Realisierung rekursiver Datenstrukturen durch generische Klassen

Andreas Vox

Institut für Softwaretechnik und Programmiersprachen
Medizinische Universität zu Lübeck
http://www.isp.mu-luebeck.de
vox@isp.mu-luebeck.de

Zusammenfassung Wir stellen das Generic-Factory-Muster vor, welches der Realisierung rekursiver Datenstrukturen in objektorientierten Sprachen dient. Dieses Muster ist eine Weiterentwicklung des bekannten Composite-Musters und nutzt den in Sprachen wie Pizza [12] oder Generic Java [3] vorhandenen parametrischen Polymorphismus, um Ergebnisse aus der gut erforschten funktionalen Welt in objektorientierten Programmen nutzbar zu machen.

1 Einführung

Rekursive Datenstrukturen beschreiben baumartige Strukturen, in denen Teilbäume den gleichen Typ wie die Wurzel haben. Derartige Strukturen werden von fast allen Programmen benötigt; Beispiele reichen von einfachen Listen bis zu abstrakten Syntaxbäumen.

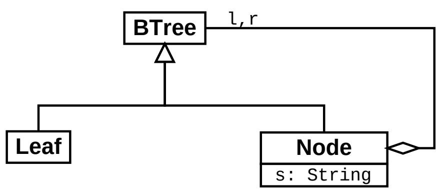
Der Aufbau von rekursiven Datenstrukturen läßt sich auf drei Bildungsprimitiva zurückführen: Tupelbildung (direktes Produkt), Alternativenbildung (direkte Summe) und Rekursion.

In funktionalen Sprachen lassen sich rekursive Datenstrukturen direkt als algebraischer Datentyp definieren:

```
datatype BTree = Leaf | Node of BTree * string * BTree
```

Hier wird Tupelbildung durch * und Alternativenbildung durch | dargestellt. Rekursion wird über Namensgleichheit erreicht.

In der objektorientierten Programmierung werden rekursive Datenstrukturen üblicherweise durch das Composite-Entwurfsmuster [5] realisiert. Danach wird die durch den obigen SML-Datentyp repräsentierte Datenstruktur durch drei Datenklassen implementiert, die folgende Klassenstruktur aufweisen:



Tupelbildung wird durch Aggregation bzw. dem Einfügen von Attributen ausgedrückt. Alternativenbildung entspricht der Spezialisierung durch Unterklassen. Rekursion wird durch Aggregation von Elementen der Oberklasse erreicht.

Bei komplexen Datenstrukturen – wie z.B. abstrakten Syntaxbäumen – wird für jede Alternative eine zusätzliche Datenklasse benötigt wird. Der Entwurf wird dadurch unhandlich und die Sichtweise verschiebt sich von der rekursiven Datenstruktur als Ganzen hin zu den einzelnen Datenklassen.

Unsere Lösung erweitert das Composite-Muster um ein Generic-Factory-Interface, welches die *Struktur* der Datenklassen beschreibt, ohne die *Typen* der Datenklassen festzulegen. Implementierungen dieses Interfaces haben die Aufgabe, konkrete Instanzen der Datenklassen zu erzeugen.

Eine `reduce`-Methode in den Datenklassen erlaubt es, Implementierungen des Generic-Factory-Interfaces ähnlich wie Visitors [5] zu verwenden. Im Gegensatz zum Visitor-Muster ist der Typ der Generic-Factory nicht von den Datenklassen abhängig. Dies ermöglicht es, Generic-Factories zur Beschreibung von Modulschnittstellen zu benutzen, ohne das Modul auf einen konkreten Datentyp festzulegen.

Diese Arbeit gliedert sich wie folgt: In Abschnitt 2 stellen wir das Generic-Factory-Muster vor und illustrieren die Technik durch mehrere Anwendungsbeispiele. Im Abschnitt 3 gehen wir auf den praktischen Einsatz dieses Musters in der Programmentwicklung ein. Der Zusammenhang mit algebraischen Methoden wird kompakt in Abschnitt 4 beschrieben. Abschnitt 5 schließt mit einer Zusammenfassung und einem Ausblick ab.

