Objektorientierte Programmierung mit MOSTflexiPL

Christian Heinlein

Hochschule Aalen – Technik und Wirtschaft
christian.heinlein@hs-aalen.de

Abstract. MOSTflexiPL (Modular, Statically Typed, Flexibly Extensible Programming Language, vgl. http://flexipl.info) ist eine statisch typisierte Programmiersprache, deren Syntax vom Anwender nahezu beliebig erweitert und angepasst werden kann. Aufbauend auf einer kleinen Menge vordefinierter Grundoperatoren, können nach Belieben weitere Operatoren für unterschiedlichste Zwecke definiert werden. Da Operatoren beliebig viele Namen besitzen und auf beliebig viele Operanden angewandt werden können, decken sie neben den üblichen Präfix-, Infix- und Postfix-Operatoren auch Mixfix-Operatoren, Kontrollstrukturen, Typkonstruktoren und Deklarationsformen ab. Die Menge der vordefinierten Grundkonstrukte ist zwar klein, aber sehr ausdrucksstark. Um dies zu belegen, wird in diesem Beitrag gezeigt, wie man eine vollwertige objektorientierte Programmiersprache mit Vererbung, Untertyp-Polymorphie und dynamischem Binden in Form von MOSTflexiPL-Syntaxerweiterungen definieren kann. Tatsächlich geht das Spektrum der Möglichkeiten sogar weit über gängige Sprachen hinaus: Neben einfacher Vererbung und dem üblichen „single dispatch“, lassen sich auch mehrfache Vererbung in unterschiedlichen „Spielarten“ sowie „multiple dispatch“ und „predicate dispatch“ realisieren. Außerdem sind Typen ohne besondere Anstrengung „offen“, d. h. sie können problemlos nachträglich und modular um weitere Obertypen, Attribute und Operationen erweitert werden.

1 Einleitung

MOSTflexiPL (Modular, Statically Typed, Flexibly Extensible Programming Language) ist eine statisch typisierte Programmiersprache, die vom Anwender nahezu beliebig erweitert und angepasst werden kann [He12, He14]. Basierend auf einer kleinen Menge vordefinierter Grundoperationen (z. B. für Arithmetik, Logik und elementare Kontrollstrukturen), können in der Sprache selbst nach Belieben neue Operatoren, Operatorkombinationen, Kontrollstrukturen, Typkonstruktoren und Deklarationsformen definiert werden. Die Grundidee besteht darin, jedes dieser syntaktischen Konstrukte als *Operator* aufzufassen, der beliebig viele *Namen* (Operatorsymbole) und *Operanden* in beliebiger Reihenfolge besitzen kann. Beispielsweise besitzt der Additionsoperator `•+•` zwei Operanden (symbolisiert durch `•`) und einen Namen `+`, der bei einer Anwendung des Operators zwischen den Operanden (infix) steht, z. B. `2 + 3`. Der aus der Mathematik bekannte Betragsoperator `|•|` hingegen besitzt einen Operanden und zwei (zufällig gleiche) Namen, die um den Operanden herum (zirkumfix) stehen, z. B. `|-5|`. Eine Fallunterscheidung kann z. B. als Operator `if•then•else•`

`end` mit drei Operanden und vier Namen oder auch als Operator `•?•:•` mit drei Operanden und zwei Namen (Fragezeichen und Doppelpunkt) definiert werden. Der Typkonstruktor `•[•]` zur Definition von Arraytypen besteht aus zwei Namen (öffnende und schließende eckige Klammer) und zwei Operanden, von denen der erste ein Typ und der zweite eine ganze Zahl sein muss, z. B. `int [10]`. Aber auch Prozeduren und Funktionen lassen sich als Operatoren definieren, z. B. `max (•,•)` (mit klassischer imperativer Syntax, bei der die Klammern und das Komma einfach weitere Namen des Operators sind) oder `print•` (mit moderner funktionaler Syntax).

Tatsächlich erfolgt die Definition von Operatoren sehr ähnlich wie die Definition von Funktionen in anderen Sprachen. Und da mit jedem neuen Operator „nebenbei" auch ein neues syntaktisches Konstrukt definiert wird, werden Syntaxerweiterungen auf die gleiche Art und Weise erstellt wie gewöhnliche Programme, d. h. es gibt hierfür keinen separaten Spezialmechanismus mit eigenen Ausdrucksmitteln und Regeln.

Da jedes syntaktische Konstrukt durch einen Operator repräsentiert wird, stellt jede Verwendung eines Konstrukts eine Operatoranwendung, d. h. einen *Ausdruck* dar. Hierbei werden auch Konstanten und Variablen sowie Literale wie z. B. `0` oder `"abc"` als nullstellige Operatoren aufgefasst, deren Anwendung einfach den jeweiligen Wert liefert. Beispielsweise ist der Ausdruck `if x >= 0 then x else -x end` eine Anwendung des Operators `if•then•else•end` auf die Operanden `x >= 0` sowie `x` und `-x`. Der Teilausdruck `x >= 0` ist wiederum eine Anwendung des Operators `•>=•` auf die Operanden `x` und `0`, bei denen es sich um Anwendungen der nullstelligen Operatoren `x` (eine Konstante oder Variable) und `0` (ein Literal) handelt. Ebenso ist `-x` eine Anwendung des Operators `-•` auf den Operanden `x`.

