Waarden en typen

Leereenheid 1

Waarden en typen

INTRODUCTIE

Er bestaat binnen de informatica geen eenduidigheid over wat precies met de term *waarde* bedoeld wordt. De auteur van het tekstboek vat het begrip heel ruim op: met een waarde bedoelt hij elke entiteit die tijdens het uitvoeren van een programma kan worden uitgerekend of opgeslagen, opgenomen kan worden in een gegevensstructuur, meegegeven kan worden als parameter, resultaat is van een functie-aanroep, enzovoort. Kort gezegd kunnen we dus *elke entiteit die tijdens een berekening kan bestaan* beschouwen als waarde.
Bij het onderscheiden van verschillende soorten waarden komen we automatisch op het begrip *type* dat in deze cursus een centrale rol speelt. Een type is in eerste instantie *een verzameling waarden*. Bovendien dwingt een type af dat we alle waarden die tot het type behoren, op gelijke wijze kunnen gebruiken in de bij het type behorende operaties. Niet elke willekeurige verzameling waarden correspondeert daarom met een type.

In elke hogere programmeertaal wordt een onderscheid gemaakt tussen *primitieve typen* en *samengestelde typen*. Daarnaast kennen veel programmeertalen ook *recursieve typen*. Binnen de verschillende talen is een zeer grote variëteit aan gegevensstructuren ontstaan: arrays, records, structures, tupels, variant records, unions, sets, strings, relaties, lijsten, bomen, enzovoort. Nadere bestudering van deze gegevensstructuren laat zien dat ze te begrijpen zijn in een aantal structureringsconcepten zoals *cartesisch product*, *disjuncte vereniging*, *functieruimte* en *recursieve typen*.

In deze leereenheid maken we ook een begin met het bestuderen van *typesystemen*; in leereenheid 7 zullen we hier uitvoerig op terugkomen. We zullen zien dat we (getypeerde) programmeertalen kunnen onderscheiden naar het moment waarop de typecontrole plaatsvindt: tijdens de vertaalfase (statische typering) of tijdens de uitvoering van het programma (dynamische typering). De keuze tussen deze twee vormen van typering heeft belangrijke consequenties voor aspecten als veiligheid en efficiëntie. Een andere keuze bij het ontwerpen van een typesysteem betreft de vorm van *type-equivalentie*. We onderscheiden in deze leereenheid twee vormen, namelijk één waarbij de structuur van typen bepalend is (structurele equivalentie) en één waarbij de naam van typen bepalend is (naamequivalentie). Ook formuleren we het eerste kwaliteitscriterium waarop een programmeertaal te beoordelen is: het *typevolledigheids-principe.* In leereenheden 3 en 4 zullen we nog een aantal van dergelijke principes tegenkomen.

In de laatste paragrafen besteden we aandacht aan *expressies*: programmaconstructies die een waarde opleveren bij evaluatie. We zullen zien dat in functionele talen zoals Haskell en ML, expressies een belangrijkere rol spelen dan in imperatieve talen.

Overigens is in de loop der jaren de kijk op het begrip type langzaam geëvolueerd. In deze leereenheid speelt de bij typen behorende typecontrole een belangrijke rol. We gebruiken typen om aan te geven hoe gegevens in het geheugen dienen te worden opgeslagen en hoe de toegang tot die gegevens vanuit de programmeertaal verloopt. De typecontrole zorgt ervoor dat de standaardfuncties van een programmeertaal de ongetypeerde bitpatronen die in het geheugen staan, altijd interpreteren zoals ze bedoeld zijn.

LEERDOELEN

Na het bestuderen van deze leereenheid wordt verwacht dat u

- in eigen woorden de volgende begrippen kunt omschrijven: waarde, type, standaardtype, discreet type, samengesteld type, recursief type, cartesisch product, disjuncte vereniging, functieruimte, statische typering, dynamische typering, type-equivalentie, structurele equivalentie, naamequivalentie, constructie, functieaanroep, expressie, conditionele expressie
- kunt aangeven welke soorten waarden voorkomen in de talen Java en Haskell
- van de verschillende soorten typen voorbeelden kunt geven in de talen Java en Haskell
- de vier structureringsconcepten kunt noemen met behulp waarvan we samengestelde typen kunnen karakteriseren en deze concepten kunnen herkennen in de verschillende samengestelde typen uit Java en Haskell
- de formules waarmee de cardinaliteiten van de verschillende samengestelde typen kunnen worden berekend, kent en kunt toepassen
- manieren kunt noemen waarop in verschillende talen strings zijn gedefinieerd en kunt aangeven welke consequenties een dergelijke keuze heeft voor de manier waarop we operaties op strings aan de taal kunnen toevoegen
- het verschil kunt uitleggen tussen statisch en dynamisch getypeerde programmeertalen, enkele relevante voorbeelden kunt noemen van beide klassen van talen en kunt aangeven wat de voor- en nadelen zijn van beide benaderingen
- het verschil kunt uitleggen tussen structurele equivalentie en naamequivalentie en weet welke vorm van type-equivalentie wordt toegepast in de talen Java en Haskell
- het typevolledigheidsprincipe kunt formuleren en in grote lijnen kunt aangeven in hoeverre de talen Java en Haskell voldoen aan dit principe
- een onderscheid kunt maken tussen de verschillende soorten expressies en kunt laten zien welke verschillen er tussen Java en Haskell bestaan met betrekking tot het gebruik van expressies.

Studeeraanwijzingen

Bij deze leereenheid hoort hoofdstuk 2 van het tekstboek van Watt. De paragrafen 2.4.3 en 2.7 zijn facultatief.

Let erop dat de nummers van de voorbeelden die uit het tekstboek zijn (dit geldt voor de hele cursus). Zo wordt in deze leereenheid voorbeeld 2.3 uit het tekstboek van Watt omgezet naar een vergelijkbaar voorbeeld in Java.

Alle Java-programma's uit deze leereenheid staan in de map Le01. Alle Haskell-programma's uit deze leereenheid zijn terug te vinden in het bestand Le01.hs in directory Examples. Wij raden u aan om zo nu en dan deze programma's uit te proberen met de Haskell-interpretator.

De studielast van deze leereenheid is circa 13 uur.

LEERKERN

1 Typen

Studeeraanwijzing

Bestudeer in Watt de introductie op hoofdstuk 2 en paragraaf 2.1.

