leq ref(datatype_{i+1}) \leq ref(datatype_{i+1}) \leq ref(datatype_i).

Sei datatype_{i,k} ein beliebiges Element / Attribut des Datentyps datatype_i. Dabei gilt: $ref(datatype_{i,k}) < ref(datatype_{i,k+1})$.

Sei datatype_{i,idx_i} ein beliebiges Element / Attribut des Datentyps datatype_i, sodass gilt: datatype_{i,idx_i} = datatype_{i+1}.



Die Berechnung der Adresse für eine beliebige Folge verschiedener Datentypen ($\mathtt{datatype_{1,idx_1}}, \ldots, \mathtt{datatype_{n,idx_n}}$), die das Resultat einer Aneinandereihung von **Zugriffen** auf **Zeigerelemente**, **Feldelemente** und **Verbundsattributte** unterschiedlicher Datentypen $\mathtt{datatype_i}$ ist (z.B. *complex_var.attr3[2]), kann mittels der Formel 0.0.3:

$$\texttt{ref}(\texttt{datatype}_{\texttt{1},\texttt{idx}_1}, \ \dots, \ \texttt{datatype}_{\texttt{n},\texttt{idx}_n}) = \texttt{ref}(\texttt{datatype}_{\texttt{1}}) + \sum_{i=1}^{n-1} \sum_{k=1}^{idx_i-1} \text{size}(\texttt{datatype}_{\texttt{i},k}) \quad (0.0.3)$$

berechnet werden.^c

Dabei darf nur der letzte Knoten datatype_n vom Datentyp Zeiger sein. Ist in einer Folge von Datentypen ein Knoten vom Datentyp Zeiger, der nicht der letzte Datentyp datatype_n in der Folge ist, so muss die Adressberechnung in 2 Adressberechnungen aufgeteilt werden, wobei die erste Adressberechnung vom ersten Datentyp datatype₁ bis direkt zum Dantentyp Zeiger geht datatype_{pntr} und die zweite Adressberechnung einen Dantentyp nach dem Datentyp Zeiger anfängt datatype_{pntr+1} und bis zum letzten Datenyp datatype_n geht. Bei der zweiten Adressberechnung muss dabei die Adresse ref(datatype₁) des Summanden aus der Formel 0.0.3 auf den Inhalt der Speicherzelle an der gerade in der zweiten Adressberechnung berechneten Adresse M [ref(datatype₁, ..., datatype_{pntr})] gesetzt werden.

Die Formel 0.0.3 stellt dabei eine Verallgemeinerung der Formel 0.0.1 dar, die für alle möglichen Aneinandereihungen von Zugriffen auf Zeigerelemente, Feldelemente und Verbundsattribute funktioniert (z.B. (*complex_var.attr2)[3]). Da die Formel allgemein sein muss, lässt sie sich nicht so elegant mit einem Produkt \prod schreiben, wie die Formel 0.0.1, da man nicht davon ausgehen kann, dass alle Elemente den gleichen Datentyp haben^d.

Die Komposition Ref(Global(num)) bzw. Ref(Stackframe(num)) repräsentiert dabei den Summanden ref(datatype₁) in der Formel.

Die Komposition Exp(Attr(Stack(Num('1')), name)) repräsentiert dabei einen Summanden $\sum_{k=1}^{idx_i-1} \text{size(datatype}_{i,k})$ in der Formel.

Die Komposition Exp(Stack(Num('1'))) repräsentiert dabei das Lesen des Inhalts $M[\text{ref}(\text{datatype}_{1,idx_1}, \ldots, \text{datatype}_{n,idx_n})]$ der Speicherzelle an der finalen Adresse $\text{ref}(\text{datatype}_{1,idx_1}, \ldots, \text{datatype}_{n,idx_n})$.

^aEs ist ein Baum, der nur die Datentypen als Knoten enthält, auf die zugegriffen wird.

^bref (datatype) steht dabei für das Schreiben der Startadresse, die dem Datentyp datatype zugeordnet ist auf den Stack.

 c Die äußere Schleife iteriert nacheinander über die Folge von Datentypen, die aus den Zugriffen auf Zeigerelmente, Feldelemente oder Verbundsattribute resultiert. Die innere Schleife iteriert über alle Elemente oder Attribute des momentan betrachteten Datentyps datatype_i, die vor dem Element / Attribut datatype_{i,idx_i} liegen. d Verbundsattribute haben unterschiedliche Größen.

Definition 0.1: Entarteter Baum

Baum bei dem jeder Knoten maximal eine ausgehende Kante hat, also maximal Außengrad 1.

Oder alternativ: Baum beim dem jeder Knoten des Baumes maximal eine eingehende Kante hat, also maximal Innengrad^a 1.

Der Baum entspricht also einer verketteten Liste.^b

^aDer Innengrad ist die Anzahl eingehener Kanten.

```
Name './example_struct_attr_access.picoc_mon',
      Block
        Name 'main.0',
6
           // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
           → Num('2'))]))
           Exp(Num('4'))
9
           Exp(Num('2'))
10
           Assign(Global(Num('0')), Stack(Num('2')))
11
           // Exp(Attr(Name('st'), Name('y')))
12
           Ref(Global(Num('0')))
13
           Ref(Attr(Stack(Num('1')), Name('y')))
14
           Exp(Stack(Num('1')))
           Return(Empty())
16
        ]
    ]
```

Code 0.31: PicoC-ANF Pass für Zugriff auf Verbundsattribut.

