Albert Ludwigs Universität Freiburg

Technische Fakultät

PicoC-Compiler

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

Bachelorarbeit

Abgabedatum: 28th April 2022

Author:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

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

Inhaltsverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

Grammatikverzeichnis

0.0.1 Umsetzung von Structs

0.0.1.1 Deklaration und Definition von Structtypen

Die **Deklaration** eines neuen **Structtyps** (z.B. `struct st {int len; int ar[2];};`) und die **Definition** einer Variable mit diesem **Structtyp** (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 Structtyps

Bevor irgendwas definiert werden kann, muss erstmal ein **Structtyp** deklariert werden. Im **Abstract Syntax Tree** in Code 0.3 wird die **Deklaration eines Structtyps** `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 **Structtyp** `struct st st_var;` wird durch die Komposition `Alloc(Writeable(), StructSpec(Name('st')), Name('st_var'))` dargestellt.

```
1  File
2    Name './example_struct_decl_def.ast',
3    [
4      StructDecl
5        Name 'st',
6        [
7          Alloc(Writeable(), IntType('int'), Name('len'))
8          Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))
9        ],
10     FunDef
11       VoidType 'void',
12       Name 'main',
13       [],
14       [
15         Exp(Alloc(Writeable(), StructSpec(Name('st')), Name('st_var')))
16       ]
17   ]
```

Code 0.2: Abstract Syntax Tree für die Deklaration eines Structtyps

Für den **Structtyp** selbst wird in der **Symboltabelle**, die in Code 0.3 dargestellt ist ein Eintrag mit dem **Schlüssel** `st` erstellt. Die Felder dieses Eintrags `type_qualifier`, `datatype`, `name`, `position` und `size` sind wie üblich belegt, allerdings sind in dem `value_address`-Feld die Attribute des **Structtyps** `[Name('len@st'), Name('ar@st')]` aufgelistet, sodass man über den **Structtyp** `st` die **Attribute** des Structtyps in der **Symboltabelle** nachschlagen kann. Die Schlüssel der **Attribute** haben einen **Suffix** `@st` angehängt, der eine Art **Scope** innerhalb des **Structtyps** für seine Attribut darstellt. Es gilt foglich, dass **innerhalb** eines **Structtyps**

zwei Attribute nicht gleich benannt werden können, aber dafür zwei **unterschiedliche Structtypen** ihre Attribute gleich benennen können.

Jedes der **Attribute** `[Name('len@st'), Name('ar@st')]` erhält auch einen eigenen Eintrag in der **Symboltabelle**, wobei die Felder `type_qualifier`, `datatype`, `name`, `value_address`, `position` und `size` wie üblich belegt werden. Die Felder `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 **Structtyp** `st` wird ein Eintrag in der **Symboltabelle** angelegt. Das `datatyp`-Feld enthält dabei den Namen des **Structtyps** als Komposition `StructSpec(Name('st'))`, wodurch jederzeit alle wichtigen Informationen zu diesem **Structyp**[1] und seinen **Attributen** in der **Symboltabelle** nachgeschlagen werden können.

> Die **Größe** einer Variable `st_var`, die ihm `size`-Feld des **Symboltabelleneintrags** eingetragen ist und mit dem **Structtyp** `struct st` $\{$`datatype`$_1$ `attr`$_1$; ... `datatype`$_n$ `attr`$_n$;$\}$;[a] definiert ist (`struct st st_var;`), berechnet sich dabei aus der Summe der **Größen** der einzelnen **Datentypen** `datatype`$_1$ ... `datatpye`$_n$ der **Attribute** `attr`$_1$, ... `attr`$_n$ des **Structtyps**: $\texttt{size(st)} = \sum_{i=1}^{n} \texttt{size(datatype}_i\texttt{)}$.
>
> [a] Hier 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 Arrays **vor** die Variable zu schreiben von L_C übernommen. Es wird so getann, als würde der komplette **Datentyp** immer **hinter** der Variable stehen: `datatype var`.

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               IntType('int')
        name:                   Name('len@st')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('15'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               ArrayDecl([Num('2')], IntType('int'))
        name:                   Name('ar@st')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('24'))
        size:                   Num('2')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               StructDecl(Name('st'), [Alloc(Writeable(), IntType('int'),
        ↪  Name('len'))Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')),
        ↪  Name('ar'))])
        name:                   Name('st')
        value or address:       [Name('len@st'), Name('ar@st')]
        position:               Pos(Num('1'), Num('7'))
        size:                   Num('3')
```

[1] Wie z.B. vor allem die **Größe** bzw. **Anzahl an Speicherzellen**, die dieser **Structtyp** einnimmt.

