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

Abbildung	sverzeich	nis	Ι
Codeverze	ichnis		II
Tabellenve	erzeichnis		III
Definitions	sverzeichr	is	IV
Grammati	kverzeich	ais	\mathbf{V}
0.0.	.1 Umset	zung von Verbunden	1
	0.0.1.1	Deklaration von Verbundstypen und Definition von Verbunden	
	0.0.1.2	Initialisierung von Verbunden	
	0.0.1.3	Zugriff auf Verbundsattribut	6
	0.0.1.4	Zuweisung an Verbundsattribut	10
0.0.	.2 Umset	Umsetzung des Zugriffs auf Zusammengesetzte Datentypen im Allgemeinen	
	0.0.2.1	Anfangsteil	15
	0.0.2.2	Mittelteil	17
	0.0.2.3	Schlussteil	20
Literatur			\mathbf{A}

Abbildungsverzeichnis

1	Allgemeine	Veranschaulichung	des Zugriffs au	if Zusammengesetzte Datent	$vpen. \dots 1$

Codeverzeichnis

0.1	PicoC-Code für die Deklaration eines Verbundstyps
0.2	Abstrakter Syntaxbaum für die Deklaration eines Verbundstyps
0.3	Symboltabelle für die Deklaration eines Verbundstyps
0.4	PicoC-Code für Initialisierung von Verbunden.
0.5	Abstrakter Syntaxbaum für Initialisierung von Verbunden
0.6	PicoC-ANF Pass für Initialisierung von Verbunden
0.7	RETI-Blocks Pass für Initialisierung von Verbunden
0.8	PicoC-Code für Zugriff auf Verbundsattribut
0.9	Abstrakter Syntaxbaum für Zugriff auf Verbundsattribut
	PicoC-ANF Pass für Zugriff auf Verbundsattribut
	RETI-Blocks Pass für Zugriff auf Verbundsattribut
	PicoC-Code für Zuweisung an Verbundsattribut
0.13	Abstrakter Syntaxbaum für Zuweisung an Verbundsattribut
	PicoC-ANF Pass für Zuweisung an Verbundsattribut
0.15	RETI-Blocks Pass für Zuweisung an Verbndsattribut
	PicoC-Code für den Anfangsteil
	Abstrakter Syntaxbaum für den Anfangsteil
	PicoC-ANF Pass für den Anfangsteil
0.19	RETI-Blocks Pass für den Anfangsteil
0.20	PicoC-Code für den Mittelteil
0.21	Abstrakter Syntaxbaum für den Mittelteil
	PicoC-ANF Pass für den Mittelteil
0.23	RETI-Blocks Pass für den Mittelteil
0.24	PicoC-Code für den Schlussteil
0.25	Abstrakter Syntaxbaum für den Schlussteil
0.26	PicoC-ANF Pass für den Schlussteil
0.27	RETI-Blocks Pass für den Schlussteil

Tabellenverzeichnis

Definitionsverzeichnis

Grammatikverzeichnis

0.0.1 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 0.0.1.1 erläutert. Des Weiteren ist die Umsetzung der Innitialisierung eines Verbundes 0.0.1.2, des **Zugriffs auf ein Verbundsattribut** 0.0.1.3 und der **Zuweisung an ein Verbundsattribut** 0.0.1.4 zu klären.

0.0.1.1 Deklaration von Verbundstypen und Definition von Verbunden

Die Umsetzung der Deklaration (Definition ??) eines neuen Verbundstyps (z.B. struct st {int len; int ar[2];}) und der Definition (Definition ??) eines Verbundes mit diesem Verbundstyp (z.B. struct st st_var;) wird im Folgenden anhand des Beispiels in Code 0.1 erläutert.

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

Code 0.1: PicoC-Code für die Deklaration eines Verbundstyps.

Bevor ein Verbund definiert werden kann, muss erstmal ein Verbundstyp deklariert werden. Im Abstrakten Syntaxbaum in Code 0.3 wird die Deklaration eines Verbundstyps struct st {int len; int ar[2];} durch die Knoten 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 Knoten Alloc(Writeable(), StructSpec(Name('st')), Name('st_var')) dargestellt.

```
1 File
    Name './example_struct_decl_def.ast',
 4
       StructDecl
         Name 'st',
           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
         Γ
           Exp(Alloc(Writeable(), StructSpec(Name('st')), Name('st_var')))
16
         ]
    ]
```

Code 0.2: Abstrakter Syntaxbaum für die Deklaration eines Verbundstyps.

Für den Verbundstyp selbst und seine Verbundsattribute werden in der Symboltabelle, die in Code 0.3 dargestellt ist Symboltabelleneintrage mit den Schlüsseln st, len@st und ar@st erstellt. Die Schlüssel der

Verbundsattribute haben einen Suffix @st angehängt, welcher für die Verbundsattribute einen Verbundstyps indirekt einen Sichtbarkeitsbereich (Definition ??) über den Verbundstyp selbst erzeugt. Im Unterkapitel ?? wird die Funktionsweise von Sichtbarkeitsbereichen genauer erläutert. Es gilt folglich, dass innerhalb eines Verbundstyps zwei Verbundsattribute nicht gleich benannt werden können, aber dafür zwei unterschiedliche Verbundstypen ihre Verbundsattribute gleich benennen können.