Im RETI-Blocks Pass in Code 0.32 werden die Kompositionen Ref(Global(Num('0'))), Ref(Attr(Stack(Num('1')), Name('y'))) und Exp(Stack(Num('1'))) durch ihre entsprechenden RETI-Knoten ersetzt.

 $^{{}^{}b}B\ddot{a}ume.$

```
LOADI ACC 4;
11
           STOREIN SP ACC 1;
12
           # Exp(Num('2'))
13
           SUBI SP 1;
           LOADI ACC 2;
15
           STOREIN SP ACC 1;
16
           # Assign(Global(Num('0')), Stack(Num('2')))
17
           LOADIN SP ACC 1;
18
           STOREIN DS ACC 1;
19
           LOADIN SP ACC 2;
20
           STOREIN DS ACC 0;
21
           ADDI SP 2;
           # // Exp(Attr(Name('st'), Name('y')))
22
23
           # Ref(Global(Num('0')))
24
           SUBI SP 1;
25
           LOADI IN1 0;
26
           ADD IN1 DS;
27
           STOREIN SP IN1 1;
28
           # Ref(Attr(Stack(Num('1')), Name('y')))
29
           LOADIN SP IN1 1;
30
           ADDI IN1 1;
31
           STOREIN SP IN1 1;
32
           # Exp(Stack(Num('1')))
33
           LOADIN SP IN1 1;
34
           LOADIN IN1 ACC 0;
35
           STOREIN SP ACC 1;
36
           # Return(Empty())
37
           LOADIN BAF PC -1;
38
         ]
39
    ]
```

Code 0.32: RETI-Blocks Pass für Zugriff auf Verbundsattribut.

0.0.3.4 Zuweisung an Verbundsattribut

Die **Zuweisung an ein Verbundsattribut** (z.B. st.y = 42) wird im Folgenden anhand des Beispiels in Code 0.33 erklärt.

```
1 struct pos {int x; int y;};
2
3 void main() {
4   struct pos st = {.x=4, .y=2};
5   st.y = 42;
6 }
```

Code 0.33: PicoC-Code für Zuweisung an Verbundsattribut.

Im Abstact Syntax Tree wird eine Zuweisung an ein Verbundsattribut (z.B. st.y = 42) durch die Komposition Assign(Attr(Name('st'), Name('y')), Num('42')) dargestellt.

```
Name './example_struct_attr_assignment.ast',
4
      StructDecl
        Name 'pos',
           Alloc(Writeable(), IntType('int'), Name('x'))
           Alloc(Writeable(), IntType('int'), Name('y'))
9
        ],
10
      FunDef
11
         VoidType 'void',
12
        Name 'main',
13
         [],
14
         Γ
15
           Assign(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),

    Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
           Assign(Attr(Name('st'), Name('y')), Num('42'))
16
17
         ]
18
    ]
```

Code 0.34: Abstrakter Syntaxbaum für Zuweisung an Verbundsattribut.

Im PicoC-ANF Pass in Code 0.35 wird die Komposition Assign(Attr(Name('st'), Name('y')), Num('42')) auf ähnliche Weise ausgewertet, wie die Komposition, die einen Zugriff auf ein Feldelement Assign(Subscr(Name('ar'), Num('2')), Num('42')) in Subkapitel 0.0.2.3 darstellt. Daher wird hier, um Wiederholung zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsnten auf das Unterkapitel 0.0.2.3 verwiesen.

Im Gegensatz zum Vorgehen in Unterkapitel 0.0.2.3 muss hier für das Auswerten des linken Knoten Attr(Name('st'), Name('y')) von Assign(Attr(Name('st'), Name('y')), Num('42')) wie in Subkapitel 0.0.3.3 vorgegangen werden.

```
Name './example_struct_attr_assignment.picoc_mon',
    Γ
      Block
        Name 'main.0',
6
           // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
           → Num('2'))]))
           Exp(Num('4'))
9
           Exp(Num('2'))
10
           Assign(Global(Num('0')), Stack(Num('2')))
11
           // Assign(Attr(Name('st'), Name('y')), Num('42'))
12
           Exp(Num('42'))
13
           Ref(Global(Num('0')))
14
           Ref(Attr(Stack(Num('1')), Name('y')))
           Assign(Stack(Num('1')), Stack(Num('2')))
16
           Return(Empty())
17
        ]
18
    ]
```