2 Das Generic-Factory-Muster

Die Hauptidee des Generic-Factory-Musters ist es, die Strukturbeschreibung einer rekursiven Datenstruktur von den Datenklassen zu trennen. Dadurch eröffnet sich die Möglichkeit, die Datenstruktur auf verschiedene Weisen zu implementieren, von denen das Composite-Muster eine „kanonische“ Realisierung beschreibt. In Abschnitt 2.1 stellen wir Generic-Factory-Interfaces vor, welche die typunabhängige Beschreibung von rekursiven Datenstrukturen leisten. In Abschnitt 2.2 beschreiben wir, wie die Datenklassen des Composite-Musters mit dem Generic-Factory-Interface zusammenhängen und wie sie über die `reduce`-Methode mit anderen Realisierungen derselben Datenstruktur zusammenarbeiten.

Darauf aufbauend zeigen wir in Abschnitt 2.3, wie das Generic-Factory-Interface benutzt werden kann, um Schnittstellen zwischen Komponenten zu beschreiben, die Exemplare dieser Datenstruktur produzieren und/oder konsumieren. Interessanterweise hat das zur Folge, dass diese Datenstrukturen gar nicht mehr erzeugt werden müssen.

2.1 Eine abstrakte Beschreibung für rekursive Datenstrukturen

Ein erster Ansatz ist es, die Factory-Klasse zu betrachten, welche die Exemplare der dem Composite-Muster entsprechenden Datenklassen erzeugt:

```
class BTreeFactory {
    BTree caseLeaf() { ... }
    BTree caseNode(BTree l, String s, BTree r) { ... }
}
```

Diese Klasse enthält bereits sämtliche Informationen über die Datenstruktur:

- `BTree`s sind aus den Klassen `Leaf` und `Node` aufgebaut und
- ein `Node` enthält zwei Teilbäume `l` und `r` sowie ein Datum `s` vom Typ `String`.

Der nächste Schritt besteht darin, diese Klasse zu generalisieren, indem die erzeugte Klasse durch einen Typparameter ersetzt wird:

```
interface GenericBTreeFactory<X> {
    X caseLeaf();
    X caseNode(X l, String s, X r);
}
```

Diese Form nennen wir ein Generic-Factory-Interface.

Um zu sehen, wie dieses Interface eine rekursive Datenstruktur beschreibt, kann es als rekursive Typgleichung

$$X = \texttt{unit} \,|\, X \times \texttt{String} \times X$$

gelesen werden (`unit` sei der einelementige Datentyp $\{()\}$).

(Seiteneffektfreie) Implementierungen von `GenericBTreeFactory` beschreiben Funktionen vom Typ $f : \texttt{unit} \,|\, X \times \texttt{String} \times X \to X$. Falls f bijektiv ist, so erfüllt X die obige Typgleichung. Dies ist zum Beispiel für `BTreeFactory` der Fall.

Im folgenden Abschnitt gehen wir darauf ein, wie beliebige (nicht bijektive) Implementierungen von `GenericBTreeFactory` sinnvoll mit `BTree`s zusammenarbeiten können.

2.2 Die `reduce`-Methode in den Datenklassen

Bei gegebenem Generic-Factory-Interface lassen sich die konkreten Datenklassen nach dem Composite-Muster automatisch erzeugen. Zusätzlich erhalten alle Datenklassen eine generische Methode

```
<R> R reduce (GenericBTreeFactory<R> fab);
```

welche in den Unterklassen jeweils die entsprechende Methode in `fab` aufruft. Dabei wird `reduce` rekursiv für diejenigen Felder aufgerufen, die vom Typ der Oberklasse sind.

