Überblick

Allgemeines

- Für eine Menge an Sorten, eine Menge von Variablen dieser Sorten und eine Menge von Operationen über diesen Sorten können wir nun Ausdrücke bilden
- Typischerweise stellt jede höhere Programmiersprache gewisse *Grunddatentypen* als Sorten zur Verfügung
- Zusätzlich werden (über entsprechende Module) auch gewisse *Grundoperationen* bereitgestellt, also eine Menge von (teilweise überladenen) Operationssymbolen
- Die Semantik dieser Operationen ist durch den zugrundeliegenden Algorithmus zur Berechnung der entsprechenden Funktion definiert
- In den meisten Programmiersprachen ist dies aber vor dem Benutzer verborgen, und es erfolgt auch keine axiomatische Spezifikation, sondern (wenn überhaupt) eine textuelle Beschreibung

Sorten in Java

- Java stellt grundlegende Sorten (Datentypen) (auch *atomare* oder *primitive* Typen genannt) bereit: für $\mathbb{B}$, $CHAR$, eine Teilmenge von $\mathbb{Z}$ und für eine Teilmenge von $\mathbb{R}$ (aber keinen eigenen Grunddatentyp für $\mathbb{N}$)
- Die Werte der primitiven Typen werden intern binär dargestellt.
- Wie bereits diskutiert unterscheiden sich die Datentypen u.a. in der Anzahl der Bits (auch Länge), die für ihre Darstellung verwendet werden und die Einfluss auf den Wertebereich des Typs haben
- Bemerkung: Als objektorientierte Sprache bietet Java zusätzlich die Möglichkeit, benutzerdefinierte Datentypen zu definieren. Diese Möglichkeiten lernen wir im Teil über objektorientierte Modellierung genauer kennen

Sorten in Java

Überblick: Atomare Datentypen (Sorten) in Java:

Name	Länge	Wertebereich
`boolean`	1 Byte	Wahrheitswerte {`true`, `false`}
`char`	2 Byte	Alle Unicodezeichen
`byte`	1 Byte	Ganze Zahlen von -2^7 bis $2^7 - 1$
`short`	2 Byte	Ganze Zahlen von -2^{15} bis $2^{15} - 1$
`int`	4 Byte	Ganze Zahlen von -2^{31} bis $2^{31} - 1$
`long`	8 Byte	Ganze Zahlen von -2^{63} bis $2^{63} - 1$
`float`	4 Byte	Gleitkommazahlen (einfache Genauigkeit)
`double`	8 Byte	Gleitkommazahlen (doppelte Genauigkeit)

Wie erwähnt, stellt Java zu diesen atomaren Datentypen auch entsprechende abstrakte Module mit Grundoperationen bereit. Die folgenden Folien stellen einige Besonderheiten dieser Module vor (für eine Liste an zur Verfügung stehenden Grundoperationen sei ansonsten auf einschlägige Java-Bücher verwiesen).

Grundtypen und -operationen in Java

Der Typ **`boolean`** enthält die Literale **`true`** und **`false`** und folgende Grundoperationen:

Zeichen	Bezeichnung	Bedeutung
`!`	logisches NICHT ($\neg$)	`!a` ergibt **`false`** wenn `a` wahr ist, sonst **`true`**.
`&`	logisches UND ($\wedge$)	`a & b` ergibt **`true`**, wenn sowohl `a` als auch `b` wahr ist. Beide Teilausdrücke werden ausgewertet.
`&&`	sequentielles UND	`a && b` ergibt **`true`**, wenn sowohl `a` als auch `b` wahr ist. Ist `a` bereits falsch, wird **`false`** zurückgegeben und `b` nicht ausgewertet.
`|`	logisches ODER ($\vee$)	`a | b` ergibt **`true`**, wenn mindestens einer der beiden Ausdrücke `a` oder `b` wahr ist. Beide Teilausdrücke (`a` und `b`) werden ausgewertet.
`||`	sequentielles ODER	`a || b` ergibt **`true`**, wenn mindestens einer der beiden Ausdrücke `a` oder `b` wahr ist. Ist bereits `a` wahr, so wird **`true`** zurückgegeben und `b` nicht ausgewertet.
`^`	exkl. ODER (XOR)	`a ^ b` ergibt **`true`**, wenn beide Ausdrücke `a` und `b` einen unterschiedlichen Wahrheitswert haben.