Die Attribute⁸ der Symboltabelleneinträge für die Verbundsattribute sind genauso belegt wie bei üblichen Variablen. Die Attribute des Symboltabelleneinträgs für den Verbundstyp type_qualifier, datatype, name, position und size sind wie üblich belegt. In dem value_address-Attribut des Symboltabelleneinträgs für den Verbundstyp sind die Verbundsattribute [Name('len@st'), Name('ar@st')] aufgelistet, sodass man über den Verbundstyp st als Schlüssel die Verbundsattribute des Verbundstyps in der Symboltabelle nachschlagen kann.

Für die Definition einer Variable st_var@main mit diesem Verbundstyp st wird ein Symboltabelleneintrag in der Symboltabelle angelegt. Das datatype-Attribut dieses Symboltabelleneintrags enthält dabei den Namen des Verbundstyps als StructSpec(Name('st')). Dadurch können jederzeit alle wichtigen Informationen zu diesem Verbundstyp⁹ und seinen Verbundsattributen in der Symboltabelle nachgeschlagen werden.

Anmerkung Q

Die Anzahl Speicherzellen die eine Variable st_var belegt^a, die mit dem Verbundstyp struct st {datatype₁ attr₁; ... datatype_n attr_n; }^b definiert ist (struct st st_var;), berechnet sich aus der Summe der Anzahl Speicherzellen, welche die einzelnen Datentypen datatype₁ ... datatype_n der Verbundsattribute attr₁, ... attr_n des Verbundstyps belegen: $size(st) = \sum_{i=1}^{n} size(datatype_i)$.^c

^aDie ihm size-Attribut des Symboltabelleneintrags eingetragen ist.

^cDie Funktion size berechnet die Anzahl Speicherzellen, die ein Datentyp belegt.

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

 $[^]b$ Hier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache L_{PicoC} nicht die manchmal etwas unpraktische Designentscheidung, auch die eckigen Klammern [] bei der Definition eines Feldes hinter die Variable zu schreiben von $L_{\mathbb{C}}$ übernommen. Es wird so getan, als würde der komplette Datentyp immer vor der Variable stehen: datatype var.

⁸Die über einen Bezeichner selektierbaren Elemente eines Symboltabelleneintrags und eines Verbunds heißen bei beiden Attribute.

⁹Wie z.B. vor allem die Größe bzw. Anzahl an Speicherzellen, die dieser Verbundstyp einnimmt.

```
19
           size:
                                    Num('2')
20
         },
21
       Symbol
22
         {
23
           type qualifier:
                                    Empty()
                                    StructDecl(Name('st'), [Alloc(Writeable(), IntType('int'),
24
           datatype:
           → Name('len'))Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')),
           → Name('ar'))])
                                    Name('st')
25
                                     [Name('len@st'), Name('ar@st')]
26
           value or address:
27
           position:
                                    Pos(Num('1'), Num('7'))
28
                                    Num('3')
           size:
29
         },
30
       Symbol
31
         {
32
           type qualifier:
                                    Empty()
33
                                    FunDecl(VoidType('void'), Name('main'), [])
           datatype:
34
                                    Name('main')
           name:
35
           value or address:
                                    Empty()
36
                                    Pos(Num('3'), Num('5'))
           position:
37
                                    Empty()
           size:
38
         },
39
       Symbol
40
41
                                    Writeable()
           type qualifier:
42
                                    StructSpec(Name('st'))
           datatype:
43
                                    Name('st_var@main')
           name:
44
           value or address:
                                    Num('0')
45
           position:
                                    Pos(Num('4'), Num('12'))
46
                                    Num('3')
           size:
47
         }
48
    ]
```

Code 0.3: Symboltabelle für die Deklaration eines Verbundstyps.

0.0.1.2 Initialisierung von Verbunden

Die Umsetzung der Initialisierung eines Verbundes wird im Folgenden mithilfe des Beispiels in Code 0.4 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 0.4: PicoC-Code für Initialisierung von Verbunden.

Im Abstrakten Syntaxbaum in Code 0.5 wird die Initialisierung eines Verbundes struct st1 st = {.attr1=var, .attr2={.attr={{&var, &var}}}} mithilfe der Knoten Assign(Alloc(Writeable(),

StructSpec(Name('st1')), Name('st')), Struct(...)) dargestellt.

```
File
    Name './example_struct_init.ast',
 4
       StructDecl
 5
         Name 'st1',
 6
           Alloc(Writeable(), ArrayDecl([Num('2')], PntrDecl(Num('1'), IntType('int'))),
           → Name('attr'))
 8
         ],
 9
       StructDecl
10
         Name 'st2',
11
         Γ
12
           Alloc(Writeable(), IntType('int'), Name('attr1'))
           Alloc(Writeable(), StructSpec(Name('st1')), Name('attr2'))
13
14
         ],
15
       FunDef
16
         VoidType 'void',
17
         Name 'main',
18
         [],
19
         Γ
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'))]))]))
22
23
    ]
```

Code 0.5: Abstrakter Syntaxbaum für Initialisierung von Verbunden.