Im Beispiel ergibt das folgenden Code:

```
abstract class BTree {
    abstract <R> R reduce(GenericBTreeFactory<R> fab);
}
class Leaf extends BTree {
    <R> R reduce(GenericBTreeFactory<R> fab) {
        return fab.caseLeaf();
}   }
class Node extends BTree {
    BTree l; String s; BTree r;
    <R> R reduce(GenericBTreeFactory<R> fab) {
        return fab.caseNode(l.reduce(fab), s, r.reduce(fab));
}   }
```

Algorithmisch realisiert `reduce` einen bottom-up Baumdurchlauf, wobei der Generic-Factory-Parameter angibt, wie die Zwischenergebnisse kombiniert werden.

Der wichtige Punkt ist, dass das `GenericBTreeFactory`-Interface ohne Verwendung von `BTree` oder seinen Unterklassen implementiert werden kann. Im Zusammenspiel mit `reduce` lassen sich so Algorithmen unabhängig vom Typ der konkreten Datenklassen formulieren. Beispielsweise liefert die Klasse

```
class Counting implements GenericBTreeFactory<Integer> {
    Integer caseLeaf() {
        return new Integer(1); }
    Integer caseNode(Integer l, String s, Integer r) {
        return new Integer(1 + l.intValue() + r.intValue()); }
}
```

über den Aufruf t.`reduce(new Counting())` die Anzahl der Knoten eines Baumes t . Ein weiteres Beispiel ist die folgende Klasse, welche die Konkatenation aller `String`-Daten in einem `Btree` liefert:

```
class AsString implements GenericBTreeFactory<String> {
    Integer caseLeaf() {
        return ""; }
    Integer caseNode(String l, String s, String r) {
        return l + s + r; }
}
```

Als drittes Beispiel repräsentiert die Klasse `BTreeFactory` die identische Abbildung auf `BTree`: der Aufruf t.`reduce(new BTreeFactory())` liefert eine tiefe Kopie von t .

Allerdings lässt sich nicht jede Abbildung f : `BTree` $\rightarrow M$ durch die Implementierung einer Generic-Factory ohne Seiteneffekte realisieren, ein Gegenbeispiel ist etwa die Funktion `insert`: `String` $\times$ `Btree` $\rightarrow$ `BTree` . Eine Implementierung von `insert` müsste sich festlegen, ob sie für ein `Leaf()` den Wert `Leaf()` oder den Wert `Node(Leaf(), s, Leaf())` zurückgibt. Im ersten Fall würde der Baum lediglich kopiert, im zweiten Fall würde `s` an allen Blättern eingefügt. Dieses Problem ließe sich durch Seiteneffekte umgehen; die Implementierung als Methode oder als Visitor ist jedoch eleganter.

2.3 Effiziente modulare Algorithmen

Generic-Factories lassen sich auf drei Weisen zur Definition von Schnittstellen zwischen Algorithmen benutzen:

A) Algorithmen, die einen Parameter vom Typ einer Generic-Factory haben, bauen rekursive Datenstrukturen auf; zum Beispiel ein Parser, der einen abstrakten Syntaxbaum erzeugt.
B) Algorithmen, die als Generic-Factory implementiert sind, konsumieren rekursive Datenstrukturen und erzeugen daraus ein Ergebnis. Ein Beispiel ist die Auswertung von synthetisierten Attributen in einem abstrakten Syntaxbaum.
C) Algorithmen, die sowohl als Generic-Factory implementiert sind als auch einen Parameter vom Typ einer (nicht notwendig derselben) Generic-Factory haben, transformieren eine rekursive Datenstruktur in eine andere. Ein Beispiel ist die Umwandlung eines konkreten Syntaxbaums in den entsprechenden abstrakten Syntaxbaum.

Durch Parameter-Instantiierung lassen sich diese Algorithmen zu neuen Algorithmen zusammensetzen. Übergibt man etwa einem Parser vom Typ A einen passenden Attributauswerter vom Typ B, so erhält man einen Parser, der direkt das synthetisierte Attribut liefert und *dabei den Syntaxbaum gar nicht mehr erzeugt*. Dieser Effekt ist aus der funktionalen Programmierung als Shortcut Deforestation bekannt [16][15].