Die Anwendungsmöglichkeiten von MOSTflexiPL sind vielfältig. In erster Linie ist es als Allzwecksprache (general purpose language) gedacht, mit der man je nach persönlicher Präferenz sowohl imperativ als auch funktional programmieren kann. Der entscheidende Unterschied und Vorteil gegenüber anderen Sprachen besteht darin, dass man bei Bedarf jederzeit syntaktische Erweiterungen vornehmen kann, um bestimmte Dinge einfacher, kürzer, „natürlicher" oder verständlicher ausdrücken zu können. Derartige Erweiterungen können entweder ad hoc für ein einzelnes Programm definiert werden oder aber in wiederverwendbaren Operatorbibliotheken zusammengefasst werden, die auch anderen Benutzern zur Verfügung gestellt werden können. Da sich Spracherweiterungen sehr leicht definieren und auch wieder ändern lassen, kann MOSTflexiPL auch als Experimentierplattform für neue Sprachkonstrukte verwendet werden, beispielsweise für objektorientierte Programmierung wie in diesem Beitrag. Weil sich die Syntax der Sprache nicht nur erweitern, sondern auch beliebig verändern und einschränken lässt, kann MOSTflexiPL schließlich auch als Hilfsmittel zur Definition und Implementierung anwendungsspezifischer Sprachen (DSLs) verwendet werden.

Die vordefinierten Grundkonstrukte und -konzepte der Sprache (die zum Teil aus dem Vorgängerprojekt APPLEs [He07] stammen), insbesondere auch das vorhandene Typsystem mit beschränkter parametrischer Polymorphie sowie benutzerdefinierbaren impliziten Typumwandlungen, sind so allgemein und ausdrucksstark, dass nicht nur „syntaktischer Zucker", sondern auch weitreichende „paradigmatische" Erweiterun-

gen definiert werden können, was im folgenden demonstriert werden soll. Die Tatsache, dass sich Benutzer der Sprache bei Bedarf nahezu alles Gewünschte selbst definieren (oder aus Bibliotheken importieren) können, ist auch der Grund, warum bestimmte, aus anderen Sprachen gewohnte „Bequemlichkeiten“ a priori nicht vorhanden sind.

Im nachfolgenden Abschnitt 2 werden einige typische Beispiele für Syntaxerweiterungen durch neue Operatoren vorgestellt und damit nebenbei wichtige Grundkonstrukte von MOSTflexiPL vorgestellt und erläutert. Abschnitt 3 zeigt dann exemplarisch einige wichtige Syntaxerweiterungen für objektorientierte Programmierung. Abschnitt 4 enthält eine kurze Diskussion der entwickelten Sprachkonstrukte, während Abschnitt 5 mit Zusammenfassung und Ausblick schließt. Weiterführende Informationen zu MOSTflexiPL sowie viele weitere Beispiele findet man auf http://flexipl.info.

2 Beispiele für Syntaxerweiterungen durch Operatoren

2.1 Einfache Operatordeklarationen

Die folgenden Zeilen zeigen eine einfache Operatordeklaration in MOSTflexiPL:

```
["n" : int]          Parameterliste
n "²" : int          Signatur und Resultattyp
{ n * n }            Implementierung
```

Sie besteht aus einer *Parameterliste* in eckigen Klammern, einer *Signatur* vor dem Doppelpunkt, einem *Resultattyp* danach sowie einer *Implementierung* in geschweiften Klammern. Die Signatur besteht ihrerseits aus Parametern und Zeichenketten in Anführungszeichen und definiert die *Syntax* des Operators, d. h. die syntaktische Form seiner Anwendungen: Jeder Parameter ist ein Platzhalter für einen Operanden, d. h. für einen Teilausdruck mit entsprechendem Typ, während eine Folge beliebiger Zeichen in Anführungszeichen einen Namen des Operators darstellt, der bei Anwendungen des Operators genau so (allerdings ohne die Anführungszeichen) hingeschrieben werden muss. Demnach ist z. B. `(2+3)²` eine korrekte Anwendung des gerade definierten Operators, weil `(2+3)` ein Teilausdruck mit Typ `int` und `²` der Name des Operators ist.

Die Parameterliste besteht aus *Parameterdeklarationen* (ggf. durch Strichpunkte getrennt), bei denen es sich ebenfalls um einfache Operatordeklarationen handelt, die (bei den hier betrachteten einfachen Beispielen) lediglich aus einer Signatur und einem Resultattyp bestehen. Daher ist jedes Auftreten des Parameters `n` in Wirklichkeit eine Anwendung des nullstelligen Operators mit Signatur `"n"` und Resultattyp `int`.

Die Implementierung schließlich ist ein beliebiger Ausdruck, dessen Typ mit dem Resultattyp übereinstimmen muss und durch dessen Auswertung das Ergebnis einer Operatoranwendung entsteht. Beispielsweise entsteht der Wert des Ausdrucks `(2+3)²`, indem zunächst der Parameter `n` mit dem Wert des zugehörigen Operanden `(2+3)` (also 5) initialisiert wird und anschließend die Implementierung `n * n` ausgewertet wird, die in diesem Fall den Wert 25 liefert.