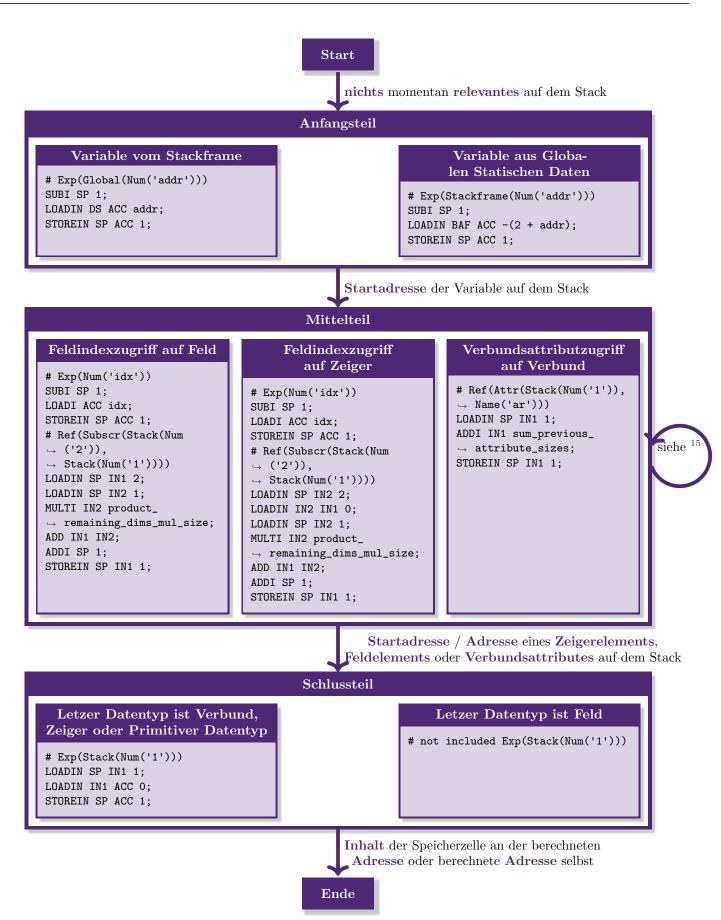
Code 0.35: PicoC-ANF Pass für Zuweisung an Verbundsattribut.

Im RETI-Blocks Pass in Code 0.36 werden die Kompositionen Exp(Num('42')), Ref(Global(Num('0'))), Ref(Attr(Stack(Num('1')), Name('y'))) und Assign(Stack(Num('1')), Stack(Num('2'))) durch ihre entsprechenden RETI-Knoten ersetzt.

```
1 File
     Name './example_struct_attr_assignment.reti_blocks',
       Block
 5
         Name 'main.0',
 6
 7
           # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
           → Num('2'))]))
           # Exp(Num('4'))
 9
           SUBI SP 1;
10
           LOADI ACC 4;
11
           STOREIN SP ACC 1;
12
           # Exp(Num('2'))
13
           SUBI SP 1;
14
           LOADI ACC 2;
15
           STOREIN SP ACC 1;
16
           # Assign(Global(Num('0')), Stack(Num('2')))
17
           LOADIN SP ACC 1;
18
           STOREIN DS ACC 1:
19
           LOADIN SP ACC 2;
20
           STOREIN DS ACC 0;
21
           ADDI SP 2;
22
           # // Assign(Attr(Name('st'), Name('y')), Num('42'))
23
           # Exp(Num('42'))
24
           SUBI SP 1;
25
           LOADI ACC 42;
26
           STOREIN SP ACC 1;
27
           # Ref(Global(Num('0')))
28
           SUBI SP 1;
29
           LOADI IN1 0;
30
           ADD IN1 DS;
           STOREIN SP IN1 1;
31
32
           # Ref(Attr(Stack(Num('1')), Name('y')))
33
           LOADIN SP IN1 1;
34
           ADDI IN1 1;
35
           STOREIN SP IN1 1;
36
           # Assign(Stack(Num('1')), Stack(Num('2')))
37
           LOADIN SP IN1 1;
38
           LOADIN SP ACC 2;
39
           ADDI SP 2;
40
           STOREIN IN1 ACC 0;
41
           # Return(Empty())
42
           LOADIN BAF PC -1;
43
         ]
    ]
```

Code 0.36: RETI-Blocks Pass für Zuweisung an Verbndsattribut.

0.0.4 Umsetzung des Zugriffs auf Zusammengesetzte Datentypen im Allgemeinen



In den Unterkapiteln 0.0.1, 0.0.2 und 0.0.3 fällt auf, dass der Zugriff auf Elemente / Attribute der in diesen Kapiteln beschriebenen Datentypen (Zeiger, Feld und Verbund) sehr ähnlich abläuft. Es lässt sich ein allgemeines Vorgehen, bestehend aus einem Anfangsteil, Mittelteil und Schlussteil darin erkennen.

Dieses Vorgehen ist in Abbildung 1 veranschaulicht. Dieses Vorgehen erlaubt es auch gemischte Ausdrücke zu schreiben, in denen die verschiedenen **Zugriffsarten** für **Elemente** / **Attribute** der Datenypen **Zeiger**, **Feld** und **Verbund** gemischt sind (z.B. (*st_var.ar)[0]).

Dies ist möglich, indem im Mittelteil, je nachdem, ob das versteckte Attribut datatype des Ref(exp, datatype)-Knotens ein ArrayDecl(nums, datatype), ein PhtrDecl(num, datatype) oder StructSpec(name) beinhaltet und die dazu passende Zugriffsoperation Subscr(exp1, exp2) oder Attr(exp, name) vorliegt, einen anderen RETI-Code generiert wird. Dieser RETI-Code berechet die Startadresse eines gewünschten Zeigerelements, Feldelements oder Verbundsattributs.

Würde man bei einem Subscr(Name('var'), exp2) den Datentyp der Variable Name('var') von ArrayDecl(nums, IntType()) zu PointerDecl(num, IntType()) ändern, müsste nur der Mittelteil ausgetauscht werden. Anfangsteil und Schlussteil bleiben unverändert.

