
Kapitel 2

Einfache Datentypen

In den vielen Jahren der Entwicklung von Programmiersprachen haben sich drei einfache Datentypklassen etabliert [2]. Es gibt Wahrheitswerte, Zahlen und Symbole.



Diese kleinsten Bausteine können wir als atomar bezeichnen. Später sehen wir, wie sich daraus komplexere Datentypen konstruieren lassen.

2.1 Wahrheitswerte

Der Datentyp `Bool` ist vielleicht der bedeutendste von allen. So werden alle primitiven Abläufe in einem Rechner physikalisch durch Strom „an“ oder „aus“ geregelt. Diese zwei Zustände werden mit `True` oder `False` bezeichnet. Im Folgenden wollen wir aufgrund der kürzeren Schreibweise eine 1 als Synonym für `True` und eine 0 als Synonym für `False` verwenden.

Mit einer 0 und einer 1 lassen sich also nur zwei Zustände speichern. Im Allgemeinen bezeichnen wir einen Wahrheitswert als Bit (*binary digit*).

Schauen wir uns dazu ein Beispiel an. Angenommen, wir haben 2 Bit b_1 und b_2 zur Verfügung, dann können wir $2^2 = 4$ unterschiedliche Kombinationen einer Folge von 0 und 1 angeben:

b_1	b_2
0	0
0	1
1	0
1	1

Bei jedem weiteren Bit verdoppelt sich der darstellbare Bereich, denn die komplette Tabelle kann einmal mit 0 und einmal mit 1 erweitert werden:

b_1	b_2	b_3
0	0	0
0	1	0
1	0	0
1	1	0
0	0	1
0	1	1
1	0	1
1	1	1

Mit n Bits lassen sich demnach 2^n unterschiedliche Zustände beschreiben. Die Angaben in Bit bei den Datentypen werden uns später verraten, wie groß die jeweiligen Wertebereiche sind.

Im nächsten Abschnitt sehen wir, dass sich mit einfachen Operationen beliebig komplexe Funktionen darstellen lassen. Dazu erweitern wir die Tabelle der möglichen Belegungen um eine Spalte, die das Ergebnis einer Funktion liefert. Diese nennen wir Werte- oder Funktionstabelle und sie könnte beispielsweise wie folgt aussehen:

b_1	b_2	$f(b_1, b_2)$
0	0	1
0	1	0
1	0	1
1	1	0

Damit können wir das Verhalten einer Funktion für die entsprechende Eingabe beschreiben. Für unser Beispiel liefert die Eingabe $f(1,0) = 1$.

2.1.1 Negation

Mit der Operation NOT (Negation) lässt sich der Wert einer Variable invertieren, aus 1 wird 0 und umgekehrt. Vor der Variable, die negiert werden soll, schreiben wir das Symbol \neg :

a	$\neg a$
0	1
1	0

In Haskell schreiben wir für die Negation `not`:

```
Hugs> not True
False
```

Der NOT-Operator bindet übrigens am stärksten. Das bedeutet, dass die Variable, die unmittelbar dahinter steht, oder ein entsprechender Klammerausdruck negiert werden.

2.1.2 Konjunktion

Die logische Operation AND (Konjunktion) erwartet zwei Eingaben a und b und liefert genau dann eine 1, wenn beide Eingaben eine 1 beinhalten und liefert eine 0 sonst. In der Literatur wird das Symbol \wedge verwendet:

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

In Haskell schreiben wir dafür `&&`. Schauen wir uns zwei Beispiele an:

```
Hugs> True && True
True

Hugs> False && False
False
```

2.1.3 Disjunktion

Für zwei Eingaben a und b , liefert die Operation OR (Disjunktion) eine 1, wenn mindestens eine der beiden Eingaben eine 1 ist. Als Symbol wird \vee verwendet:

a	b	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

In Haskell schreiben wir `||` als Symbol für das OR. Schauen wir uns ein Beispiel an, in dem die vorhergehenden Funktionen mitverwendet werden:

```
Hugs> False || True
True

Hugs> False || (not False && False)
False
```