```
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('main'), [])
        name:                   Name('main')
        value or address:       Empty()
        position:               Pos(Num('3'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               StructSpec(Name('st'))
        name:                   Name('st_var@main')
        value or address:       Num('0')
        position:               Pos(Num('4'), Num('12'))
        size:                   Num('3')
      }
  ]
```

Code 0.3: Symboltabelle für die Deklaration eines Structtyps

0.0.1.2 Initialisierung von Structs

Die **Initialisierung eines Structs** 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 Structs

Im **Abstract Syntax Tree** in Code 0.5 wird die **Initialisierung eines Structs** `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',
3   [
4     StructDecl
5       Name 'st1',
6       [
7         Alloc(Writeable(), ArrayDecl([Num('2')], PntrDecl(Num('1'), IntType('int'))),
          ↪  Name('attr'))
8       ],
```

```
    StructDecl
      Name 'st2',
      [
        Alloc(Writeable(), IntType('int'), Name('attr1'))
        Alloc(Writeable(), StructSpec(Name('st1')), Name('attr2'))
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
        Assign(Alloc(Writeable(), StructSpec(Name('st2')), Name('st')),
        ↪  Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'),
        ↪  Struct([Assign(Name('attr'), Array([Array([Ref(Name('var')),
        ↪  Ref(Name('var'))])])])])])
      ]
  ]
```

Code 0.5: Abstract Syntax Tree für Initialisierung von Structs

Im **PicoC-Mon Pass** in Code 0.6 wird die **Komposition** `Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')), Struct(...))` auf fast dieselbe Weise ausgewertet, wie bei der **Initialisierung eines Arrays** in Subkapitel **??** daher wird um keine Wiederholung zu betreiben auf Subkapitel **??** verwiesen. Um das ganze interressanter zu gestalten wurde das Beispiel in Code 0.4 so gewählt, dass sich daran eine komplexere, mehrstufige Initialisierung mit **verschiedenen** Datentypen erklären lässt.

Der **Struct-Initializer** Teilbaum `Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Array([Ref(Name('var')), Ref(Name('var'))])])])])`, der beim **Struct-Initializer Container-Knoten** anfängt, wird auf dieselbe Weise nach dem **Depth-First-Search** Prinzip von **links-nach-rechts** ausgewertet, wie es bei der **Initialisierung eines Arrays** in Subkapitel **??** bereits erklärt wurde.

Beim **Iterieren** über den **Teilbaum**, muss beim **Struct-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 **Arrays** oder **Structs** 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 **Struct-Initialiser**, einem **Array-Initialiser** oder einem **Logischen Ausdruck** zu tuen. Bei **Array-** und **Struct-Initialisier** wird einfach über diese nach dem **Depth-First-Search** Schema 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.

```
File
  Name './example_struct_init.picoc_mon',
  [
    Block
      Name 'main.0',
      [
```

```
7         // Assign(Name('var'), Num('42'))
8         Exp(Num('42'))
9         Assign(Global(Num('0')), Stack(Num('1')))
10        // Assign(Name('st'), Struct([Assign(Name('attr1'), Name('var')),
          ↪  Assign(Name('attr2'), Struct([Assign(Name('attr'),
          ↪  Array([Array([Ref(Name('var')), Ref(Name('var'))])])])])]))
11        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-Mon Pass für Initialisierung von Structs

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

```
1 File
2   Name './example_struct_init.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('var'), Num('42'))
8         # 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([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')))
27        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        ]
42    ]
```

Code 0.7: RETI-Blocks Pass für Initialisierung von Structs

0.0.1.3 Zugriff auf Structattribut

Der **Zugriff auf ein Structattribut** (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 Structattribut

Im **Abstract Syntax Tree** in Code 0.9 wird der **Zugriff auf ein Structattribut** `st.y` mithilfe der **Komposition** `Exp(Attr(Name('st'), Name('y')))` dargestellt.

```
1  File
2    Name './example_struct_attr_access.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',
12       Name 'main',
13       [],
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 0.9: Abstract Syntax Tree für Zugriff auf Structattribut

Im **PicoC-Mon Pass** in Code 0.10 wird die Komposition `Exp(Attr(Name('st'), Name('y')))` auf ähnliche Weise ausgewertet, wie die Komposition, die einen **Zugriff auf ein Arrayelement** `Exp(Subscr(Name('ar'), Num('0')))` in Subkapitel **??** darstellt. Daher wird hier, um Wiederholung zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsnten auf das Subkapitel **??** verwiesen.

Die Komposition `Exp(Attr(Name('st'), Name('y')))` wird genauso, wie in Subkapitel **??** durch Kompositionen 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 Subkapitel **??**.

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 **Structs** 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 Arrayindex** in Subkapitel **??**, muss hier vorher nichts anderes als die **Anfangsadresse** des **Structs** auf dem **Stack** liegen. Das **Structattribut** auf welches zugegriffen wird steht bereits in der Komposition `Ref(Attr(Stack(Num('1')), Name('y')))`, nämlich `Name('y')`. Den **Structtyp**, dem dieses Attribut gehört, kann man aus dem versteckten Attribut `datatype` herauslesen. Das versteckte Attribut wird während des Kompiliervorgangs im **PiocC-Mon Pass** dem **Container-Knoten** `Ref(exp, datatype)` angehängt.