Zeichen (Character) in Java

- Java hat einen eigenen Typ **`char`** für (Unicode-)Zeichen.
- Werte (Literale) werden in einfachen Hochkommata gesetzt, z.B. `'A'` für das Zeichen “A”.
- Einige Sonderzeichen können mit Hilfe von Standard-Escape-Sequenzen dargestellt werden:

Sequenz	Bedeutung
`\b`	Backspace (Rückschritt)
`\t`	Tabulator (horizontal)
`\n`	Newline (Zeilenumbruch)
`\f`	Seitenumbruch (Formfeed)
`\r`	Wagenrücklauf (Carriage return)
`\"`	doppeltes Anführungszeichen
`\'`	einfaches Anführungszeichen
`\\`	Backslash

Ganze Zahlen in Java

- Java hat vier Datentypen für ganze (vorzeichenbehaftete) Zahlen: **`byte`** (Länge: 8 Bit), **`short`** (Länge: 16 Bit), **`int`** (Länge: 32 Bit) und **`long`** (Länge: 64 Bit).
- Werte (*Konstanten*) können geschrieben werden in
  - Dezimalform: bestehen aus den Ziffern `0`, ..., `9`
  - Oktalform: beginnen mit dem Präfix `0` und bestehen aus Ziffern `0`, ..., `7`
  - Hexadezimalform: beginnen mit dem Präfix `0x` und bestehen aus Ziffern `0`, ..., `9` und den Buchstaben `a`, ..., `f` (bzw. `A`, ..., `F`)
- Negative Zahlen erhalten ein vorangestelltes `-`.
- Gehört dieses vorangestellte `-` zum Literal?

Vorzeichen-Operatoren in Java

- Vorzeichen-Operatoren haben die Signatur

$$S \to S$$

mit $S \in \{\texttt{byte}, \texttt{short}, \texttt{int}, \ldots\}$.

- Operationen:

Operator	Bezeichnung	Bedeutung
+	Positives Vorzeichen	`+n` ist gleichbedeutend mit `n`
−	Negatives Vorzeichen	`-n` kehrt das Vorzeichen von `n` um

- Bemerkung: Offenbar ist der Vorzeichen-Operator überladen!!!

Gleitkommazahlen in Java

- Java hat zwei Datentypen für Fließkommazahlen: **`float`** (Länge: 32 Bit) und **`double`** (Länge: 64 Bit).
- Werte (Literale) werden immer in Dezimalnotation geschrieben und bestehen maximal aus
  - Vorkommateil
  - Dezimalpunkt (*)
  - Nachkommateil
  - Exponent `e` oder `E` (Präfix `-` möglich) (*)
  - Suffix `f` oder `F` (**`float`**) oder `d` oder `D` (**`double`**) (*)

  wobei mindestens einer der mit (*) gekennzeichneten Bestandteile vorhanden sein muss.
- Negative Zahlen erhalten ein vorangestelltes `-`.
- Beispiele:
  - double: `6.23`, `623E-2`, `62.3e-1`
  - float: `6.23f`, `623E-2F`, `62.2e-1f`

Arithmetische Operationen in Java

- Arithmetische Operationen haben die Signatur

$$S \times S \to S$$

mit $S \in \{\texttt{byte}, \texttt{short}, \texttt{int}, \ldots\}$.

- Operationen:

Operator	Bezeichnung	Bedeutung
+	Summe	`a+b` ergibt die Summe von `a` und `b`
−	Differenz	`a-b` ergibt die Differenz von `a` und `b`
*	Produkt	`a*b` ergibt das Produkt aus `a` und `b`
/	Quotient	`a/b` ergibt den Quotienten von `a` und `b`
%	Modulo	`a%b` ergibt den Rest der ganzzahligen Division von a durch b

- Bemerkung: Auch diese Operatoren sind überladen

Inkrement-Operationen in Java

- Inkrement-Operationen haben die Signatur

$$S \to S$$

mit $S \in \{\texttt{byte}, \texttt{short}, \texttt{int}, \ldots\}$.

