Family Polymorphism

Michael Schnell

23. - 25. Januar 2006

Seminar: Java, Grundlagen und Erweiterungen
Arbeitsbereich Programmiersprachen
Institut für Informatik
Universität Freiburg
Wintersemester 2005/2006

1 Einleitung

Diese Arbeit ist eine schriftliche Ausarbeitung im Rahmen des Seminars *Java, Grundlagen und Erweiterungen* aus dem Wintersemester 2005/2006. Sie basiert hauptsächlich auf dem Artikel *Family Polymorphism* von Erik Ernst. [1]

Eine Familie kann aus einem Vater, einer Mutter und einer Tochter bestehen. Zusammen bilden diese drei eben eine Familie. Es gibt aber sehr viel mehr als eine Familie auf der Welt und daher auch sehr viel mehr Väter, Mütter und Töchter. Und nicht jede Kombination von einem Vater, einer Mutter und einer Tochter ergeben tatsächlich eine Familie, da sie alle zu unterschiedlichen Familien gehören können.

Beim Programmieren arbeitet man meist mehr als mit nur einem Objekt einer Klasse. Dabei stehen die Klassen oftmals zueinander in Relationen und bilden sozusagen eine Familie, eine Klassenfamilie. Und ebenso, wie es sich anbietet von einzelnen Klassen abzuleiten, so bietet es sich auch an von ganzen Familien abzuleiten. Familienpolymorphismus ist ein Programmiersprachenfeature, das es uns ermöglicht solche Familien zu definieren und dabei sicherstellt, daß die Familien nicht vermischt werden.

Betrachten wir einmal, was uns traditionelle Vererbung, Polymorphismus und späte Bindung bieten. Eine polymorphe Referenz `x` kann zur Laufzeit auf eine Instanz einer Klasse C_i verweisen. C_i kann dabei eine Klasse aus der Menge $C = \{C_0, \cdots, C_n\}$ sein. Dabei kann C_0 die Basisklasse sein, und die C_1 bis C_n sind direkte oder indirekte Subklassen von C_0. Es ist uns dann möglich eine Methode `m` von `x` aufzurufen, syntaktisch meist durch `x.m()`. Jeder der Klassen aus C kann seine eigene Implementation von `m` haben. Durch späte Bindung wird sichergestellt, daß zur Laufzeit stets die zu C_i gehörige Implementation von `m` aufgerufen wird. Mittels einer statischen Analyse kann sichergestellt werden, daß es auch für jeden Aufruf von `x.m()` eine Implementation gibt.

Diese Mechanismen bieten uns also die Flexibilität eine unbegrenzte Anzahl von Klassen und Implementationen von `m` mit ein und dem selben Aufruf `x.m()` zu benutzen und bietet gleichzeitig sie Sicherheit, daß es stets eine Implementation gibt und zur Laufzeit auch immer die zum aktuellen Objekt gehörige Implementation aufgerufen wird.

Diese tolle Eigenschaften gelten aber nur bei Verwendung von einem Objekt. Sobald ein zweites Objekt dazukommt gehen diese Eigenschaften ein Stück weit verloren. Betrachten wir jetzt also einmal den Fall mit einem weiteren Objekt, das als Argument der Methode `m` übergeben wird. Syntaktisch: `x.m(y)`. Diesen Ausdruck verbinden traditionelle objektorientierte Programmiersprachen, wie Java und C++, nur mit zwei statisch bekannten Klassen, der Klasse C_x von `x` und der Klasse C_y von `y`. Zur Laufzeit kann `x` auf eine beliebige Instanz einer Subklasse von C_x und `y` auf eine beliebige Instanz einer beliebigen Subklasse von C_y zeigen. Es gibt keine Möglichkeit statisch sicherzustellen, daß eine bestimmte Subklasse von C_x stets mit einer bestimmten Subklasse von C_y auftreten muss.

Es ist mit dem traditionellen Verständnis von Polymorphie daher nicht möglich Relationen zwischen mehreren Objekten und ihrer Methoden zu definieren. Im zweiten Abschnitt werden wir sehen, daß wir Relationen zwischen mehreren Objekten nur dadurch erreichen können, daß wir entweder auf Typsicherheit oder Flexibilität verzichten.

Familienpolymorphismus ist eine erweiterte Art von Polymorphismus. Er ermöglicht es Beziehungen zwischen mehreren Klassen statisch zu definieren, so daß diese Klassen eine Familie bilden. Er bietet die Möglichkeit Methoden zu implementieren, die Mitglieder einer Klassenfamilie als Argumente entgegennehmen, und stellt statisch sicher, daß verschiedene Familien nicht vermischt werden. Und das ohne dabei statisch die exakte Familie zu kennen.

Es gibt einige Anzeichen dafür, daß Klassenfamilien in Zukunft immer wichtiger werden. Und daher wird oder ist es auch wichtig, daß man bei Verwendung von Klassenfamilien stets sowohl Typsicherheit als auch Flexibilität in Form von Wiederverwendbarkeit des Codes gleichzeitig bieten kann. Es geht dabei im Prinzip immer darum, daß mehrere Klassen beteiligt sind. Beispielsweise braucht man in der Generativen Programmierung in größerem Rahmen Variabilität, also im Bezug auf mehrere Klassen. Auch in der aspektbezogenen Programmierung, in der man versucht Anforderungen an ein Programm zu kapseln, sind bei so genannten Cross-Cutting-Concerns meistens mehr als eine Klasse beteiligt. Auch in traditionellen Sprachen (wie Java/C++) kann man Familien definieren. Jedoch geht dies dort nur implizit mit oben genannten Problemen und könnte daher einen Mechanismus gebrauchen der Typsicherheit und Flexibilität gleichzeitig bietet.