Die Zugriffsoperation muss dabei zum Datentyp im versteckten Attribut datatype passen, ansonsten gibt es eine DatatypeMismatch-Fehlermeldung. Ein Zugriff auf ein Feldindex Subscr(exp1, epp2) kann dabei mit den Datentypen Feld ArrayDecl(nums, datatype) und Zeiger PntrDecl(num, datatype) kombiniert werden. Allerdings benötigen beide Kombinationen unterschiedlichen RETI-Code. Das liegt daran, dass bei einem Zeiger PntrDecl(num, datatype) die Adresse, die auf dem Stack liegt auf eine Speicherzelle mit einer weiteren Adresse zeigt und das gewünschte Element erst zu finden ist, wenn man der letzteren Adresse folgt. Ein Zugriff auf ein Verbundsattribut Attr(exp, name) kann nur mit dem Datentyp Struct StructSpec(name) kombiniert werden.

Anmerkung Q

Um Verwirrung vorzubeugen, wird hier vorausschauend nochmal darauf hingewiesen, dass eine Dereferenzierung in der Form Deref(exp1, exp2) nicht mehr existiert, denn wie in Unterkapitel 0.0.1 bereits erklärt wurde, wurde der Knoten Deref(exp1, exp2) im PicoC-Shrink Pass durch Subscr(exp1, exp2) ersetzt. Das hatte den Zweck, doppelten Code zu vermeiden, da die Dereferenzierung und der Zugriff auf ein Feldelement jeweils gegenseitig austauschbar sind. Der Zugriff auf einen Feldindex steht also gleichermaßen auch für eine Dereferenzierung.

Das versteckte Attribut datatype beinhaltet den Unterdatentyp, in welchem der Zugriff auf ein Zeigerelment, Feldelement oder Verbundsattribut erfolgt. Der Unterdatentyp ist dabei ein Teilbaum des Baumes, der vom gesamten Datentyp der Variable gebildet wird. Wobei man sich allerdings nur für den obersten Knoten in diesem Unterdatentyp interessiert und die möglicherweise unter diesem momentan betrachteten Knoten liegenden Knoten in einem anderen Ref(exp, versteckte Attribut)-Knoten, dem jeweiligen versteckten Attribut datatype zugeordnet sind. Das versteckte Attribut datatype enthält also die Information auf welchen Unterdatentyp im dem momentanen Kontext gerade zugegriffen wird.

Der Anfangsteil, der durch die Komposition Ref(Name('var')) repräsentiert wird, ist dafür zuständig die Startadresse der Variablen Name('var') auf den Stack zu schreiben und je nachdem, ob diese Variable in den Globalen Statischen Daten oder auf dem Stackframe liegt einen anderen RETI-Code zu generieren.

Der Schlussteil wird durch die Komposition Exp(Stack(Num('1')), datatype) dargestellt. Je nachdem, ob das versteckte Attribut datatype ein CharType(), IntType(), PntrDecl(num, datatype) oder StructType(name) ist, wird ein entsprechender RETI-Code generiert, der die Adresse, die auf dem Stack liegt dazu nutzt, um den Inhalt der Speicherzelle an dieser Adresse auf den Stack zu schreiben. Dabei wird die Speicherzelle der Adresse mit dem Inhalt auf den sie selbst zeigt überschreiben. Bei einem ArrayDecl(nums, datatype) hingegen wird kein weiterer RETI-Code generiert, die Adresse, die auf dem Stack liegt, stellt bereits das

gewünschte Ergebnis dar.

Felder haben in der Sprache L_C und somit auch in L_{PiocC} die Eigenheit, dass wenn auf ein gesamtes Feld zugegriffen wird¹⁶, die Adresse des ersten Elements ausgegeben wird und nicht der Inhalt der Speicherzelle des ersten Elements. Bei allen anderen in der Sprache L_{PicoC} implementieren Datentypen wird immer der Inhalt der Speicherzelle ausgegeben, die an der Adresse zu finden ist, die auf dem Stack liegt.

Anmerkung Q

Implementieren lässt sich dieses Vorgehen, indem beim Antreffen eines Subscr(exp1, exp2) oder Attr(exp, name) Ausdrucks ein Exp(Stack(Num('1'))) an die Spitze einer Liste der generierten Ausdrücke gesetzt wird und der Ausdruck selbst als exp-Attribut des Ref(exp)-Knotens gesetzt wird und hinter dem Exp(Stack(Num('1')))-Knoten in der Liste eingefügt wird. Beim Antreffen eines Ref(exp) wird fast gleich vorgegangen, wie beim Antreffen eines Subscr(exp1, exp2) oder Attr(exp, name), nur, dass kein Exp(Stack(Num('1'))) vorne an die Spitze der Liste der generierten Ausdrücke gesetzt wird. Und ein Ref(exp) bei dem exp direkt ein Name(str) ist, wird dieser einfach direkt durch Ref(Global(num)) bzw. Ref(Stackframe(num)) ersetzt.

Es wird solange dem jeweiligen exp1 des Subscr(exp1, exp2)-Knoten, dem exp des Attr(exp, name)-Knoten oder dem exp des Ref(exp)-Knoten gefolgt und der jeweilige Knoten selbst als exp des Ref(exp)-Knoten eingesetzt und hinten in die Liste der generierten Ausdrücke eingefügt, bis man bei einem Name(name) ankommt. Der Name(name)-Knoten wird zu einem Ref(Global(num)) oder Ref(Stackframe(num)) umgewandelt und ebenfalls ganz hinten in die Liste der generierten Ausdrücke eingefügt. Wenn man dem exp Attribut eines Ref(exp)-Knoten folgt, wird allerdings kein Ref(exp) in die Liste der generierten Ausdrücke eingefügt, sondern das datatype-Attribut des zuletzt eingefügten Ref(exp, datatype) manipuliert, sodass dessen datatype in ein ArrayDecl([Num('1')], datatype) eingebettet ist und so ein auf das Ref(exp) folgendes Deref(exp1, exp2) oder Subscr(exp1, exp2) direkt behandelt wird.