Ein weiterer Operator `|•|`, der den absoluten Betrag seines Operanden als Ergebnis liefert, kann wie folgt definiert werden:

```
["x" : int]
"|" x "|" : int
{ if x >= 0 then x else -x end }
```

Hier besteht die Implementierung aus einer Anwendung des vordefinierten Verzweigungsoperators `if•then•else•end`, der, abhängig vom Wahrheitswert seines ersten Operanden, entweder den Wert seines zweiten oder den seines dritten Operanden als Ergebnis liefert. Da die Signatur aus einem senkrechten Strich in Anführungszeichen, dem `int`-Parameter `x` und einem weiteren senkrechten Strich in Anführungszeichen besteht, sind `|0|` und `|2-5|` exemplarische Anwendungen des Operators.

Um mehrere Ausdrücke nacheinander auszuwerten, kann man sie mit dem vordefinierten Operator `•;•` verknüpfen, bei dessen Anwendung – wie bei jeder Operatoranwendung – zunächst seine beiden Operanden (von links nach rechts) ausgewertet werden und der als Ergebnis einfach den Wert seines rechten Operanden liefert.

2.2 Konstanten, Typen und Variablen

Für einen beliebigen Typ `T` definiert eine Deklaration der Gestalt `"x" : T` eine *eindeutige Konstante* des Typs `T`, d. h. einen nullstelligen Operator `x`, der bei jeder Anwendung denselben eindeutigen Wert liefert und daher auch als *statischer Operator* bezeichnet wird. (Operatoren mit Implementierung, wie z. B. `"random" : int { ..... }`, die prinzipiell bei jeder Anwendung einen anderen Wert liefern können, werden zur Unterscheidung als *dynamische Operatoren* bezeichnet.)

Wenn man für `T` den vordefinierten Metatyp `type` verwendet, z. B. `"Person" : type`, erhält man einen neuen Typ `Person`, der verschieden von allen anderen Typen ist. (Dementsprechend sind vordefinierte Typen wie z. B. `int` auch nichts anderes als solche typwertigen Konstanten.) Anschließend kann man eindeutige Werte des Typs wie z. B. `"p" : Person` definieren, die vergleichbar mit Objekten in anderen Sprachen sind.

Wenn eine Deklaration eines statischen Operators Parameter besitzt, z. B. `["T" : type] "List" T : type`, so liefert der dadurch definierte Operator `List•` für jeden Wert seines Parameters `T` einen anderen eindeutigen Wert, sodass z. B. `List int` und `List Person` verschiedene Werte des Typs `type`, d. h. verschiedene Typen sind. Umgekehrt bezeichnet `List int` natürlich jedesmal den gleichen Typ.

Allgemein liefert ein statischer Operator bei Anwendung auf die gleichen Parameterwerte also immer das gleiche Ergebnis (was für einen dynamischen Operator mit Implementierung wiederum nicht garantiert werden kann) und bei Anwendung auf unterschiedliche Werte unterschiedliche Ergebnisse. Daraus folgt, dass zwei *statische Ausdrücke*, d. h. Ausdrücke, die nur statische Operatoren enthalten, den gleichen Wert liefern, wenn sie *strukturgleich* sind, d. h. wenn es sich um Anwendungen desselben (statischen) Operators auf paarweise strukturgleiche Operanden handelt.

Aufgrund dieser für den Compiler wichtigen Eigenschaft, dürfen statische Operatoren als *Typkonstruktoren* verwendet werden, während dynamische Operatoren in Typ-

ausdrücken verboten sind. Tatsächlich sind Typen in MOSTflexiPL einfach als statische Ausdrücke mit Typ `type` definiert.

Der vordefinierte Typkonstruktor `•?` liefert zu jedem Typ `T` den zugehörigen *Variablentyp* `T?`, dessen Werte jeweils eindeutige Variablen mit *Inhaltstyp* `T` sind. Beispielsweise deklariert `"i" : int?` eine Variable mit Inhaltstyp `int`, d. h. eine eindeutige Speicherzelle zur Speicherung von `int`-Werten, deren Inhalt durch eine Zuweisung wie z. B. `i = i + 1` verändert werden kann.

Eine parametrisierte Variablendeklaration wie z. B.

```
["p" : Person]
p "." "name" : string?
```

liefert für jeden Wert des Parameters `p` eine andere eindeutige Variable `p.name`, d. h. sie ordnet jeder Person eine Variable mit Inhaltstyp `string` zu, in der der Name der Person gespeichert werden kann:

```
p.name = "Heinlein";
print p.name
```

Auf diese Weise lassen sich indirekt Datenstrukturen definieren, die bei Bedarf modular um neue „Attribute“ erweitert werden können (sog. *offene Typen* [He07]).