- Operationen:

Operator	Bezeichnung	Bedeutung
`++`	Präinkrement	`++a` ergibt `a+1` und erhöht `a` um `1`
`++`	Postinkrement	`a++` ergibt `a` und erhöht `a` um `1`
`--`	Prädekrement	`--a` ergibt `a-1` und verringert `a` um `1`
`--`	Postdekrement	`a--` ergibt `a` und verringert `a` um `1`

- Bemerkung: Diese Operatoren haben offenbar *Nebeneffekte*:
  Der Ausdruck `a++` hat einen Wert, verändert aber auch die Größe `a`.
  Dieser Effekt ist im funktionalen Paradigma eigentlich unerwünscht, wir werden ihn aber später insbesondere bei der imperativen Programmierung noch genauer besprechen.

Vergleichsoperationen in Java

- Vergleichsoperatoren haben die Signatur

$$S \times S \to \texttt{boolean}$$

mit $S \in \{\texttt{byte}, \texttt{short}, \texttt{int}, \ldots\}$.

- Operationen:

Operator	Bezeichnung	Bedeutung
==	Gleich	a==b ergibt **true**, wenn a gleich b ist
!=	Ungleich	a!=b ergibt **true**, wenn a ungleich b ist
<	Kleiner	a<b ergibt **true**, wenn a kleiner b ist
<=	Kleiner gleich	a<=b ergibt **true**, wenn a kleiner oder gleich b ist
>	Größer	a>b ergibt **true**, wenn a größer b ist
>=	Größer gleich	a>=b ergibt **true**, wenn a größer oder gleich b ist

- Bemerkung: Operatoren sind natürlich ebenfalls überladen

Ausdrücke in Java

- Wir können auch in Java aus Operatoren Ausdrücke (zunächst ohne Variablen) bilden, so wie im vorherigen Kapitel besprochen.
- Laut induktiver Definition von Ausdrücken ist ein Literal ein Ausdruck.
- Interessanter ist, aus Literalen (z.B. den **`int`** Werten `6` und `8`) und mehrstelligen Operatoren (z.B. `+`, `*`, `<`, `&&`) Ausdrücke zu bilden.
- Ein gültiger Ausdruck hat, wie wir wissen, selbst wieder einen Wert (der über die Semantik der beteiligten Operationen definiert ist) und einen Typ (der durch die Ergebnissorte des angewendeten Operators definiert ist):
  - `6 + 8 //Wert: 14 vom Typ int`
  - `6 * 8 //Wert: 48 vom Typ int`
  - `6 < 8 //Wert: true vom Typ boolean`
  - `6 && 8 //ungueltiger Ausdruck`

Typkonversion

- Motivation:
  - Was passiert eigentlich, wenn man verschiedene Sorten/Typen miteinander verarbeiten will?
  - Ist der Java-Ausdruck `6 + 7.3` erlaubt?
  - Eigentlich nicht: Die Operation $+$ ist zwar überladen, d.h.

$$+ : S \times S \to S$$

ist definiert für beliebige primitive Datentypen S, aber es gibt keine Operation

$$+ : S \times T \to U$$

für *verschiedene* primitive Datentypen $S \neq T$.

Das Konzept Typkonversion im Allgemeinen

- Aber es gilt $\mathbb{N}_0 \subseteq \mathbb{Z} \subseteq \mathbb{R}$, d.h. der Ausdruck ist eigentlich sinnvoll wenn man die Zahl 6 als $6.0 \in$ **`double`** interpretieren würde.
- Ganz allgemein nennt man das Konzept der Umwandlung eines Ausdrucks mit einem bestimmten Typ in einen Ausdruck mit einem anderen Typ *Typkonversion*.
- In vielen Programmiersprachen gibt es eine automatische Typkonversion meist vom spezielleren in den allgemeineren Typ (dazu gleich mehr).
- Eine Typkonversion vom allgemeineren in den spezielleren Typ muss (wenn erlaubt) sinnvollerweise immer explizit durch einen *Typecasting*-Operator herbeigeführt werden.
- Es gibt aber auch Programmiersprachen, in denen man grundsätzlich zur Typkonversion ein entsprechendes Typecasting explizit durchführen muss.

Automatische Typkonversion in Java

- Unabhängig davon kennen Sie jetzt das allgemeine Konzept und die Problematik solch einer Typkonversion. Unterschätzen Sie diese nicht als Fehlerquelle bei der Entwicklung von Algorithmen bzw. der Erstellung von Programmen!
- Der Ausdruck `6 + 7.3` ist in Java tatsächlich auch erlaubt, d.h. hier findet offenbar eine explizite Typkonversion statt.
- *Wann* passiert *was* und *wie* bei der Auswertung des Ausdrucks `6 + 7.3`?

