Family Polymorphism

Erik Ernst

Department of Computer Science
University of Aalborg, Denmark
eernst@cs.auc.dk

Abstract. This paper takes polymorphism to the multi-object level. Traditional inheritance, polymorphism, and late binding interact nicely to provide both *flexibility* and *safety* – when a method is invoked on an object via a polymorphic reference, late binding ensures that we get the appropriate implementation of that method for the actual object. We are granted the flexibility of using different kinds of objects and different method implementations, and we are guaranteed the safety of the combination. Nested classes, polymorphism, and late binding of nested classes interact similarly to provide both safety and flexibility at the level of multi-object systems. We are granted the flexibility of using different *families* of kinds of objects, and we are guaranteed the safety of the combination. This paper highlights the inability of traditional polymorphism to handle multiple objects, and presents family polymorphism as a way to overcome this problem. Family polymorphism has been implemented in the programming language `gbeta`, a generalized version of BETA, and the source code of this implementation is available under GPL.[1]

1 Introduction

Imagine a hotel lobby with a few people standing around, waiting. The receptionist decides to get things going by asking a man "Are you a husband?" and asking a woman "Are you a wife?". Upon receiving two affirmative – though slightly baffled – answers, those two people are assigned to the same room, together with the little girl who said "Erm, yeah, and I'm a daughter!"

The reason why this might not be entirely appropriate is that those people may very well be 'husband', 'wife', and 'daughter', but it makes a big difference whether or not they play these roles in the *same family*.

Family polymorphism is a programming language feature that allows us to express and manage multi-object relations, thus ensuring both the flexibility of using any of an unbounded number of families, and the safety guarantee that families will not be mixed. It is, in a sense, a programming language feature that solves problems with the same structure as the hotel room assignment problem.

Traditional inheritance, polymorphism, and late binding of methods provide both flexibility and safety in the following sense. A polymorphic reference `x` may at run-time refer to an object which is an instance of some class C_i chosen from

[1] `http://www.cs.auc.dk/~eernst/gbeta/`.

J. Lindskov Knudsen (Ed.): ECOOP 2001, LNCS 2072, pp. 303–326, 2001.

a set of classes $\mathcal{C} = \{C_0 \ldots C_k\}$. We may invoke a method `m` on `x`, typically using syntax such as `x.m()`, and each of the classes may provide its own method implementation for `m` or inherit an implementation defined elsewhere. Late binding ensures that the chosen implementation of `m` is the one associated with C_i (if any), i.e., the appropriate implementation for the actual object. Static type checking may be used to ensure that there is indeed an implementation for every invocation.

All in all, this provides the flexibility of using several classes and several method implementations, and the safety of ensuring that the chosen method implementation is always appropriate for the actual object. It is important to note that the same call-site, `x.m()`, is *reused* with all those pairs consisting of a class and a method implementation; that it does not depend on the exact class of `x` or the exact choice of implementation of `m`; and moreover that the set of class/method pairs is open-ended.

Now consider the situation where two or more objects are involved, for instance where one object is given as an argument to a method on the other object, `x.m(y)`. In this case, traditional object-oriented languages such as the Java™ programming language [2] and C++ [25] will only allow us to associate two compile-time constant classes with this expression, namely the statically known class of `x`, $C_{\texttt{x}}$, and the statically known argument type of `m`, the class $C_{\texttt{m}}$. At run-time, `x` may refer to an instance of any subclass of $C_{\texttt{x}}$ and `y` may refer to an instance of any subclass of $C_{\texttt{m}}$. There is no way to ensure statically that a particular subclass $C'_{\texttt{x}}$ of $C_{\texttt{x}}$ is always paired up with a particular subclass $C'_{\texttt{m}}$ of $C_{\texttt{m}}$.

Note that multiple dispatch [5,10,23] solves a *different* problem: With multiple dispatch it is possible to choose a method implementation based on the actual classes of `x` and `y` – but we do not want to choose a method implementation for an arbitrary combination of classes, we want to ensure that the combinations of classes are *not* arbitrary.

The fact is that the traditional notion of polymorphism is unable to capture relations between several objects and their methods – it only handles the case with one object and its methods, and multi-object relations are always specified in terms of a fixed number of compile-time constant classes, i.e., essentially monomorphically. As we shall see in Sect. 3, this means that we must give up either flexibility or safety, we cannot have both at the same time.