Mit den folgenden Operatoren `•->•` und `•.•` wird die Definition und Verwendung solcher Attribute noch weiter erleichtert:

```
["X" : type; "Y" : type]
X "->" Y : type;

["X" : type; "Y" : type; "x" : X; "a" : X -> Y]
x "." a : Y?
```

Für zwei beliebige Typen `X` und `Y` liefert der statische Operator `•->•` jeweils einen eindeutigen Typ `X -> Y`, der zur Repräsentation von Attributen des Typs `X` mit Zieltyp `Y` verwendet werden kann, z. B.:

```
"Date" : type;
"day" : Date -> int;
"month" : Date -> int;
"year" : Date -> int;

"dob" : Person -> Date
```

Für ein Objekt `x` eines beliebigen Typs `X` und ein Attribut `a` eines zugehörigen Attributtyps `X -> Y` liefert der statische Operator `•.•` jeweils eine eindeutige Variable `x.a` mit Inhaltstyp `Y`, die zur Speicherung des entsprechenden Attributwerts verwendet werden kann, z. B.:

```
"d" : Date;
d.day = 8; d.month = 2; d.year = 1965;
p.dob = d
```

Variablen, denen noch kein Wert zugewiesen wurde, enthalten den Sonderwert nil, der

die Abwesenheit eines echten Werts anzeigt. Dementsprechend liefern Zugriffe auf Attribute, denen noch kein Wert zugewiesen wurde, ebenfalls nil.

2.3 Implizite Typumwandlungen

Wenn ein Operator genau einen Operanden und keinen Namen besitzt, definiert er in natürlicher Weise eine implizite Typumwandlung vom Typ seines Operanden in seinen Resultattyp, z. B.:

```
["d" : Date]
d : string
{ d.day ++ "." ++ d.month ++ "." d.year }
```

Aufgrund seiner besonderen syntaktischen Struktur kann dieser Operator – ohne weitere Eingabesymbole zu verbrauchen – auf jeden Teilausdruck mit Typ `Date` angewandt werden und liefert als Resultat einen Wert des Typs `string` (konkret z. B. `"8.2.1965"`). Da der Compiler grundsätzlich jeden anwendbaren Operator „ausprobiert“ und Ausdrücke, die nicht typkorrekt sind, wieder „aussortiert“, wird er diesen Operator letztlich immer genau dann verwenden, wenn an einer bestimmten Stelle eine Umwandlung von `Date` nach `string` erforderlich ist, z. B.:

```
"s" : string?;
s = p.dob
```

2.4 Virtuelle Operatoren

Wenn man `var T` als Synonym für Variablentypen `T?` (vgl. Abschnitt 2.2) verwenden möchte, kann man versuchen, den Operator `var •` wie folgt zu definieren:

```
["T" : type]
"var" T : type
{ T? }
```

Da Typen vollwertige Werte sind, wird man zur Laufzeit tatsächlich feststellen, dass Vergleiche wie `var int == int?` als Resultat `true` liefern. Trotzdem ist diese Definition von `var •` relativ nutzlos, weil der Compiler dynamische Operatoren mit Implementierung in Typausdrücken nicht akzeptiert (vgl. Abschnitt 2.2) und eine Deklaration der Art `"i" : var int` daher fehlerhaft wäre.

Damit der Operator `var •` wirklich nützlich ist, muss er wie folgt als *virtueller Operator* definiert werden:

```
["T" : type]
"var" T = T?
```

Allgemein besitzt die Deklaration eines virtuellen Operators ebenfalls eine Parameterliste in eckigen Klammern sowie eine anschließende Signatur (im Beispiel `"var" T`). Anstelle von Resultattyp und Implementierung folgt dann jedoch eine sog. *Realisierung* nach einem Gleichheitszeichen (im Beispiel `T?`). Der Resultattyp des Operators ergibt sich implizit aus dem Typ der Realisierung (im Beispiel lautet er `type`).

Die Anwendung eines virtuellen Operators unterscheidet sich syntaktisch nicht von der Anwendung eines anderen Operators. Beispielsweise sind `var int` und `var List int` korrekte Anwendungen des Operators `var •`, während `var 2` fehlerhaft ist, weil der Operand `2` nicht den geforderten Typ `type` besitzt.

Die einzige Besonderheit besteht darin, dass eine Anwendung eines virtuellen Operators, nachdem sie erfolgreich auf Typkorrektheit überprüft wurde, vom Compiler sofort durch die Realisierung des Operators *ersetzt* wird, in der die Parameter des Operators wiederum durch die entsprechenden Operanden ersetzt werden. Demnach wird ein Ausdruck wie `var int` sofort durch die Realisierung `T?` ersetzt, in der der Parameter `T` wiederum durch den Operanden `int` ersetzt wird, d. h. der endgültige Ausdruck lautet `int?`.

Da diese Ersetzung bereits zur Übersetzungszeit stattfindet, wird eine Deklaration der Art `"i" : var int` jetzt vom Compiler akzeptiert, weil der Teilausdruck `var int` sofort durch `int?` ersetzt wird und die Deklaration daher vollkommen gleichbedeutend mit `"i" : int?` ist.

Da `var int` „in Wirklichkeit“ also `int?` bedeutet und nur „scheinbar“ etwas anderes darstellt, wird `var •` als „virtueller“ Operator und `T?` als seine „Realisierung“ bezeichnet.

2.5 Benutzerdefinierte Deklarationsoperatoren

Eine weitere Kategorie von Operatoren, die virtuell definiert werden müssen, damit sie den gewünschten Effekt haben, sind Deklarationsoperatoren, d. h. Operatoren, bei deren Anwendung andere Operatoren deklariert werden.