Sobald ein System mehr als eine Variante einer Klassenfamilie enthält muss dafür gesorgt werden, daß sich diese Varianten nicht vermischt benutzt werden.

Der Aufbau der restlichen Arbeit ist folgender: Der zweite Abschnitt führt anhand von Beispielen die Probleme auf, die mit traditionellen Techniken entstehen. Abschnitt drei stellt gbeta und den Familienpolymorphismus als Lösung dieser Probleme vor. Abschnitt 4 geht kurz auf das Typsystem von gbeta ein. Der fünfte Abschnitt geht auf verwandte Arbeiten ein und der letzte Abschnitt enthält Schlussfolgerungen.

2 Familien mit traditionellem Polymorphismus

In diesem Abschnitt wird anhand einer Beispielfamilie gezeigt, was mit traditionellem Polymorphismus möglich ist und was nicht. Vor allem wird aufgezeigt, daß man hierbei zu einer Wahl zwischen Typsicherheit und Flexibilität in Form von Wiederverwendbarkeit gezwungen ist.

Das Beispiel betrachtet Graphen, die als eine Menge von Knoten und eine Menge von Kanten gegeben sind. Kanten verbinden die Knoten zu Graphen. Es bilden also die Klassen `Node` und `Edge` die Familie *Graph*. Eine weitere Familie ist *OnOffGraph*, die aus `OnOffNode` und `OnOffEdge` besteht. Dabei erweitert ein *OnOffGraph* einen *Graph* so, daß eine vorhandene Kante auch ausgeschaltet werden kann, um z.B. ein Kommunikationsnetzwerk abzubilden, in dem einzelne Verbindungen auch mal abreißen können. Es kann auch noch viele andere Familien geben. Beispiele wären beschriftete Graphen, gerichtete Graphen, gewichtete Graphen usw.

```
class Node {
  boolean touches(Edge e) { return (this == e.n1) || (this==e.n2);  }
}

class Edge { Node n1, n2; }

class OnOffNode extends Node {
  boolean touches(Edge e) {
    return ((OnOffEdge)e).enabled? super.touches(e) : false;
  }
}

class OnOffEdge extends Edge {
  boolean enabled;
  OnOffEdge() { this.enabled = false; }
}

public class Main {
  static void build(Node n, Edge e, boolean b){
    e.n1 = e.n2 = n;
    if(b == n.touches(e)) System.out.println("OK");
    else System.out.println("not OK");
  }

  public static void main(String[] args){
    build(new Node(), new Edge(), true);
    build(new OnOffNode(), new OnOffEdge(), false);
    build(new OnOffNode(), new Edge(), true); // ClassCastException!
    build(new Node(), new OnOffEdge(), true); // "Works"
  }
}
```

Abbildung 1: Implementation in Java

2.1 Implementation in Java

Eine Implementation in Java ist in Abb. 1 zu sehen. Dieses Beispiel ist direkt aus dem Artikel [1] übernommen worden.

Es gibt eine Klasse `Node` mit einer Methode `touches` und eine Klasse `Edge`, welche schlicht zwei Referenzvariablen vom Typ `Node` hat. `touches` liefert einen `boolean`, welcher angibt, ob eine gegebene Kante mit diesem Knoten verbunden ist.

Des weiteren gibt es eine Klasse `OnOffEdge`, welche die Klasse `Edge` erweitert und einen `boolean enabled` erhält. Dieser wird im Beispiel der Einfachheit halber stets auf `false` gesetzt. `OnOffNode` erweitert `Node` und überschreibt die Methode `touches`, so daß der Rückgabewert nicht nur von der Struktur des Graphen abhängt, sondern auch davon, ob die Kante „aktiviert“ ist oder nicht. Um auf `enabled` von `OnOffEdge` zugreifen zu können ist hierbei ein Cast nötig.

Die Klasse `Main` benutzt diese zwei Familien. Sie hat eine Methode `build`, die einen Knoten `n` mit einer Kante `e` verbindet. (Der Einfachheit halber verbindet sie den Knoten mit sich selbst.) Sie vergleicht anschließend noch den Rückgabewert von `n.touches(e)` mit ihrem dritten Parameter, über den man den erwarteten Wert angeben kann.

Die `main`-Methode ruft nun `build` mit allen Kombinationen aus Knoten und Kanten auf. Die ersten beiden Aufrufe sind korrekt. Es wird jeweils ein Knoten und eine Kante aus der gleichen Familie gepaart. Beim dritten und vierten Aufruf von `build` werden jedoch die zwei Familien gemischt. Beide Aufrufe werden vom Typsystem als typkorrekt anerkannt, da eine Referenzvariable vom Typ `Node` auch auf eine Instanz vom Typ `OnOffNode` zeigen darf und gleiches analog für `Edge` und `OnOffEdge` gilt.

Der dritte Aufruf verursacht jedoch eine `ClassCastException` in der Methode `build` der Klasse `OnOffNode`, da sich eine Instanz der Klasse `Edge` nunmal nicht in eine Referenz der Klasse `OnOffEdge` umwandeln lässt und auch keine Instanzvariable `enabled` hat. Der vierte Aufruf verursacht zwar in diesem Beispiel keine direkten Laufzeitfehler, ist aber ebenfalls kritisch. In so einem Fall würden Fehler vielleicht erst beim Erweitern des Programms auftreten.