Im PicoC-ANF Pass in Code 0.6 wird Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st1')), Struct(...)) auf fast dieselbe Weise ausgewertet, wie bei der Initialisierung eines Feldes in Unterkapitel ??. Für genauere Details wird an dieser Stelle daher auf Unterkapitel ?? verwiesen. Um das Ganze interessanter zu gestalten, wurde das Beispiel in Code 0.4 so gewählt, dass sich daran eine komplexere, mehrstufige Initialisierung mit verschiedenen Datentypen erklären lässt.

Der Teilbaum Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'), Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'), Struct([Assign(Name('attr1'), Name('var'))])]))]), der beim äußersten Verbund-Initializer-Knoten Struct(...) anfängt, wird auf dieselbe Weise nach dem Prinzip der Tiefensuche von links-nachrechts ausgewertet, wie es bei der Initialisierung eines Feldes in Unterkapitel ?? bereits erklärt wurde. Beim Iterieren über den Teilbaum, muss bei einem Verbund-Initializer-Knoten Struct(...) nur beachtet werden, dass bei den Assign(lhs, exp)-Knoten der Teilbaum beim rechten exp Attribut weitergeht.

Im Allgemeinen gibt es im Teilbaum beim Initialisieren eines Feldes oder Verbundes auf der rechten Seite immer nur 3 Fälle. Auf der rechten Seite hat man es entweder mit einem Verbund-Initialiser, einem Feld-Initialiser oder einem Logischen Ausdruck zu tun. Bei einem Feld- oder Verbund-Initialiser wird über diesen nach dem Prinzip der Tiefensuche von links-nach-rechts iteriert und mithilfe von Exp(exp)-Knoten die Auswertung der Logischen Ausdrücke in den Blättern auf den Stack dargestellt. Der Fall, dass ein Logischer Ausdruck vorliegt erübrigt sich hiermit.

¹⁰Über welche die Attributzuweisung (z.B. attr1=var) als z.B. Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Array([Ref(Name('var')), Ref(Name('var'))]])])))) dargestellt wird.

```
Name './example_struct_init.picoc_mon',
4
      Block
        Name 'main.0',
           // 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')))
12
          Ref(Global(Num('0')))
13
          Ref(Global(Num('0')))
14
          Assign(Global(Num('1')), Stack(Num('3')))
15
          Return(Empty())
16
        ]
17
    ]
```

Code 0.6: PicoC-ANF Pass für Initialisierung von Verbunden.

Im RETI-Blocks Pass in Code 0.7 werden die PicoC-Knoten Exp(Global(Num('0'))), Ref(Global(Num('0'))) und Assign(Global(Num('1')),Stack(Num('3'))) durch ihre semantisch entsprechenden RETI-Knoten ersetzt.

```
2
    Name './example_struct_init.reti_blocks',
       Block
         Name 'main.0',
 6
           # // Assign(Name('var'), Num('42'))
           # Exp(Num('42'))
 9
           SUBI SP 1;
10
           LOADI ACC 42;
           STOREIN SP ACC 1;
11
12
           # Assign(Global(Num('0')), Stack(Num('1')))
13
           LOADIN SP ACC 1;
           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'))]))]))))))))
           # 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;
           # Ref(Global(Num('0')))
```

```
SUBI SP 1;
28
           LOADI IN1 0;
29
           ADD IN1 DS;
30
           STOREIN SP IN1 1;
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
         ]
     ]
```

Code 0.7: RETI-Blocks Pass für Initialisierung von Verbunden.

0.0.1.3 Zugriff auf Verbundsattribut

Die Umsetzung des **Zugriffs auf ein Verbundsattribut** (z.B. st.y) wird im Folgenden mithilfe des Beispiels in Code 0.8 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 0.8: PicoC-Code für Zugriff auf Verbundsattribut.

Im Abstrakten Syntaxbaum in Code 0.9 wird der Zugriff auf ein Verbundsattribut st.y mithilfe der Knoten Exp(Attr(Name('st'), Name('y'))) dargestellt.

```
1 File
    Name './example_struct_attr_access.ast',
       StructDecl
         Name 'pos',
 7
8
9
           Alloc(Writeable(), IntType('int'), Name('x'))
           Alloc(Writeable(), IntType('int'), Name('y'))
         ],
10
       FunDef
11
         VoidType 'void',
12
         Name 'main',
13
         [],
14
         [
```

```
Assign(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),

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

Exp(Attr(Name('st'), Name('y')))

[ ]

[ ]
```

Code 0.9: Abstrakter Syntaxbaum für Zugriff auf Verbundsattribut.

Im PicoC-ANF Pass in Code 0.10 werden die Knoten Exp(Attr(Name('st'), Name('y'))) auf eine ähnliche Weise ausgewertet, wie die Knoten Exp(Subscr(Name('ar'), Num('0'))), die in Unterkapitel ?? einen Zugriff auf ein Feldelement darstellen. Daher wird hier, um Redundanz zu vermeiden, nur auf wichtige Aspekte hingewiesen und ansonsten auf das Unterkapitel ?? verwiesen.

Die Knoten Exp(Attr(Name('st'), Name('y'))) werden genauso, wie in Unterkapitel ?? durch Knoten ersetzt, die sich in Anfangsteil 0.0.2.1, Mittelteil 0.0.2.2 und Schlussteil 0.0.2.3 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 Unterkapitel ??.

