

Typen und Objektorientierung in Factor

(BlockTag2, SEKS, WS2009, 2009-12-08)

Einleitung

Factor ist eine typisierte Programmiersprache. Das heißt, jeder Wert, mit dem Worte in Factor arbeiten, hat *mindestens* einen Typen. 3 ist vom Typ Zahl, aber auch vom Typ Ganzzahl. "Hello" ist vom Typ String, aber auch vom Typ Sequenz.

Ähnlich wie Mengen, erlauben es Typen, Werte in eindeutig identifizierbare, mit Namen versehene Gruppen einzuteilen. Damit ist die Grundlage dafür geschaffen, in Programmen nicht nur mit konkreten Werten (2, „Hello“,t), sondern abstrakter mit Gruppen von Werten (Zahl, String, Boolean) zu arbeiten.

Typen sind generell für mehrere Dinge wichtig:

- Sie bieten ein Ausdrucksmittel zur feineren Spezifikation/Deklaration von Funktionen. Beispielsweise wenn wir sagen, dass die Additionsfunktion zwei Zahlen erwartet und eine Zahl liefert.
- Die Anwendung eines Wortes auf Daten, für die dieses Wort nicht definiert ist – beispielsweise wenn die Additionsfunktion mit einem String aufgerufen wird – kann zu schwerwiegenden Fehlern, wie etwa Speicherüberläufen, führen. Typprüfung verhindert solche Fehler.
- Typfehler sind meist besser nachvollziehbar und können konsequenter behandelt werden als Fehler, die sich durch fehlerhafte Behandlung von Daten ergeben. Beispielsweise würde in einer nicht-typisierten Sprache der Aufruf der Additionsfunktion mit einem String möglicherweise ein Ergebnis liefern, weil der String einfach numerisch interpretiert wird. Das Programm würde mit diesem falschen Wert weiterrechnen und der Fehler könnte erst spät auftreten oder gar unerkannt bleiben.
- Typen kombiniert mit Vererbung und generischen Worten ermöglichen die Strukturierung und Wiederverwendung von Code. Generische Methoden bieten die Möglichkeit eine Funktion auf Typen zu spezialisieren, also z.B. für jeden Zahlentyp (Ganzzahl, Fließkommazahl, komplexe Zahl, ...) eine unterschiedliche Implementierung von + anzugeben. So etwas ist nur möglich, wenn das Konzept des Typs in der Programmiersprache existent ist
- Struktur-Typen ermöglichen die Strukturierung von Daten. Beispielsweise führen Listen und Tupel in Factor zusammengesetzte Datentypen ein.

In Factor gibt es spezielle Worte, die es erlauben Werte als Instanzen eines Typen zu deklarieren. Diese reine Art der Gruppierung oder Klassifizierung wird häufig mit der Konstruktion neuer Werte bzw. Datenstrukturen kombiniert. Beispielsweise definiert eine Klasse eine neue Datenstruktur und klassifiziert diese gleichzeitig über den Klassennamen.

Typen werden in Factor zur Laufzeit überprüft. Das wird auch dynamische Typprüfung genannt. In Sprachen wie etwa Java wird ein großer Teil der Typprüfung statisch, also zur Compilezeit, durchgeführt. Java kann vor der Ausführung eines Programms ermitteln, ob an einer bestimmten Stelle im Programm ein Typfehler auftritt. Factor weiß dies erst zur Laufzeit. Der Preis, den Java-Programmierer für statische Typprüfung zahlen, ist, dass Typdeklarationen vom Programmierer angegeben werden müssen und dynamische Eigenschaften wie die Veränderung von Klassen zur Laufzeit und Metaprogrammierung stark eingeschränkt sind. Mit anderen Worten: Vieles was zur Compilezeit überprüft wird, muss auch zur Compilezeit bereits in Stein gemeißelt sein.

Der Erstkontakt mit Factor beinhaltet meist den Umgang mit den eingebauten Typen für Zahlen, Booleans, Strings oder Listen. Basierend auf diesen eingebauten Typen erlauben es die im Folgenden vorgestellten Wörter neue Typen zu definieren. Dies geschieht durch Einschränkung, Vereinigung, Schnittmengenbildung, Strukturierung oder Unterklassenbildung.