Schaut man sich die Methode `touches` von `OnOffNode` an, so sieht man, daß das Argument bei jedem Aufruf ein `OnOffEdge` sein muss. Also hätte man den Parameter eigentlich auf `OnOffEdge` setzen müssen. Dies wäre in Java dann aber keine Überschreibung der Methode mehr. (Sie würde statt dessen überladen.) Der Typ des Parameters lässt sich in Java beim Überschreiben von Methoden nicht ändern bzw. einschränken. Also lassen sich in Java zwar Klassenfamilien implizit definieren, aber den typsicheren Gebrauch kann man nicht statisch garantieren. Die Implementation ist also nicht typsicher.

Dafür lässt sich in dieser Implementation die Methode `build` mit jeder Familie benutzen. Sie hängt von keiner konkreten Familie ab. Sie würde sich sogar mit einer neuen Familie, z.B. `LabelledGraph` benutzen lassen, ohne neu compiliert zu werden.

In diesem Beispiel wurde also Flexibilität in Form von Wiederverwendbarkeit durch Vernachlässigung statischer Typsicherheit erreicht.

2.2 Implementation mit Generics

Für ein zweites Beispiel wurde in [1] die Sprache C++ mit Templates benutzt, da Java zu der Zeit noch keine Generics kannte. Inzwischen ist Java aber in der Version 1.5 (bzw. 5.0) erschienen und bietet Generezität durch Generics. Daher habe ich das Beispiel weitgehend identisch auf Java mit Generics übertragen.

In diesem Ansatz wurden die Familienmitglieder von Klassen abgeleitet, die eine Art „Vor-Familie“ bilden. Diese besteht aus den typparametrisierten Klassen `NodeF` und `EdgeF`. Beide sind rekursive über den Knotentyp N und den Kantentyp E der Familie parametrisiert, d.h. der Parameter N ist ein Subtyp von NodeF¡N,E¿ und der Parameter E ist ein Subtyp von EdegeF¡N,E¿. `NodeF<N,E>` besitzt eine Methode `touches`, die ein Argument vom Typ des Typparameters `E` entgegennimmt. Sie leistet ansonsten das gleiche wie in dem vorherigen Beispiel. `EdgeF<N,E>` besitzt hier wieder nur zwei Instanzvariablen vom Typ `NodeF<N,E>`. Die Familie *Graph*, bestehend aus `Node` und `Edge`, entsteht nun durch ableiten dieser zwei Klassen, wobei an die Typparameter der Vor-Familien jeweils `Node` und `Edge` übergeben wird.

Auch die Familie *OnOffGraph* hat eine Vor-Familie. Diese besteht aus den Klassen `OnOffNodeF<ON,OE>` und `OnOffEdgeF<ON,OE>`, welche wiederum von `NodeF<N,E>` und `EdgeF<N,E>` abgeleitet sind. Die Parametrisierung wirkt hier etwas unübersichtlich. Bei näherer Betrachtung sieht man aber, daß sie im Prinzip genau so aufgebaut ist wie bei `NodeF` und `EdgeF`. Sie wirkt nur kompliziert durch die längeren Namen und durch die Tatsache, daß die Klassen zusätzlich von parametrisierten Klassen abgeleitet werden.
`OnOffNodeF<ON,OE>` besitzt ebenfalls eine Methode `touches`. Diese überschreibt nun die Methode `touches` von `NodeF<N,E>`, obwohl sie ein Argument vom Typ `OE` entgegennimmt. Ein dynamischer Cast ist hier keiner nötig. `OnOffEdgeF<ON,OE>` erweitert `EdgeF<N,E>` analog zum vorherigen Beispiel um die Instanzvariable `enabled`, die auf `false` gesetzt wird. Die Familie *OnOffGraph* entsteht dann wieder analog zur Familie *Graph* durch ableiten der Vor-Klassenfamilie.

In der Main-Klasse gibt es nun analog zum Template-Ansatz aus [1] zwei Methoden `build`. Die eine nimmt als Argumente Klassen der Familie *Graph*, also `Node` und `Edge`, entgegen, die andere Klassen der Familie *OnOffGraph*. Ansonsten enthalten sie beide exakt den selben Quellcode, wie sie schon die Methode `build` als dem ersten Beispiel enthielt.

Analog zum vorherigen Beispiel gibt es eine `main`-Methode, welche die `build`-Methoden aufruft. Die ersten beiden Aufrufe mischen die Familien nicht und produzieren das gewünschte Ergebnis. Die nächsten beiden auskommentierten Aufrufe von `build` versuchen die Familien zu mischen. Dies wird jedoch zur Compilezeit vom Typsystem mit einem Typfehler quittiert.

Dieser Ansatz ist also bereits statisch vollkommen typsicher. Es sind zur Laufzeit keine Typüberprüfungen für dynamische Casts nötig. An der doppelt vorhandenen `build`-Methode erkennt man jedoch, daß dieser Ansatz die Wiederverwendbarkeit des Codes nicht bietet.

Es wurde also Typsicherheit auf Kosten der Wiederverwendbarkeit erreicht.