Die Berechnung der **Adresse**, für eine Aneinanderreihung von **Zugriffen auf Structattributte** $\texttt{st\_var.attr}_{1,j_1}\ \ldots\ \texttt{.attr}_{n,j_n}$ mehrerer **Structs** $\texttt{struct st}_i\ \texttt{st\_var}$ unterschiedlicher **Structtypen** $\texttt{struct st}_i\ \{\texttt{datatype}_{i,1}\ \texttt{attr}_{i,1};\ \ldots\ \texttt{datatype}_{i,m}\ \texttt{attr}_{i,m};\}$, kann mittels der Formel 0.0.1:

$$\texttt{ref}(\texttt{st\_var.attr}_{1,j_1}\ \ldots\ \texttt{.attr}_{n,j_n}) = \texttt{ref}(\texttt{st\_var}) + \sum_{i=1}^{n}\sum_{k=1}^{j_i-1} \text{size}(\texttt{datatype}_{i,k}) \tag{0.0.1}$$

berechnet werden.[a][b][c]

Die Komposition `Ref(Global(Num('0')))` repräsentiert dabei den Summanden `ref(st_var)` in der Formel.

Der Komposition `Exp(Attr(Name('st'), Name('y')))` repräsentiert dabei einen Summanden $\sum_{k=1}^{j_i-1}\text{size}(\texttt{datatype}_{i,k})$ in der Formel.

Die Komposition `Exp(Stack(Num('1')))` repräsentiert dabei das Lesen des **Inhalts** $\texttt{M}\left[\texttt{ref}(\texttt{st.attr}_{1,j_1}\ \ldots\ \texttt{.attr}_{n,j_n})\right]$ der Speicherzelle an der finalen **Adresse** $\texttt{ref}(\texttt{st.attr}_{1,j_1}\ \ldots\ \texttt{.attr}_{n,j_n})$.

In Unterkapitel 0.0.2.2 wird eine allgemeine Formel für **Zugriffe** auf **Structattribute**, **Arrayelemente** und **Pointer** erklärt, die auf der obigen Formel 0.0.1 und der Formel **??** aufbaut.

[a] `ref(exp)` steht dabei für das Schreiben der **Adresse** von `exp` auf den Stack, wobei `exp` z.B. `st_var.attr` sein könnte.
[b] Die **äußere Schleife** iteriert nacheinander über die **Zugriffe auf Structattribute**. Die **innere Schleife** iteriert über alle **Attribute** des momentan betrachteten **Structtyps** $\texttt{st}_i$, die vor dem Attribut mit der Nummer $\texttt{j}_i$ liegen.
[c] Die Formel baut auf dem Abschnitt über **Structs** aus der Betriebssysteme Vorlesung (Scholl, „Betriebssysteme“) auf.

```
1 File
2   Name './example_struct_attr_access.picoc_mon',
3   [
```

```
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
          ↪  Num('2'))]))
8         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')))
15        Return(Empty())
16      ]
17  ]
```

Code 0.10: PicoC-Mon Pass für Zugriff auf Structattribut

Im **RETI-Blocks Pass** in Code 0.11 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.

```
1 File
2   Name './example_struct_attr_access.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
          ↪  Num('2'))]))
8         # 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        # // 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        ]
39    ]
```

Code 0.11: RETI-Blocks Pass für Zugriff auf Structattribut

0.0.1.4 Zuweisung an Structattribut

Die **Zuweisung an ein Structattribut** (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 Structattribut

Im **Abstact Syntax Tree** wird eine **Zuweisung an ein Structattribut** (z.B. `st.y = 42;`) durch die Komposition `Assign(Attr(Name('st'), 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',
12       Name 'main',
13       [],
14       [
15         Assign(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),
           ↪  Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
16         Assign(Attr(Name('st'), Name('y')), Num('42'))
17       ]
18   ]
```

Code 0.13: Abstract Syntax Tree für Zuweisung an Structattribut

Im **PicoC-Mon Pass** in Code 0.14 wird die Komposition `Assign(Attr(Name('st'), Name('y')), Num('42'))` auf ähnliche Weise ausgewertet, wie die Komposition, die einen **Zugriff auf ein Arrayelement** `Assign(Subscr(Name('ar'), Num('2')), Num('42'))` in Subkapitel **??** darstellt. Daher wird hier, um Wiederholung zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsnten auf das Unterkapitel **??** verwiesen.

Im Gegensatz zum Vorgehen in Unterkapitel **??** muss hier für das Auswerten des **linken** Container-Knoten `Attr(Name('st'), Name('y'))` von `Assign(Attr(Name('st'), Name('y')), Num('42'))` wie in Subkapitel 0.0.1.3 vorgegangen werden.