Nur für den Mittelteil werden andere Knoten Ref(Attr(Stack(Num('1')), Name('y'))) gebraucht. Diese Knoten Ref(Attr(Stack(Num('1')), Name('y'))) stellen die Aufgabe dar, die Anfangsadresse des Attributs auf welches zugegriffen wird zu berechnen und auf den Stack zu legen. Hierfür wird die Anfangsadresse des Verbundes, in dem dieses Attribut liegt verwendet. Das auf den Stack-Speichern dieser Anfangsadresse wird durch Knoten des Anfangsteils dargstellt.

Beim Zugriff auf einen Feldindex muss vorher durch z.B. Exp(Num('3')) die Berechnung des Indexwerts und das auf den Stack legen des Ergebnisses dargestellt werden. Beim Zugriff auf ein Verbundsattribut steht der Bezeichner des Verbundsattributs Name('y') dagegen bereits während des Kompilierens in Ref(Attr(Stack(Num('1')), Name('y'))) zur Verfügung. Der Verbundstyp, dem dieses Attribut gehört, wird im Mittelteil aus dem versteckten Attribut datatype des Knoten Ref(exp, datatype) herausgelesen. Der Verbundstyp wird während des Kompiliervorgangs im PiocC-ANF Pass dem Knoten Ref(exp, datatype) über das versteckten Attribut datatype angehängt.

Anmerkung Q

Sei datatype_i ein Folgeglied einer Folge (datatype_i) $_{i \in \mathbb{N}}$, dessen erstes Folgeglied datatype_i ist. Dabei steht i für eine Ebene eines Baumes. Die Folgeglieder der Folge lassen sich Startadressen $ref(\text{datatype}_i)$ von Speicherbereichen $ref(\text{datatype}_i)$... $ref(\text{datatype}_i) + size(\text{datatype}_i)$ im Hauptspeicher zuordnen. Hierbei gilt, dass $ref(\text{datatype}_i) \le ref(\text{datatype}_{i+1}) < ref(\text{datatype}_i) + size(\text{datatype}_i)$.

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

Sei datatype_{i,idx_i} das Element / Attribut des Datentyps datatype_i für das gilt: datatype_{i,idx_i} = datatype_{i+1}.

In Abbildung 0.0.1 ist das ganze veranschaulicht. Die ausgegrauten Knoten stellen die verschiedenen Elemente / Attribute datatype_{i,k} des Datentyps datatype_i dar. Allerdings können nur die Knoten datatype_i bzw. datatype_{i,idx} Folgeglieder der Folge (datatype_i)_{i\inN} darstellen.



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 datatype_i ist (z.B. *complex_var.attr3[2]), kann mittels der Formel 0.0.2:

$$ref(\texttt{datatype}_{\texttt{1},\texttt{idx}_1}, \ \dots, \ \texttt{datatype}_{\texttt{n},\texttt{idx}_n}) = ref(\texttt{datatype}_{\texttt{1}}) + \sum_{i=1}^{n-1} \sum_{k=1}^{idx_i-1} size(\texttt{datatype}_{\texttt{i},k}) \quad (0.0.2)$$

berechnet werden. $^{b\ c}$

Dabei darf nur das letzte Folgenglied 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. Dabei geht die erste Adressberechnung vom ersten Datentyp datatype₁ bis direkt zum Zeiger-Datentyp datatype_{pntr} und die zweite Adressberechnung fängt einen Datentyp nach dem Zeiger-Datentyp datatype_{pntr+1} an und geht bis zum letzten Datenyp datatype_n. Bei der zweiten Adressberechnung muss dabei die Adresse $ref(\text{datatype}_1)$ des Summanden aus der Formel 0.0.2 auf den Inhalt^d der Speicherzelle an der Adresse, welche in der ersten Adressberechnung^e $ref(\text{datatype}_1, \ldots, \text{datatype}_{pntr})$ berechnet wurde gesetzt werden: M [$ref(\text{datatype}_1, \ldots, \text{datatype}_{pntr}$)].

Die Formel 0.0.2 stellt dabei eine Verallgemeinerung der Formel ?? 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 ??, da man nicht davon ausgehen kann, dass alle Elemente den gleichen Datentyp haben^f.

Die Knoten Ref(Global(num)) bzw. Ref(Stackframe(num)) repräsentieren dabei den Summanden $ref(datatype_1)$ in der Formel.

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

Die Knoten $\operatorname{Exp}(\operatorname{Stack}(\operatorname{Num}('1')))$ repräsentieren dabei das Lesen des Inhalts $M[ref(\operatorname{datatype}_{1,\operatorname{idx}_1},\ldots,\operatorname{datatype}_{n,\operatorname{idx}_n})]$ der Speicherzelle an der finalen Adresse $\operatorname{ref}(\operatorname{datatype}_{1,\operatorname{idx}_1},\ldots,\operatorname{datatype}_{n,\operatorname{idx}_n})$.

aref(datatype) ordent dabei dem Datentyp datatype eine Startadresse zu.

 $[^]b$ Die Funktion size berechnet die Anzahl Speicherzellen, die ein Datentyp belegt.

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

^dDer Inhalt dieser Speicherzelle ist eine Adresse, da im momentanen Kontext ein Zeiger betrachtet wird.

^eHierbei kommt die Adresse des Zeigers selbst raus.