Einer der größeren Unterschiede zwischen diesem und dem Letzten Ansatz ist der, daß

```
class NodeF<N extends NodeF<N,E>, E extends EdgeF<N,E>> {
    public boolean touches(E e){
        return (this == e.n1) || (this == e.n2);
    }
}
class EdgeF<N extends NodeF<N,E>, E extends EdgeF<N,E>> {
    public N n1, n2;
}
class Node extends NodeF<Node, Edge> {}
class Edge extends EdgeF<Node, Edge> {}

class OnOffNodeF<ON extends OnOffNodeF<ON,OE>, OE extends OnOffEdgeF<ON,OE>>
                                                        extends NodeF<ON, OE> {
    public boolean touches(OE e){
        return e.enabled? super.touches(e) : false;
    }
}
class OnOffEdgeF<ON extends OnOffNodeF<ON,OE>, OE extends OnOffEdgeF<ON,OE>>
                                                         extends EdgeF<ON,OE> {
    boolean enabled;
    public OnOffEdgeF(){
        enabled = false;
    }
}
class OnOffNode extends OnOffNodeF<OnOffNode, OnOffEdge> {}
class OnOffEdge extends OnOffEdgeF<OnOffNode, OnOffEdge> {}

public class GenericsMain {
    static void build(Node n, Edge e, boolean b){
        e.n1 = e.n2 = n;
        if(b == n.touches(e)) System.out.println("OK");
        else System.err.println("nicht OK");
    }
    static void build(OnOffNode n, OnOffEdge e, boolean b){
        e.n1 = e.n2 = n;
        if(b == n.touches(e)) System.out.println("OK");
        else System.err.println("nicht OK");
    }
    public static void main(String[] args){
        build(new Node(), new Edge(), true);
        build(new OnOffNode(), new OnOffEdge(), false);
        //build(new OnOffNode(), new Edge(), false); // type error
        //build(new Node(), new OnOffEdge(), false); // type error
    }
}
```

Abbildung 2: Implementation in Java mit Generics

```
static <N extends NodeF<N,E>, E extends EdgeF<N,E>>
                  void build(N n, E e, boolean b) {
  e.n1 = e.n2 = n;
  if(b == n.touches(e)) System.out.println("OK");
  else System.err.println("nicht OK");
}
```

Abbildung 3: Parametrisierte build-Methode

hier `OnOffNode` *nicht* von `Node` abgeleitet wurde. Für das Typsystem besteht hier also keine Subtyprelation zwischen diesen Klassen. Dies verhindert das vermischen der Familien.

Parametrisierte build-Methode

In [1] wurde passend zu dem Ansatz in C++ mit Templates auch noch der Ansatz betrachtet die `build`-Funktionen durch eine Templatefunktion zu ersetzen. Da es mit Generics ebenfalls möglich ist parametrisierte Methoden zu erstellen habe ich dies ebenfalls untersucht. Ersetzten wir also mal in Abb. 2 die `build`-Methoden durch die aus Abb. 3.

Das Ergebnis ist überaschend gut. Die Typsicherheit des Programms bleibt erhalten und die parametrisierte `build`-Methode arbeitet wie gewünscht. Sie arbeitet auch mit einer beliebigen neuen Klassenfamilie aus Knoten und Kanten. Wir haben also ein statisch typsicheres Programm und können die `build`-Methode wiederverwenden.

Problematisch ist jedoch, daß in einer Aufrufskette, in der jede Methode die Klassen einer Familie weiterreichen möchte, genau so parametrisiert ist. Beispielsweise müsste eine Methode `prebuild`, die build aufrufen möchte, wie folgt aussehen:

```
static <N extends NodeF<N,E>, E extends EdgeF<N,E>>
       void prebuild(N n, E e, boolean b) { build(n,e,b); }
```

Dies wird bei Familien mit mehr Mitgliedern schnell unübersichtlich und stellt einen gewissen Verwaltungsaufwand dar, der erst recht zum Problem wird, wenn die Familien um ein neues Familienmitglied erweitert werden sollen. Denn das neue Mitglied muss als neuer Typparameter zu allen über die Familie parametrisierten Methoden (wie z.B. build und prebuild) hinzugefügt werden, selbst dann, wenn so eine Methode das neue Mitglied nicht braucht.

Vergleich mit dem Template-Ansatz aus [1]

Ein wesentlicher Unterschied zwischen Generics und Templates ist der, daß die Typüberprüfung für eine generische Javaklasse einmal als ganzes gemacht wird und vom Compiler nur einmal ein Bytecode erstellt wird. Bei der Templateinstanziierung wird hingegen macroähnlich jede Klasse eingesetzt, die Typüberprüfung von Anfang an neu durchgeführt und jedes mal neuer Code erstellt.

Im Vergleich mit dem Template-Ansatz aus [1] entstehen mit Generics weniger Probleme. Durch die macroähnliche Templateinstanziierung in C++ entstehen einige statische Abhängigkeiten. Auch sind virtuelle Templatemethoden in C++ nicht möglich. Dies stellt mit Generics kein Problem dar. In Java sind ja alle Methoden virtuell, und damit auch die typparametrisierten Methoden.

2.3 Umfang des Problems

Auf ein weiteres Problem gleicher Art stößt man, wenn man versucht Datenstrukturen zu erstellen, die nur Objekte einer bestimmten Familie aufnehmen sollen. Entweder man verzichtet auf Typsicherheit und kann die Datenstruktur für jede Familie nutzen, oder man fixiert die Datenstruktur auf eine bestimmte Familie.

Man beachte, daß der Ansatz aus C++ jede Template-Instanziierung separat zu prüfen, in gewisser Weise der Flexibilit t ist. Jede Einschränkung der Typparameter, die es ermöglicht den Typ einmalig als Ganzes zu überprüfen, wäre nur dann stark genug, wenn er die Implementierung auch wirklich typsicher macht. Für diese Fälle ist jedoch auch der C++ Ansatz erfolgreich. Eingeschränkte Typparameter können uns also nicht weiterbringen.

Daß man bei der Verwendung von Klassenfamilien zu einer Wahl zwischen Typsicherheit und Flexibilität gezwungen ist, ist ein Problem, das sich nicht nur auf Java und C++ beschränkt, sondern in vielen Sprachdesigns vorhanden ist. Die einzige gute bisher bekannte Lösung des Problems ist Familienpolymorphismus.