```
File
  Name './example_struct_attr_assignment.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
        ↪  Num('2'))]))
        Exp(Num('4'))
        Exp(Num('2'))
        Assign(Global(Num('0')), Stack(Num('2')))
        // Assign(Attr(Name('st'), Name('y')), Num('42'))
        Exp(Num('42'))
        Ref(Global(Num('0')))
        Ref(Attr(Stack(Num('1')), Name('y')))
        Assign(Stack(Num('1')), Stack(Num('2')))
        Return(Empty())
      ]
  ]
```

Code 0.14: PicoC-Mon Pass für Zuweisung an Structattribut

Im **RETI-Blocks Pass** in Code 0.15 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.

```
File
  Name './example_struct_attr_assignment.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
        ↪  Num('2'))]))
        # Exp(Num('4'))
        SUBI SP 1;
        LOADI ACC 4;
        STOREIN SP ACC 1;
        # Exp(Num('2'))
        SUBI SP 1;
        LOADI ACC 2;
        STOREIN SP ACC 1;
        # 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;
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;
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      ]
44  ]
```

Code 0.15: RETI-Blocks Pass für Zuweisung an Structattribut

0.0.2 Umsetzung des Zugriffs auf Derived Datatypes im Allgemeinen

In den Unterkapiteln **??**, **??** und 0.0.1 fällt auf, dass der **Zugriff** auf **Elemente** / **Attribute** der in diesen Kapiteln beschriebenen Datentypen (**Pointer**, **Array** und **Struct**) 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 **Pointer**, **Array** und **Struct** 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)`-Container-Knotens ein `ArrayDecl(nums, datatype)`, ein `PntrDecl(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 **Pointerelmements**, **Arrayelements** oder **Structattributs**.

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 Arrayindex** `Subscr(exp1, epp2)` kann dabei mit den Datentypen **Array** `ArrayDecl(nums, datatype)` und **Pointer** `PntrDecl(num, datatype)` kombiniert

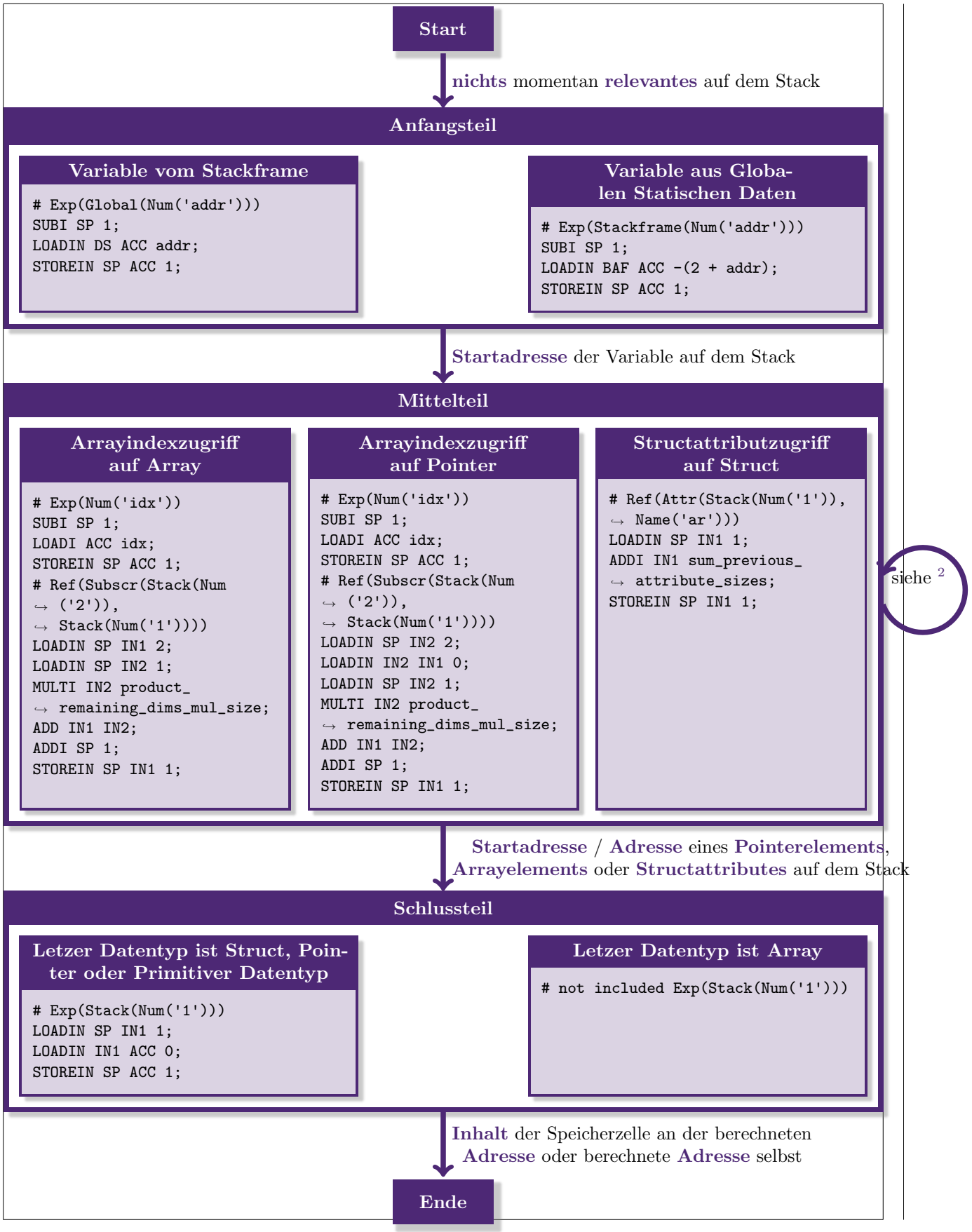


Abbildung 1: Allgemeine Veranschaulichung des Zugriffs auf Derived Datatypes

werden. Allerdings benötigen beide Kombinationen unterschiedlichen **RETI-Code**. Das liegt daran, dass bei einem **Pointer** `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 Structattribut** `Attr(exp, name)` kann nur mit dem Datentyp **Struct** `StructSpec(name)` kombiniert werden.

> 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 **??** bereits erklärt wurde, wurde der **Container-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 Arrayelement** jeweils gegenseitig austauschbar sind. Der **Zugriff auf einen Arrayindex** steht also gleichermaßen auch für eine **Dereferenzierung**.

Das versteckte Attribut `datatype` beinhaltet den **Unterdatentyp**, in welchem der Zugriff auf ein **Pointerelement**, **Arrayelement** oder **Structattribut** 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 **Container-Knoten** oder **Token-Knoten** in diesem **Unterdatentyp** interessiert und die möglicherweise unter diesem momentan betrachteten **Knoten** liegenden **Container-Knoten** und **Token-Knoten** in einem anderen `Ref(exp, versteckte Attribut)`-Container-Knoten dem versteckte Attribut 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 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.

Arrays haben in der Sprache L_C und somit auch in L_{PiocC} die Eigenheit, dass wenn auf ein gesamtes **Array** zugegriffen wird[3], 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.

> **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')))`-Container-Knoten in der Liste eingefügt wird.
>
> Es wird solange dem jeweiligen `exp1` des `Subscr(exp1, exp2)`-Knoten, dem `exp` des `Attr(exp, name)` Knoten oder dem `exp` des `Ref(exp)`-Knotens gefolgt und der jeweilige **Container-Knoten** selbst in ein `Ref(exp)` 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 Aus-**

[3] Und nicht auf ein **Element** des Arrays.
[3] **Startadresse / Adresse** eines **Pointerelements, Arrayelements** oder **Structattributes** auf dem Stack.

drücke eingefügt.

Beim Antreffen eines `Ref(exp)` wird direkt so vorgegangen, wie

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 **Datenetyp** in ihrem `datatype`-Attribut zugewiesen bekommen haben. Alles andere führt zu einer **Fehlermeldung**, für die das versteckte Attribut `error_data` genutzt wird.[a]

[a] Man kann diese Implementierung gut mit dem **Auseinanderrollen** und **Wieder-Einrollen** eines **Schnecken-Gebäcks** vergleichen.

Im Folgenden werden anhand mehrerer Beispiele die einzelnen Abschnitte **Anfangsteil** 0.0.2.1, **Mittelteil** 0.0.2.2 und **Schlussteil** 0.0.2.3 bei der Kompilierung von **Zugriffen** auf **Pointerelemente**, **Arrayelemente**, **Structattribute** bei gemischten Ausdrücken, wie `(*st_first.ar)[0];` einzeln isoliert betrachtet und erläutert.

0.0.2.1 Anfangsteil für Globale Statische Daten und Stackframe