Als Beispiel sei ein Parser und eine einfache Generic-Factory für Listenstrukturen gegeben:

```
interface GenericListFactory<L> {
    L caseNil();
    L caseCons(String head, L tail);
}
class Length implements GenericListFactory<Integer> {
    Integer caseNil() {
        return new Integer(0); }
    Integer caseCons(String head, Integer tail) {
        return new Integer(1 + tail.intValue()); }
}
static <R> R parse(String eingabe, GenericListFactory<R> fab) {
    int pos = s.indexOf(";");
    if (pos < 0) {
        return fab.caseNil(); }
    else {
        return fab.caseCons(s.substring(0, pos), parse(s.substring(pos+1)));
}   }
```

Der Aufruf `parse("a;b;c;d;", new Length())` liefert die Länge der durch `"a;b;c;d;"` repräsentierten Liste, ohne diese Liste zu konstruieren.

Als Beispiel für Typ-C-Algorithmen dient folgende Klasse, die eine Listenstruktur in einen entarteten Binärbaum transformiert:

```
class List2BTreeConverter<T> implements GenericListFactory<T> {
    GenericBTreeFactory<T> fab;
    List2BTreeConverter(GenericBTreeFactory<T> fab) {
        this.fab = fab; }
    T caseNil() {
        return fab.caseLeaf();
    }
    T caseCons(String head, T tail) {
        return fab.caseNode(fab.caseLeaf(), head, tail);
    }
}
```

Der Generic-Factory-Parameter wird dem Konstruktor übergeben und als Referenz in `List2BTreeConverter` gespeichert. Über folgende Pipeline ist es jetzt möglich, die für `BTree`s definierte Klasse `AsString` auch auf Listen anzuwenden:

```
GenericBTreeFactory<String> pipe1 = new AsString();
GenericListFactory<String> pipe2 = new List2BTreeConverter(pipe1);
String text = parse(pipe2);
```

Durch die automatische Shortcut Deforestation werden die dazwischenliegenden Datenstrukturen nicht aufgebaut:

$$\boxed{\texttt{parse}} \xrightarrow{(List)} \boxed{\texttt{List2BTreeConverter}} \xrightarrow{(BTree)} \boxed{\texttt{AsString}} \longrightarrow \texttt{String}$$

3 Programmentwicklung mit Generic-Factories

In diesem Abschnitt soll gezeigt werden, wie sich das Generic-Factory-Muster in den Entwicklungsprozess einordnet. Die Verwendung von Generic-Factories gliedert sich in drei aufeinander aufbauende Schritte:

1. Entwurf einer rekursiven Datenstruktur als Generic-Factory-Interface.
   Ein Generic-Factory-Interface beschreibt kompakt die Alternativen und Felder einer rekursiven Datenstruktur. Die Datenstruktur wird dadurch als eine Einheit und nicht als Sammlung kooperierender Klassen aufgefasst, wie im Composite-Muster.
2. Entwurf von Algorithmen und Modulen, die mit rekursiven Datenstrukturen arbeiten.
   Der Einsatz von Generic-Factories zur Definition von Modulschnittstellen bietet den Vorteil, dass nur die *Struktur*, nicht jedoch der *Typ* der Daten festgelegt wird. Da die Struktur für die meisten Algorithmen wichtiger ist als der konkrete Typ, eröffnet dies mehr Möglichkeiten für Kombination und Wiederverwendung.
   Zusätzlich kann das Visitor-Muster in den Fällen eingesetzt werden, in denen eine Implementierung als Generic-Factory nicht möglich oder sinnvoll ist.