Ein Ansatz, der auf (evtl. eingeschränkter) Typparametrisierung beruht, gibt uns keine Möglichkeit Familienpolymorphismus zu erreichen. Entsteht eine Familie durch Ableiten der Mitglieder einer anderen Familie, so besteht hier stets die Gefahr die Familien zu vermischen. Um Familienpolymorphismus zu erreichen muss also eine andere Technik her.

Im folgenden Abschnitt wird Familienpolymorphismus vorgestellt, welcher in gbeta möglich gemacht wurde.

3 Familienpolymorphismus

Das Problem der bisher gezeigten Ansätze ist, daß die Familien nur implizit vorhanden sind, und nicht explizit angegeben wurden. Dies ist bisher auch nicht möglich. Würden wir jedoch Klassen als Attribute von Objekten auffassen, dann wird es möglich Objekte als Repository von Klassen zu betrachten. Diese Klassen bilden zusammen dann eine Familie. Durch späte Bindung dieser Klassenattribute wird es möglich mehrere Familien durch gewöhnliche Vererbung zu definieren. Über das umschließende Objekt, das Familienobjekt, kann dann auf die Klassen zugegriffen werden. Von jedem Familienobjekt ist statisch bekannt, daß es ein Repository von Klassen irgendeiner Variante einer Familie ist, die durch den statisch bekannten Typ des Familienobjekts gegeben ist. Es ist jedoch statisch nicht bekannt, welche Variante es ist.

Die eben beschriebene Idee wurde in der Programmiersprache gbeta umgesetzt. gbeta ist eine von BETA abgeleitete Sprache. Um das folgende Beispiel zu verstehen wird dem

Leser hier zunächst die Syntax und Konzepte der Sprache gbeta etwas näher gebracht.

3.1 gbeta

In gbeta (wie auch in BETA) wurde von den Konzepten wie Klassen und Methoden abstrahiert. Es gibt ein vereinheitlichtes Konstrukt, das sich *Pattern* nennt. So kann ein Pattern je nach Kontext bzw. Definition als Klasse oder als Methode verstanden werden. Vergleichbar zu Java, wo jedes Programm in eine Klasse gepackt wird, so wird in gbeta jedes Programm in ein Pattern gepackt. Das äußerste Pattern hat hierbei jedoch keinen Bezeichner. Ein Pattern definiert man mit `(# #)`, dabei ist `(#` als öffnende Klammer und `#)` als schließende Klammer zu verstehen. Ein so definiertes Pattern stellt bereits ein gbeta-Programm dar, das jedoch nichts tut. Ein Pattern kann Attribute mit Bezeichner haben.

```
(# foo: (# #) #)
```

Dieses Pattern besitzt ein Attribut `foo`, das ein (leeres) Pattern ist. Wie schon erwähnt trägt das äußerste Pattern keinen Bezeichner. Davon abgesehen definiert man ein Pattern stets durch einen Bezeichner gefolgt von einem Doppelpunkt gefolgt von einem Teil, der in [2] `<Merge>` genannt wird. Im Abschnitt 3.2 Abb. 4 ist dies meist eben eine solche Patterndefinition mit `(# #)`. Es kann aber auch beispielsweise `integer`, `real` oder `boolean` sein. Ebenso möglich sind dynamische Referenzen mit `^` und statische Referenzen mit `@`. Dynamische Referenzen sind mit Referenzvariablen in Java vergleichbar. Statische Referenzen verweisen konstant auf ein Objekt. Man kann sich also vorstellen, daß dies die Objekte selbst sind. Folgendes Beispiel verdeutlicht dieses Konzept:

```
(#
  foo: (# b: @boolean #);
bar: (#
    fo: ^foo;
    c: @foo
  #);
f: @foo
#)
```

Das äußerste Pattern hat drei Patternattribute: `foo`, `bar` und `f`. Das Patternattribut `foo` hat ein Attribut b, welches vom Typ boolean ist. `bar` hat zwei Attribute: `fo`, welches eine dynamische Referenz vom Typ `foo` ist und `c`, welches ein `foo`-Objekt ist. Und schließlich ist `f` ein Objekt vom Typ `foo`.

Wie schon erwähnt kann ein Pattern auch als Methode verstanden werden. Dafür gibt es in der Patterndefinition drei weitere Abschnitte, die mit Schlüsselwörtern eingeleitet werden. Mit `enter` beginnt der Abschnitt, in dem man definiert, welche der Attribute Parameter darstellen. Mit `do` beginnt der Abschnitt, in dem imperative Anweisungen stehen. Mit `exit` beginnt schließlich der Abschnitt, in dem man den „Rückgabewert“ angibt. So berechnet z.B. folgendes Pattern das Quadrat einer Zahl:

```
quad: (#
 r: @real;
 enter r
 exit r*r
#);
```

Methodenaufrufe und Zuweisungen werden in gbeta mit dem Pfeiloperator `->` gemacht. Dabei fließen die Daten von links nach rechts, also andersherum als in den meisten gängigen Programmiersprachen. Die Anweisung `(4)->quad->r` ruft also quad mit dem Argument `4` auf und speichert das Ergebnis in `r`

Es ist auch noch erwähnenswert, daß Pattern, die als Klassen aufgefasst werden sollen, den Abschnitt `exit this(Patternname)` haben sollten. Denn die Auswertung eines Patterns gibt nicht automatisch eine Referenz auf auf sich selbst zurück. Dafür ist der „`new`“-Operator in gbeta implizit. `f: @foo` erzeugt also automatisch eine neue Instanz des Pattern `foo`.