```
1  struct ar_with_len {int len; int ar[2];};
2
3  void main() {
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 0.16: PicoC-Code für den Anfangsteil

```
1  File
2    Name './example_derived_dts_introduction_part.ast',
3    [
4      StructDecl
5        Name 'ar_with_len',
6        [
7          Alloc(Writeable(), IntType('int'), Name('len'))
8          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')))
16        Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], PntrDecl(Num('1'),
          ↪  IntType('int')))), Name('complex_var')))
17        Exp(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')))
25        Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
          ↪  Name('complex_var')))
26        Exp(Name('complex_var'))
27      ]
28  ]
```

Code 0.17: Abstract Syntax Tree für den Anfangsteil

```
1 File
2   Name './example_derived_dts_introduction_part.picoc_mon',
3   [
4     Block
5       Name 'main.1',
6       [
7         // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
          ↪  Name('st_ar')))
8         // Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], PntrDecl(Num('1'),
          ↪  IntType('int')))), Name('complex_var')))
9         // Exp(Name('complex_var'))
10        Exp(Global(Num('9')))
11        Return(Empty())
12      ],
13    Block
14      Name 'fun.0',
15      [
16        // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
          ↪  Name('st_ar')))
17        // Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
          ↪  Name('complex_var')))
18        // Exp(Name('complex_var'))
19        Exp(Stackframe(Num('9')))
20        Return(Empty())
21      ]
22  ]
```