^fVerbundsattribute haben unterschiedliche Größen.

```
File
2
    Name './example_struct_attr_access.picoc_mon',
    Γ
4
      Block
5
        Name 'main.0',
6
           // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
           → Num('2'))]))
8
           Exp(Num('4'))
           Exp(Num('2'))
9
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')))
15
           Return(Empty())
16
        ]
    ]
```

Code 0.10: Pico C-ANF Pass für Zugriff auf Verbundsattribut.

Im RETI-Blocks Pass in Code 0.11 werden die PicoC-Knoten Ref(Global(Num('0'))), Ref(Attr(Stack(Num('1')),Name('y'))) und Exp(Stack(Num('1'))) durch ihre semantisch entsprechenden RETI-Knoten ersetzt.

```
1 File
2
    Name './example_struct_attr_access.reti_blocks',
4
      Block
        Name 'main.0',
6
           # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
           → Num('2'))]))
           # Exp(Num('4'))
           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;
           LOADIN SP ACC 2;
```

```
STOREIN DS ACC 0;
21
           ADDI SP 2;
22
           # // Exp(Attr(Name('st'), Name('y')))
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
         ٦
39
    ]
```

Code 0.11: RETI-Blocks Pass für Zugriff auf Verbundsattribut.

0.0.1.4 Zuweisung an Verbundsattribut

Die Umsetzung der Zuweisung an ein Verbundsattribut (z.B. st.y = 42) wird im Folgenden anhand des Beispiels in Code 0.12 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 0.12: PicoC-Code für Zuweisung an Verbundsattribut.

Im Abstrakten Syntaxbaum wird eine Zuweisung an ein Verbundsattribut st.y = 42 durch die Knoten Assign(Attr(Name('yt'), Name('y')), Num('42')) dargestellt.

```
1 File
2   Name './example_struct_attr_assignment.ast',
3   [
4    StructDecl
5    Name 'pos',
6    [
7         Alloc(Writeable(), IntType('int'), Name('x'))
8         Alloc(Writeable(), IntType('int'), Name('y'))
9   ],
10   FunDef
11   VoidType 'void',
```

Code 0.13: Abstrakter Syntaxbaum für Zuweisung an Verbundsattribut.

Im PicoC-ANF Pass in Code 0.14 werden die Knoten Assign(Attr(Name('st'), Name('y')), Num('42')) auf eine ähnliche Weise ausgewertet, wie die Knoten Assign(Subscr(Name('ar'), Num('2')), Num('42')), die in Unterkapitel ?? einen Zugriff auf ein Feldelement darstellen. Daher wird hier, um Redundanz zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsten auf das Unterkapitel ?? verwiesen.

Im Gegensatz zum Vorgehen in Unterkapitel ?? muss hier zum Auswerten des linken Knoten Attr(Name('st'),Name('y')),Num('42')) wie in Unterkapitel 0.0.1.3 vorgegangen werden.

```
Name './example_struct_attr_assignment.picoc_mon',
4
      Block
        Name 'main.0',
6
           // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
           → Num('2'))]))
           Exp(Num('4'))
           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')))
15
           Assign(Stack(Num('1')), Stack(Num('2')))
16
           Return(Empty())
17
        ]
```

Code 0.14: PicoC-ANF Pass für Zuweisung an Verbundsattribut.

Im RETI-Blocks Pass in Code 0.15 werden die PicoC-Knoten Exp(Num('42')), Ref(Global(Num('0'))), Ref(Attr(Stack(Num('1')), Name('y'))) und Assign(Stack(Num('1')), Stack(Num('2'))) durch ihre semantisch entsprechenden RETI-Knoten ersetzt.