OPGAVE 1.1

Geef een zo volledig mogelijk overzicht van de in Haskell voorkomende soorten/typen van waarden, opgesplitst in de volgende soorten waarden:
– primitieve waarden
– samengestelde waarden
– overige waarden.
Geef ook voorbeelden van de verschillende soorten waarden.

2 Primitieve typen

Studeeraanwijzing

Bestudeer in Watt paragraaf 2.2 inclusief de paragrafen 2.2.1, 2.2.2 en 2.2.3.

Haskell

– Haskell kent als primitieve typen (primitive types): Int, Integer, Float, Double en Char.
– De typen Bool en Rational zijn *geen* primitieve typen, maar worden gemaakt met behulp van constructor-functies en constructor-operaties.
– Haskell kent *geen* deelintervaltype (subrange type).
– Een opsommingstype (enumeration type) is in Haskell *geen* primitief type, maar kan wel gecreëerd worden met behulp van constructor-functies zonder parameters.

Voorbeeld 2.3 in Haskell

In Haskell kan het type Month met behulp van een datadeclaratie worden gedefinieerd.

```
data Month  =  Jan | Feb | Mar | Apr | May | Jun
            |  Jul | Aug | Sep | Oct | Nov | Dec
```

Java

– Java kent als primitieve typen: byte, short, int, long, char, float, double en boolean.
– Java kent *geen* deelintervaltype.
– Een opsommingstype is in Java *geen* primitief type, maar kan wel gecreëerd worden met behulp van een klasse van type enum. Dit is een samengesteld type waarvan de componenten constanten zijn.

Voorbeeld 2.3 in Java

In Java kan het enumeratietype Month worden gedefinieerd met behulp van constanten die instanties zijn van klasse Month zelf.

```
enum Month {
  JAN, FEB, MAR,
  APR, MAY, JUN,
  JUL, AUG, SEP,
  OCT, NOV, DEC;

  public String toString() {
    switch(this) {
      case JAN : return "jan";
      case FEB : return "feb";
      case MAR : return "mar";
      case APR : return "apr";
      case MAY : return "may";
      case JUN : return "jun";
      case JUL : return "jul";
      case AUG : return "aug";
      case SEP : return "sep";
      case OCT : return "oct";
      case NOV : return "nov";
      case DEC : return "dec";
      default  : return "";
    }
  }
}
```

De waarden van type Month zijn aan te duiden met Month.JAN, Month.FEB, enzovoort.

3 Samengestelde typen

3.1 CARTESISCHE PRODUCTEN, STRUCTUREN EN RECORDS

Studeeraanwijzing

Bestudeer in Watt de introductie op paragraaf 2.3 en paragraaf 2.3.1.

Java

Java kent geen record-type als in Ada. Een cartesisch product kan in Java worden gemodelleerd als klassetype waarin voor alle componenten van het product variabelen zijn gedeclareerd.

Voorbeeld 2.5 in Java

Het type Date is in Java als volgt te definiëren:

```
class Date {
  Month m;
  int d;  // Java kent geen deelintervaltype
}
```

Na de declaratie van een variabele van type Date moet een instantie worden gecreëerd voordat er aan de componenten m en d waarden kunnen worden toegekend.

```
Date someday;
someday = new Date();
someday.d = 29;
someday.m = Month.FEB;
```

OPGAVE 1.2
Stel we gebruiken type byte (er zijn 256 waarden van type byte) in plaats van int voor het type van attribuut d in klasse Date. Wat is dan de cardinaliteit van klasse Date?

Haskell

– Haskell kent het tupeltype, dat echter genoteerd wordt als $(T_1, ..., T_n)$, met $T_1 \times ... \times T_n$ als verzameling waarden. Elke tupel $(e_1, ..., e_n)$ met $e_i \in T_i$ behoort tot dit tupeltype.
– Als $n = 0$, krijgen we het 0-tupel ( ). Dit type heeft slechts één waarde, die eveneens met ( ) wordt aangeduid. Zie ook de opmerkingen in het tekstboek over Unit aan het einde van de paragraaf. Haskell kent geen 1-tupels.

Haskell

Met behulp van een typedefinitie kan in Haskell het type Person worden gedefinieerd:

```
type Person  =  (String, String, Int, Float)
```

Een waarde author_textbook van het type Person kan nu als volgt gedefinieerd worden:

```
author_textbook  ::  Person
author_textbook  =  ("Watt", "David", 43, 1.85)
```

Een voorbeeld van een functie die gebruikmaakt van het type Person:

```
voting_age  ::  Person -> Bool
voting_age   (_, _, age, _)
            | age >= 18  =  True
            | otherwise  =  False
```

Of nog wat korter:

```
voting_age (_, _, age, _)  =  age >= 18
```

Merk op dat we in Haskell gebruik kunnen maken van anonieme of 'don't care'-variabelen (_).

Java

Evenals C kent Java het type void waarmee het Unit-type wordt aangegeven.

3.2 FUNCTIERUIMTEN, ARRAYS EN FUNCTIES

Studeeraanwijzing

Bestudeer in Watt paragraaf 2.3.2.

Arrays in Java

In Ada kan de indexverzameling van een array van ieder discreet type zijn, dus Integer, Char, Boolean, deelintervaltypen en opsommingstypen. In Java is de indexverzameling van een array altijd het deelinterval $\{0, 1, ..., n - 1\}$, waarbij n de actuele lengte van de array voorstelt. De lengte van een array maakt geen deel uit van zijn type, maar wordt pas vastgelegd bij de creatie van de array. Het type van een array waarvan de componenten type T hebben, wordt in Java geschreven als:

```
T[]
```

Als een array met lengte n is gecreëerd, dan kunnen we een element van dit type beschouwen als een waarde uit de verzameling

$\{0, ..., n-1\} \rightarrow T$

De cardinaliteit is

$(\#T)^n$

Currying bij arrays in Java

Met betrekking tot meerdimensionale arrays wordt op bladzijde 26 van het tekstboek opgemerkt dat we een n-dimensionale array kunnen beschouwen als een functie van een n-tupel naar een ander type. Zo kunnen we in Java een element van een tweedimensionale array van het type

`T[][]`

beschouwen als een waarde uit de verzameling

$\{0, ..., n_1 - 1\} \times \{0, ..., n_2 - 1\} \rightarrow T.$

Een tweedimensionale array is in Java echter een array waarvan elk element weer een array is; het bovenbeschreven type is dus 'een array van een array van T'. Een element van dit type kunnen we nu beschouwen als een waarde uit de verzameling

$\{0, ..., n_1 - 1\} \rightarrow (\{0, ..., n_2 - 1\} \rightarrow T\ )$

We zien dat hier sprake is van een vorm van currying!
We moeten een element a ook indiceren als $a[i][j]$, in plaats van $a[i, j]$. Daarmee kunnen we a dus beschouwen als een functie die i als parameter heeft, die vervolgens een functie oplevert met j als parameter.