Wenn man beispielsweise Variablen (ähnlich wie in C, C++ und Java) in der Form `int "i"` anstelle von `"i" : int?` deklarieren möchte, kann man hierfür den folgenden virtuellen Operator `• •` verwenden:

```
["T" : type; "name" : string]
T name =
name : T?
```

Eine Anwendung wie z. B. `int "i"` (die nur aus Operanden besteht, weil der Operator keine Namen besitzt) wird wiederum durch die Realisierung des Operators, d. h. durch den Ausdruck `name : T?` ersetzt, in dem die Parameter `name` und `T` durch die Operanden `"i"` bzw. `int` ersetzt werden, sodass schließlich der Ausdruck `"i" : int?` entsteht.

3 Syntaxerweiterungen für objektorientierte Programmierung

Im folgenden werden exemplarisch einige Spracherweiterungen vorgestellt, die es erlauben, mit MOSTflexiPL objektorientiert zu programmieren. Aus Platzgründen müssen zwar zahlreiche Details weggelassen werden, aber anhand der gezeigten Beispiele kann man trotzdem einen guten Eindruck von den Möglichkeiten der Sprache gewinnen.

3.1 Hilfsoperatoren

Für zwei beliebige Typen `X` und `Y` stellt `<X,Y>` ein eindeutiges Attribut mit Typ `X -> Y` dar (vgl. §2.2), das verwendet werden kann, um für jedes Objekt des Typs `X` einen Verweis auf ein Objekt des Typs `Y` zu speichern:

```
["X" : type; "Y" : type]
"<" X "," Y ">" : X -> Y
```

Für Objekte `x` und `y` mit beliebigen Typen `X` bzw. `Y` stellt `x <-> y` unter Verwendung der Attribute `<X,Y>` und `<Y,X>` eine bidirektionale Verbindung zwischen diesen beiden Objekten her:

```
["X" : type; "Y" : type; "x" : X; "y" : Y]
x "<->" y : bool
{
    x.<X,Y> = y;
    y.<Y,X> = x;
    true
}
```

Der Resultattyp `bool` und der Resultatwert `true` haben keine weitere Bedeutung. Sie werden nur gebraucht, weil ein Operator immer einen Resultattyp besitzen und einen Resultatwert liefern muss.

3.2 Offene Typen

Mit den in §2.2 definierten Operatoren `•->•` und `•.•` können Datentypen sehr bequem – und nachträglich erweiterbar – definiert und verwendet werden, z. B.:

```
"Person" : type;
"name" : Person -> string;

"Date" : type;
"day" : Date -> int;
"month" : Date -> int;
"year" : Date -> int;

"dob" : Person -> Date;

"p" : Person;
p.name = "Heinlein";

"d" : Date;
d.day = 8; d.month = 2; d.year = 1965;
p.dob = d
```

3.3 Untertypen

Wenn in einer objektorientierten Sprache ein Typ (z. B. `Citizen`) als Untertyp eines anderen (z. B. `Person`) definiert wird, hat dies u. a. folgende Auswirkungen:

1. Ein Objekt des Untertyps kann implizit in den Obertyp umgewandelt werden (implizite Aufwärtsumwandlung).
2. Damit kann ein Objekt des Untertyps überall verwendet werden, wo ein Objekt des Obertyps erwartet wird (Ersetzbarkeit).
3. Insbesondere können alle Attribute des Obertyps auch für Objekte des Untertyps verwendet werden (Vererbung).
4. Für ein Objekt des Obertyps kann überprüft werden, ob es sich eigentlich um ein Objekt des Untertyps handelt (dynamischer Typtest). Wenn dies der Fall ist, kann das Objekt explizit in den Untertyp umgewandelt werden (explizite Abwärtsumwandlung).

Die erste dieser vier Eigenschaften, aus der die nächsten beiden automatisch folgen, kann in MOSTflexiPL wie folgt durch eine implizite Typumwandlung nachgebildet werden (vgl. §2.3):

```
"Citizen" : type;
"state" : Citizen -> string;
"idno" : Citizen -> string;

["c" : Citizen]
c : Person
{
    if c.<Citizen,Person> then
        c.<Citizen,Person>
    else
        "p" : Person;
        p <-> c;
        p
    end
}
```

Hier wird `Citizen` zunächst als normaler offener Typ mit Attributen `state` und `idno` definiert. Der anschließend definierte Operator ermöglicht eine implizite Aufwärtsumwandlung eines `Citizen`-Objekts `c` in ein „assoziiertes“ `Person`-Objekt `c.<Citizen,Person>`. Falls dieses noch nicht existiert, wird es als neues Objekt `p` erzeugt und mit dem Objekt `c` verbunden. Damit kann ein `Citizen`-Objekt z. B. wie folgt erstellt werden:

```
"c" : Citizen
c.name = "Heinlein";
c.state = "Germany"
```

Der folgende Operator realisiert die vierte Eigenschaft, indem er zu einem `Person`-Objekt `p` entweder das assoziierte `Citizen`-Objekt `p.<Person,Citizen>` liefert, sofern dieses existiert, oder den Sonderwert nil (vgl. §2.2):

```
["p" : Person]
p "?" "Citizen" : Citizen
{
    p.<Person,Citizen>
}
```