Für Faminienpolymorphismus sind vor allem die virtuell definierten Pattern wichtig. Virtuelle Pattern sind ein Konzept, das es in dieser Form in traditionellen objektorientierten Programmiersprachen wie Java und C++ nicht gibt. Es lässt sich teilweise mit virtuellen Methoden vergleichen.

Enthält ein Pattern `P` ein virtuelles Pattern `Q`
`P: (# Q:<(# i:@integer do 'Hallo\n' -> stdio #) #)`
so kann ein von `P` abgeleitetes Pattern `P2` die Definition des Pattern `Q` überschreiben:

```
P2 : P (#
  Q:: <!(# r:@real do 'Welt\n' -> stdio #)
#);
```

Eine dynamische Referenz `p: ^ P;` kann zur Laufzeit ein Objekt vom Typ `P` oder `P2` zugewiesen bekommen (`&P[]->p[];` oder `&P2[]->p[];`). Ein anschließender Aufruf von `p.Q;` arbeitet das Pattern Q des zur Laufzeit tatsächlichen Objekts ab. Dies ist soweit wie in Java oder C++ mit virtuellen Methoden. In gbeta kann ein Pattern jedoch auch als Klasse aufgefasst werden. So erzeugt z.B. `pq: @p.Q;` ein neues Objekt (`pq`), das je nach Typ des aktuellen Objekts `p` von `P.Q` oder `P2.Q` erzeugt wird.

gbeta ist eine komplette Programmiersprache und es gibt noch viel mehr das man erläutern könnte. Sie steht unter GPL und ist zusammen mit Tutorials und vielem mehr unter [2] erhältlich.

3.2 Implementation in gbeta

Die Implementation der Graphen in gbeta ist in Abb. 4 zu sehen. Als erstes fällt auf, daß die Familien jetzt explizit als Pattern `Graph` und `OnOffGraph` gegeben sind. (Auf ähnliche Weise hätte man im ersten Javabeispiel syntaktisch `Node` und `Edge` als innere Klassen definieren können. Dies wäre aber eine wirklich rein syntaktische Änderung gewesen und

```
(#  putline: (# enter stdio do '\n' -> stdio; #);

    Graph: (#
      Node: <(#
          touches: <(#
              e: ^Edge; b: @boolean;
            enter e[]
            do (this(Node)=e.n1) or (this(Node)=e.n2) -> b
            exit b
          #);
        exit this(Node)[]
      #);
      Edge: <(#
          n1: ^Node; n2: ^Node;
        exit this(Edge)[]
      #);
    #);

    OnOffGraph: Graph (#
        Node:: <(#
            touches:: <!(# do (if e.enabled then INNER if) #)
        #);
        Edge:: <(# enabled: @boolean; #);
    #);

    build: (#
        g: < @Graph; n: ^ g.Node; e: ^ g.Edge; b: @ boolean;
      enter (n[], e[], b)
      do
        n->e.n1[]->e.n2[];
        (if (e->n.touches)=b then 'OK'->putline else 'Nicht OK!' -> putline if)
    #);

    g1 : @Graph; g2 : @OnOffGraph;

  do
    (g1.Node, g1.Edge, true) -> build(#g::@g1#);
    (g2.Node, g2.Edge, false) -> build(#g::@g2#);
  (* type errors: *)
  (*(g1.Node, g2.Edge, false) -> build(#g::@g1#);*)
  (*(g1.Node, g2.Edge, false) -> build(#g::@g2#);*)
  (*(g2.Node, g1.Edge, false) -> build(#g::@g1#);*)
  (*(g2.Node, g1.Edge, false) -> build(#g::@g2#);*)
#)
```

Abbildung 4: Implementation in gbeta

hätte uns der Lösung nicht näher gebracht.) Das Pattern `Graph` hat zwei Attribute, welche virtuelle Pattern sind (späte Bindung), nämlich `Node` und `Edge`. `OnOffGraph` ist ein Subpattern von `Graph`. Dadurch hat ein `OnOffGraph` automatisch auch die Attribute `Node` und `Edge`.

Auf diese Weise bilden die zwei Pattern `Node` und `Edge` eine Familie. Ein (Familien) Objekt `g1` des Pattern `Graph` enthält also zusammengehörige Pattern („Klassen“), die eine Familie bilden, nämlich die Pattern `g1.Node` und `g1.Edge`.

Stellt `g2` ein anderes Objekt dar (z.B. vom Pattern `OnOffGraph`, aber auch von `Graph`), so betrachtet das Typsystem von gbeta die Pattern `g1.Node` und `g2.Node` nicht als verwandt. Dies gilt, solange nicht statisch bekannt ist, daß `g1` und `g2` exakt das gleiche Objekt darstellen. Auf diese Weise kann sichergestellt werden, daß Familien nicht vermischt werden.

Die Pattern in diesem Beispiel haben die gleichen Attribute wie die Klassen in den vorherigen. Es gibt ein Pattern `build`, welches als Methode aufzufassen ist. Sie wird im Beispiel zweimal aufgerufen. Einmal mit `Node` und `Edge` von der Familie `Graph` und einmal von der Familie `OnOffGraph`. Die im Beispiel auskommentierten Aufrufe werden vom Typsystem zurückgewiesen, da sie die Familien vermischen.

Beim Aufruf von `build` wird `g` auf ein festes (Familien-)Objekt gebunden. Es ist hierbei egal welche Instanz dies ist. Sie muss nur vom Typ oder Subtyp `Graph` sein. Dadurch müssen die Argumente `n` und `e` von der angegebenen Familie sein, denn `n` ist vom Typ `g.Node` und `e` ist vom Typ `g.Edge`. Ein Mischen wird so also ausgeschlossen. Das Familienobjekt muss dabei durch die gesamte Auswertung konstant sein. Würde man es als normales Attribut in Form einer dynamischen Referenz übergeben, dann würde es das gbeta-Typsystem als unsicher zurückweisen.