Automatische Typkonversion in Java

- Wann:
  Während des Übersetzens des Programmcodes in Bytecode durch den Compiler.
- Was:
  Der Compiler kann dem Ausdruck keinen Typ (und damit auch keinen Wert) zuweisen, da es keine erfüllbare Signatur gibt. Solange kein *Informationsverlust* auftritt, versucht der Compiler diese Situation zu retten.
- Wie:
  Der Compiler konvertiert automatisch den Ausdruck `6` vom Typ **`int`** in einen Ausdruck vom Typ **`double`**, so dass die Operation

$$+ : \texttt{double} \times \texttt{double} \rightarrow \texttt{double}$$

  angewendet werden kann.

Automatische Typkonversion in Java

- Formal gesehen ist diese Konvertierung eine Operation `i->d` mit der Signatur

$$\texttt{i->d} : \textbf{int} \rightarrow \textbf{double}$$

  d.h. der Compiler wandelt den Ausdruck

```
6 + 7.3
```

  um in den Ausdruck

```
i->d(6) + 7.3
```

- Dieser Ausdruck hat offensichtlich einen eindeutigen Typ und damit auch einen eindeutig definierten Wert.

Typkonversion in Java

- Was bedeuted “Informationsverlust”?
- Es gilt folgende “Kleiner-Beziehung” (auch “spezieller/allgemeiner-Beziehung”) zwischen Datentypen:

```
byte < short < int < long < float < double
```

- Beispiele:
  - `1 + 1.7` ist vom Typ **`double`**
  - `1.0f + 2` ist vom Typ **`float`**
  - `1.0f + 2.0` ist vom Typ **`double`**
- Java konvertiert Ausdrücke automatisch in den allgemeineren (“größeren”) Typ, da dabei kein Informationsverlust auftritt.

Explizite Typkonversion in Java: Type-Casting

- Will man eine Typkonversion zum spezielleren Typ durchführen, so muss man dies in Java explizit angeben.
- Dies nennt man allgemein *Type-Casting*.
- In Java erzwingt man die Typkonversion zum spezielleren Typ `type` durch Voranstellen von `(type)`.
- Der Ausdruck

  ```
  (type) a
  ```

  wandelt den Ausdruck `a` in einen Ausdruck vom Typ `type` um.
- Beispiele:
  - `(byte)3` ist vom Typ **`byte`**
  - `(int)(2.0 + 5.0)` ist vom Typ **`int`**
  - `(float)1.3e-7` ist vom Typ **`float`**

Explizite Typkonversion in Java: Type-Casting

- Bei der Typkonversion in einen spezielleren Typ kann Information verloren gehen.
- Beispiele:
  - `(int)5.2` ergibt `5`
  - `(int)-5.2` ergibt `-5`

Der Type-Cast-Operator in Java

Im Ausdruck

```
(type) a
```

ist `(type)` ein Operator. Type-Cast-Operatoren bilden zusammen mit einem Ausdruck wieder einen Ausdruck.
Der Typ des Operators ist z.B.:

$$\texttt{(int)} \quad : \quad \texttt{charUbyteUshortUintUlongUfloatUdouble} \rightarrow \texttt{int}$$

$$\texttt{(float)} \quad : \quad \texttt{charUbyteUshortUintUlongUfloatUdouble} \rightarrow \texttt{float}$$

Sie können also z.B. auch **`char`** in **`int`** umwandeln.

Klingt komisch? Ist aber so! Und was passiert da?

Beispiele

- `(byte) 3` ist vom Typ **`byte`**
- `(int) (2.0 + 5.0)` ist vom Typ **`int`**
- `(float) 1.3e-7` ist vom Typ **`float`**

VORSICHT: Bei der Typkonversion in einen spezielleren Typ kann Information verloren gehen:

- `(int) 5.2` ergibt `5` (ein späteres `(double) 5` ergibt `5.0`)
- `(int) -1.2` ergibt `-1`

Überblick

Funktionen

- Das Konzept der Ausdrücke erlaub uns nun, einfache funktionale Algorithmen zu entwerfen
- Zur Veranschaulichung dient folgendes Beispiel aus der elementaren Physik:

  Ein frei beweglicher Körper mit der Masse m (> 0) werde aus der Ruhelage ein Zeit t lang mit einer auf ihn einwirkenden konstanten Kraft k bewegt. Gesucht ist ein Algorithmus, der (in Abhängigkeit von m, t und k) die Strecke s bestimmt, um die der Körper aus seiner ursprünglichen Lage fortbewegt wird.