Voorbeeld 2.9 in Java (aanvulling)

We geven een toepassing van deze vorm van currying aansluitend bij voorbeeld 2.9 uit het tekstboek. Het type Pixel hebben we vervangen door int. Na onderstaande declaraties en array-creatie in een Java-programma

```
int[] firstline;
int[][] mywindow;
int firstelem;

mywindow = new int[256][512];
```

is de volgende toekenning mogelijk:

```
firstline = mywindow[0];
```

Uitgaande van de array (functie) mywindow van het (gecurryde) type

$\{0, ..., 255\} \rightarrow \{0, ..., 511\} \rightarrow$ int

hebben we door middel van indiceren (partiële parametrisatie) een eendimensionale array (functie) van het type {0, ..., 511} → int gecreëerd en toegekend aan firstline. Uiteraard is ook volledige parametrisatie mogelijk:

```
firstelem = firstline[0];
firstelem = mywindow[0][0];
```

Haskell

Haskell kent geen arrays.

Voorbeeld 2.10 in Java

Een functieabstractie of kortweg functie is in Java een methode. De beide definities van de functie even in Java luiden:

```
boolean isEven (int n) {
  return n % 2 == 0;
}

boolean isEven (int n) {
  n = Math.abs(n);
  while (n > 1)
    n = n - 2;
  return n == 0;
}
```

OPGAVE 1.3 (Voorbeeld 2.10 in Haskell)
In voorbeeld 2.10 van het tekstboek worden twee definities gegeven van een functieabstractie even in C++.
a Geef een functie voor isEven in Haskell op basis van het eerste algoritme (de mod-operator in Haskell heet `mod`).
b Geef een functie voor isEven in Haskell op basis van het tweede algoritme.
Aanwijzing: maak gebruik van recursie.

OPGAVE 1.4
Gegeven zijn de verzamelingen $T = \{a, b\}$ en $V = \{x, y, z\}$.
a Geef een voorbeeld van een functieruimte $T \to V$.
b Wat is de cardinaliteit van $T \to V$?

3.3 DISJUNCTE VERENIGINGEN, DISCRIMINATED RECORDS EN OBJECTEN

Studeeraanwijzing

Bestudeer in Watt paragraaf 2.3.3.

Een discriminated record is een record (samengesteld type) die geparametriseerd wordt. In voorbeeld 2.13 wordt het type Number gedefinieerd. Een waarde van type Number kan ofwel exact dan wel inexact zijn. De parameter acc die de discriminant (of tag) is, is van type Accuracy. In de declaratie krijgt de discriminant de defaultwaarde exact.

In Ada wordt een disjuncte vereniging gemodelleerd met een discriminated record.

Java

Java kent geen discriminated records als in Ada. Een disjuncte vereniging kan in Java worden gemodelleerd met voor elke component van de vereniging een klasse waarin de betreffende component wordt gedeclareerd. Elke aldus verkregen klasse moet dezelfde interface implementeren. In deze interface kunnen we de methoden specificeren die op de disjuncte vereniging werken.

Voorbeeld 2.12 in Java

Het type Number wordt in Java gerealiseerd met interface Number met voor de componenten exact en approx respectievelijk de klassen Exact en Approx die Number implementeren.

```
interface Number { }

class Exact implements Number {
  int ival;

  Exact(int i) {
    ival = i;
  }
}

class Inexact implements Number {
  float rval;

  Inexact(float r) {
    rval = r;
  }
}
```

De klassen spelen de rol van tag (discriminant) en bepalen welke waarde is opgeslagen. Om te bepalen of we met een Exact-instantie of met een Inexact-instantie te maken hebben, kunnen we gebruikmaken van de operator instanceof.

```
int rounded;
...
if (num instanceof Exact)
  rounded = ((Exact)num).ival;
if (num instanceof Inexact)
  rounded = Math.round(((Inexact)num).rval);
```

Gebruikmakend van het type Number kunnen we bijvoorbeeld een array creëren die zowel integers als floats bevat:

```
Number[] arrayOfNumbers =
      {new Exact(6), new Inexact (3.1416f),
       new Inexact (2.718f), new Exact(4)};
```

Als we de waarden in de array willen afronden naar het dichtstbijliggende gehele getal, ligt het voor de hand om in Number hiervoor een methode (bijvoorbeeld rounded) te specificeren die in beide klassen wordt geïmplementeerd.

```
interface Number {
  int rounded();
}
```

```
class Exact implements Number {
  int ival;

  Exact(int i) {
    ival = i;
  }

  public int rounded() {
    return ival;
  }
}

class Inexact implements Number {
  float rval;

  Inexact (float r) {
    rval = r;
  }

  public int rounded() {
    return Math.round(rval);
  }
}
```

We hoeven nu niet meer de operator instanceof te gebruiken om te bepalen met welke instantie we te maken hebben. Doordat methoden in Java dynamisch worden gebonden, wordt automatisch de juiste definitie van rounded gekozen:

```
for (int i=0; i<arrayOfNumbers.length; i++) {
  rounded = arrayOfNumbers[i].rounded();
  ...
}
```

OPGAVE 1.5

Gegeven zijn de verzamelingen $T = \{a, b\}$ en $V = \{x, y, z\}$. Wat is de cardinaliteit van $T + V$?