Prädikatstypen (Predicates)

Ein Prädikatstyp wird durch Einschränkung eines existierenden Typen definiert. Dies entspricht dem intuitiven, natürlich-sprachlichen Vorgehen, wenn wir etwas sagen wie „Alle Studenten, die älter als 25 sind“. Eine Gruppe von Studenten über 25 wird basierend auf der Gruppe der Studenten geschaffen, eingeschränkt durch das Prädikat „älter 25“. Die Definition eines Prädikatstypen besteht aus dem einzuschränkenden Typen und einem Prädikat, das definiert, welche Werte dieses Typen Teil des Prädikatstypen sind. Ein Prädikat ist ein Programm, dessen Resultat als t oder f interpretiert wird. Wenn T_p ein Prädikatstyp ist, T der Basistyp für T_p und P das Prädikat zur Definition von T_p , dann ist ein Wert W genau dann vom Typ T_p , wenn W vom Typ T ist und P angewendet auf W wahr ist. Im folgenden Beispiel wird der Typ aller Integer, die größer als 10 sind, definiert.

```
PREDICATE: greater10 < integer 10 > ;
```

Der Basistyp ist integer, das Prädikat `10 >`. Lassen Sie sich hier nicht von dem ersten `<` verwirren, es gehört zur Syntax des Parse-Wortes „PREDICATE:“.

Die Überprüfung

```
11 greater10 instance?
```

legt `t` auf den Stack, während

```
10 greater10
```

`f` auf den Stack legt. Alternativ kann für diese Überprüfung auch das aus der Typdefinition generierte Wort `greater10?` verwendet werden:

```
11 greater10?
```

Vereinigungstypen (Unions)

Ein Union-Typ wird durch die Vereinigung existierender Typen definiert. Er entspricht natürlich einer Gruppierung wie „Das Hochschulpersonal sind die Professoren und Mitarbeiter“. Die Funktionsweise entspricht dem Konzept der Mengenvereinigung in der Mengentheorie und dem logischen Oder, wenn es sich auf die Frage der Mengenzugehörigkeit bezieht. Wenn Typ T_{12} ein UNION-Typ aus T_{p1} und T_{p2} ist, dann ist ein Wert W Teil von T_{12} , wenn er vom Typ T_{p1} oder vom Typ T_{p2} ist.

Folgendes Beispiel führt einen Typen `stringint` als UNION von `string` und `integer` ein:

```
UNION: stringint string integer ;
```

Mit `stringint?` kann nun getestet werden, ob ein Wert vom Typ `stringint` ist:

2 stringint?

und

"Hello" stringint?

ergeben jeweils t, während

2.3 stringint?

f ergibt.

Mixin-Typen

Ein Mixin ist prinzipiell dasselbe wie ein Union. Der Unterschied besteht darin, dass nach der initialen Typdefinition noch Elemente zur Vereinigung hinzugefügt werden können. Der Typ ist also veränderbar.

Die Definition

```
MIXIN: mytype
INSTANCE: string mytype
INSTANCE: integer mytype
```

entspricht also

```
UNION: mytype string integer ;
```

Die Erweiterbarkeit des MIXIN erlaubt jedoch folgendes:

Das Programm

```
t mytype?
```

ergibt vor Eingabe von

```
INSTANCE: boolean mytype
```

f, danach aber t.

Schnittmengen-Typen (Intersections)

Ein Schnittmengentyp wird als Schnittmenge mehrerer existierender Typen definiert. Natürlichsprachlich entspricht es einer Aussage wie „Alle Studenten, die sowohl in Mathematik als auch in Grundlagen der Informatik eine 1 hatten“. Dies entspricht dem Konzept der Schnittmenge in der Mengentheorie und dem logischen Und wenn es sich auf die Frage der Mengenzugehörigkeit bezieht. Wenn Typ T12 ein INTERSECTION-Typ aus T1 und T2 ist, dann ist ein Wert W Teil von T12, wenn er vom Typ1 *und* vom Typ2 ist.

Folgendes Programm definiert den Typen aller Integer, die größer als 10 und kleiner als 100 sind.

```
PREDICATE: greater10 < integer 10 > ;
PREDICATE: smaller100 < integer 100 < ;
INTERSECTION: greater10smaller100 greater10 smaller100 ;
```

Das Programm

```
50 greater10smaller100 instance?
```

legt `t` auf den Stack, während

```
123 greater10smaller100 instance?
```

`f` auf den Stack legt.

Singletons

Ein Singleton ist ein Typ, von dem es nur eine Instanz gibt. Das Klassenwort selbst ist diese eine Instanz.

Das Programm

```
SINGLETON: token
```

definiert einen Singleton mit einem einzigen Wert `token`.

Das Programm

```
token token instance?
```

legt `t` auf den Stack. Es zeigt, dass `token` eine Instanz - die einzige - von `token` ist.