Code 0.18: PicoC-Mon Pass für den Anfangsteil

```
File
  Name './example_derived_dts_introduction_part.reti_blocks',
  [
    Block
      Name 'main.1',
      [
        # // 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'), IntType('int')))), Name('complex_var')))
        # // Exp(Name('complex_var'))
        # Exp(Global(Num('9')))
        SUBI SP 1;
        LOADIN DS ACC 9;
        STOREIN SP ACC 1;
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun.0',
      [
        # // 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'))), Name('complex_var')))
        # // Exp(Name('complex_var'))
        # Exp(Stackframe(Num('9')))
        SUBI SP 1;
        LOADIN BAF ACC -11;
        STOREIN SP ACC 1;
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

Code 0.19: RETI-Blocks Pass für den Anfangsteil

0.0.2.2 Mittelteil für die verschiedenen Derived Datatypes

```c
struct st {int (*ar)[1];};

void main() {
  int var[1] = {42};
  struct st complex_var = {.ar=&var};
  (*complex_var.ar)[0];
}
```

Code 0.20: PicoC-Code für den Mittelteil

```
File
  Name './example_derived_dts_main_part.ast',
  [
```

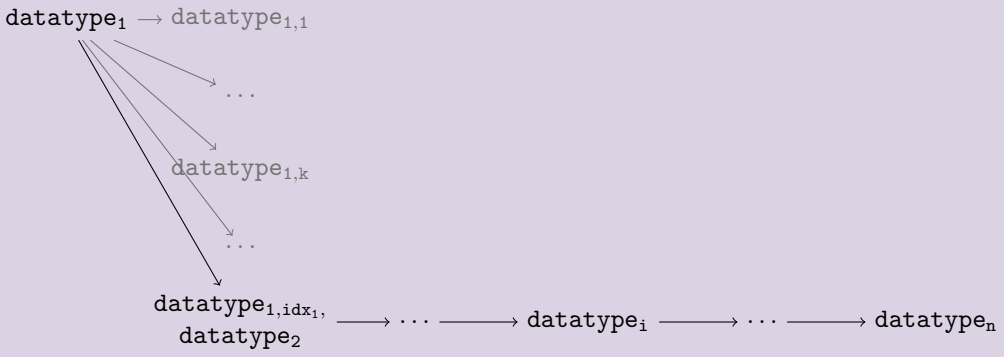
```
4     StructDecl
5       Name 'st',
6       [
7         Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('1')], IntType('int'))),
          ↪  Name('ar'))
8       ],
9     FunDef
10      VoidType 'void',
11      Name 'main',
12      [],
13      [
14        Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('var')),
          ↪  Array([Num('42')]))
15        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')), Num('0')))
17      ]
18  ]
```

Code 0.21: Abstract Syntax Tree für den Mittelteil

Sei $\texttt{datatype}_\texttt{i}$ ein Knotes eines **entarteten Baumes** (wie in Abbildung 0.0.2), dessen Wurzel $\texttt{datatype}_1$ ist. Dabei steht $\texttt{i}$ für eine **Ebene** des entarteten Baumes. Die Knoten des entarteten Baumes lassen sich **Startadressen** $\texttt{ref}(\texttt{datatype}_\texttt{i})$ von Speicherbereichen $\texttt{ref}(\texttt{datatype}_\texttt{i}) \ \ldots \ \texttt{ref}(\texttt{datatype}_\texttt{i}) + \texttt{size}(\texttt{datatype}_\texttt{i})$ im Hauptspeicher zuordnen, wobei gilt, dass $\texttt{ref}(\texttt{datatype}_\texttt{i}) \leq \texttt{ref}(\texttt{datatype}_{\texttt{i+1}}) < \texttt{ref}(\texttt{datatype}_\texttt{i}) + \texttt{size}(\texttt{datatype}_\texttt{i})$[a].

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

Sei $\texttt{datatype}_{\texttt{i,idx}_\texttt{i}}$ ein beliebiges **Element / Attribut** des Datentyps `datatype_i`, sodass gilt: $\texttt{datatype}_{\texttt{i,idx}_\texttt{i}} = \texttt{datatype}_{\texttt{i+1}}$.



(0.0.2)

Die Berechnung der **Adresse** für eine beliebige Folge verschiedener Datentypen $(\texttt{datatype}_{\texttt{1,idx}_1}, \ \ldots, \ \texttt{datatype}_{\texttt{n,idx}_\texttt{n}})$, die das Resultat einer Aneinandereihung von **Zugriffen** auf **Arrayelemente** und **Structattribute** unterschiedlicher Datentypen $\texttt{datatype}_\texttt{i}$ ist (z.B.

`*complex_var.attr2[3]`), kann mittels der Formel 0.0.3:

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

berechnet werden.[bcde]