Voorbeeld 2.15 (aanvulling)

In Java is combineren van het cartesisch product en de disjuncte vereniging mogelijk als we in plaats van een interface gebruikmaken van een abstracte klasse. In tegenstelling tot een interface, kunnen we in een abstracte klasse variabelen declareren. Het type Figure kan in Java worden gemaakt met behulp van een abstracte klasse met drie subklassen.

```
abstract class Figure {
  float x, y;
}

class Point extends Figure {
  Point(float x, float y) {
    this.x = x;
    this.y = y;
  }
}

class Circle extends Figure {
  float radius;
```

```
  Circle(float x, float y, float radius) {
    this.x = x;
    this.y = y;
    this.radius = radius;
  }
}

class Rectangle extends Figure {
  float height, width;

  Rectangle(float x, float y, float height, float width) {
    this.x = x;
    this.y = y;
    this.height = height;
    this.width = width;
  }
}
```

Op de volgende wijze creëren we een array van het type Figure[ ]:

```
Figure[] arrayOfFigures =
                   {new Point(1.0f, 2.0f),
                    new Circle(0.0f, 0.0f, 5.0f),
                    new Rectangle(1.5f, 2.0f, 3.0f, 4.0f)};
```

OPGAVE 1.6 (Voorbeelden 2.14 en 2.15 in Haskell)
Beschouw voorbeeld 2.14 in het tekstboek waarin met behulp van een discriminated-record in Ada een type Figure wordt gedefinieerd en voorbeeld 2.15 waarin ook een type Figure wordt gedefinieerd in Java met behulp van een abstracte klasse.
a Definieer een soortgelijk type Figure in Haskell, zonder daarbij het cartesisch product en de disjuncte vereniging te combineren.
b Definieer een functie

```
area :: Figure -> Float
```

die van een willekeurige figuur de bijbehorende oppervlakte bepaalt.

---

4 Recursieve typen

4.1 LIJSTEN

Studeeraanwijzing

Bestudeer in Watt de introductie op paragraaf 2.4 en paragraaf 2.4.1.

Voorbeeld 2.17 (aanvulling)

De lege lijst wordt gecreëerd door de constructor van IntList aan te roepen met null als parameter.
De functie cons die een element op kop van een lijst toevoegt, kan als volgt worden geïmplementeerd in klasse IntList:

```
public IntList cons(int elem) {
  IntNode list = new IntNode(elem, first);
  IntList newList = new IntList(list);
  return newList;
}
```

OPGAVE 1.7 (Voorbeeld 2.18 in Haskell)
a Definieer in Haskell een functie

```
empty :: IntList -> Bool
```

waarmee getest kan worden of een integer-lijst al of niet leeg is.
b Definieer in Haskell een functie

```
lastElem :: IntList -> Int
```

die het laatste element van een (niet-lege) lijst oplevert.

4.2 STRINGS

Studeeraanwijzing

Bestudeer in Watt paragraaf 2.4.2.

Haskell

In Haskell is een string – net als in Prolog en Miranda – gedefinieerd als een lijst van characters:

```
type String = [Char]
```

Op deze wijze zijn alle standaardoperaties op lijsten ook toepasbaar op strings, zoals head, tail, map en alle infix-operaties voor het testen van gelijkheid (==) of lexicografische ordening (<, <=, >, >=), alsmede de concatenatie van lijsten (++). De module Char bevat nog extra functies op karakters zoals toUpper en toLower.

Voorbeelden in Haskell

De volgende interactie met de Haskell-interpretator is dus mogelijk:

```
Char> head (tail "abc")
'b'
Char > map toUpper "abc"
"ABC"
Char > "Hello" ++ " world"
"Hello world"
Char > "Hello" < "hello"
True
Char >
```

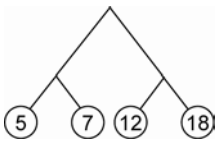
4.3 RECURSIEVE TYPEN IN HET ALGEMEEN

Studeeraanwijzing

Paragraaf 2.4.3 uit het tekstboek behoort niet tot de verplichte leerstof.

Voorbeeld 2.19 in Haskell

Een binaire boom kan op verschillende manieren worden gedefinieerd. De recursieve definitie luidt hier:
Een binaire boom is:
– een element
of
– een linker binaire boom en een rechter binaire boom.
Alleen de eindpunten (bladeren) van een boom bevatten een element en de overige punten niet. Bovendien heeft ieder punt dat geen eindpunt is precies twee takken. In figuur 1.1 ziet u een tekening van het derde voorbeeld in het tekstboek.



FIGUUR 1.1 De binaire boom: Branch (Branch (Leaf 5, Leaf7), Branch (Leaf 12, Leaf 18)

---

5 Typesystemen

5.1 STATISCHE EN DYNAMISCHE TYPERING

Studeeraanwijzing

Bestudeer in Watt de introductie op paragraaf 2.5 en paragraaf 2.5.1.

Haskell

Haskell is statisch getypeerd. Bij het inlezen van het prelude-bestand – of elk ander Haskell-bestand – wordt vóór het vertalen een typecontrole uitgevoerd.

Java

Ook Java is – in tegenstelling tot de OO-taal Smalltalk – een statisch getypeerde taal. Merk op dat bij statische binding door gebruik te maken van een cast dynamische typering wordt afgedwongen. Als de cast ontbreekt, dan moet de typering kloppen.

Zie tekstboek over dynamische typering

Omdat in dynamisch getypeerde talen pas op het laatste moment het type gecontroleerd wordt, zijn lossere typeregimes mogelijk. Het is dan niet langer nodig dat het type van een waarde direct is af te leiden vanuit de programmatekst. Zo wordt de variabele m in voorbeeld 2.21 op een gegeven moment vergeleken met een string en even later met een getal.

OPGAVE 1.8

Op welke manier kunt u in de praktijk nagaan of een door u gebruikte programmeertaal statisch dan wel dynamisch getypeerd is?
Probeer uw methode uit op de Haskell-interpretator.

5.2 TYPE-EQUIVALENTIE

Studeeraanwijzing

Bestudeer in Watt paragraaf 2.5.2.

Bij naamequivalentie zijn twee typen equivalent als ze dezelfde naam hebben. Bij structuurequivalentie zijn twee typen gelijk als ze dezelfde structuur (verzameling waarden) hebben.

Java

– Java maakt gebruik van *naamequivalentie*.
– Twee interface- of klassetypen zijn equivalent als ze dezelfde gekwalificeerde naam hebben – zoals java.awt.Button of mypackage.MyClass$Inner (een binnenklasse) – en dus dezelfde interface of klasse aanduiden.
– Twee array-typen zijn equivalent als hun componenttypen equivalent zijn.
– Java kent geen *typedefinitie* zoals Haskell en Pascal waarmee we aan een bestaand type een nieuwe naam kunnen geven.