3. Automatische Codegenerierung für die Datenklassen.
   Wir entwicklen zur Zeit einen Codegenerator, der aus einem Generic-Factory-Interface die zugehörigen Datenklassen erzeugt. Die verbleibenden Entwurfsentscheidungen für die Implementierung der rekursiven Datenstruktur werden in Vorlagen für den Codegenerator gekapselt, um so eine gleichförmige Implementierung der Datenklassen zu gewährleisten. Manuelle Änderungen an einzelnen Datenklassen sind nicht wünschenswert und auch nicht nötig, da Visitors, Generic-Factories und externe Tabellen ausreichen, um zusätzliche Funktionalität zu implementieren.
   Der eingesetzte Codegenerator sollte außer den Datenklassen auch das passende Visitor-Interface und eine Reihe von Standardalgorithmen (Iteratoren, Depth-First-Traversal, Level-Order-Traversal, equals, Ein-/Ausgabe, etc.) erzeugen.

4 Vergleich zu algebraischen Ansätzen

Die meisten Ideen für das Generic-Factory-Muster stammen aus der Welt der funktionalen Programmiersprachen [2][4], polytypischer Programmierung [10] und der universellen Algebra [11][7]. In Abschnitt 4.1 fassen wir kompakt die algebraischen Grundlagen rekursiver Datentypen zusammen und setzen sie in 4.2 zum Generic-Factory-Musters in Beziehung. Abschnitt 4.3 zeigt die Grenzen dieser Entsprechung auf.

4.1 F-Algebren

Ein (polynomialer) Funktor ist eine durch Konstanten, direkte Summe und direktes Produkt gebildete Mengenabbildung, die gleichzeitig die Funktionenräume aufeinander abbildet. Ist etwa $\phi(M) := \{()\} + A \times M$ ein Funktor, so wird eine Funktion $f : M \to M'$ durch ϕ auf die Funktion $\phi(f) : \phi(M) \to \phi(M'), \phi(f) = \mathrm{id}_{\{()\}} + \mathrm{id}_A \times f$ abgebildet ($\phi(f)(()) = ()$ und $\phi(f)(a, x) = (a, f(x))$).

Eine F-Algebra $(A, a : \varphi(A) \to A)$ ist ein Paar aus einer Trägermenge A und einer Strukturoperation a. Die Strukturoperation legt fest, wie aus Konstanten und Werten aus A neue Werte in A gebildet werden können.

Eine Varietät ist die Klasse aller F-Algebren, die denselben Funktor ϕ benutzen. Ein Homomorphismus ist eine Struktur erhaltende Abbildung zwischen F-Algebren der gleichen Varietät. Eine F-Algebra $(I, i : \phi(I) \to I)$ ist initial, wenn es zu jeder F-Algebra $(A, a : \phi(A) \to A)$ genau einen Homomorphismus $(\!|a|\!)$ gibt, der (I, i) auf (A, a) abbildet. Diese Homomorphismen werden auch Catamorphismen [2] [1] genannt; zu ihnen sind in der funktionalen Welt eine Reihe von Umformungsgesetzen bekannt.

Ein für die Semantik von algebraischen Datentypen wichtiges Ergebnis ist, dass es in allen Varietäten, die auf einem polynomialen Funktor beruhen, eine initiale Algebra gibt, die bis auf Isomorphie eindeutig ist [6]. Anders ausgedrückt definiert jeder polynomiale Funktor eine Datenstruktur.

4.2 Entsprechungen im Generic-Factory-Muster

Im Generic-Factory-Muster beschreibt das Generic-Factory-Interface genau die Signatur der Strukturabbildungen $a : \phi(A) \to A$. Ein Generic-Factory-Interface definiert daher eine Varietät. Jede seiteneffektfreie Implementierung definiert eine Strukturabbildung und legt über den Typparameter die Trägermenge fest, daher sind seiteneffektfreie Implementierungen des Generic-Factory-Interfaces nichts anderes als F-Algebren. Die Generic-Factory, welche die Datenklassen nach dem Composite-Muster erzeugt, entspricht der initialen Algebra, wobei die `reduce`-Methode den Homomorphismus $(\!| \cdot |\!)$ von der initialen Algebra in eine beliebige andere Algebra realisiert.