Parallel wird eine Liste der Ref (exp)-Knoten geführt, deren versteckte Attribute datatype und error_data die entsprechenden Informationen zugewiesen bekommen müssen. Sobald man beim Name (name)-Knoten angekommen ist und mithilfe dieses in der Symboltabelle den Dantentyp der Variable nachsehen kann, wird der Datentyp der Variable nun ebenfalls, wie die Ausdrücke Subscr(exp1, exp2) und Attr(exp, name) schrittweise durchiteriert und dem jeweils nächsten datatype-Attribut gefolgt werden. Das Iterieren über den Datentyp wird solange durchgeführt, bis alle Ref(exp)-Knoten ihren im jeweiligen Kontext vorliegenden Datentyp in ihrem datatype-Attribut zugewiesen bekommen haben. Alles andere führt zu einer Fehlermeldung, für die das versteckte Attribut error_data genutzt wird.

Im Folgenden werden anhand mehrerer Beispiele die einzelnen Abschnitte Anfangsteil 0.0.4.1, Mittelteil 0.0.4.2 und Schlussteil 0.0.4.3 bei der Kompilierung von Zugriffen auf Zeigerelemente, Feldelemente, Verbundsattribute bei gemischten Ausdrücken, wie (*st_first_ar)[0]; einzeln isoliert betrachtet und erläutert.

0.0.4.1 Anfangsteil

Der Anfangsteil, bei dem die Adresse einer Variable auf den Stack geschrieben wird (z.B. &st), wird im Folgenden mithilfe des Beispiels in Code 0.37 erklärt.

```
1 struct ar_with_len {int len; int ar[2];};
2
3 void main() {
```

¹⁶Und nicht auf ein Element des Feldes.

¹⁶Startadresse / Adresse eines Zeigerelements, Feldelements oder Verbundsattributes auf dem Stack.

```
4   struct ar_with_len st_ar[3];
5   int *(*complex_var)[3];
6   &complex_var;
7 }
8   
9 void fun() {
10   struct ar_with_len st_ar[3];
11   int (*complex_var)[3];
12   &complex_var;
13 }
```

Code 0.37: PicoC-Code für den Anfangsteil.

Im Abstrakten Syntaxbaum in Code 0.38 wird die Refererenzierung &complex_var mit der Komposition Exp(Ref(Name('complex_var'))) dargestellt. Üblicherweise wird aber einfach nur Ref(Name('complex_var') geschrieben, aber da beim Erstellen des Abstract Syntx Tree jeder Logischer Ausdruck in ein Exp(exp) eingebettet wird, ist das Ref(Name('complex_var')) in ein Exp() eingebettet. Man müsste an vielen Stellen eine gesonderte Fallunterschiedung aufstellen, um von Exp(Ref(Name('complex_var'))) das Exp() zu entfernen, obwohl das Exp() in den darauffolgenden Passes so oder so herausgefiltet wird. Daher wurde darauf verzichtet den Code ohne triftigen Grund komplexer zu machen.

```
1
    Name './example_derived_dts_introduction_part.ast',
     Ε
       StructDecl
         Name 'ar_with_len',
           Alloc(Writeable(), IntType('int'), Name('len'))
           Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))
9
         ],
10
       FunDef
11
         VoidType 'void',
12
         Name 'main',
13
         [],
14
15
           Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),

→ Name('st_ar')))
          Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], PntrDecl(Num('1'),
16
           → IntType('int')))), Name('complex_var')))
17
           Exp(Ref(Name('complex_var')))
18
         ],
19
       FunDef
20
         VoidType 'void',
21
         Name 'fun',
22
         [],
23
24
           Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
           → Name('st_ar')))
           Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
25
           → Name('complex_var')))
           Exp(Ref(Name('complex_var')))
26
         ]
27
    ]
```

Code 0.38: Abstrakter Syntaxbaum für den Anfangsteil.

Im PicoC-ANF Pass in Code 0.39 wird die Komposition Exp(Ref(Name('complex_var'))) durch die Komposition Ref(Global(Num('9'))) bzw. Ref(Stackframe(Num('9'))) ersetzt, je nachdem, ob die Variable Name('complex_var') in den Globalen Statischen Daten oder auf dem Stack liegt.

```
1
  File
    Name './example_derived_dts_introduction_part.picoc_mon',
     Ε
       Block
         Name 'main.1',
7
8
9
           // Exp(Ref(Name('complex_var')))
           Ref(Global(Num('9')))
           Return(Empty())
10
         ],
       Block
12
         Name 'fun.0',
13
14
           // Exp(Ref(Name('complex_var')))
15
           Ref(Stackframe(Num('9')))
16
           Return(Empty())
17
         ]
18
    ]
```

Code 0.39: PicoC-ANF Pass für den Anfangsteil.