We use the term *family polymorphism* to describe a generalized kind of polymorphism that will allow us to statically declare and manage relations between several classes polymorphically, i.e., in such a way that a given set of classes may be known to constitute a family – that family being characterized by having certain relations between its members – but it is not known statically exactly what classes they are.

The contributions of this work are the concept of family polymorphism, the underlying programming language mechanism, the associated static analysis techniques, and the implementation in a full-scale programming language, gbeta [13]. The notion of dependent types that makes family polymorphism pos-

sible has been present in some form in the BETA [19] community for many years (it is described informally in [19]), but the static analysis and the implementation have not been complete before gbeta.

The rest of this paper is structured as follows: Section 2 argues that the multi-class perspective is becoming more and more important. Section 3 describes the problems with current approaches in more detail, by means of a running example in Java and C++. The way these problems are solved with family polymorphism is described in Sect. 4. Finally, Sect. 5 covers related work, and Sect. 6 concludes.

2 We Need Class Families

Traditional object-orientation allows us to express a concept and several variations and/or implementations thereof by means of the class and inheritance mechanisms. However, there are many signs that this single-class perspective is becoming obsolete or at least insufficient.

A main motivational point of generative programming [12] and software product line approaches (e.g. [4]) is that modern software engineering must support variability at a more global scale than the individual class. This means that variants must be composed consistently across an application.

Languages and systems supporting advanced separation of concerns – such as aspect-oriented programming [14], composition filters [1], and multi-dimensional separation of concerns [26] – often emphasize the handling of cross-cutting concerns, i.e., issues involving more than one class. This means that they add support for the creation of class family variants.

Even in more traditional languages like Java and C++ it is possible to express class families, and the momentum behind the abovementioned research efforts supports the claim that the multi-class perspective cannot be ignored.

When a system contains more than one variant of a class family at the same time, it becomes necessary to maintain consistency in the usage of family members, i.e., to avoid mixing the families inappropriately. In this case it is not sufficient to be able to choose variants statically, there must also be support for management of multiple class family variants at run-time. As described in the next section, this causes a dilemma. Family polymorphism is a mechanism that can be used to resolve this dilemma.

3 Handling Graphs with Traditional Polymorphism

In this section we will present an example of a class family, and draw the attention to an unfortunate choice between safety and flexibility in reuse that we are forced to make. It is a property of the type systems of Java and C++ that we cannot have both safety and reuse flexibility at the same time, but this property is shared with more advanced type systems such as those of GJ [6] and Cecil [11,15]. We will return to this topic in Sect. 3.3 and 4.2.

Consider the concept of a *graph*, consisting of a set of *nodes* connected in some way by a set of *edges*. The graph concept plays the "organizing" role,

offering a common frame of reference under which the concept of node and the concept of edge make sense. Moreover, there are many different kinds of graphs – colored graphs, weighted graphs, labelled graphs, etc.

In this context we will concentrate on a simple `Graph` and an `OnOffGraph`. The latter adds support for switching each edge on and off, for instance to model communication networks where individual links may now and then be broken.

A `Graph` is not just one graph in the usual sense, it represents all nodes and edges of a particular *kind*, and these nodes and edges may then be organized and reorganized into any number of concrete graphs. In Sect. 4 we shall look at some data structures that may be used to hold the nodes and edges of a concrete graph.

3.1 The Naïve Approach

It is not hard to express graphs by means of two families of classes as described in the previous section. A definition of such class families in Java is given in Fig. 1. The first family consists of the classes `Node` and `Edge`, and the second family consists of the classes `OnOffNode` and `OnOffEdge`. A sample class `Main` contains code to show usage of these class families. It might seem more natural to express a class family by means of inner classes in Java, but since they would add syntactic complexity and would exhibit the same problems, we chose to avoid inner classes in this example.

The method `touches` on `Node` tests whether or not a given `Edge` is connected to the receiver `Node`. It would presumably be used to find paths through the graph. In a simple `Graph` the answer only depends on the graph structure, but in an `OnOffGraph` it also depends on the enabledness of the `Edge`.

The `main` method in `Main` invokes a method `build` three times, with different arguments. The method `build` expects to receive a `Node` and an `Edge`, both from the same class family. It then proceeds to connect the `Node` and the `Edge`, and finally invokes the method `touches` on the `Node` with the `Edge` as an argument. The third argument to `build` is a `boolean` which shows the expected result.