Dabei darf nur der letzte Knoten $\texttt{datatype}_\texttt{n}$ den Datentyp **Pointer** haben. Ist in einer Folge von **Datentypen** ein Knoten vom Datentyp **Pointer**, der nicht der **letzte Datentyp** $\texttt{datatype}_\texttt{n}$ in der Folge ist, so muss die **Adressberechnung** in 2 Adressberechnungen aufgeteilt werden, wobei die **erste Adressberechnung** vom ersten Datentyp $\texttt{datatype}_1$ bis direkt zum Dantentyp Pointer geht $\texttt{datatype}_\texttt{pntr}$ und die **zweite Adressberechnung** einen Dantentyp nach dem Datentyp Pointer anfängt $\texttt{datatpye}_{\texttt{pntr}+1}$ und bis zum letzten Datenyp $\texttt{datatype}_\texttt{n}$ geht. Bei der **zweiten Adressberechnung** muss dabei die **Adresse** $\texttt{ref}(\texttt{datatype}_1)$ des Summanden aus der Formel 0.0.3 auf den Inhalt der Speicherzelle an der gerade in der **zweiten Adressberechnung** berechneten Adresse $\texttt{M}\left[\texttt{refdatatype}_1 \ldots \texttt{datatype}_\texttt{pntr}\right]$ gesetzt werden.

Die Formel 0.0.3 stellt dabei eine **Verallgemeinerung** der Formel **??** dar, die für alle möglichen Aneinandereihungen von Zugriffen auf **Arrayelemente** und **Structattribute** funktioniert (Z.B. `(*complex_var.attr2)[3]`). Da die Formel allgemein sein muss, lässt sie sich nich 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 Komposition `Ref(Global(num))` bzw. `Ref(Stackframe(num))` aus Unterkapitel 0.0.2.1 repräsentiert dabei den Summanden $\texttt{ref}(\texttt{datatype}_1)$ in der Formel.

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

Die Komposition `Exp(Stack(Num('1')))` repräsentiert dabei das Lesen des **Inhalts** $\texttt{M}\left[\texttt{ref}(\texttt{st.attr}_{1,\texttt{idx}_1} \ \ldots\ .\texttt{attr}_{\texttt{n},\texttt{idx}_\texttt{n}})\right]$ der Speicherzelle an der finalen **Adresse** $\texttt{ref}(\texttt{st.attr}_{1,\texttt{idx}_1} \ \ldots\ .\texttt{attr}_{\texttt{n},\texttt{idx}_\texttt{n}})$.

[a] Es ist Baum, der nur die Datentypen als Knoten enthält, auf die Zugegriffen wurde
[b] `ref(exp)` steht dabei für das Schreiben der **Adresse** von `exp` auf den Stack, wobei `exp` z.B. `complex_var.attr` oder `complex_var[2]` sein könnte.
[c] Die **äußere Schleife** iteriert nacheinander über die **Zugriffe** auf **Pointerlemente**, `Arrayelmente` oder **Structattribute**. Die **innere Schleife** iteriert über alle **Elemente** oder **Attribute** des momentan betrachteten **Datentyps** $\texttt{st}_\texttt{i}$, die vor dem **Element** / **Attribut** mit der Nummer $\texttt{idx}_\texttt{i}$ liegen.
[d] Die Formel 0.0.3 baut auf den beiden Formeln **??** und 0.0.1 auf.
[e]
[f] Structattribute haben unterschiedliche Größen.

```
1  File
2    Name './example_derived_dts_main_part.picoc_mon',
3    [
4      Block
5        Name 'main.0',
6        [
7          // 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')))]))
```

```
        Ref(Global(Num('0')))
        Assign(Global(Num('1')), Stack(Num('1')))
        // Exp(Subscr(Subscr(Attr(Name('complex_var'), Name('ar')), Num('0')), Num('0')))
        Ref(Global(Num('1')))
        Ref(Attr(Stack(Num('1')), Name('ar')))
        Exp(Num('0'))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Exp(Num('0'))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Exp(Stack(Num('1')))
        Return(Empty())
      ]
  ]
```

Code 0.22: PicoC-Mon Pass für den Mittelteil

