3.3.7 Zeiger

Ein **Zeiger** ist ein Verweis auf einen Behälter, in dem ein Wert gespeichert ist. Somit ist der Wert (R-Wert) einer Zeigervariablen immer ein L-Wert.

Die wichtigste Operation auf Zeigervariablen ist das **Dereferenzieren**, was den Übergang von einem L-Wert zu dem dort gespeicherten Wert bezeichnet.

Beispiel 3.7 (Zeiger in Pascal)**.**
Zeigerdeklaration: `var p: ^`τ`;`
Dereferenzieren: `p^`

Beispiel 3.8 (Zeiger in C)**.**

```
int *ip; // Wert von ip: Zeiger auf Behaelter, in dem ein int steht
int i;
ip = &i; // Adresse (L-Wert) der Variablen i
i = *ip; // dereferenzierter Wert von ip
```

Die Notation „`int *ip`“ ist zwar üblich in C, aber etwas missverständlich, denn eigentlich ist der Typ der Variablen `ip` ein Zeiger auf einen Behälter, der ein `int` enthält, was der Notation „`*int ip`“ entsprechen würde. Tatsächlich verwendet z.B. die Programmiersprache Rust diese Notation:

```
let i: i32 = 42;
let ip: &i32 = &i;
let j: i32 = *ip;
```

Da jeder Datentyp einen Initialwert haben sollte, sollte dies auch für Zeiger gelten. Aus diesem Grund ist es üblich, einen ausgezeichneten Zeigerwert zu definieren: **der leere Zeiger**, der auf kein Objekt verweist.

Beispiel 3.9. Der leere Zeiger hat in verschiedenen Sprachen unterschiedliche Namen:

Pascal:	`nil`
C:	`NULL`
Java:	`null`
Go:	`nil`

Durch die Verwendung von leeren Zeigen ergibt sich ein Problem: beim Dereferenzieren darf der Zeiger nicht leer sein sondern er muss auf einen definierten Behälter zeigen. Somit ist das Dereferenzieren von leeren Zeigern nicht möglich und führt zu einem Laufzeitfehler. Wir betrachten hierzu die Fortsetzung des Beispiels von oben:

```
ip = NULL;
i = *ip; // Laufzeitfehler
```

Dies ist ein sehr häufiger Fehler bei der Programmierung mit Zeigern, sodass der berühmte Informatiker Tony Hoare die Einführung von leeren Zeigern als "billion-dollar mistake" auf einer Konferenz (QCon London) in 2009 bezeichnet hat:[5]

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Aus diesem Grund erlaubt die Sprache Rust zwar Zeiger, aber stellt keinen leeren Zeiger zur Verfügung. Dies hat zur Konsequenz, dass bei der Einführung einer Zeigervariablen immer ein Objekt angegeben werden muss, auf das diese Zeigervariable verweist. Wenn man trotzdem manchmal Zeigervariablen benötigt, die eventuell auf kein definiertes Objekt verweisen, kann man den Typ `Option` verwenden, der dem Typ `Maybe` in Haskell entspricht und wie folgt in Rust vordefiniert ist:

```
enum Option<T> {
    Some(T),
    None,
}
```

Bei der Verwendung eines `Option`-Zeigertyps muss man vor jeder Derferenzierung prüfen, ob es sich eventuell um einen leeren Zeigerwert handelt und dann entsprechend reagieren. Dies ist zwar etwas aufwändiger, vermeidet aber die von Hoare erwähnten unzähligen Programmfehler, die in Systemabstürzen enden.

Eine andere Alternative, Systemabstürze infolge von leeren Zeiger zu vermeiden, ist eine automatische Analyse von Zeigerbenutzungen in Programmen. Da dies im allgemeinen ein unentscheidbares Problem darstellt, wird manchmal der Ansatz verfolgt, Methoden der Programmverifikation mit mächtigen automatischen Programmanalyseverfahren zu kombinieren, wie es z.B. für die Programmiersprache Eiffel gemacht wurde (Meyer, 2017).

Da es aber Zeiger in vielen Programmiersprachen gibt, wollen wir auch deren Semantik definieren. Da die Verwendung von Zeigen als Spiel mit L- und R-Werten betrachtet werden kann, können wir die **Semantik von Zeigern** einfach mit unserer Semantik von L/R-Werten definieren.