Im RETI-Blocks Pass in Code 0.40 werden die Komposition Ref(Global(Num('9'))) bzw. Ref(Stackframe(Num('9'))) durch ihre entsprechenden RETI-Knoten ersetzt.

```
1
 2
    Name './example_derived_dts_introduction_part.reti_blocks',
     Γ
       Block
         Name 'main.1',
 7
8
9
           # // Exp(Ref(Name('complex_var')))
           # Ref(Global(Num('9')))
           SUBI SP 1;
10
           LOADI IN1 9;
11
           ADD IN1 DS;
12
           STOREIN SP IN1 1;
13
           # Return(Empty())
14
           LOADIN BAF PC -1;
15
         ],
16
       Block
17
         Name 'fun.0',
18
19
           # // Exp(Ref(Name('complex_var')))
20
           # Ref(Stackframe(Num('9')))
21
           SUBI SP 1;
```

```
22 MOVE BAF IN1;

23 SUBI IN1 11;

24 STOREIN SP IN1 1;

25 # Return(Empty())

26 LOADIN BAF PC -1;

27 ]
```

Code 0.40: RETI-Blocks Pass für den Anfangsteil.

0.0.4.2 Mittelteil

Der Mittelteil, bei dem die Startadresse / Adresse einer Aneinandereihung von Zugriffen auf Zeigerelemente, Feldelemente oder Verbundsattribute berechnet wird (z.B. (*complex_var.ar)[2-2]), wird im Folgenden mithilfe des Beispiels in Code 0.41 erklärt.

```
1 struct st {int (*ar)[1];};
2
3 void main() {
4   int var[1] = {42};
5   struct st complex_var = {.ar=&var};
6   (*complex_var.ar)[2-2];
7 }
```

Code 0.41: PicoC-Code für den Mittelteil.

Im Abstrakten Syntaxbaum in Code 0.42 wird die Aneinandererihung von Zugriffen auf Zeigerelemente, Feldelemente und Verbundsattribute (*complex_var.ar)[2-2] durch die Komposition Exp(Subscr(Deref(Attr(Name('complex_var'), Name('ar')), Num('0')), BinOp(Num('2'), Sub('-'), Num('2')))) dargestellt.

```
1 File
2
    Name './example_derived_dts_main_part.ast',
4
      StructDecl
        Name 'st',
6
          Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('1')], IntType('int'))),
           → Name('ar'))
        ],
9
      FunDef
        VoidType 'void',
10
11
        Name 'main',
12
         [],
13
         Ε
14
          Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('var')),
           → Array([Num('42')]))
          Assign(Alloc(Writeable(), StructSpec(Name('st')), Name('complex_var')),

    Struct([Assign(Name('ar'), Ref(Name('var')))]))

16
          Exp(Subscr(Deref(Attr(Name('complex_var'), Name('ar')), Num('0')), BinOp(Num('2'),

    Sub('-'), Num('2'))))
```

```
17 ]
18 ]
```

Code 0.42: Abstrakter Syntaxbaum für den Mittelteil.

Im PicoC-ANF Pass in Code 0.43 wird die Komposition Exp(Subscr(Deref(Attr(Name('complex_var'), Name('ar')), Num('0')), BinOp(Num('2'), Sub('-'), Num('2')))) durch die Kompositionen Ref(Attr(Stack(Num('1')), Name('ar'))), Exp(Num('2')), Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1')))), Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) und Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) ersetzt. Bei Subscr(exp1, exp2) wird dieser Knoten einfach dem exp Attribut des Ref(exp)-Knoten zugewiesen und die Indexberechnung für exp2 davorgezogen (in diesem Fall dargestellt durch Exp(Num('2')) und Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))) und über Stack(Num('1')) auf das Ergebnis der Indexberechnung auf dem Stack zugegriffen: Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1')))).

```
2
3
    Name './example_derived_dts_main_part.picoc_mon',
     Γ
 4
5
6
7
8
       Block
         Name 'main.0',
           // Assign(Name('var'), Array([Num('42')]))
           Exp(Num('42'))
           Assign(Global(Num('0')), Stack(Num('1')))
10
           // Assign(Name('complex_var'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
           Ref(Global(Num('0')))
11
12
           Assign(Global(Num('1')), Stack(Num('1')))
13
           // Exp(Subscr(Subscr(Attr(Name('complex_var'), Name('ar')), Num('0')),
              BinOp(Num('2'), Sub('-'), Num('2'))))
           Ref(Global(Num('1')))
15
           Ref(Attr(Stack(Num('1')), Name('ar')))
16
           Exp(Num('0'))
17
           Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
18
           Exp(Num('2'))
19
           Exp(Num('2'))
20
           Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
21
           Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
22
           Exp(Stack(Num('1')))
23
           Return(Empty())
24
         ]
25
    ]
```

Code 0.43: PicoC-ANF Pass für den Mittelteil.