Funktionen

- Die Datendarstellung ist in diesem Beispiel sehr einfach: Die Größen m, t und k sind offensichtlich aus $\mathbb{R}$ (mit $m > 0$ und $t \geq 0$)
- Der gesuchte Algorithmus kann damit als Abbildung (Funktion)

$$\mathit{Strecke} : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$

formuliert werden, wobei die Berechnungsvorschrift explizit (als Ausdruck) unter Verwendung von Grundoperationen angegeben werden soll

Funktionen

- Die Idee für diesen Algorithmus ergibt sich aus folgenden einfachen Gesetzen:
  - Die auf einen Körper mit Masse m einwirkende Kraft k erzeugt eine konstante Beschleunigung
$$b = \frac{k}{m}$$
  - Der in der Zeit t zurückgelegte Weg bei einer Bewegung mit konstanter Beschleunigung b ist
$$s = \frac{1}{2} \cdot b \cdot t^2$$
  - Einsetzen liefert
$$Strecke(m, t, k) = (k \cdot t^2)/(2 \cdot m)$$
  - Dies beschreibt, wie man s durch Anwendung verschiedener Grundoperationen auf m, t und k bestimmen kann

Funktionen

Algorithmus 8 (Strecke)

$$Strecke : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$

mit

$$Strecke(m, t, k) = (k \cdot t^2)/(2 \cdot m)$$

(Bedingung: $m > 0$, $t \geq 0$)

Dabei ist

- $Strecke$ der Name des Algorithmus (der Funktion)
- m, t und k Variablen der Sorte $\mathbb{R}$, die sog. (*formalen*) *Parameter* von Strecke.
- $(k \cdot t^2)/(2 \cdot m)$ ein Ausdruck (mit implizitem Typcast) der Sorte $\mathbb{R}$ auch (*Funktions-*) *Rumpf* genannt (die Berechnungsvorschrift); hier kann ein beliebiger Ausdruck stehen; als Variablen für diesen Ausdruck sind zunächst nur die formalen Parameter des Algorithmus erlaubt
- Einschränkungen des Definitionsbereichs werden *Vorbedingungen* genannt

Erweiterung des Modulkonzepts

- Wir können nun das Modulkonzept wie angekündigt erweitern
- Zur Erinnerung: ein Modul ist eine Menge von Sorten mit einer Menge von Operationen über diesen Sorten
- Wir können nun die Menge der Sorten $\{\mathbb{R}\}$ und die Menge der Operationen bestehend aus $\{Strecke\}$ als Modul (das wir vielleicht passenderweise *Strecke* nennen könnten) auffassen
- Anders als bisher, sind die Operationen (in diesem Fall die Funktion $Strecke$) nun nicht mehr abstrakt, sondern durch einen (funktionalen) Algorithmus explizit angegeben.

Benutzerdefinierte Funktionen in Ausdrücken

- Der Algorithmus bzw. die Funktion *Strecke* kann nun auch innerhalb anderer Algorithmen verwendet werden genauso wie die bisherigen Basisoperationen
- Dazu kann die Syntax von Ausdrücken entsprechend erweitert werden:

  Sind $a_1, \ldots, a_n$ Ausdrücke der Sorten $S_1, \ldots, S_n$ und f eine Funktion (Algorithmus) mit der Signatur $S_1 \times \ldots \times S_n \rightarrow S_{n+1}$ (wobei $S_1, \ldots, S_n \in S$), dann ist $f(a_1, \ldots, a_n)$ ein Ausdruck der Sorte S_{n+1}.