Generische Wörter

Generische Wörter bilden zusammen mit den weiter unten noch zu besprechenden Tupeltypen den Kern von dem ab, was in Sprachen wie Java als Objektorientierung verstanden wird. Generische Wörter erlauben unter anderem das, was in Java als dynamische Methodenbindung bekannt ist.

Generische Wörter sind Wörter, die auf Objekten mit verschiedenen Typen arbeiten können. Beispiele für generische Wörter in Factor sind etwa die arithmetischen Operationen: Das Wort `+` ist nicht nur für Ganzzahlen, sondern auch für Fließkommazahlen und andere Zahlentypen definiert. Intern existiert für jeden Zahlentypen eine eigene Implementierung der Addition. Basierend auf den Argumenttypen entscheidet Factor, welche Implementierung verwendet wird. Ohne einen solchen Mechanismus wäre es nötig, statt einem einzigen `+` separate Wörter für die einzelnen Zahlentypen zu verwenden wie z.B. `+int` für die Addition von Integern, `+float` für die Addition von Fließkommazahlen usw..

Im folgenden Beispiel wird ein generisches Wort `add` definiert, welches sowohl auf Sequenzen als auch auf Zahlen arbeitet.

```
GENERIC: add ( x y -- z )
M: sequence add append ;
M: number add + ;
```

Die Implementierung ist `+`, falls es sich um eine Zahl handelt und `append`, falls es sich um eine Sequenz handelt. Die folgenden beiden Programme haben also die gleiche Bedeutung:

```
3 4 add
3 4 +
```

Da Strings Sequenzen sind, sind folgende beiden Programme gleichbedeutend.

```
"hello" "world" add
"hello" "world" append
```

Standardmäßig erfolgt die Auswahl der Implementierung basierend auf Argumenttypen (*dispatch on type*) anhand des obersten Arguments auf dem Stack. Im Beispiel

```
3 4 add
```

erkennt der Factor-Interpreter also, dass es sich bei der 4 um eine `number` handelt und wählt daher die passende Implementierung `+`.

Das Wort `GENERIC:` definiert zunächst ein generisches Wort ohne Implementierung. Die Anwendung des so definierten Wortes gibt für alle Argumente einen Fehler. Die Implementierungen für verschiedene Typen werden durch das Methodendefinitionswort `M:` hinzugefügt. Mit dem Wort `GENERIC` kann definiert werden, dass die Auswahl der Implementierung anhand einer anderen Stelle auf dem Stack als der obersten erfolgen soll.

Im folgenden Beispiel wird basierend auf den Singletons `token1` und `token2` durch Vereinigung ein Typ `token` definiert. Das generische Wort `what-are-you?` liefert auf einen `token` angewendet die Antwort `"I am a token"`.

```
SINGLETON: token1
SINGLETON: token2
UNION: token token1 token2 ;
GENERIC: what-are-you? ( t -- a )
M: token what-are-you? drop "I am a token" ;
```

Eine Erweiterung auf Zahlen sieht so aus:

```
M: number what-are-you? drop "I am a number" ;
```

Tupeltypen (Tuples)

Tupeltypen bilden zusammen mit den oben beschriebenen generischen Wörtern den Kern von dem ab, was in Sprachen wie Java als Objektorientierung verstanden wird. Im Sinne solcher Sprachen entspricht ein Tupel in etwa einer Klassendefinitionen, die ausschließlich Membervariablen enthält.

Ein Tupeltyp ist eine Sequenz mit benannten Feldern (*dictionary*, Schlüssel/Wert-Tabelle). Ein Tupel wird

durch Nennung der Schlüssel definiert. Optional können Standardwerte, Typen und Einschränkungen wie etwa Nur-Lese-Zugriff für jedes Feld angegeben werden. Factor generiert basierend auf der Definition von Tupeln automatische Zugriffswörter (getter und setter).

Das Programm

```
TUPLE: person name age ;
```

definiert ein Tupel mit dem Namen `person` und zwei Feldern `name` und `age`. Durch Anwenden des Konstruktors `new` wird eine Tupelinstanz erzeugt und auf dem Stack abgelegt:

```
person new
```

Die Werte für `name` und `age` werden standardmäßig auf `f` gesetzt.

Mit den erzeugten Lesewörtern `name>>` und `age>>` werden Werte gelesen. Das Programm

```
person new >>name
```

ergibt `f`, den Standardwert für Felder.