```
File
  Name './example_derived_dts_main_part.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Name('var'), Array([Num('42')]))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 0;
        ADDI SP 1;
        # // Assign(Name('complex_var'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Assign(Global(Num('1')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 1;
        ADDI SP 1;
        # // Exp(Subscr(Subscr(Attr(Name('complex_var'), Name('ar')), Num('0')), Num('0')))
        # Ref(Global(Num('1')))
        SUBI SP 1;
        LOADI IN1 1;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Ref(Attr(Stack(Num('1')), Name('ar')))
        LOADIN SP IN1 1;
        ADDI IN1 0;
        STOREIN SP IN1 1;
        # Exp(Num('0'))
        SUBI SP 1;
        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('0'))
49        SUBI SP 1;
50        LOADI ACC 0;
51        STOREIN SP ACC 1;
52        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
53        LOADIN SP IN1 2;
54        LOADIN SP IN2 1;
55        MULTI IN2 1;
56        ADD IN1 IN2;
57        ADDI SP 1;
58        STOREIN SP IN1 1;
59        # Exp(Stack(Num('1')))
60        LOADIN SP IN1 1;
61        LOADIN IN1 ACC 0;
62        STOREIN SP ACC 1;
63        # Return(Empty())
64        LOADIN BAF PC -1;
65      ]
66  ]
```

Code 0.23: RETI-Blocks Pass für den Mittelteil

0.0.2.3 Schlussteil für die verschiedenen Derived Datatypes

```
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

```
1 File
2   Name './example_derived_dts_final_part.ast',
3   [
```

```
4     StructDecl
5       Name 'st',
6       [
7         Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr'))
8       ],
9     FunDef
10      VoidType 'void',
11      Name 'main',
12      [],
13      [
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')))
16        Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
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')))
20        Exp(Subscr(Name('complex_var2'), Num('0')))
21        Exp(Deref(Name('complex_var3'), Num('0')))
22      ]
23  ]
```

Code 0.25: Abstract Syntax Tree für den Schlussteil

```
1 File
2   Name './example_derived_dts_final_part.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Exp(Alloc(Writeable(), ArrayDecl([Num('1'), Num('2')], IntType('int')),
          ↪  Name('complex_var1')))
8         // Exp(Alloc(Writeable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),
          ↪  Name('complex_var2')))
9         // Assign(Name('var'), Num('42'))
10        Exp(Num('42'))
11        Assign(Global(Num('4')), Stack(Num('1')))
12        // Assign(Name('pntr1'), Ref(Name('var')))
13        Ref(Global(Num('4')))
14        Assign(Global(Num('5')), Stack(Num('1')))
15        // Assign(Name('complex_var3'), Ref(Name('pntr1')))
16        Ref(Global(Num('5')))
17        Assign(Global(Num('6')), Stack(Num('1')))
18        // Exp(Subscr(Name('complex_var1'), Num('0')))
19        Ref(Global(Num('0')))
20        Exp(Num('0'))
21        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
22        Exp(Stack(Num('1')))
23        // Exp(Subscr(Name('complex_var2'), Num('0')))
24        Ref(Global(Num('2')))
25        Exp(Num('0'))
```

```
26        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
27        Exp(Stack(Num('1')))
28        // Exp(Subscr(Name('complex_var3'), Num('0')))
29        Ref(Global(Num('6')))
30        Exp(Num('0'))
31        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
32        Exp(Stack(Num('1')))
33        Return(Empty())
34      ]
35    ]
```

Code 0.26: PicoC-Mon Pass für den Schlussteil

```
1  File
2    Name './example_derived_dts_final_part.reti_blocks',
3    [
4      Block
5        Name 'main.0',
6        [
7          # // Exp(Alloc(Writeable(), ArrayDecl([Num('1'), Num('2')], IntType('int')),
           ↪  Name('complex_var1')))
8          # // Exp(Alloc(Writeable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),
           ↪  Name('complex_var2')))
9          # // Assign(Name('var'), Num('42'))
10         # Exp(Num('42'))
11         SUBI SP 1;
12         LOADI ACC 42;
13         STOREIN SP ACC 1;
14         # Assign(Global(Num('4')), Stack(Num('1')))
15         LOADIN SP ACC 1;
16         STOREIN DS ACC 4;
17         ADDI SP 1;
18         # // Assign(Name('pntr1'), Ref(Name('var')))
19         # Ref(Global(Num('4')))
20         SUBI SP 1;
21         LOADI IN1 4;
22         ADD IN1 DS;
23         STOREIN SP IN1 1;
24         # Assign(Global(Num('5')), Stack(Num('1')))
25         LOADIN SP ACC 1;
26         STOREIN DS ACC 5;
27         ADDI SP 1;
28         # // Assign(Name('complex_var3'), Ref(Name('pntr1')))
29         # Ref(Global(Num('5')))
30         SUBI SP 1;
31         LOADI IN1 5;
32         ADD IN1 DS;
33         STOREIN SP IN1 1;
34         # Assign(Global(Num('6')), Stack(Num('1')))
35         LOADIN SP ACC 1;
36         STOREIN DS ACC 6;
37         ADDI SP 1;
38         # // Exp(Subscr(Name('complex_var1'), Num('0')))
39         # Ref(Global(Num('0')))
```

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

```
97          LOADIN IN1 ACC 0;
98          STOREIN SP ACC 1;
99          # Return(Empty())
100         LOADIN BAF PC -1;
101       ]
102   ]
```

Code 0.27: RETI-Blocks Pass für den Schlussteil

Literatur

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).