Im RETI-Blocks Pass in Code 0.44 werden die Kompositionen Ref(Attr(Stack(Num('1')), Name('ar'))), Exp(Num('2')), Exp(Num('2')), Stack(Num('1'))), Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) und Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) durch ihre entsprechenden RETI-Knoten ersetzt. Bei der Generierung des RETI-Code muss auch das versteckte Attribut datatype im Ref(exp, datatype)-Knoten berücksichtigt werden, was in Unterkapitel 0.0.4 zusammen mit der Abbildung 1 bereits erklärt wurde.

```
Name './example_derived_dts_main_part.reti_blocks',
     Γ
 4
       Block
         Name 'main.0',
           # // Assign(Name('var'), Array([Num('42')]))
           # Exp(Num('42'))
 9
           SUBI SP 1;
10
           LOADI ACC 42;
11
           STOREIN SP ACC 1;
12
           # Assign(Global(Num('0')), Stack(Num('1')))
13
           LOADIN SP ACC 1;
14
           STOREIN DS ACC 0;
15
           ADDI SP 1;
16
           # // Assign(Name('complex_var'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
           # Ref(Global(Num('0')))
18
           SUBI SP 1;
19
           LOADI IN1 0:
20
           ADD IN1 DS;
21
           STOREIN SP IN1 1;
22
           # Assign(Global(Num('1')), Stack(Num('1')))
23
           LOADIN SP ACC 1;
24
           STOREIN DS ACC 1;
25
           ADDI SP 1;
26
           # // Exp(Subscr(Subscr(Attr(Name('complex_var'), Name('ar')), Num('0')),

→ BinOp(Num('2'), Sub('-'), Num('2'))))
           # Ref(Global(Num('1')))
27
           SUBI SP 1;
28
29
           LOADI IN1 1;
30
           ADD IN1 DS;
31
           STOREIN SP IN1 1;
32
           # Ref(Attr(Stack(Num('1')), Name('ar')))
33
           LOADIN SP IN1 1;
34
           ADDI IN1 0;
35
           STOREIN SP IN1 1;
36
           # Exp(Num('0'))
37
           SUBI SP 1;
38
           LOADI ACC 0;
39
           STOREIN SP ACC 1;
40
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
41
           LOADIN SP IN2 2;
42
           LOADIN IN2 IN1 0;
43
           LOADIN SP IN2 1;
44
           MULTI IN2 1;
45
           ADD IN1 IN2;
46
           ADDI SP 1;
47
           STOREIN SP IN1 1;
48
           # Exp(Num('2'))
49
           SUBI SP 1;
50
           LOADI ACC 2;
51
           STOREIN SP ACC 1;
52
           # Exp(Num('2'))
53
           SUBI SP 1;
54
           LOADI ACC 2;
55
           STOREIN SP ACC 1;
56
           # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
```

```
LOADIN SP ACC 2;
58
           LOADIN SP IN2 1;
59
           SUB ACC IN2;
60
           STOREIN SP ACC 2;
61
           ADDI SP 1;
62
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
63
           LOADIN SP IN1 2;
64
           LOADIN SP IN2 1;
65
           MULTI IN2 1;
66
           ADD IN1 IN2;
67
           ADDI SP 1;
68
           STOREIN SP IN1 1;
69
           # Exp(Stack(Num('1')))
           LOADIN SP IN1 1;
70
71
           LOADIN IN1 ACC 0;
72
           STOREIN SP ACC 1;
73
           # Return(Empty())
           LOADIN BAF PC -1;
75
         ]
    ]
```

Code 0.44: RETI-Blocks Pass für den Mittelteil.

0.0.4.3 Schlussteil

Der Schlussteil, bei dem der Inhalt der Speicherzelle an der Adresse, die im Anfangsteil 0.0.4.1 und Mittelteil 0.0.4.2 auf dem Stack berechnet wurde, auf den Stack gespeichert wird¹⁷, wird im Folgenden mithilfe des Beispiels in Code 0.45 erklärt.

```
1 struct st {int attr[2];};
2
3 void main() {
4    int complex_var1[1][2];
5    struct st complex_var2[1];
6    int var = 42;
7    int *pntr1 = &var;
8    int **complex_var3 = &pntr1;
9
10    complex_var1[0];
11    complex_var2[0];
12    *complex_var3;
13 }
```

Code 0.45: PicoC-Code für den Schlussteil.

Das Generieren des Abstrakten Syntaxbaumes in Code 0.46 verläuft wie üblich.

```
1 File
2 Name './example_derived_dts_final_part.ast',
```

¹⁷Und dabei die Speicherzelle der Adresse selbst überschreibt.

```
4
      StructDecl
        Name 'st',
7
8
          Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr'))
        ],
9
      FunDef
10
         VoidType 'void',
11
        Name 'main',
12
         [],
13
         Γ
14
          Exp(Alloc(Writeable(), ArrayDecl([Num('1'), Num('2')], IntType('int')),

→ Name('complex_var1')))
15
          Exp(Alloc(Writeable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),
           → Name('complex_var2')))
          Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
16
17
          Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr1')),

→ Ref(Name('var')))
18
          Assign(Alloc(Writeable(), PntrDecl(Num('2'), IntType('int')), Name('complex_var3')),

→ Ref(Name('pntr1')))
19
          Exp(Subscr(Name('complex_var1'), Num('0')))
          Exp(Subscr(Name('complex_var2'), Num('0')))
20
21
          Exp(Deref(Name('complex_var3'), Num('0')))
22
    ]
```

Code 0.46: Abstrakter Syntaxbaum für den Schlussteil.

Im PicoC-ANF Pass in Code 0.47 wird das eben angesprochene auf den Stack speichern des Inhalts der berechneten Adresse mit der Komposition Exp(Stack(Num('1'))) dargestellt.

```
2
    Name './example_derived_dts_final_part.picoc_mon',
    Γ
      Block
        Name 'main.0',
7
8
           // Assign(Name('var'), Num('42'))
          Exp(Num('42'))
           Assign(Global(Num('4')), Stack(Num('1')))
10
           // Assign(Name('pntr1'), Ref(Name('var')))
11
           Ref(Global(Num('4')))
12
           Assign(Global(Num('5')), Stack(Num('1')))
13
           // Assign(Name('complex_var3'), Ref(Name('pntr1')))
14
          Ref(Global(Num('5')))
15
           Assign(Global(Num('6')), Stack(Num('1')))
16
           // Exp(Subscr(Name('complex_var1'), Num('0')))
17
           Ref(Global(Num('0')))
           Exp(Num('0'))
18
19
          Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
20
           Exp(Stack(Num('1')))
21
           // Exp(Subscr(Name('complex_var2'), Num('0')))
22
           Ref(Global(Num('2')))
           Exp(Num('0'))
```

```
Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
25
           Exp(Stack(Num('1')))
26
           // Exp(Subscr(Name('complex_var3'), Num('0')))
27
           Ref(Global(Num('6')))
           Exp(Num('0'))
29
           Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
30
           Exp(Stack(Num('1')))
31
           Return(Empty())
32
         ]
33
    ]
```

Code 0.47: PicoC-ANF Pass für den Schlussteil.