OPGAVE 1.9

Op welke manier kunt u nagaan welke vorm van type-equivalentie – structurele equivalentie dan wel naamequivalentie – een gegeven programmeertaal heeft? Probeer uw methode uit op de Haskell-interpretator.

5.3 HET TYPEVOLLEDIGHEIDSPRINCIPE

Studeeraanwijzing

Bestudeer in Watt paragraaf 2.5.3.

Alternatieve formulering

We geven een alternatieve formulering van het typevolledigheids-principe: wat met waarden van één type kan, moet – voor zover zinvol – ook met waarden van een ander type kunnen.

In tabel 1.1 geven we voor de taal Java een overzicht welke taalconstructies welke typen mogen hebben. Horizontaal zijn respectievelijk de volgende constructies opgenomen: constante, argument van een methode, resultaat van een methode, een object-component en een array-component. Verticaal zijn de verschillende typen opgenomen zoals al eerder opgesomd in paragraaf 1: primitieve typen; samengestelde typen die, vanwege het verschil in behandeling in Java, opgesplitst zijn in klassetypen en String; referenties (pointers), variabele-referenties (we komen hier in leereenheid 2 op terug) en methoden (komt terug in leereenheid 5).

TABEL 1.1 Overzicht van de toepassing van het typevolledigheidsprincipe in de taal Java

type	*constante*	*methode-argument*	*methode-resultaat*	*object-component*	*array-component*
primitief	ja	ja	ja	ja	ja
samengesteld:					
– klasse	nee	nee	nee	ja	nee
– String	ja	nee	nee	ja	nee
referentie (pointer)	nee	ja	ja	ja	ja
variabele-referentie	nee	nee	nee	ja	nee
methode	nee	nee	nee	ja	nee

NB: Het is niet nodig dat u deze tabel uit het hoofd kent. Wel is het belangrijk om in te zien in hoeverre en op welke belangrijke punten het typevolledigheidsprincipe bij de taal Java geweld wordt aangedaan.

Toelichting op de tabel

– In Java kunnen constanten alleen van een primitief type of van het type String zijn; dit laatste is vreemd omdat in Java het type String een klassetype is, zie paragraaf 4.3.
– Een samengesteld type kan in Java niet worden meegegeven als argument aan een methodeaanroep, wel kunnen we een *referentie* naar een samengesteld type meegeven).
– Een methode kan in Java niet worden meegegeven als argument aan een methodeaanroep.
– Het resultaat van een methode kan in Java geen methode of samengesteld type zijn, wel een referentie naar een samengesteld type.
– In Java kunnen methoden en variabele-referenties wel componenten zijn van een object, maar niet van een array.

OPGAVE 1.10
Voor de functionele taal Haskell is er geen verschil tussen de verschillende soorten samengestelde typen, zodat deze in één rij zijn samengevat (zie tabel 1.2). Ook kent Haskell geen (variabele-)referenties.

TABEL 1.2 Overzicht van de toepassing van het typevolledigheidsprincipe in de taal Haskell (niet volledig ingevuld)

type	*constante*	*functie-argument*	*functie-resultaat*	*tupel-component*	*lijstcomponent*
primitief	ja	ja	ja	ja	ja
samengesteld	?	ja	ja	ja	?
functie	ja	?	?	?	ja

We zien dat in elk van de ingevulde plaatsen in tabel 1.2 ja staat. Verwacht u dat dit ook voor de vijf overgebleven plaatsen geldt? Verzin voorbeelden en probeer deze eventueel uit met de Haskell-interpretator.

6 Expressies

Expressies spelen in functionele talen een veel belangrijker rol dan in imperatieve talen. Dit komt doordat besturingsstructuren zoals die in imperatieve talen voorkomen, in functionele talen gerealiseerd worden met expressies.

6.1 LITERALS

Studeeraanwijzing

Bestudeer in Watt de introductie op paragraaf 2.6 en paragraaf 2.6.1.

Haskell en Java

Zowel Java als Haskell kennen numerieke letterlijke waarden (numeric literals) onderverdeeld in *gehele getallen* (zoals 1 en –5), *drijvende-punt-getallen* (zoals 3.5 en -0.7e-3), *karakters* (zoals 'a' en '%') en *strings* (zoals "aap", "a" en ""). Al deze expressies stellen vaste, onveranderlijke waarden voor.
Java kent bovendien nog de letterlijke waarde null.

6.2 CONSTRUCTIES

Studeeraanwijzing

Bestudeer in Watt paragraaf 2.6.2.

Voorbeeld in Java

Een ander voorbeeld van het gebruik van constructies vinden we in Java in de vorm van array-initializers. Hierin is het mogelijk om een array bij de declaratie te initialiseren, bijvoorbeeld:

```
int[] a = {7, 2, 5};
```

Wat bij declaratie wél mag, is in Java *niet* toegestaan bij een toekenning, zoals bijvoorbeeld in:

```
a = {3, 4, 2};
```

Wel is de volgende toekenning toegestaan:

```
a = new int[] {7, 2, 5};
```

Voorbeeld 2.28 (aanvulling)

Met de volgende functiedefinitie

```
thisyear :: Int
thisyear  = 2007

days_of_month  ::  [Int]
days_of_month  =
   [31, if isLeap thisyear then 29 else 28,
    31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

krijgen we:

```
Le01> days_of_month
[31,28,31,30,31,30,31,31,30,31,30,31]
Le01>
```

6.3 FUNCTIEAANROEPEN

Studeeraanwijzing

Bestudeer in Watt paragraaf 2.6.3.

Haskell

Haskell behandelt functieabstracties als eerste-klaswaarden (zie ook tabel 1.2 bij opgave 1.10). Dit betekent onder meer dat we in een functieaanroep *F(AP)* voor *F* niet noodzakelijk een functienaam (identifier) hoeven te gebruiken, maar dat elke expressie die een functieabstractie oplevert, ook toegestaan is.
Het volgende voorbeeld is vergelijkbaar met het voorbeeld uit het tekstboek.

Voorbeeld in Haskell

In Haskell is de volgende definitie mogelijk:

```
twotimes x  =  2 * x
threetimes x  =  3 * x

two_or_threetimes x  =
   (if x < 0 then twotimes else threetimes) x
```

Deze functie kan als volgt worden toegepast:

```
Le01> two_or_threetimes 5
15
Le01> two_or_threetimes (-5)
-10
Le01>
```

Haskell

In Haskell kunnen we functies – met minstens twee parameters – als operator gebruiken, door ze tussen back-quotes te plaatsen, en andersom kunnen we operatoren als functie gebruiken, door ze tussen haakjes te plaatsen. Operatoren zijn in Haskell ook op dezelfde wijze als functies te definiëren en zijn dus – net als bij Ada en C++ – *volledig equivalent* aan functies. We geven nog enkele voorbeelden in Haskell.

Voorbeelden in Haskell

De conjunctie-operator en de relationele operatoren zijn in de Haskell-prelude als operator gedefinieerd, maar zijn ook als functie (prefix) te gebruiken.