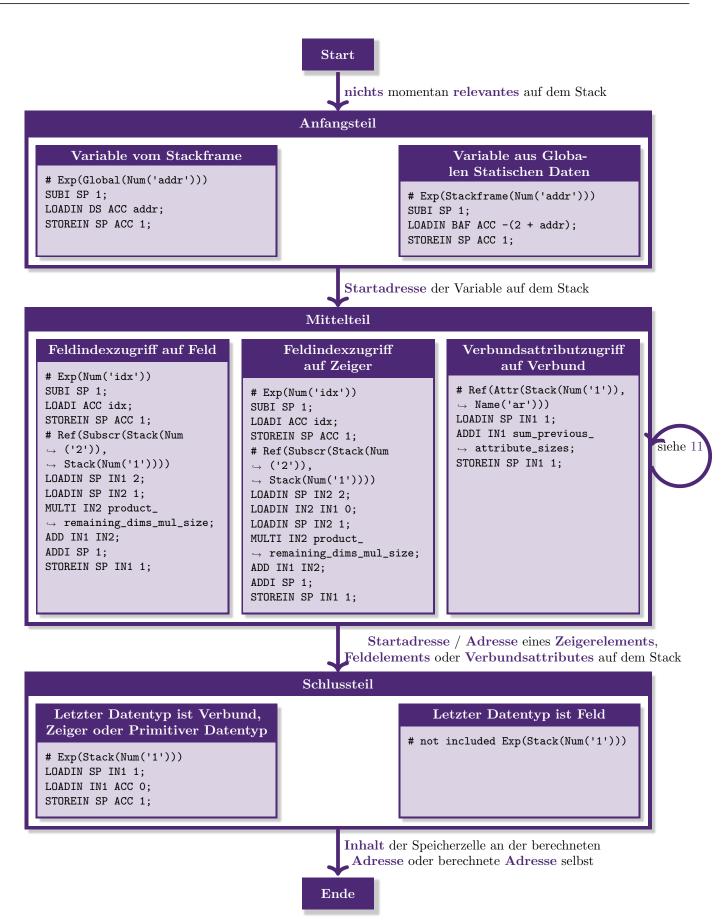
```
1 File
2 Name './example_struct_attr_assignment.reti_blocks',
3 [
4 Block
```

```
Name 'main.0',
 6
           # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
           # Exp(Num('4'))
           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')))
           LOADIN SP ACC 1;
17
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;
           LOADI ACC 42;
25
26
           STOREIN SP ACC 1;
27
           # Ref(Global(Num('0')))
28
           SUBI SP 1;
29
           LOADI IN1 0;
30
           ADD IN1 DS;
31
           STOREIN SP IN1 1;
32
           # Ref(Attr(Stack(Num('1')), Name('y')))
33
           LOADIN SP IN1 1;
34
           ADDI IN1 1;
35
           STOREIN SP IN1 1;
           # Assign(Stack(Num('1')), Stack(Num('2')))
36
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 0.15: RETI-Blocks Pass für Zuweisung an Verbndsattribut.

0.0.2 Umsetzung des Zugriffs auf Zusammengesetzte Datentypen im Allgemeinen

In den Unterkapiteln ??, ?? und 0.0.1 fällt auf, dass der Zugriff auf Elemente / Attribute der in diesen Kapiteln vorkommenden Datentypen (Zeiger, Feld und Verbund) sehr ähnlich abläuft. Es lässt sich ein allgemeines Vorgehen, bestehend aus einem Anfangsteil 0.0.2.1, Mittelteil 0.0.2.2 und Schlussteil 0.0.2.3 darin erkennen. In diesem allgemeinen Vorgehen lassen sich die verschiedenen Zugriffsarten für Elemente bzw. Attribute von Zeigern (z.B. *(pntr + i)), Feldern (z.B. ar[i]) und Verbunden (z.B. st.attr) miteinander kombinieren und so gemischte Ausdrücke, wie z.B. (*st_first.ar) [0] bilden. Dieses allgemeine Vorgehen ist in Abbildung 1 veranschaulicht.



Gemischte Ausdrücke sind möglich, indem im Mittelteil, je nachdem, ob das versteckte Attribut datatype des Ref(exp, datatype)-Knotens ein ArrayDecl(nums, datatype), ein PntrDecl(num, datatype) oder StructSpec(name) beinhaltet ein anderer RETI-Code generiert wird. Hierzu muss im exp-Attribut des Ref(exp, datatype)-Knoten die passende Zugriffsoperation Subscr(exp1, exp2) oder Attr(exp, name) vorliegen.

Der gerade erwähnte RETI-Code berechnet die Startadresse eines gewünschten Zeigerelements, Feldelements oder Verbundsattributs. Zur Berechnung wird die Startadresse des Zeigers, Feldes oder Verbundes, dessen Attribut oder Element berechnet werden soll verwendet. Die Startadresse wird in einem vorherigen Berechnungschritt oder im Anfangsteil auf den Stack geschrieben. Bei einem Zugriff auf einen Feldindex wird zudem mithilfe von entsprechendem RETI-Code dafür gesorgt, dass beim Ausführen zur Laufzeit der Wert des Index berechnet wird und nach der Startadresse auf den Stack geschrieben wird. Dies wurde in Unterkapitel ?? bereits veranschaulicht.

Würde man den Datentyp der Variable Name('var') von ArrayDecl(nums, IntType()) zu PointerDecl(num, IntType()) ändern und hätte eine Operation Subscr(Name('var'), exp2) für die RETI-Code erzeugt werden soll, müssten nur die RETI-Befehle des Mittelteils ausgetauscht werden. Die RETI-Befehle des Anfangsteils und Schlussteils würden unverändert bleiben.

Im Ref(exp, datatype)-Knoten muss die Zugriffsoperation im exp-Attribut zum Datentyp im versteckten Attribut datatype passen. Im Fall, dass Operation und Datentyp nicht zusammenpassen, gibt es eine DatatypeMismatch-Fehlermeldung. Ein Zugriff auf einen Feldindex Subscr(exp1, exp2) kann dabei mit den Datentypen Feld ArrayDecl(nums, datatype) und Zeiger PntrDecl(num, datatype) kombiniert werden. Allerdings wird für beide Kombinationen unterschiedlicher RETI-Code generiert. Das liegt daran, dass in der Speicherzelle des Zeigers PntrDecl(num, datatype) eine Adresse steht und das gewünschte Element erst zu finden ist, wenn man dieser Adresse folgt. Hierfür muss ein anderer RETI-Code erzeugt werden, wie für ein Feld ArrayDecl(nums, datatype), bei dem direkt auf dessen Elemente zugegriffen werden kann. Ein Zugriff auf ein Verbundsattribut Attr(exp, name) kann nur mit dem Datentyp Struct StructSpec(name) kombiniert werden.¹²

Anmerkung Q

Um Verwirrung vorzubeugen, wird hier vorausschauend nochmal darauf hingewiesen, dass eine Dereferenzierung in der Form Deref(exp1, exp2) nicht mehr existiert. In Unterkapitel ?? wurde bereits erklärt, dass alle Knoten Deref(exp1, exp2) im PicoC-Shrink Pass durch Subscr(exp1, exp2) ersetzt wurden. 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 des Ref(exp, datatype)-Knoten beinhaltet den Unterdatentyp (Definition ??), in welchem der Zugriff auf ein Zeigerelment, Feldelement oder Verbundsattribut erfolgt. 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)

¹¹Startadresse / Adresse eines Zeigerelements, Feldelements oder Verbundsattributes auf dem Stack.

¹²Die Bedeutung aller hier erwähnten Knoten und Kompositionen von Knoten wird in den Tabellen der Kapitel ??, ?? und ?? erläutert.

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 wird¹³, 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.

0.0.2.1 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 0.16 erklärt.