2.1.4 Exklusiv-Oder

Die Operation XOR (Exklusiv-Oder, Antivalenz) ist der OR-Operation sehr ähnlich. Sie liefert eine 1, wenn genau eine der beiden Eingaben eine 1 enthält:

a	b	$a \otimes b$
0	0	0
0	1	1
1	0	1
1	1	0

Da in Haskell kein Operator für die XOR-Funktion vordefiniert ist, schreiben wir uns diesen jetzt selbst. Anhand der Wahrheitstabelle sehen wir, dass XOR immer dann wahr ist, wenn a und b ungleich zueinander sind. Das kann mit booleschen Operatoren, die wir bereits kennen, ausgedrückt werden:

$$(\text{not } a \ \&\& \ b) \ || \ (a \ \&\& \ \text{not } b)$$

Es gibt aber eine kürzere Möglichkeit. Haskell bietet Operatoren an, um Werte auf Gleichheit zu testen. So liefert `a==b` ein `True`, wenn a und b denselben Wert haben (äquivalent) und `/=` ist `True`, wenn sie ungleich sind (antivalent).

Hier nun die Definition der Funktion `xor` mit dem Ungleichoperator:

```
xor :: Bool -> Bool -> Bool
xor x y = x /= y
```

Was diese Zeilen genau bedeuten, werden wir bald verstehen und auf diese an einer späteren Stelle noch einmal zurückkommen. Wir wollen sie in einem Haskellskript speichern, laden und wie folgt testen:

```
Hugs> xor True (True && False)
True
```

Die anderen logischen Funktionen AND, OR und NOT sind zwar bereits in der Prelude enthalten, ließen sich jedoch analog implementieren.

2.1.5 Boolesche Algebra

Die boolesche Algebra ist ein formales System zur Darstellung von Aussagen mit Wahrheitswerten. Sie hat die Informatik und gerade den Bau von Computern motiviert und geprägt [8].

Für boolesche Funktionen f mit $f : \mathcal{B} \times \mathcal{B} \rightarrow \mathcal{B}$, die zwei boolesche Werte als Argumente erhalten und einen als Ergebnis liefern, wie beispielsweise `&&` oder `| |`, gibt es 2^4 verschiedene Möglichkeiten.

Die drei Funktionen XOR (f_6, \otimes), AND (f_8, \wedge) und OR (f_{14}, \vee) haben wir bereits kennengelernt:

a	b	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Weitere wichtige Funktionen, die wir kennen sollten, sind: Kontradiktion (f_0), NOR (f_1, ∇), NAND ($f_7, \bar{\wedge}$), Äquivalenz (f_9, \Leftrightarrow), Implikation (f_{11}, \Rightarrow) und Tautologie (f_{15}).

2.1.6 Boolesche Gesetze

Zwei boolesche Funktionen sind semantisch äquivalent, wenn für alle Belegungen die Funktionsergebnisse gleich sind. Wir werden das Symbol \equiv für semantische Äquivalenz verwenden. Dies lässt sich z.B. mithilfe einer Wertetabelle überprüfen.

Beispielsweise gilt:

$$\neg(a \vee (\neg a \wedge b)) \equiv \neg a \wedge \neg b$$

Schauen wir uns das in der Wertetabelle an:

a	b	$\neg a$	$\neg b$	$\neg a \wedge b$	$a \vee (\neg a \wedge b)$	$\neg(a \vee (\neg a \wedge b))$	$\neg a \wedge \neg b$
0	0	1	1	0	0	1	1
0	1	1	0	1	1	0	0
1	0	0	1	0	1	0	0
1	1	0	0	0	1	0	0

Da die beiden letzten Spalten die gleichen Einträge aufweisen, haben wir gezeigt, dass die semantische Äquivalenz der Aussage gilt.

Einige semantische Äquivalenzen haben den Rang eines Gesetzes. Die wichtigsten Gesetze sind hier zusammengestellt:

Gesetz	Bezeichnung
$\neg\neg x \equiv x$	Involution
$x \wedge y \equiv y \wedge x$	Kommutativität
$x \vee y \equiv y \vee x$	Kommutativität
$(x \wedge y) \wedge z \equiv x \wedge (y \wedge z)$	Assoziativität
$(x \vee y) \vee z \equiv x \vee (y \vee z)$	Assoziativität
$x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$	Distributivität
$x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$	Distributivität
$x \wedge x \equiv x$	Idempotenz
$x \vee x \equiv x$	Idempotenz
$x \wedge (x \vee y) \equiv x$	Absorption
$x \vee (x \wedge y) \equiv x$	Absorption
$\neg(x \wedge y) \equiv \neg x \vee \neg y$	De Morgan
$\neg(x \vee y) \equiv \neg x \wedge \neg y$	De Morgan
$x \vee (y \wedge \neg y) \equiv x$	Neutralität
$x \wedge (y \vee \neg y) \equiv x$	Neutralität
$x \wedge \neg x \equiv 0$	Komplementarität
$x \vee \neg x \equiv 1$	Komplementarität