```
Hugs> :type (&&)
(&&) :: Bool -> Bool -> Bool
Hugs> (&&) ((<) 1 3) ((>=) 5 5)
True
Hugs>
```

De grootste gemene deler (greatest common divisor) is in de Haskell-prelude gedefinieerd als (prefix-)functie, maar is zonder bezwaar als (infix-)operator te gebruiken.

```
Hugs> :type gcd
gcd :: Integral a => a -> a -> a
Hugs> gcd 12 20
4
Hugs> 30 `gcd` 12
6
Hugs>
```

6.4 CONDITIONELE EXPRESSIES

Studeeraanwijzing

Bestudeer in Watt paragraaf 2.6.4.

Haskell

Haskell kent zowel een *if*-expressie als een *case*-expressie.
Omdat Haskell geen opdrachten (commands) kent, kent deze taal ook geen conditionele of keuzeopdrachten (zoals in Pascal en Java); deze behandelen we in paragraaf 7.6 van leereenheid 2: Keuzeopdrachten.

Voorbeelden 2.30 en 2.31 in Haskell

In Haskell kan met behulp van een conditionele expressie eenvoudig een functie max' worden gedefinieerd:

```
max' x y  =  if x > y then x else y
```

Overigens is de functie max in de Haskell-prelude gedefinieerd zonder gebruik te maken van een conditionele expressie:

```
max x y | x <= y    =  y
        | otherwise =  x
```

In Haskell is met behulp van een case-expressie als volgt een functie monthsize te definiëren:

```
monthsize  ::  String -> Int -> Int
monthsize thismonth thisyear
   =  case thismonth of
      "Feb" -> if leap thisyear then 29 else 28
      "Apr" -> 30
      "Jun" -> 30
      "Sep" -> 30
      "Nov" -> 30
      _     -> 31
```

Voorbeelden 2.30 en 2.31 in Java

Ook Java kent conditionele expressies, deze komen overeen met if-expressies.
Voorbeeld 2.30 kunnen we in Java als volgt noteren:

```
double max(double x, double y) {
  return x > y ? x : y;
}
```

Voorbeeld 2.31 in Java geschreven met een conditionele expressie:

```
int monthsize(String thismonth, int thisyear) {
  return thismonth.equals("Feb") ?
         (leap(thisyear) ? 29 : 28) :
         (thismonth.equals("Apr") ||
         thismonth.equals("Jun") ||
         thismonth.equals("Sep") ||
         thismonth.equals("Nov") ? 30 : 31);
}
```

6.5 ITERATIEVE EXPRESSIES

Studeeraanwijzing

Bestudeer in Watt paragraaf 2.6.5.

Voorbeeld 2.32

Het eerste voorbeeld kan als volgt in de module Char uitgetest worden:

```
Char> [if isLower c then toUpper c else c | c <- cs]
        where cs = ['C', 'a', 'r', 'o', 'l']
"CAROL"
Char>
```

Het tweede voorbeeld kan als volgt worden getest:

```
Hugs> [y | y <- ys, y `mod` 100 == 0]
        where ys = [1900, 1946, 2000, 2004]
[1900,2000]
Hugs>
```

6.6 CONSTANTE- EN VARIABELE-ACCESS

Studeeraanwijzing

Bestudeer in Watt paragraaf 2.6.6.

Voorbeeld in Java

In Java luiden de constante- en variabeledeclaraties als volgt:

```
final float pi = 3.1416f;
float r;
```

7 Implementatie

Studeeraanwijzing

Paragraaf 2.7 in Watt behoort niet tot de leerstof.

ZELFTOETS

1 Omschrijf in eigen woorden de volgende begrippen:
   a waarde
   b type
   c recursief type
   d functieruimte $S \to T$
   e dynamische typering
   f typevolledigheidsprincipe
   g conditionele expressie.

2 a Welke primitieve en samengestelde typen kent de taal Java?
   b Welke primitieve en samengestelde typen kent de taal Haskell?
   c Geef aan hoe de structureringsconcepten (cartesisch product, disjuncte vereniging, functieruimte en recursief type) worden gerealiseerd in Java voor het construeren van een samengestelde type.
   d Geef aan hoe deze structureringsconcepten worden gerealiseerd in Haskell.

3 Als twee typen S en T respectievelijk cardinaliteit $\#S = s$ en $\#T = t$ hebben, wat is dan de cardinaliteit van de samengestelde typen $S \times T$, S^n, $S + T$ en $S \to T$?

4 a Wat zijn de relatieve voordelen van statische typering?
   b Wat zijn de relatieve voordelen van dynamische typering?

5 Geef een aantal punten aan waarin de taal Java niet voldoet aan het typevolledigheidprincipe.

6 a Welke vormen van constructies kennen Java en Haskell?
   b Geef een overzicht van mogelijkheden in het gebruik van functies in Haskell die niet in Java bestaan.

TERUGKOPPELING

1 Uitwerking van de opgaven

1.1 Haskell kent de volgende *primitieve waarden*:
– karakters
– gehele getallen
– drijvende-puntgetallen.
Haskell kent de volgende *samengestelde waarden*:
– lijsten (waaronder strings), zoals [12, 0], [['a'], ['2', '<']], [sin, cos, tan] en "ABC"
– tupels, zoals ("aap", True, 2), ( ) en (18, Aug, 1992)
– dataconstructies, waaronder waarheidswaarden True en False (True en False zijn constructorfuncties met 0 parameters), als ook breuken, zoals 5 % 10 en 2 % 3 (% is de breuk-operator).
De enige andere waarden die Haskell kent, zijn *functieabstracties* of kortweg functies.

1.2 De cardinaliteit van Date is:

#(Date) = #(Month) x #(byte)
= 12 x 256
= 3072

1.3 a Een functie voor isEven in Haskell is gebaseerd op het eerste algoritme:

```
isEven'  ::  Integer -> Bool
isEven' n  =  (n `mod` 2) == 0
```

Deze definitie is ook in de prelude gebruikt, alleen met `rem` in plaats van `mod`; `mod` hoort bij `div` en `rem` hoort bij `quot.`. Zij geven een ander resultaat bij een negatieve deler, wat voor deze definitie geen verschil maakt. De definitie van `rem` is gelijk aan de definitie van de operator % in Java.

b Omdat functionele talen geen lussen kennen, dienen we recursie te gebruiken. Merk op dat we twee basisdefinities moeten geven!