4.3 Unterschiede

Die oben aufgezählten Entsprechungen sind nur Idealvorstellungen, die in der imperativen objektorientierten Welt nicht erfüllt sein müssen. Es gibt drei offensichtliche Probleme, welche die direkte Übertragung algebraischer Methoden auf objektorientierte Programmiersprachen wie Java behindern.

Der erste Unterschied ist die Möglichkeit von Seiteneffekten und Nichtdeterminismus. In der funktionalen Welt werden Seiteneffekte durch Monaden [17] gekapselt oder durch ein Effektsystem beschrieben [18]. Die Übertragung dieser Methoden auf objektorientierte Programme steht noch aus.

Ein subtilerer Unterschied ist das Vorhandensein einer Objektidentität. Dies impliziert, dass in der objektorientierten Welt eine feinere Gleichheit existiert als in der algebraischen Welt. Ein Ausweg kann sein, statt `==` nur die strukturelle Gleichheit mittels `equals` zuzulassen.

Eine weitere Besonderheit objektorientierter Sprachen ist der Nullwert. Da `null` automatisch Element jedes neu definierten Datentyps ist, werden die als Generic-Factories aufgeschriebenen Funktoren automatisch als $\phi(M) + \{\texttt{null}\}$ interpretiert. Dies ist allerdings keine wesentliche Einschränkung.

5 Zusammenfassung und Ausblick

Das Generic-Factory-Muster stellt den Entwurf rekursiver Datenstrukturen auf eine abstraktere Ebene, da die Struktur allein aus dem Generic-Factory-Interface ersichtlich ist, und die einzelnen Datenklassen für das Verständnis nicht mehr wichtig sind. Die Formulierung von Algorithmen als Generic-Factory oder als Visitor läuft auf der gleichen Abstraktionsebene ab. Die Datenklassen können später durch ein Werkzeug automatisch generiert werden.

Modulschnittstellen, die mit Hilfe eines Generic-Factory-Interface beschrieben werden, legen nur die Struktur der rekursiven Datenstruktur fest, nicht den konkreten Typ. Die Module werden dadurch leichter wiederverwendbar. Als zusätzlicher Vorteil ist die Komposition solcher Module effizienter als der traditionelle Austausch von Datenstrukturen, da zwischen den Modulen automatisch Shortcut Deforestation stattfindet.

Verwandte Arbeiten im objektorientierten Bereich schlagen andere Wege ein als wir: Pizza [12], eine Erweiterung von Java und ein Vorläufer von GJ [3], kennt algebraische Datentypen und switch-Anweisungen über algebraischen Datentypen. Diese Erweiterungen sind aber nicht generisch und erlauben daher nicht die selbe Flexibilität wie unsere Lösung. Es gibt eine Reihe Ansätze, das Visitor-Muster zu verbessern oder zu erweitern [9] [13] [14] [8], die jedoch nicht unsere abstrakte Sichtweise einnehmen, eine rekursive Datenstruktur als Einheit zu betrachten. Teilweise opfern diese Ansätze Typsicherheit für Flexibilität.

Aus der funktionalen Programmierung ist eine Vielzahl von Arbeiten bekannt, die Programmtransformation nutzen um die Effizienz von Programmen zu steigern. Wir gehen davon aus, dass die Verwendung von Generic-Factories Ansatzpunkte für weitere Optimierungen liefern wird, die in normalen objektorientierten Programmen nicht bestehen. Shortcut Deforestation zeigt, dass generische Programmierung auch Laufzeitvorteile bringen kann.

Ziel laufender Forschung ist es, den Zusammenhang zwischen algebraischer Welt und objektorientierten Programmen zu formalisieren und Vorbedingungen zu identifizieren, unter denen algebraische Transformationsgesetze auf objektorientierte Programme übertragen werden können.