Damit kann dieser Operator sowohl für dynamische Typtests als auch für Abwärtsumwandlungen verwendet werden, z. B.:

```
if p?Citizen then
    print p?Citizen.state
end
```

Da die Definition einer Vererbungsbeziehung immer nach dem gleichen Schema erfolgt, ist es zweckmäßig, sie wiederum syntaktisch zu „verpacken“, z. B. mit Hilfe eines virtuellen Operators `•=>•`, dessen Definition hier aus Platzgründen weggelassen wird und der dann einfach wie folgt verwendet werden kann:

```
"Man" : type;
"bearded" : Man -> bool;
Man => Person
```

Da die impliziten Umwandlungen, die durch den Operator `•=>•` definiert werden, bei Bedarf auch transitiv angewandt werden, funktionieren mehrstufige Untertypbeziehungen ganz genauso.

3.4 Mehrfachvererbung

Durch mehrfache Verwendung des Operators `•=>•` kann ein Typ auch mehrere Obertypen besitzen, z. B.:

```
"User" : type;
"username" : User -> string;
"password" : User -> string;

"Employee" : type;
Employee => Person;
Employee => User
```

Damit kann ein Objekt des Typs `Employee` je nach Bedarf sowohl in `Person` als auch in `User` umgewandelt und entsprechend verwendet werden:

```
"e" : Employee;
e.name = "Heinlein";
e.username = "cheinl"
```

Da die Typen `Man` und `Citizen` beide Untertypen von `Person` sind und damit sowohl `Man`- als auch `Citizen`-Objekte jeweils ein assoziiertes `Person`-Objekt besitzen, entsteht im folgenden Beispiel durch Mehrfachvererbung die unerwünschte Situation, dass ein `MaleCitizen`-Objekt zwei verschiedene (indirekt) assoziierte `Person`-Objekte besitzt und deshalb die Umwandlung von `MaleCitizen` nach `Person` mehrdeutig ist (vgl. Abb. 1 links):

```
"MaleCitizen" : type;
MaleCitizen => Man;
MaleCitizen => Citizen
```

Abb. 1 rechts zeigt die eigentlich gewünschte Rautenstruktur, bei der es zu einem `MaleCitizen`-Objekt nur *ein* assoziiertes `Person`-Objekt gibt und die Umwandlung von `MaleCitizen` nach `Person` dementsprechend eindeutig ist. Um dies zu erreichen, sind mehrere Maßnahmen erforderlich:

- Die beiden indirekten Umwandlungen von `MaleCitizen` über `Man` bzw. `Citizen` nach `Person` müssen verboten werden, was sich mit Hilfe von *Ausschlussdeklarationen* realisieren lässt.[1]
- Stattdessen muss eine direkte Umwandlung von `MaleCitizen` nach `Person` definiert werden (gestrichelter Pfeil in der Abbildung).
- Bei der Erzeugung der assoziierten Objekte zu einem `MaleCitizen`-Objekt muss darauf geachtet werden, dass das `Man`- und das `Citizen`-Objekt auf dasselbe `Person`-Objekt verweisen.

Die entsprechenden Definitionen, die im Detail etwas „verzwickt“ sind, können wiederum hinter einem Operator mit „schöner“ Syntax versteckt werden, der dann z. B. wie folgt verwendet werden kann:

```
MaleCitizen => { Man | Citizen } => Person
```

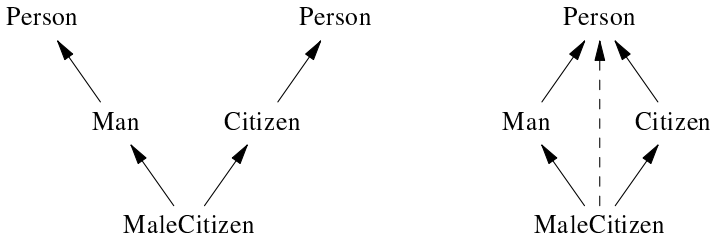

Abbildung 1: Mehrfachvererbung in V- bzw. Rautenform

[1] Ausschlussdeklarationen verbieten grundsätzlich bestimmte Verschachtelungen von Operatoranwendungen und werden primär zur Definition von Operatorvorrang eingesetzt. Um beispielsweise die bekannte Punkt-vor-Strich-Regel für arithmetische Operatoren zu implementieren, werden direkte Anwendungen der multiplikativen Operatoren •`*`• und •`/`• auf Anwendungen der additiven Operatoren •`+`• und •`-`• ausgeschlossen. Damit kann ein prinzipiell mehrdeutiger Ausdruck wie z. B. `a + b * c` nur noch als `a + (b * c)` interpretiert werden, weil die Interpretation als `(a + b) * c` ausgeschlossen ist.