```
struct ar_with_len {int len; int ar[2];};

void main() {
    struct ar_with_len st_ar[3];
    int *(*complex_var)[3];
    &complex_var;
}

void fun() {
    struct ar_with_len st_ar[3];
    int (*complex_var)[3];
    &complex_var;
}

&complex_var;
}
```

Code 0.16: PicoC-Code für den Anfangsteil.

Im Abstrakten Syntaxbaum in Code 0.17 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.

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

[
StructDecl
Name 'ar_with_len',

[
Alloc(Writeable(), IntType('int'), Name('len'))
Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))
],

FunDef
VoidType 'void',
```

¹³Und nicht auf ein **Element** des Feldes.

```
Name 'main',
13
         [],
14
           Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
15
           → 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
         Γ
24
           Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
           → Name('st_ar')))
25
           Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
           → Name('complex_var')))
           Exp(Ref(Name('complex_var')))
26
27
        ]
28
    ]
```

Code 0.17: Abstrakter Syntaxbaum für den Anfangsteil.

Im PicoC-ANF Pass in Code 0.18 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.

```
File
    Name './example_derived_dts_introduction_part.picoc_mon',
       Block
         Name 'main.1',
 6
7
8
9
           // Exp(Ref(Name('complex_var')))
           Ref(Global(Num('9')))
           Return(Empty())
10
         ],
11
       Block
12
         Name 'fun.0',
13
14
           // Exp(Ref(Name('complex_var')))
15
           Ref(Stackframe(Num('9')))
16
           Return(Empty())
         ]
18
    ]
```

Code 0.18: PicoC-ANF Pass für den Anfangsteil.

Im **RETI-Blocks** Pass in Code 0.19 werden die Komposition Ref(Global(Num('9'))) bzw. Ref(Stackframe(Num('9'))) durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
1 File
     Name './example_derived_dts_introduction_part.reti_blocks',
     Ε
       Block
         Name 'main.1',
           # // Exp(Ref(Name('complex_var')))
           # Ref(Global(Num('9')))
 9
           SUBI SP 1;
10
           LOADI IN1 9;
11
           ADD IN1 DS;
           STOREIN SP IN1 1;
12
13
           # Return(Empty())
14
           LOADIN BAF PC -1;
15
         ],
16
       Block
         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
         ]
28
    ]
```

Code 0.19: RETI-Blocks Pass für den Anfangsteil.

0.0.2.2 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 0.20 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 0.20: PicoC-Code für den Mittelteil.

Im Abstrakten Syntaxbaum in Code 0.21 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.

```
Name './example_derived_dts_main_part.ast',
4
      StructDecl
        Name 'st',
6
         Γ
7
           Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('1')], IntType('int'))),
           → Name('ar'))
        ],
9
      FunDef
10
         VoidType 'void',
11
        Name 'main',
12
         [],
13
         Γ
14
           Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('var')),
           → Array([Num('42')]))
           Assign(Alloc(Writeable(), StructSpec(Name('st')), Name('complex_var')),
15

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

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

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

17
         1
18
    ]
```

Code 0.21: Abstrakter Syntaxbaum für den Mittelteil.

Im PicoC-ANF Pass in Code 0.22 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')))) 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
    Name './example_derived_dts_main_part.picoc_mon',
      Block
        Name 'main.0',
          // Assign(Name('var'), Array([Num('42')]))
8
          Exp(Num('42'))
9
          Assign(Global(Num('0')), Stack(Num('1')))
10
          // Assign(Name('complex_var'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
11
          Ref(Global(Num('0')))
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'))))
14
          Ref(Global(Num('1')))
15
          Ref(Attr(Stack(Num('1')), Name('ar')))
16
          Exp(Num('0'))
          Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
```

Code 0.22: PicoC-ANF Pass für den Mittelteil.