Benutzerdefinierte Funktionen in Ausdrücken

Beispiel:

- Die Arbeit berechnet sich aus Kraft multipliziert mit der Strecke:

  Algorithmus 9 (Arbeit)

  $$Arbeit : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$

  mit

  $$Arbeit(m, t, k) = k \cdot Strecke(m, t, k)$$

  (Bedingung: $m > 0$, $t \geq 0$)

- Im Rumpf des Algorithmus wird ein Ausdruck mit der Funktion $Strecke$ gebildet. Die Funktion/der Algorithmus $Strecke$ wird hier *aufgerufen*

Funktionsausführung/-aufruf

- Die Ausführung eines Algorithmus ist ein *Funktionsaufruf*. Was passiert bei so einem Funktionsaufruf?
- Der Aufruf des Algorithmus $Strecke$ mit konkreten Werten ist nichts anderes als eine Substitution
  - Der Algorithmus wird durch einen Ausdruck definiert.
  - Der *Wert* dieses Audrucks ergibt sich aus der Substitution der Eingabevariablen durch die entsprechenden Werte, mit denen der Algorithmus aufgerufen wird
  - Diese Substitution σ wird auch *Variablenbelegung* genannt; die Eingabevariablen werden jeweils mit einem Literal der entsprechenden Sorte belegt
  - $V(\sigma)$ bezeichnet die Menge der (Eingabe-)Variablen in dieser Substitution
  - z.B. beim Aufruf von $Strecke(1.0, 2.0, 3.0)$ werden die Variablen m,t,k mit den Literalen 1.0,2.0,$3.0 \in \mathbb{R}$ belegt, d.h. $V(\sigma) = \{m, t, k\}$ und

$$\sigma = [m/1.0, t/2.0, k/3.0]$$

Wert eines Ausdrucks

Formal

- Gegeben eine Substitution/Variablenbelegung σ ($V(\sigma)$ bezeichnet die Variablen in σ)
- Rekursive Definition des *Wertes* $W_\sigma(u)$ *eines Ausdrucks* u bzgl. σ:
  - Wenn u eine Variable $v \in V(\sigma)$ ist, so ist $W_\sigma(u) = u\,\sigma$
  - Wenn u ein Literal op ist, so ist $W_\sigma(u) = op$
  - Wenn u eine Anwendung eines n-stelligen Operators $op(a_1, \ldots, a_n)$ ist, so ist $W_\sigma(u) = op(W_\sigma(a_1), \ldots, W_\sigma(a_n))$
  - Wenn u ein Funktionsaufruf $f(a_1, \ldots, a_n)$ mit Funktionsrumpf r ist, so ist $W_\sigma(u) = W_\sigma(r)$
- Bemerkungen
  - $W_\sigma(u)$ ergibt als Wert ein Objekt der Sorte von u, bezeichnet durch das entsprechende Literal der Sorte
  - Wir schreiben $W(u)$ statt $W_\sigma(u)$, wenn σ aus dem Kontext klar ersichtlich ist
  - Wenn u eine Variable $v \notin V(\sigma)$ ist, so ist $W_\sigma(u)$ *undefiniert*

Wert eines Ausdrucks

Beispiel:

Algorithmus $f : \mathbb{N}_0 \to \mathbb{N}_0$ mit $f(x) = (x \cdot x) + 4$

Der Aufruf $f(2)$ entspricht einer Substitution $\sigma = [x/2]$ mit $V(\sigma) = \{x\}$

$$
\begin{aligned}
\text{Der Wert } W(f(x)) &= W((x \cdot x) + 4) \\
&= W(x \cdot x) + W(4) \\
&= (W(x) \cdot W(x)) + 4 \\
&= (x[x/2] \cdot x[x/2]) + 4 \\
&= (2 \cdot 2) + 4 \\
&= 4 + 4 \\
&= 8
\end{aligned}
$$

Wert eines Ausdrucks

Weiteres Beispiel:

Aufruf von $Strecke(1.0, 2.0, 3.0)$ entspricht einer Substitution $\sigma = [m/1.0, t/2.0, k/3.0]$ mit $V(\sigma) = \{m, t, k\}$

$$
\begin{aligned}
W(Strecke(1.0, 2.0, 3.0)) &= W((k \cdot t \cdot t)/(2 \cdot m)) \\
&= W(k \cdot t \cdot t)/W(2 \cdot m) \\
&= (W(k) \cdot W(t) \cdot W(t))/(W(2) \cdot W(m)) \\
&= (k[k/3.0] \cdot t[t/2.0] \cdot t[t/2.0])/(2 \cdot m[m/1.0]) \\
&= (3.0 \cdot 2.0 \cdot 2.0)/(2 \cdot 1.0) \\
&= 12.0/2.0 \\
&= 6.0
\end{aligned}
$$

Beachten Sie die Sortenanpassung/Typumwandlung.