Mit einer naheliegenden Verallgemeinerung dieses Operators lassen sich dann auch noch komplexere Vererbungsbeziehungen modellieren, z. B. Doppelstaatsbürger, die zwar zwei verschiedene `Citizen`-Teilobjekte, aber nur *ein* gemeinsames `Person`-Teilobjekt besitzen sollen (vgl. Abb. 2):

```
"Citizen1" : type;
"Citizen2" : type;
"DualCitizen" : type;

DualCitizen =>
    { Citizen1 => Citizen | Citizen2 => Citizen } => Person
```

Die Hilfstypen `Citizen1` und `Citizen2` werden gebraucht, um die beiden `Citizen`-Teilobjekte unterscheiden und ansprechen zu können, z. B.:

```
"dc" : DualCitizen;
"c1" : Citizen1 = dc; c1.state = "Germany";
"c2" : Citizen2 = dc; c2.state = "USA";
```

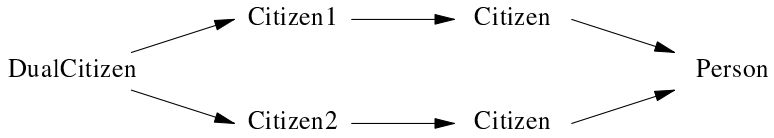

Abbildung 2: Doppelstaatsbürger

Durch mehrfache Anwendung dieser Operatoren lassen sich schließlich auch hochkomplexe Strukturen wie z. B. männliche oder weibliche Doppelstaatsbürger modellieren, die mehrere verschränkte Rauten enthalten.

3.5 Weiterführende Möglichkeiten

Ebenso wie man Attribute auch nachträglich zu offenen Typen hinzufügen kann, kann man mit den zuvor beschriebenen Operatoren auch Vererbungsbeziehungen nachträglich definieren – eine Möglichkeit, die objektorientierte Programmiersprachen normalerweise nicht bieten. Insbesondere ist es möglich, nachträglich Obertypen zu einem Typ hinzuzufügen.

Außerdem kann die Tatsache, dass ein Objekt eines Untertyps in Wirklichkeit aus mehreren miteinander verbundenen Teilobjekten besteht, ausgenutzt werden, um auf einfache Weise *dynamische Objektevolution* zu implementieren – eine Möglichkeit, die man in gängigen objektorientierten Programmiersprachen ebenfalls schmerzlich vermisst. Um beispielsweise nachträglich aus einer gewöhnlichen Person einen Mann oder einen Staatsbürger zu machen, genügt es, ein entsprechendes `Man`- oder `Citizen`-Objekt zu erzeugen und mit dem bereits vorhandenen `Person`-Objekt zu verbinden. Mit einer geeigneten syntaktischen Verpackung kann man dann als Anwender z. B. schreiben:

```
"p" : Person;
p.name = "Heinlein";

"c" : Citizen = p!Citizen;
c.state = "Germany"
```

3.6 Dynamisch gebundene Operationen

Dynamisch gebundene Operationen, die abhängig vom tatsächlichen Typ des Aufrufobjekts unterschiedliche Implementierungen ausführen, werden von Compilern üblicherweise durch „virtual function tables“ implementiert. Wenn eine Programmiersprache Funktionszeiger o. ä. unterstützt, d. h. die Möglichkeit bietet, Funktionen o. ä. in Variablen zu speichern, können derartige Tabellen aber auch auf Anwendungsebene realisiert werden. Da Operatoren in MOSTflexiPL als Werte verwendet werden können, ist diese Möglichkeit gegeben, d. h. dynamisches Binden lässt sich prinzipiell auf Anwendungsebene implementieren.

Um nicht nur das gebräuchliche „single dispatch“, sondern das wesentlich flexiblere „multiple dispatch“ anbieten zu können, bei dem die dynamischen Typen *aller* Aufrufparameter bei der Auswahl der passenden Implementierung berücksichtigt werden können, sind komplexere Tabellenstrukturen erforderlich, die sich aber ebenfalls auf Anwendungsebene implementieren lassen.

Eine ansprechende syntaktische Verpackung könnte dann z. B. wie folgt aussehen:

```
"equal" ("p1" : Person; "p2" : Person) : bool
{
    p1.name == p2.name
};

"equal" ("c1" : Person?Citizen; "c2" : Person?Citizen) : bool
{
    c1.name == c2.name &
    c1.state == c2.state &
    c1.idno == c2.idno
}
```

Die erste Implementierung der Methode `equal` ist die allgemeinste, die für beliebige Personen `p1` und `p2` aufgerufen werden kann. Die zweite Implementierung stellt eine Spezialisierung dar, die nur ausgewählt wird, wenn beide Parameter den dynamischen Typ `Citizen` besitzen, d. h. `Person`-Objekte mit assoziierten `Citizen`-Objekten sind.

Wenn bei der Auswahl der passenden Methodenimplementierung nicht nur die Typen, sondern beliebige Eigenschaften der Aufrufparameter berücksichtigt werden können, spricht man von „predicate dispatching“ [Er98]. Auch dieses Konzept lässt sich mit MOSTflexiPL prinzipiell umsetzen.

4 Diskussion

Das primäre Ziel dieses Beitrags ist es, die vielfältigen Möglichkeiten von MOSTflexiPL anhand einer praxisrelevanten „Aufgabenstellung“ – Unterstützung für objektorientierte Programmierung – zu demonstrieren. Da die konkrete „Lösung“ dieser Aufgabe, d. h. die exakte Syntax und Semantik der hierfür definierten Operatoren, für dieses Ziel sekundär ist, sollen weder die konkret gewählte Lösung noch mögliche Alternativen an dieser Stelle ausführlich diskutiert werden, sondern lediglich einige wesentliche Unterschiede zu „gängigen“ Ansätzen aufgezählt werden. Eine ausführlichere Diskussion findet sich in [He07], wo viele der hier vorgestellten Ideen bereits im Zusammenhang mit der Programmiersprache C+++ vorgestellt wurden.