In the first two cases, `build` is used as it was intended, and it produces the expected result. However, in the third case we break the "rules" and invoke `build` with two objects from different class families. This causes a `ClassCastException` at run-time. The fourth case is confusing and probably unintended, but does not directly cause run-time errors.

The third invocation is type correct, since an `OnOffNode` is-a `Node` and an `Edge` is-an `Edge`. And if the `OnOffNode` had only been known statically as a `Node`, the failing third invocation of `build` would have looked just like the successful first invocation, according to the type system.

The problem is obviously that we have been unable to express the *actual* requirements. As we can see in the implementation of `touches` in the class `OnOffNode`, the argument of type `Edge` must really be an `OnOffEdge` – otherwise the dynamic cast will fail. Since method arguments in Java are in-variant, we must use `Edge` as the argument type and then use a dynamic cast in the method body. Of course, we may then invoke the method with an instance of `Edge` as

```
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");
  }
  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"
  }
}
```

Fig. 1. Reuse: Yes – Safety: No

argument, and the error will only be detected at run-time. The type system will not allow us to express the connection between the members of a class family, it will only allow us to create a type hole such that all *combinations* of members of these families, including the correct combinations, are allowed.

It may be argued that this is not a "type hole", it is a dynamic cast, and the people who use dynamic casts deserve what they get. The point is that the programmer is forced into writing a program with incomplete compile time type checking because the discipline which *should* be imposed on the choice of arguments cannot be expressed. So it is a type hole, even if it is one we have explicitly asked for.

Apart from this, the example exhibits the very nice property that the method `build` works both for a simple `Graph` and for an `OnOffGraph`. In other words, we are allowed to reuse the method `build` with several different class families, without any static dependency on the actual choice of family.

3.2 Working Out Safety

We have the option of shifting the trade-off in favor of safety, giving up on some reuse opportunities. An alternative expression of the class families is given in Fig. 2, and it is obviously a bit less straightforward than the previous version.

In this case we use the language C++, because Java does not (currently) support genericity and hence does not allow this kind of solution. For brevity, we use the keyword `struct` and not `class`, thus avoiding the need for accessibility declarations.

In this approach, we use type parameters to establish "pre-families", i.e., sets of type parameterized classes such that mutually recursive families of classes can be created by template instantiation, as with `Node` and `Edge`, and with `OnOffNode` and `OnOffEdge`.

In line with Fig. 1 there is a `main` function where the two class families are used, and the usage is expressed in two almost identical functions, `build1` and `build2`. These two functions have the same functionality as the method `build` in Fig. 1.

The difference between the situation in Fig. 1 and the situation in Fig. 2 is that the members of the class families are related in different ways according to the type systems.[2] In Fig. 1, `OnOffNode` is a subclass of `Node` and `OnOffEdge` is a subclass of `Edge`. This is not the case in Fig. 2. In other words, in the first figure the families are related by a memberwise subclass relation, and in the second figure the families consist of unrelated classes.

Since there is no relation between a member of one family and a member of another family, there is no danger of mixing members of different families. Hence, this closes the type hole – as we should also expect, given that the second example is expressed without dynamic casts. Statements mixing the two families, like the two function calls which are commented out in `main`, will cause the program to be rejected at compile time.

The result is that we have gained safety and lost reuse.

Note that we could also have achieved the same trade-off in Java by textually copying the inherited material from `Node` to `OnOffNode`, and from `Edge` to `OnOffEdge`, and then removing the '`extends`' clauses – except of course that textual copying creates maintenance and comprehension problems.

At this point, C++ programmers would immediately remark that the loss of reuse is a non-problem: We could simply change `build` into a template function with the argument types being template arguments. That would make it possible to write just one (template) function `build` with textually the same body as `build1` and `build2`. We could then invoke it in place of both `build1` and `build2` in `main`.