Deklaration von Zeigervariablen:

$$\langle E \mid M \rangle \quad \tau * x \quad \langle E; x : (l, *\tau) \mid M[l/NULL] \rangle$$

wobei $l \in free(M)$

[5] `https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/`

Dereferenzierung von Zeigern:

$$\frac{\langle E \mid M\rangle \vdash^R e : l}{\langle E \mid M\rangle \vdash^L *e : l}$$

$$\frac{\langle E \mid M\rangle \vdash^R e : l}{\langle E \mid M\rangle \vdash^R *e : M(l)} \qquad \text{falls } l \neq NULL$$

Hierbei ist l immer eine Speicherreferenz, was in statisch getypten Programmiersprachen durch das Typsystem sichergestellt wird. In dynamisch getypten Programmiersprachen müsste dies beim Dereferenzieren explizit getestet werden.

Beispiel 3.10 (Zeigerverwendung in C)**.**

```
*ip = 3;   // Setze den Behälter, auf den ip verweist, auf 3.
ip = addr; // Setze den Wert von ip (d.h. eine Adresse oder L-Wert)
           // auf addr.
```

Adressoperator:

$$\frac{\langle E \mid M\rangle \vdash^L e : l}{\langle E \mid M\rangle \vdash^R \&e : l}$$

Beispiel 3.11 (Definition von Listen in C)**.**

```
typedef struct {
  int elem;          // Listenelemente: Zahlen
  struct List *next; // Restliste: Zeiger auf den Listenrest
} List;

struct List *head;
// neue Listenstruktur
head = (struct List *) malloc(sizeof(struct List));
(*head).elem = 3;  // Kurzschreibweise: head->elem = 3;
```

Somit wird die leere Liste durch den leeren Zeiger `NULL` repräsentiert, wohingegen eine nichtleere Liste ein Zeiger auf einen `List`-Verbund ist.

Zeigerarithmetik

In manchen Sprachen (Assembler, C) werden Referenzen (L-Werte) mit ganzen Zahlen identifiziert. Als Konsequenz sind dann arithmetische Operationen auf Referenzen erlaubt, z. B. „`(&i)+4`“.
Dies ist eventuell relevant für die Systemprogrammierung, aber

- der Code wird undurchschaubar,
- das Programm ist fehleranfällig (keine Kontrolle der korrekten Zeigerbenutzung durch das Laufzeitsystem), und
- man erhält nicht-portablen Code.

Daher sollte Zeigerarithmetik in höheren Programmiersprachen nicht erlaubt sein. Z.B. unterstützt Rust zwar Zeiger, aber keine Zeigerarithmetik.

Als Alternative bieten z.B. Modula-3 oder auch Java (mittels JNI: Java Native Language Interface) eine klare Trennung von höherer Programmierung und low-level Systemprogrammierung.

Allgemeine Nachteile von Zeigern und Zeigerprogrammierung:

- fehleranfällig: existieren die Objekte, auf die ein Zeiger verweist? (dangling pointers)
- Zeiger verlangen eine explizite Handhabung des Dereferenzierens (z. B. C: `**p`, doppelte Verweise).
- Zeigerarithmetik (wenn das zulässig ist) erlaubt keine Laufzeitprüfungen.
- Zeiger sind unstrukturiert („pointers are gotos“).
- Falls die Programmiersprache keine automatische Speicherverwaltung bietet, sondern der Programmierer den Speicher explizit verwalten muss (`malloc`, `free`...), dann können durch eine fehlerhafte Speicherverwaltung schwer zu durchschauende Fehlersituationen entstehen.

Lösung in funktionalen, logischen und objektorientierte Sprachen (Smalltalk, Eiffel, Java):

- statt Zeiger nur Referenzen auf Objekte
- keine explizite Unterscheidung zwischen Referenz und Wert
- implizites Dereferenzieren

Beispiel 3.12 (Listen in Java)**.**

```
class List {
  int elem;
  List next;
}
...
List head;
head = new List(); // Erzeugung einer neuen Listenstruktur
```

```
head.elem = 3;
```

Dieser Ansatz hat allerdings auch einige Nachteile:

- Alle Werte sind nur durch Dereferenzieren erreichbar.
- Es erfolgt bei einem Wertzugriff immer eine Indirektion durch Zeiger.
- Dies ist umständlich und überflüssig für Grunddatentypen, wie `int`, `char`, ...

Daher wird in manchen objektorientierten Sprache wie Java ein gemischtes Konzept angeboten: Grunddatentypen sind direkt zugreifbar, strukturierte Typen immer über Referenzen.