Literatur

1. Backhouse, Roland, Patrik Jansson, Johan Jeuring und Lambert Meertens: *Generic Programming –An Introduction–*. In: Swierstra, S. Doaitse, Pedro R. Henriques und José N. Oliveira (Herausgeber): *Revised Lectures 3rd Int. School on Advanced Functional Programming, AFP'98, Braga, Portugal, 12–19 Sept. 1998*, Band 1608 der Reihe *Lecture Notes in Computer Science*, Seiten 28–115. Springer-Verlag, Berlin, 1999.
2. Bird, Richard S.: *An Introduction to the Theory of Lists*. In: Broy, M. (Herausgeber): *Logic of Programming and Calculi of Discrete Design*, Band 36 der Reihe *NATO ASI Series F*, Seiten 3–42. Springer-Verlag, Berlin, 1987.
3. Bracha, Gilad, Martin Odersky, David Stoutamire und Philip Wadler: *Making the future safe for the past: Adding Genericity to the Java Programming Language*. In: *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, Vancouver, October 1998. ACM.
4. Duris, Etienne, Didier Parigot, Gilles Roussel und Martin Jourdan: *Attribute Grammars and Folds : Generic Control Operators*. Technical Report RR-2957, Inria, Institut National de Recherche en Informatique et en Automatique.
5. Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
6. Goguen, J. A., J. W. Thatcher und E. G. Wagner: *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types*. In: Yeh, Raymond T. (Herausgeber): *Current Trends in Programming Methodology*, Band 4, Seiten 80–149. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1978.
7. Jacobs, Bart und Jan Rutten: *A Tutorial on (Co)Algebras and (Co)Induction*. Bulletin of the EATCS, 62:222–259, 1996.
8. Krishnamurthi, Shriram, Matthias Felleisen und Danial P. Friedman: *Synthesizing Object-Oriented and Functional Design to Promote Re-use*. In: *Euro-*

pean Conference on Object-Oriented Programming (ECOOP), Seite 91ff, Brussels, July 1998.

9. Lieberherr, Karl J. und Doug Orleans: *Preventive Program Maintenance in Demeter/Java.* In: *Proceedings of the 19th International Conference on Software Engineering*, Seiten 604–605. ACM Press, Mai 1997.
10. Meertens, Lambert: *Calculate Polytypically!* In: Kuchen, H. und S. D. Swiestra (Herausgeber): *Proceedings 8th Int. Symp. on Programming Languages: Implementations, Logics, and Programs, PLILP'96, Aachen, Germany, 24–27 Sept 1996*, Band 1140 der Reihe *Lecture Notes in Computer Science*, Seiten 1–16. Springer-Verlag, Berlin, 1996.
11. Meinke, K. und J. V. Tucker: *Handbook of Logic in Computer Science*, Band 1, Kapitel Universal Algebra, Seite 189ff. Oxford Science Publications, 1992.
12. Odersky, Martin und Philip Wadler: *Pizza into Java: Translating theory into practice.* In: *24'th ACM Symposium on Principles of Programming Languages*, Paris, January 1997.
13. Ovlinger, Johan und Mitchell Wand: *A Language for Specifying Recursive Traversals of Object Structures.* In: *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, Seiten 70–81, November 1999.
14. Palsberg, J. und C.B. Jay: *The Essence of the Visitor Pattern.* Technischer Bericht05, 1997. COMPSAC'98, to appear.
15. Takano, Akihiko und Erik Meijer: *Shortcut Deforestation in Calculational Form.* In: *Conf. Record 7th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, Seiten 306–313. ACM Press, New York, 1995.
16. Wadler, Philip: *Theorems for free!* In: *4'th International Conference on Functional Programming Languages and Computer Architecture*, London, UK, September 1989.
17. Wadler, Philip: *Monads for functional programming.* In: Jeuring, J. und E. Meijer (Herausgeber): *Advanced Functional Programming*, Band 925 der Reihe *LNCS*. Springer Verlag, 1995.
18. Wadler, Philip: *The marriage of effects and monads.* In: *International Conference on Functional Programming*, Seiten 63–74, Baltimore, September 1998. ACM.