Die semantische Äquivalenz lässt sich auch über die Anwendung der booleschen Gesetze nachweisen. Auf unser Beispiel, das wir mit Hilfe der Wertetabelle beweisen konnten, wenden wir jetzt die Gesetze an und formen die Gleichung solange um, bis wir die obige Aussage erhalten:

$$\begin{aligned}
 \neg(a \vee (\neg a \wedge b)) &\equiv \neg a \wedge (a \vee \neg b) \\
 &\equiv (a \wedge \neg b) \vee (\neg a \wedge \neg b) \\
 &\equiv 0 \vee (\neg a \wedge \neg b) \\
 &\equiv \neg a \wedge \neg b
 \end{aligned}$$

In der folgenden Reihenfolge wurden die Gesetze angewendet: De Morgan, Distributivität, Komplementarität und Neutralität. Obwohl es in diesem Beispiel den Anschein hat, sind Umformungen im Allgemeinen leider nicht schneller als das Aufstellen von Wahrheitstabellen.

2.2 Zahlentypen

In Haskell wird den Zahlen, wie übrigens in den meisten anderen Programmiersprachen ebenfalls, eine besondere Aufmerksamkeit geschenkt. Je nach Anwendungsfall wollen wir beispielsweise ganze oder reelle Zahlen, mal mit mehr und mal mit weniger Genauigkeit, verwenden.

Für die gängigsten Zahlenklassen werden entsprechende vordefinierte Datentypen angeboten. Zusätzlich zu den Zahlentypen stehen Operationen und Methoden ebenfalls zur Verfügung. Diese wollen wir mit der Einführung der Datentypen parallel vorstellen.

2.2.1 Datentyp *Int*

Für den Datentyp `Int` stehen 32 Bit zur Verfügung. Das heißt, dass wir Zahlen im Bereich von -2147483648 bis 2147483647 darstellen können. Dabei wird der Speicher zyklisch verwendet, das bedeutet, dass die maximal positive Zahl 2147483647 addiert mit einer 1 entsprechend wieder mit der kleinsten Zahl -2147483648 beginnt.

Später werden wir uns intensiv mit der Erstellung von Funktionen beschäftigen, an dieser Stelle wollen wir aber eine Funktion vorgeben, mit deren Hilfe genau diese zyklische Eigenschaft gezeigt werden kann. Dazu schreiben wir eine Funktion `plus`, die zwei Zahlen vom Datentyp `Int` addiert:

```
plus :: Int -> Int -> Int
plus a b = a+b
```

Auch diese Programmzeilen können wir in unser Haskellskript übernehmen. Jetzt wollen wir diese testen und schauen, was an den Randbereichen des Datentyps `Int` passiert:

```
Hugs> plus 4 5
9

Hugs> plus 2147483647 0
2147483647
```

```
Hugs> plus 2147483647 1
-2147483648
```

Das dritte Beispiel zeigt den Sprung im Vorzeichen und den Wechsel vom größten zum kleinsten darstellbaren Wert eines `Int`. Ein `Int` kann auch normal addiert, subtrahiert, multipliziert und dividiert werden, wie wir es schon in Abschn. 1.3.3 gesehen haben.

Bei der Division gibt es allerdings ein paar kleine Einschränkungen, da `Ints` nur ganze Zahlen darstellen können, das Teilen aber nicht immer aufgeht. Deswegen ist der Operator `/` bei der Teilung zweier `Ints` ungeeignet. Stattdessen gibt es die Funktionen `div` und `mod`, die den Ganzzahlquotienten bzw. den Rest bei der Teilung liefern.

Hier sehen wir dazu ein kleines Beispiel:

```
Hugs> div 123 10
12

Hugs> mod 123 10
3
```

Wenn wir 123 ganzzahlig durch 10 teilen, erhalten wir 12, da $12 \cdot 10 = 120$. Den Rest 3, der ganzzahlig nicht durch 10 teilbar ist, erhalten wir mit dem Modulo-Operator.