Im RETI-Blocks Pass in Code 0.23 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, datatpye)-Knoten berücksichtigt werden, was in Unterkapitel 0.0.2 zusammen mit der Abbildung 1 bereits erklärt wurde.

```
1 File
 2
    Name './example_derived_dts_main_part.reti_blocks',
       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')))]))
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;
           # // Exp(Subscr(Subscr(Attr(Name('complex_var'), Name('ar')), Num('0')),

→ BinOp(Num('2'), Sub('-'), Num('2'))))
           # Ref(Global(Num('1')))
28
           SUBI SP 1;
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;
```

```
STOREIN SP IN1 1;
36
           # Exp(Num('0'))
37
           SUBI SP 1;
38
           LOADI ACC 0;
           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;
           STOREIN SP IN1 1;
47
48
           # Exp(Num('2'))
49
           SUBI SP 1;
50
           LOADI ACC 2;
           STOREIN SP ACC 1;
52
           # Exp(Num('2'))
53
           SUBI SP 1;
           LOADI ACC 2;
54
55
           STOREIN SP ACC 1;
56
           # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
57
           LOADIN SP ACC 2;
58
           LOADIN SP IN2 1;
59
           SUB ACC IN2;
60
           STOREIN SP ACC 2;
           ADDI SP 1;
61
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
62
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')))
70
           LOADIN SP IN1 1;
71
           LOADIN IN1 ACC 0;
           STOREIN SP ACC 1;
73
           # Return(Empty())
           LOADIN BAF PC -1;
74
         ]
     ]
```

Code 0.23: RETI-Blocks Pass für den Mittelteil.

0.0.2.3 Schlussteil

Der Schlussteil, bei dem der Inhalt der Speicherzelle an der Adresse, die im Anfangsteil 0.0.2.1 und Mittelteil 0.0.2.2 auf dem Stack berechnet wurde, auf den Stack gespeichert wird¹⁴, wird im Folgenden mithilfe des Beispiels in Code 0.24 erklärt.

¹⁴Und dabei die Speicherzelle der Adresse selbst überschreibt.

```
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 0.24: PicoC-Code für den Schlussteil.

Das Generieren des Abstrakten Syntaxbaumes in Code 0.25 verläuft wie üblich.

```
File
2
    Name './example_derived_dts_final_part.ast',
4
      StructDecl
5
        Name 'st',
6
7
8
9
           Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr'))
        ],
      FunDef
10
        VoidType 'void',
11
        Name 'main',
12
         [],
13
          Exp(Alloc(Writeable(), ArrayDecl([Num('1'), Num('2')], IntType('int')),
14

→ Name('complex_var1')))
          Exp(Alloc(Writeable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),
15

    Name('complex_var2')))

           Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
17
           Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr1')),

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

→ Ref(Name('pntr1')))
           Exp(Subscr(Name('complex_var1'), Num('0')))
19
           Exp(Subscr(Name('complex_var2'), Num('0')))
20
           Exp(Deref(Name('complex_var3'), Num('0')))
21
22
23
    ]
```

Code 0.25: Abstrakter Syntaxbaum für den Schlussteil.

Im PicoC-ANF Pass in Code 0.26 wird das eben angesprochene auf den Stack speichern des Inhalts der berechneten Adresse mit der Komposition Exp(Stack(Num('1'))) dargestellt.

```
Name './example_derived_dts_final_part.picoc_mon',
 4
       Block
         Name 'main.0',
           // Assign(Name('var'), Num('42'))
           Exp(Num('42'))
 9
           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')))
           // 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')))
18
           Exp(Num('0'))
19
           Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
20
           Exp(Stack(Num('1')))
           // Exp(Subscr(Name('complex_var2'), Num('0')))
21
22
           Ref(Global(Num('2')))
23
           Exp(Num('0'))
24
           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')))
28
           Exp(Num('0'))
29
           Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
30
           Exp(Stack(Num('1')))
31
           Return(Empty())
32
         ]
33
    ]
```

Code 0.26: PicoC-ANF Pass für den Schlussteil.

Im RETI-Blocks Pass in Code 0.27 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 0.0.2 zusammen mit der Abbildung 1 bereits erklärt.

```
LOADIN SP ACC 1;
14
           STOREIN DS ACC 4;
15
           ADDI SP 1;
16
           # // Assign(Name('pntr1'), Ref(Name('var')))
           # 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:
32
           # Assign(Global(Num('6')), Stack(Num('1')))
33
           LOADIN SP ACC 1;
34
           STOREIN DS ACC 6;
35
           ADDI SP 1;
36
           # // Exp(Subscr(Name('complex_var1'), Num('0')))
37
           # Ref(Global(Num('0')))
38
           SUBI SP 1;
           LOADI IN1 0;
39
40
           ADD IN1 DS;
41
           STOREIN SP IN1 1;
42
           # Exp(Num('0'))
43
           SUBI SP 1;
           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;
62
           LOADI ACC 0;
63
           STOREIN SP ACC 1;
64
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
65
           LOADIN SP IN1 2;
           LOADIN SP IN2 1;
66
67
           MULTI IN2 2;
68
           ADD IN1 IN2;
           ADDI SP 1;
```

```
STOREIN SP IN1 1;
           # Exp(Stack(Num('1')))
           LOADIN SP IN1 1;
           LOADIN IN1 ACC O;
           STOREIN SP ACC 1;
           # // Exp(Subscr(Name('complex_var3'), Num('0')))
76
           # Ref(Global(Num('6')))
           SUBI SP 1;
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;
           ADD IN1 IN2;
90
91
           ADDI SP 1;
92
           STOREIN SP IN1 1;
93
           # Exp(Stack(Num('1')))
94
           LOADIN SP IN1 1;
95
           LOADIN IN1 ACC 0;
96
           STOREIN SP ACC 1;
97
           # Return(Empty())
98
           LOADIN BAF PC -1;
99
         ]
100
    ]
```

Code 0.27: RETI-Blocks Pass für den Schlussteil.

Literatur