- Auf das Konzept einer Klasse als feste Zusammenfassung einer Datenstruktur und der zugehörigen Operationen, wurde bewusst verzichtet.
- Stattdessen können Datentypen in beliebiger Reihenfolge definiert, mit Attributen versehen und in Vererbungsbeziehungen zueinander gesetzt werden. Damit sind nachträgliche Erweiterungen und Anpassungen viel leichter möglich als mit „starren“ Klassen.
- (Multi-)Methoden werden prinzipiell unabhängig von Klassen bzw. Typen definiert (ähnlich wie „generic functions“ in CLOS) und können daher ebenfalls problemlos nachträglich hinzugefügt werden. Damit existiert das für normale objektorientierte Sprachen schwerwiegende „expression problem“ [To04] in dieser Art schlicht und einfach nicht.
- Mehrfachvererbung wird ohne Einschränkungen unterstützt, weil sie für viele Anwendungen nützlich ist und das Prinzip von Vererbung und Untertyp-Polymorphie konsequent fortsetzt.
- Das dadurch unvermeidliche „diamond inheritance problem“ wird einfach, elegant und umfassend gelöst. Im Gegensatz zu anderen Sprachen mit Mehrfachvererbung (namentlich CLOS, Eiffel und C++), lassen sich beliebig komplexe Vererbungsstrukturen (wie z. B. männliche und weibliche Doppelstaatsbürger) ohne große Mühe modellieren.
- Ganz „nebenbei“ wird mit dynamischer Objektevolution auch noch ein Konzept unterstützt, das in statisch typisierten Programmiersprachen üblicherweise komplett fehlt.

Aus Platzgründen wurde das Prinzip des „information hiding“ [Pa72] komplett ausgeklammert. Aber auch hierfür bietet MOSTflexiPL mit sogenannten Sichtbarkeitsdeklarationen [He12] adäquate Unterstützung.

5 Zusammenfassung und Ausblick

In diesem Beitrag wurde gezeigt, wie man in MOSTflexiPL Operatoren zur Unterstützung objektorientierter Programmierung definieren und verwenden kann. Neben der hier vorgestellten „Lösung“ dieser „Aufgabe“, sind natürlich auch vielfältige alternative Lösungsansätze denkbar.

Ein essentielles Kernkonzept der Sprache, das zur Lösung der Aufgabe eingesetzt wird, sind benutzerdefinierbare implizite Typumwandlungen. Dieses muss im Detail noch etwas weiterentwickelt und verbessert werden, um beispielsweise unerwartete und unerwünschte Mehrdeutigkeiten aufgrund impliziter Umwandlungen zu eliminieren.

Eine weitere große „Baustelle“ von MOSTflexiPL ist nach wie vor die Ausgabe sinnvoller und hilfreicher Fehlermeldungen zur Übersetzungszeit sowie die Fortsetzung des Übersetzungsvorgangs nach einem Fehler. Außerdem muss für einen produktiven Einsatz von MOSTflexiPL einerseits die Effizienz des Compilers noch deutlich verbessert werden und andererseits das im Namen der Sprache bereits verankerte, aber momentan noch nicht verfügbare Modulkonzept implementiert werden.

Literaturverzeichnis

[Er98] M. Ernst, C. Kaplan, C. Chambers: “Predicate Dispatching: A Unified Theory of Dispatch.” In: E. Jul (ed.): *ECOOP'98 – Object-Oriented Programming* (12th European Conference; Brussels, Belgium, July 1998; Proceedings). Lecture Notes in Computer Science 1445, Springer-Verlag, Berlin, 1998, 186–211.

[He07] C. Heinlein: “Open Types and Bidirectional Relationships as an Alternative to Classes and Inheritance.” *Journal of Object Technology* 6 (3) March/April 2007, 101–151, http://www.jot.fm/issues/issue_2007_03/article3.

[He12] C. Heinlein: “MOSTflexiPL – Modular, Statically Typed, Flexibly Extensible Programming Language.” In: J. Edwards (ed.): *Proc. ACM Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2012)* (Tucson, AZ, October 2012), 159–178.

[He14] C. Heinlein: “Fortgeschrittene Syntaxerweiterungen durch virtuelle Operatoren in MOSTflexiPL.” In: K. Schmid et al. (ed.): *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2014* (Kiel, February 2014). CEUR Workshop Proceedings, 193–212, http://ceur-ws.org/Vol-1129/paper4A.pdf.

[Pa72] D. L. Parnas: “On the Criteria to Be Used in Decomposing Systems into Modules.” *Communications of the ACM* 15 (12) December 1972, 1053–1058.

[To04] M. Torgersen: “The Expression Problem Revisited. Four New Solutions Using Generics.” In: M. Odersky (ed.): *ECOOP 2004 – Object-Oriented Programming* (18th European Conference; Oslo, Norway, June 2004; Proceedings). Lecture Notes in Computer Science 3086, Springer-Verlag, Berlin, 2004, 123–143.