2.2.2 Datentyp *Integer*

Der Datentyp `Integer` kann eine beliebig große ganze Zahl repräsentieren. Mit beliebig ist gemeint, dass die Größe lediglich vom Speicher des zur Verfügung stehenden Rechners abhängig ist und nicht von der Sprache Haskell. Ansonsten verhält sich der Datentyp gleich zu einem `Int`.

Wir können so beispielsweise zwei 40-stellige Dezimalzahlen schnell und problemlos addieren:

```
Hugs> (+) 9274826610483620118330284558476215798664
        6497782537618849557316449587356449827318
15772609148102469675646734145832665625982
```

Versuchen Sie das mit einem handelsüblichen Taschenrechner zu machen. An dieser Stelle sollten Sie das gleich mal selbst in Haskell ausprobieren und sich ab jetzt nicht mehr von großen Zahlen beeindrucken lassen.

2.2.3 Datentypen *Float* und *Double*

Gleitkommazahlen, also Zahlen mit einem Komma, können durch den Datentyp `Float` mit einer Beschränkung von 32 Bit dargestellt werden. So sind beispielsweise 1.01, 3.1415 und -12.3 gültige Gleitkommazahlen.

Schauen wir uns ein paar Beispiele auf der Konsole an:

```
Hugs> 16 / 6
2.666666666666667

Hugs> div 16 6
2

Hugs> mod 16 6
4
```

Der Operator `/` erzeugt einen `Float`. Für den Fall, dass eine ganzzahlige Division gewünscht ist, lässt sich die `div`-Funktion verwenden. Für das Beispiel liefert `div 16 6` den Wert 2, da $2 \cdot 6$ kleiner als 16 ist und $3 \cdot 6$ bereits darüber liegt. Mit der Funktion `mod` erhalten wir den Rest. Es gilt für $c = \text{div } a \ b$ und $d = \text{mod } a \ b$ die Gleichung $a = b \cdot c + d$.

Der Datentyp `Double` wird mit doppelter Genauigkeit angegeben und umfasst 64 Bit, die zur Verfügung stehenden Methoden bleiben gleich.

2.3 Zeichen und Symbole

Neben den Wahrheitswerten und Zahlen werden in der Programmierung Zeichen benötigt, um beispielsweise Text auszugeben. Ein Text besteht dabei aus einer Menge von Zeichen und Symbolen.

Der Datentyp `Char` repräsentiert ein einzelnes Zeichen oder ein Symbol des Standard-Unicode [69]. Wenn wir einen `Char` angeben wollen, wird dieser mit ‚a‘ notiert.

Die Reihenfolge des Auftretens im Unicode ist für den Vergleich sehr hilfreich, so sind Buchstaben lexikographisch sortiert:

```
Hugs> 'a' == 'b'
False

Hugs> 'a' <= 'b'
True

Hugs> 'a' >= 'b'
False
```

```
Hugs> 'a' > 'A'  
True
```

Es gibt viele Funktionen, die für die Arbeit mit dem Datentyp `Char` nützlich sind. Im Laufe der Kapitel werden wir einige davon noch kennenlernen.

2.4 Übungsaufgaben

Aufgabe 1) Installieren Sie Haskell auf Ihrem Computer und machen sich mit dem Umgang vertraut. Das Abtippen der bisherigen Beispiele kann dabei sehr hilfreich sein.

Aufgabe 2) Definieren Sie die logischen Operationen `AND`, `OR` und `NOT` in Haskell analog zu `XOR` in Abschn. 2.1.4. Geben Sie sowohl eine Version mit der Verwendung der Operatoren `&&` und `||` an, als auch eine ohne.

Aufgabe 3) Überprüfen Sie die semantische Äquivalenz der beiden De Morgan-Gesetze aus Abschn. 2.1.6 auf Seite 20.

Aufgabe 4) Ein Byte wird durch 8 Bit repräsentiert. Wie viele unterschiedliche Zustände lassen sich mit einem Byte repräsentieren? An der Stelle lohnt es sich die Zweierpotenzen von 2^0 bis 2^{10} aufzuschreiben und einzuprägen.

Aufgabe 5) Welche Bedeutung besitzen vollständige Basen in der booleschen Algebra? Recherchieren Sie und machen Sie sich klar, welche Bedeutung das mit sich bringt.

Haskell-Intensivkurs

Ein kompakter Einstieg in die funktionale
Programmierung

Block, M.; Neumann, A.

2011, XX, 300 S. 64 Abb., 5 Abb. in Farbe. Mit

Online-Extras., Softcover

ISBN: 978-3-642-04717-6