Im RETI-Blocks Pass in Code 0.48 wird die Komposition Exp(Stack(Num('1'))) durch ihre entsprechenden RETI-Knoten ersetzt, wenn das versteckte Attribut datatype im Exp(exp,datatpye)-Knoten kein Feld ArrayDecl(nums, datatype) enthält, ansonsten ist bei einem Feld die Adresse auf dem Stack bereits das gewünschte Ergebnis. Genaueres wurde in Unterkapitel 0.0.4 zusammen mit der Abbildung 1 bereits erklärt.

```
1 File
    Name './example_derived_dts_final_part.reti_blocks',
     Γ
       Block
         Name 'main.0',
           # // Assign(Name('var'), Num('42'))
           # Exp(Num('42'))
           SUBI SP 1:
10
           LOADI ACC 42;
11
           STOREIN SP ACC 1;
12
           # Assign(Global(Num('4')), Stack(Num('1')))
13
           LOADIN SP ACC 1;
14
           STOREIN DS ACC 4;
15
           ADDI SP 1;
           # // Assign(Name('pntr1'), Ref(Name('var')))
16
17
           # Ref(Global(Num('4')))
18
           SUBI SP 1;
19
           LOADI IN1 4;
20
           ADD IN1 DS;
21
           STOREIN SP IN1 1;
22
           # Assign(Global(Num('5')), Stack(Num('1')))
23
           LOADIN SP ACC 1;
24
           STOREIN DS ACC 5;
25
           ADDI SP 1;
26
           # // Assign(Name('complex_var3'), Ref(Name('pntr1')))
27
           # Ref(Global(Num('5')))
28
           SUBI SP 1;
29
           LOADI IN1 5;
30
           ADD IN1 DS;
31
           STOREIN SP IN1 1;
           # Assign(Global(Num('6')), Stack(Num('1')))
32
33
           LOADIN SP ACC 1;
34
           STOREIN DS ACC 6;
           ADDI SP 1;
```

```
# // Exp(Subscr(Name('complex_var1'), Num('0')))
           # Ref(Global(Num('0')))
37
38
           SUBI SP 1;
39
           LOADI IN1 0;
           ADD IN1 DS;
41
           STOREIN SP IN1 1;
42
           # Exp(Num('0'))
43
           SUBI SP 1;
44
           LOADI ACC 0;
45
           STOREIN SP ACC 1;
46
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
47
           LOADIN SP IN1 2;
48
           LOADIN SP IN2 1;
49
           MULTI IN2 2;
50
           ADD IN1 IN2;
51
           ADDI SP 1;
52
           STOREIN SP IN1 1;
53
           # // not included Exp(Stack(Num('1')))
54
           # // Exp(Subscr(Name('complex_var2'), Num('0')))
55
           # Ref(Global(Num('2')))
56
           SUBI SP 1;
57
           LOADI IN1 2;
58
           ADD IN1 DS;
59
           STOREIN SP IN1 1;
60
           # Exp(Num('0'))
61
           SUBI SP 1;
           LOADI ACC 0;
62
63
           STOREIN SP ACC 1;
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
64
65
           LOADIN SP IN1 2;
           LOADIN SP IN2 1;
67
           MULTI IN2 2;
68
           ADD IN1 IN2;
69
           ADDI SP 1;
70
           STOREIN SP IN1 1;
71
           # Exp(Stack(Num('1')))
72
           LOADIN SP IN1 1;
           LOADIN IN1 ACC 0;
74
           STOREIN SP ACC 1;
75
           # // Exp(Subscr(Name('complex_var3'), Num('0')))
76
           # Ref(Global(Num('6')))
           SUBI SP 1;
77
78
           LOADI IN1 6;
79
           ADD IN1 DS;
80
           STOREIN SP IN1 1;
81
           # Exp(Num('0'))
82
           SUBI SP 1;
83
           LOADI ACC 0;
84
           STOREIN SP ACC 1;
85
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
86
           LOADIN SP IN2 2;
87
           LOADIN IN2 IN1 0;
88
           LOADIN SP IN2 1;
89
           MULTI IN2 1;
90
           ADD IN1 IN2;
91
           ADDI SP 1;
           STOREIN SP IN1 1;
```

Grammatikverzeichnis

```
# Exp(Stack(Num('1')))

LOADIN SP IN1 1;

LOADIN IN1 ACC 0;

STOREIN SP ACC 1;

Return(Empty())

LOADIN BAF PC -1;

99 ]
```

Code 0.48: RETI-Blocks Pass für den Schlussteil.

Literatur

Online

 \bullet $B\"{a}ume.$ URL: https://www.stefan-marr.de/pages/informatik-abivorbereitung/baume/ (besucht am 17.07.2022).

Vorlesungen

• 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).