Wert eines Ausdrucks

Und noch ein Beispiel:

Aufruf von $Arbeit(1.0, 2.0, 3.0)$ entspricht einer Substitution $\sigma = [m/1.0, t/2.0, k/3.0]$ mit $V(\sigma) = \{m, t, k\}$

$$
\begin{aligned}
W(Arbeit(1.0, 2.0, 3.0)) &= W(k \cdot Strecke(m, t, k)) \\
&= W(k) \cdot W(Strecke(m, t, k)) \\
&= k[k/3.0] \cdot W((k \cdot t \cdot t)/(2 \cdot m)) \\
&= 3.0 \cdot \ldots \\
&= 3.0 \cdot 6.0 \\
&= 18.0
\end{aligned}
$$

Beachten Sie die Sortenanpassung/Typumwandlung.

Bedingte Ausdrücke

- Um Fallunterscheidung zu modellieren, benötigen wir nun noch das Konzept der *bedingten Ausdrücke* (*Terme*)
- Dazu erweitern wir die induktive Definition der Ausdrücke um folgenden Fall
  - Voraussetzung: die Menge der Sorten S enthält die Sorte $\mathbb{B}$
  - Ist b ein Ausdruck der Sorte $\mathbb{B}$ und sind a_1 und a_2 Ausdrücke der selben Sorte $S_0 \in S$, dann ist

    if b **then** a_1 **else** a_2 **endif**

    ein Ausdruck der Sorte S_0.
  - b heißt *Wächter*, a_1 und a_2 heißen *Zweige*.
- Bemerkung: a_1 und/oder a_2 können offenbar wiederum bedingte Ausdrücke (der Sorte S_0) sein, d.h. man kann bedingte Ausdrücke (beliebig) ineinander schachteln

Bedingte Ausdrücke

Beispiele

- Für $x \in \mathbb{Z}$ ist der Absolutbetrag von x bestimmt durch folgenden

Algorithmus 10 (Absolutbetrag)

$$ABS : \mathbb{Z} \to \mathbb{N}_0$$

mit

$$ABS(x) = \textbf{if } x \geq 0 \textbf{ then } x \textbf{ else } -x \textbf{ endif}$$

Bedingte Ausdrücke

Beispiele

- Die der Größe nach mittlere von drei natürlichen Zahlen $x, y, z \in \mathbb{N}_0$ ist bestimmt durch folgenden

Algorithmus 11 (Mitte von drei Zahlen)

$$\mathit{Mitte} : \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$$

mit

$\mathit{Mitte}(x, y, z) =$ **if** $(x < y) \wedge (y < z)$ **then** y **else**
if $(x < z) \wedge (z < y)$ **then** z **else**
if $(y < x) \wedge (x < z)$ **then** x **else**
if $(y < z) \wedge (z < x)$ **then** z **else**
if $(z < x) \wedge (x < y)$ **then** x **else** y **endif**
endif endif endif endif

Bedingte Ausdrücke

- Der Wert $W(u)$ eines bedingten Ausdrucks u

  if b **then** a_1 **else** a_2 **endif**

  ist abhängig von $W(b)$:
  - Ist $W(b) = TRUE$, so ist $W(u) = W(a_1)$
  - Ist $W(b) = FALSE$, so ist $W(u) = W(a_2)$

Bedingte Ausdrücke

- Beispiel: Aufruf von $ABS(-3)$ entspricht einer Substitution $\sigma = [x/3]$

$$
\begin{aligned}
W(ABS(-3)) &= W(\textbf{if } x \geq 0 \textbf{ then } x \textbf{ else } -x \textbf{ endif}) \\
\text{Dazu: } W(x \geq 0) &= (x \geq 0)[x/-3] \\
&= x[x/-3] \geq 0[x/3] \\
&= -3 \geq 0 = FALSE
\end{aligned}
$$

Also:

$$
\begin{aligned}
W(\textbf{if } x \geq 0 \textbf{ then } x \textbf{ else } -x \textbf{ endif}) &= W(-x) = -x[x/-3] \\
&= --3 = 3
\end{aligned}
$$