Durch die Tatsachen, daß die Familien hier mit Familienobjekten und nicht durch Klassen ausgedrückt werden und der Tatsache, daß die Familienobjekte untereinander eine Subtyprelation haben ist es möglich, das Pattern `build` mit einer unbegrenzten Anzahl von Familien zu benutzen ohne Gefahr zu laufen, daß diese vermischt werden. Man kann das Beispiel auch problemlos um ein weiteres Subpattern von `Graph` erweitern und die selbe Implementation des Pattern `build` weiterbenutzen. Und anders als im Beispiel mit Generics muss `build` nicht parametrisiert werden.

So ist Familienpolymorphismus erreicht.

Wie man Familienpolymorphismus auch für typsichere Datenstrukturen verwenden kann zeigt der folgende Abschnitt.

Datenstruktur

Eine Datenstruktur, die Knoten und Kanten aufnehmen kann, wäre z.B.

```
NodesAndEdges: (#
    g: < @Graph;
    nodes: @list(# element::g.Node #);
    edges: @list(# element::g.Edge #);
#);
```

Diese Datenstruktur enthält zwei Listen, die wiederum Knoten und Kanten einer bestimmten Familie enthalten, die durch `g` gegeben ist. Da `g` konstant und damit unveränderbar ist, ist sichergestellt, daß `g.Node` und `g.Edge` der selben Familie angehören.

Um diese Datenstruktur nun zu benutzen erstellt man zunächst einen Graphen, also ein Familienobjekt eines Pattern oder Subpattern von `Graph` und anschließend ein Objekt von `NodesAndEdges`, bei dem `g` auf das Familienobjekt gebunden wird:

```
myGraph : @OnOffGraph;
myNodesAndEdges: NodesAndEdges(# g::@myGraph #);
```

Nun kann man diese Datenstruktur beispielsweise an folgende Methode (die ja in gbeta ebenfalls ein Pattern ist) übergeben:

```
listBuild: (#
    ne: < @NodesAndEdges;
    n: ^ ne.g.Node;
    e: ^ ne.g.Edge;
  do ne.nodes.head-> n[];
     ne.edges.head-> e[];
     (n,e,true) -> build(#g::@ne.g#)
#)
```

Durch einen Aufruf wie `listBuild(# ne::@myNodesAndEdges #)` wird die Konstante `ne` auf `myNodesAndEdges` gebunden und gibt dadurch auf das konstante Familienobjekt Zugriff über `ne.g`. Dadurch ist sichergestellt, daß der Methode `build` beim Aufruf von `(n,e,true) -> build(#g::@ne.g#)` nur Knoten und Kanten der selben Familie übergeben werden. Dabei hätte eine Referenz auf `myNodesAndEdges` auch über mehrere Methoden weitergereicht werden können, um `ne` von `listBuild` letztlich konstant auf dieses Objekt zu binden:

```
m1: (# x:^NodesAndEdges enter x[] do listBild(#ne::@x#)#);
m2: (# x:^NodesAndEdges enter x[] do x[]->m1 #);
(* ... *)
```

Da es nicht nötig ist den exakten Typ der beteiligten Klassen (hier Pattern) statisch zu kennen, ist die Wiederverwendbarkeit der Methoden (hier ebenfalls Pattern) `build` und `listBuild` gegeben. Und gleichzeitig kann statisch zur Compilezeit die Typsicherheit garantiert werden. Es sind zur Laufzeit keinerlei dynamische Cast oder Typüberprüfungen nötig.

4 Typsystem von gbeta

In gbeta sind virtuelle Pattern Attribute von Objekten, nicht Attribute von Klassen. Daher sind virtuelle Typen keine Attribute von Typen. Dies wäre auch nicht so gut wie folgende Überlegung zeigt.

Nehmen wir an ein Typsystem einer Sprache mit virtuellen Attributen hätte folgende Eigenschaft: Sei x ein Objekt vom Typ T und V ein virtuelles Attribut von T mit dem Typ T_V, dann hätte $x.V$ den Typ $T.V$ und $T.V$ wäre ein existentieller Typ wie $\exists T'_V \leq T_V.T'_V$, das heißt ein Typ, der zu T gehört, aber nur durch seine obere Grenze T_V bekannt wäre. Bei zwei Objekten x und y vom Typ T hätten dann $x.V$ und $y.V$ den gleichen Typ. Daher wären sie zuweisungskompatibel. Ist y nun eine Instanz eines echten Subtyps von T, bei dem V ein echter Subtyp T_V^y von T_V ist, so wäre eine Zuweisung einer Referenz vom Typ T_V auf eine Variable vom Typ T_V^y zulässig. Dies ist jedoch typinkorrekt.

Würde man Zuweisungen dieser Art komplett verhindern, so könnte eine Methode `m1` die ein Argument vom Typ V nimmt keine weitere Methode `m2`, die ebenfalls ein Argument vom Typ V nimmt, aufrufen, selbst, wenn `m1` und `m2` in der selben Klasse sind. Damit wäre sinnvolles Programmieren mit virtuellen Typen unmöglich.