```
isEven''  ::  Integer -> Bool
isEven'' 0  =  True
isEven'' 1  =  False
isEven'' n  |  n > 1  =  isEven'' (n-2)
            |  n < 0  =  isEven'' (-n)
```

1.4 a Een voorbeeld van $T \to V$ is $\{a \to z, b \to z\}$.
Merk op dat de twee elementen van T in het voorbeeld worden afgebeeld op hetzelfde element van V.
Een ander voorbeeld is $\{a \to z, b \to x\}$
b De cardinaliteit van $T \to V$ is 3^2 ofwel 9.

1.5 De cardinaliteit van $T + V$ is 5.

1.6 a In Haskell kan het datatype Figure geconstrueerd worden door gebruik te maken van een datadeclaratie:

```
data Figure  =  Point     Float Float
             |  Circle    Float Float Float
             |  Rectangle Float Float Float Float
```

b De functie area is met behulp van patronen als volgt te definiëren.

```
area  ::  Figure -> Float
area (Point  x y)  =  0.0
area (Circle x y radius)  =  3.14159 * radius * radius
area (Rectangle x y height width)  =  height * width
```

1.7 a We definiëren eerst de drie waarden uit het tekstboek:

```
intList1  =  Nil
intList2  =  Cons 13 Nil
intList3  =  Cons 2 (Cons 3 (Cons 5 (Cons 7 Nil)))
```

Een functie waarmee getest kan worden of een lijst van integers al of niet leeg is, is als volgt te definiëren:

```
empty  ::  IntList -> Bool
empty Nil         =  True
empty (Cons _ _)  =  False
```

Toegepast op de geconstrueerde voorbeelden krijgen we de volgende interactie:

```
Le01> empty intList1
True
Le01> empty intList2
False
Le01>
```

b Een functie die van een (niet-lege) lijst het laatste element dient op te leveren, kan als volgt worden gedefinieerd:

```
lastElem  ::  IntList -> Int
lastElem (Cons elem Nil )  =  elem
lastElem (Cons _    rest)  =  lastElem rest
```

Toegepast op de voorbeelden krijgen we de volgende interactie:

```
Le01> lastElem intList2
13
Le01> lastElem intList3
7
Le01>
```

1.8 U kunt natuurlijk altijd in de meegeleverde documentatie opzoeken of de gebruikte programmeertaal statisch dan wel dynamisch getypeerd is. Indien u hierover niets vindt of als u geen zin hebt om dit op te zoeken, kunt u eenvoudig aan het vertaalprogramma een verkeerd getypeerd programmaatje aanbieden. Indien de compiler dit accepteert en pas bij de uitvoering van het programma een foutmelding geeft, is de taal dynamisch getypeerd; bij ongetypeerde talen kunnen zelfs hier geen fouten gesignaleerd worden! Bij statisch getypeerde talen wordt al tijdens de vertaalfase de typeringsfout geconstateerd.
Bij Haskell kunnen we bijvoorbeeld het bestandje met de volgende inhoud proberen te laden.

```
module Mixed where
mixedList = ['a', 'b', 'c'] ++ [True, False]
```

Haskell geeft dan de volgende foutmelding:

```
Hugs> :load "D:\\Mixed.hs"
ERROR file:.\Mixed.hs:2 - Type error in application
*** Expression     : ['a','b','c'] ++ [True,False]
*** Term           : ['a','b','c']
*** Type           : [Char]
*** Does not match : [Bool]

Hugs>
```

In Haskell moeten lijsten homogeen zijn, dat wil zeggen dat ze elementen moeten bevatten van hetzelfde type. Indien we een lijst met karakters aan een lijst met booleans willen koppelen (concateneren), dan leidt dit in de vertaalfase al tot een typeringsfout.

1.9 U kunt bijvoorbeeld een programmaatje maken waarin een functie voorkomt die expliciet een bepaald type als parameter heeft. Als we deze functie een argument geven van een ander type – een op een andere plaats gedefinieerd type, echter met dezelfde waardeverzameling als het verwachte type – dan zal een taal die gebruikmaakt van naamequivalentie, deze constructie niet accepteren, en een taal waarin structurele equivalentie wordt gebruikt, wel.
Bekijk het volgende voorbeeldprogrammaatje in Haskell.

```
type Streng  =  [Char]

string1  ::  String
string1  =  "abc"

string2  ::  Streng
string2  =  "defg"

string3  =  string1 ++ string2

stringLength  ::  String -> Int
stringLength ""  =  0
stringLength (x:xs)  =  1 + stringLength xs
```

Dit bestand wordt zonder bezwaar geaccepteerd. Verder accepteert de functie stringLength niet alleen string1 van het type String, maar ook string2 van het type Streng, getuige de volgende interactie:

```
Le01> :type stringLength
stringLength :: String -> Int
Le01> :type string1
string1 :: String
Le01> stringLength string1
3
Le01> :type string2
string2 :: Streng
Le01> stringLength string2
4
Le01> :type string3
string3 :: [Char]
Le01> stringLength string3
7
Le01>
```

We kunnen in dit geval dus concluderen dat Haskell niet gebruikmaakt van naamequivalentie, maar van structurele equivalentie.

1.10 Haskell voldoet volledig aan het typevolledigheidsprincipe. Dus ook op alle overige plaatsen kunnen we 'ja' invullen.

type	*constante*	*functie-argument*	*functie-resultaat*	*tupel-component*	*lijstcomponent*
primitief	ja	ja	ja	ja	ja
samengesteld	*ja*	ja	ja	ja	*ja*
functie	ja	*ja*	*ja*	*ja*	ja

We geven voor elk van de antwoorden een voorbeeld:
– Haskell kent drie samengestelde typen: lijsten, tupels en datatypen. In alle gevallen is een constantedefinitie mogelijk:

```
author  =  "David Watt"
title   =  "Programming Language Design Concepts"
book    =  (auteur, titel, 2004)
boom    =  Branch (Leaf 11) (Leaf 5)
```

– Een functie kan in Haskell *argument* en *resultaat* zijn van een functie. De functie (**.**) waarmee we functies kunnen samenstellen, heeft bijvoorbeeld altijd functies als argumenten en een functie als resultaat, zoals in het volgende voorbeeld van de functie oneven:

```
oneven  =  (.) not even
```

– Een tupel-component kan in Haskell zonder problemen een functie bevatten. Hier is een voorbeeld van een tupel van een string en een functie:

```
("aap", even)
```

– Lijst-componenten kunnen lijsten zijn, maar ook tupels of data-constructies. De componenttypen van de tupels moeten gelijk zijn; elk element van een lijst moet immers hetzelfde type hebben. Hier is een voorbeeld van een lijst met twee 2-tupels en een lijst met drie binaire bomen.

```
[("aap", even), ("noot", oneven)]
[Branch (Leaf 11) (Leaf 5), Leaf 11,
  Branch (Leaf 12) (Leaf 18)]
```

2 Uitwerking van de zelftoets

1 De gevraagde begrippen kunnen we als volgt omschrijven:
a Een waarde is elke entiteit die tijdens het verwerken van een programma kan worden uitgerekend of opgeslagen, opgenomen kan worden in een gegevensstructuur, meegegeven kan worden als parameter, resultaat is van een functieaanroep, enzovoort.
Anders gezegd: een waarde kunnen we beschouwen als een entiteit die kan bestaan tijdens een berekening.
b Een type is een verzameling waarden die door gemeenschappelijke eigenschappen van die waarden is gekarakteriseerd. Onder gemeenschappelijke eigenschappen vallen de operaties die op de waarden kunnen worden toegepast.
c Een recursief type is een type dat gedefinieerd is in termen van zichzelf.
d Een functieruimte $S \rightarrow T$ is de verzameling van alle afbeeldingen f: $S \rightarrow T$ waarvoor geldt dat, als x tot S behoort, $f(x)$ tot T behoort.
e Dynamische typering is een vorm van typecontrole die plaatsvindt tijdens de uitvoering van het programma.
f Het typevolledigheidsprincipe is een principe dat stelt, dat geen enkele operatie zonder speciale redenen beperkt zou moeten worden in de typen van de betrokken waarden. Anders gezegd: wat met waarden van één type kan, moet – voor zover zinvol – ook met waarden van een ander type kunnen.
g Een conditionele expressie is een expressie die verschillende deel-expressies bevat, waarvan er, afhankelijk van de uitkomst van een of meer voorwaarden, slechts één berekend wordt en waarvan het resultaat tevens de waarde van de gehele expressie vormt.

2 a Java kent de volgende typen:
– *primitieve typen*: gehele getallen (byte, short, int, long), drijvende-puntgetallen (float, double), waarheidswaarden (boolean), karakters (char)
– *samengestelde typen*: klassen, interfaces en arrays.
Type String is een klassetype. Methoden zijn in Java tweede-klas-waarden.
b Haskell kent de volgende typen:
– *primitieve typen*: gehele getallen (Int, Integer), drijvende-puntgetallen (Float, Double) en karakters (Char)
– *samengestelde typen*: tupels, lijsten, datatypen en functies.
Type String is een lijst van karakters. De typen Bool en Rational zijn datatypen.
Haskell kent alleen eersteklaswaarden.

c In Java worden de structureringsconcepten als volgt gerealiseerd:

cartesisch product	klassetype
disjuncte vereniging	interfacetype met voor iedere component van de vereniging een klassetype die de interface implementeert (zoals voorbeeld 2.12 in Java)
functieruimte	array
recursief type	geconstrueerd met behulp van referenties naar objecten (zoals voorbeeld 2.17 in Java)

d In Haskell worden de structureringsconcepten als volgt gerealiseerd:

cartesisch product	tupeltype
disjuncte vereniging	datatype met voor iedere component van de vereniging een constructor-functie (zoals voorbeeld 2.12 in Haskell)
functieruimte	functie
recursief type	lijst, datatype (zoals voorbeeld 2.19 in Haskell)

3 Als de typen S en T respectievelijk cardinaliteit s en t hebben dan hebben de gevraagde samenstellingen van deze typen respectievelijk de volgende cardinaliteiten:
$\#(S \times T) = s \cdot t$
$\#(S^n) = s^n$
$\#(S + T) = s + t$
$\#(S \rightarrow T) = t^s$

4 a Voordelen van statische typering – ten opzichte van dynamische typering – zijn: veiligheid (omdat typeringsfouten al door de vertaler worden ontdekt) en in mindere mate efficiëntie (tijdens de uitvoering van het programma hoeft geen typecontrole meer plaatst te vinden).
b Voordeel van dynamische typering – ten opzichte van statische typering – is vooral de flexibiliteit. Ten opzichte van ongetypeerde talen zijn dynamisch getypeerde talen natuurlijk weer veiliger!

5 Op de volgende punten voldoet Java niet aan het typevolledigheids-principe:
– In Java kunnen constanten alleen van een primitief type of van het type String zijn.
– Een methode of samengesteld type kan in Java niet worden mee-gegeven als argument aan een methodeaanroep; wel een referentie naar een samengesteld type.
– Het resultaat van een methode kan in Java geen methode of samen-gesteld type zijn; wel een referentie naar een samengesteld type.
– In Java kunnen methoden en variabele-referenties wel componenten zijn van een object, maar niet van een array.
Zie ook tabel 1.1.

6 a Java kent het gebruik van constructies alleen in de vorm van array-initializers en constructoraanroepen. Haskell kent zowel de tupel-constructie als de lijst-constructie (zie voorbeeld 2.28). Haskell kent ook constructorfuncties.
b Mogelijkheden met functies die in Haskell wel en in Java niet bestaan, zijn:
– Een functie in Haskell kan niet alleen argument zijn van een functie (denk aan map), maar ook resultaat zijn van een functie (denk aan odd = not . even).
– Vanwege de equivalentie in Haskell tussen operatoren en functies kan een functie ook argument én resultaat zijn van een operator; dit kan in Java allebei niet. Denk aan mogelijkheden in Haskell als: opvolger = (+1).