Mit den ebenfalls erzeugten Schreibwörtern `>>name` und `>>age` können diese Werte gesetzt werden, das modifizierte Tuple bleibt dabei auf dem Stack:

```
"Tom Green" >>name  
26 >>age
```

Neben den Schreibwörtern, die der Namenskonvention `>>field` folgen, werden auch Schreibwörter der Form `(>>field)` generiert. Diese haben nicht den Stackeffekt „(obj -- obj)“, sondern „(obj --)“. Das heißt, sie modifizieren ein Tuple zwar, für die Erhaltung des Tupels auf dem Stack muss vom Aufrufer selbst gesorgt werden. Die folgenden beiden Zeilen sind gleichbedeutend:

```
person new 23 over (>>age)  
person new 23 >>age
```

Für die Anwendung von Programmen auf Werte eines Felds in einem Tupel werden Wörter nach der Namenskonvention `change-field` generiert. In folgendem Beispiel wird der Name der Person umgedreht:

```
person new "Tom" >>name [ reverse ] change-name
```

Neben dem Konstruktor `new` gibt es weitere Möglichkeiten Tupel-Instanzen zu erzeugen. Der Konstruktor `boa` (*by order of arguments*) schreibt Werte vom Stack direkt an die korrespondierenden Stellen im Tupel. Beispiel:

```
"Tom Green" 26 person boa
```

Des Weiteren lassen sich in Kombination mit `new` oder `boa` spezielle Konstruktoren definieren, die per Konvention mit spitzen Klammern beginnen und enden. Das Programm

```
: <person> person new 0 >>age "John Doe" >>name ;
```

definiert einen Konstruktor, der eine Person mit Standardwerten initialisiert. Eine Kurznotation für Konstruktordefinitionen bietet das Wort `C:`. Folgende zwei Zeilen haben dieselbe Bedeutung:

```
C: <person> person
: <person> person boa ;
```

Eine erweiterte Form der Tupel-Definition bietet die Möglichkeit Typen und Standardwerte direkt anzugeben:

```
TUPLE: person { age int initial: 0 } { person string initial: "John Doe" } ;
```

Das Programm

```
person new
>>age "0"
```

verursacht einen Typfehler.

Tuple bieten die Möglichkeit Vererbungshierarchien zu definieren. Ein Tupel `T`, welches von einem anderen Tuple `TP` erbt (Unterklasse von `TP` ist), besitzt alle Felder von `TP` und ist gleichzeitig vom Typ `T` und vom Typ `TP`. In folgendem Beispiel wird `student` als Unterklasse von `person` definiert.

```
TUPLE: student < person id semester ;
```

Instanzen von `student` haben die Felder `name`, `age`, `id`, `semester`. Wegen der Tatsache, dass `student`-Instanzen auch vom Typ `person` sind, sind automatisch auch alle Methoden von `person` für `student` definiert. Folgendes erweiterter Beispiel zeigt dies:

```
TUPLE: person fname lname age ;
TUPLE: student < person id semester ;
GENERIC: first-and-last-name ( obj -- string )
M: person first-and-last-name dup fname>> " " append swap lname>> append ;
"Tom" "Green" 26 1234 6 student boa
first-and-last-name
```

Das Wort `first-and-last-name`, das für Tuple vom Typ `person` definiert ist, funktioniert auch für Tuple, die Instanzen von `student` sind.

Objektorientierung in Factor und Java

Eine Klassendefinitionen in Java hat sowohl strukturelle als auch operationale Anteile. Die Struktur der Objekte einer Klasse wird durch Membervariablen definiert. Vom Objekt unterstützte Operationen werden durch Methoden innerhalb der Klassendefinition angegeben.

Factor verfolgt hier einen anderen Ansatz: Es trennt Struktur- und Operationsdefinition. Die strukturellen Anteile einer Klasse werden durch Tuple ausgedrückt, die operationalen Anteile durch generische Wörter, die Implementierungen für das Tuple haben.

Mit Tupeln, Tuplevererbung und generischen Wörtern kann in Factor prinzipiell Code geschrieben werden, der dem Java-Verständnis von Objektorientierung entspricht. Allerdings bietet Factor mit den



Typkonstruktoren `UNION:`, `INTERSECTION:`, `SINGLETON:`, `PREDICATE:` und `MIXIN:` Möglichkeiten zur Typdefinition, die in Java keine Entsprechung haben.

Auch unterliegt generischen Wörtern ein anderes Verständnis, als es dem von Methoden in Java. Während dynamische Methodenbindung ebenfalls durch die Auswahl einer Implementierung basierend auf Typen (*type dispatch*) funktioniert, ist dies in Java nur innerhalb einer Vererbungslinie möglich – eine Ausnahme sind Objekte, deren Klassen dasselbe Interface implementieren. Ein generisches Wort hingegen kann für beliebige Menge von Typen definiert sein.