Weder BETA noch gbeta wählen einen dieser Möglichkeiten. Stattdessen hat ein Pattern Q, das innerhalb eines Pattern P definiert wurde, für jede Instanz von P einen anderen Typ, denn jede Instanz von P hat ein eigenes Pattern Q. Grundsätzlich haben zwei Objekte a und b also verschiedene Pattern $a.Q$ und $b.Q$. In der Praxis wird jedoch meist so Programmiert, daß durch statische Analyse nachweisbar ist, daß a und b das selbe Objekt ist. In diesem Fall gilt natürlich auch $a.Q = b.Q$.

Betrachten wir nochmal die Beispiele in gbeta (Abb. 4 und die Datenstruktur einschließlich des Pattern `listBuild` aus Abschnitt 3) und wenden dieses Wissen an. Wenn wir eine konstante unveränderbare Referenz zu einem Objekt haben (z.B. durch `g:<@Graph`), dann sind alle Referenzen zu virtuellen Attributen eines Objekts `g` (Attribute `g.Node` und `g.Edge`) virtuelle Typen von genau dieser konstanten Instanz. Das heißt `g.Node` und `g.Edge` sind virtuelle Typen von exakt der selben Instanz `g`.

Nur Referenzen, die den selben virtuellen Typ haben, also virtuelle Typen des selben Objekts haben, sind zuweisungskompatibel.

Z.B. ist `ne.nodes.head` in der Methode `listBuild` vom Typ `ne.g.Node`, genau wie das lokale Attribut `n` von `listBuild`. Also sind sie zuweisungskompatibel, obwohl wir nicht genau wissen von welchem Pattern `ne` erzeugt wurde. Ebenso kann man `n` durch die Bindung `g::@ne.g` beim Aufruf von `build` an diese Methode übergeben. Denn `build` möchte Argumente vom Typ `g.Node` und `n` ist vom Typ `ne.g.Node` und durch die Bindung ist sichergestellt, daß `g` und `ne.g` das selbe Objekt ist.

gbeta bietet also statische Typsicherheit. Es sollte jedoch erwähnt werden, daß virtuelle Pattern bzw Klassen bis vor kurzem noch nicht vollständig formalisiert wurden, und daß ein Korrektheitsbeweis ebenfalls noch aus stand. Dies wurde erst in [3] nachgeholt

5 Verwandte Arbeiten

Die Sprache gbeta ist eine generalisierte Version von BETA. Die Typsysteme sind sich sehr ähnlich, die Implementationen sind jedoch sehr unterschiedlich. Die statische Analyse virtueller Typen von BETA ist nicht zu Familienpolymorphismus in der Lage, da sie virtuelle Pattern zweier Objekte zu oft als identisch betrachtet.

In OCaml ist es möglich Familien von gegenseitig rekursiven Klassen zu definieren und Subfamilien durch Vererbung zu erstellen. Jedoch ist dies nur möglich, wenn die Typen der Familienmitglieder in einem Typüberprüfungskontext bekannt sind, also z.B. einer einzigen `let` Anweisung. Erstellt man zunächst ein Objekt eines Familienmitglieds und später ein Objekt eines anderen Mitglieds der selben Familie, dann müsste man die Typen explizit angeben. Zudem hat dieser Ansatz das gleiche Problem wie Ansätze mit parametrischem Polymorphismus.

Im Bereich der Funktionalen Sprachen gibt es viele Arbeiten rund um abhängige Typen. Ein abhängiger Typ ist ein Typ, der während der Ausführung von Laufzeitwerten abhängen darf. Er wird meistens dazu benutzt detaillierte Eigenschaften von Berechnungen zu beweisen. Das gbeta-Typsystem verwendet abhängige Typen insofern, daß Patterntypen den Laufzeitkontext, in dem sie definiert wurden, beinhalten. Für das Typsystem sind zwei Typen nur dann äquivalent, wenn sie mit dem selben Laufzeitkontext verbunden sind, und natürlich die gleichen Attribute mit den gleichen Typen haben. Um festzustellen, ob zwei Ausdrücke das selbe Objekt referenzieren wird keine Flußanalyse gemacht. Stattdessen wird nur überprüft, ob die zwei Ausdrücke über äquivalente Pfade von konstanten Referenzen auf das selbe Objekt zeigen. Dies hat sich in der Praxis bewährt.

6 Schlussfolgerungen

In dieser Ausarbeitung wurde aufgezeigt, daß es mit traditionellem Polymorphismus nicht möglich ist Familien von Klassen in einer typsicheren Art und Weise zu implementieren und gleichzeitig die Wiederverwendbarkeit des Quellcodes zu ermöglichen. Als Lösung zu diesem Problem wurde Familienpolymorphismus vorgestellt.

Virtuelle Pattern bieten in gbeta polymorphen Zugriff zu solchen Familien. Sie sind Typen, die von der Identität des umschließenden Objekts, dem Familienobjekt, abhängen. Das Familienobjekt stellt so ein Repository von Klassen einer Familie bereit. Dies löst die genannten Probleme mit minimalem Programmieraufwand, denn es ist nur nötig, das Familienobjekt mit den Instanzen der Familienmitglieder zu übergeben.

Familienpolymorphismus wird gebraucht um die Vorteile von traditionellem objektorientiertem Programmieren auch mit Klassenfamilien bieten zu können, die in Zukunft vermutlich immer wichtiger werden.

Literatur

[1] Erik Ernst (2001). Family Polymorphism In *Jørgen Lindskov Knudsen, editor, Proceedings ECOOP 2001, number 2072 in LNCS, pages 303-326, Heidelberg, Germany, 2001. Springer-Verlag.*

[2] Compiler, Tutorial: `http://www.daimi.au.dk/~eernst/gbeta/`

[3] Erik Ernst, Klaus Ostermann, and William R. Cook. *A Virtual Class Calculus.* Extended version of the POPL'06 paper with the same title.