The reason why this is *not* a solution is that each call-site for this `build` template function would be associated with a compile-time fixed choice of family. E.g., the first call-site in `main` would then be an invocation of `build<Node,Edge>`,

[2] Since types and classes may be considered to coincide for the subset of Java and C++ that we are concerned with, we will sometimes use expressions such as 'the class `X`' where 'the type associated with the class `X`' would have been more precise.

```
template <class N, class E> struct NodeF;
template <class N, class E> struct EdgeF { N *n1,*n2; };
template <class N, class E> struct NodeF {
    virtual bool touches(E* e)
      { return (this==e->n1) || (this==e->n2); }
};

struct Edge;
struct Node: public NodeF<Node,Edge> {};
struct Edge: public EdgeF<Node,Edge> {};

template <class ON, class OE>
struct OnOffEdgeF: public EdgeF<ON,OE> {
    bool enabled;
    OnOffEdgeF(): enabled(false) {}
};
template <class ON, class OE>
struct OnOffNodeF: public NodeF<ON,OE> {
    bool touches(OE* e) {
        return e->enabled? NodeF<ON,OE>::touches(e) : false;
    }
};

struct OnOffEdge;
struct OnOffNode: public OnOffNodeF<OnOffNode,OnOffEdge> {};
struct OnOffEdge: public OnOffEdgeF<OnOffNode,OnOffEdge> {};

void build1(Node* n, Edge* e, bool b) {
    e->n1=e->n2=n;
    if (b == n->touches(e)) cout << "OK\n";
}

void build2(OnOffNode* n, OnOffEdge* e, bool b) {
    e->n1=e->n2=n;
    if (b == n->touches(e)) cout << "OK\n";
}

int main(int argc, char *argv[]) {
    build1(new Node(), new Edge(), true);
    build2(new OnOffNode(), new OnOffEdge(), false);
    // build1(new OnOffNode(), new Edge(), false); // type error
    // build2(new OnOffNode(), new Edge(), false); // type error
    return 0;
}
```

Fig. 2. Safety: Yes – Reuse: No

and the second call-site an invocation of `build<OnOffNode,OnOffEdge>`, even though there is no need to explicitly write the part in angle brackets.

In spite of the fact that the template function call would look very much like a function taking dynamically polymorphic arguments, the difference has far-reaching consequences:

1. First, whenever a node and an edge should be delivered to a template function such as `build` via a number of intermediate functions, every function in the entire call chain must be a template function, and the exact types of those objects must be known statically at the original call-site (either exactly `Node` and `Edge`, or exactly `OnOffNode` and `OnOffEdge`, never anything like "any pair of classes that makes up a consistent subfamily of `Graph`").
2. Second, a template function may be a member function, but it cannot be a virtual member function. This means that we must not only know the exact type of every node and edge everywhere, we must also know statically what methods implementations *of other classes* are being used on them.
3. Third, we cannot have lists, sets, hash tables, or other data structures containing nodes and edges belonging together. We can only have data structures containing members of one, statically selected and then fixed family of nodes and edges.
4. Finally, it is perfectly reasonable to assume that a sub-family of `Node` and `Edge` would provide an implementation of an interface specified by `Node` and `Edge`. When using this implementation sub-family in a large, complex system, the template based approach would make large parts of this complex system statically dependent on that sub-family, because all usages of nodes and edges would have to be performed in a context where the exact classes of the members of the sub-family are known statically. This would make the system as a whole more fragile, as would any forced dependency on implementation details.

In short, the lack of dynamic polymorphism in multi-object settings causes the same kinds of problems that would arise if we were to give up dynamic polymorphism in the well-known single-object setting.

3.3 The Scope of This Problem

If the problems outlined in the previous sections were specific for the Java and C++ language designs and well-known solutions were available elsewhere, the issue would not be of much interest. We will therefore argue that those problems arise in many different language designs, and no good solutions are known to us – apart from the family polymorphism which is the main topic of this paper.

An approach which is similar to the one taken in Fig. 2 can be used in other languages with support for genericity based on type parameterization. In the following we will consider the relation between these approaches.

Many different proposals have been made for the addition of genericity to Java based on parametric polymorphism [21,6,9,24, and others]. F-bounds [8]

make it possible to design the genericity mechanism in such a way that a type parameterized class may be type checked once and for all – as opposed to C++ templates where type checking must essentially be performed from scratch at each instantiation. Moreover, using a homogeneous translation scheme (as in GJ [6]) just one version of the code is generated for one generic entity, thus making it possible to support "virtual template methods" in Java (late binding is by default used for all methods in Java, so an ordinary Java method would correspond to a virtual member function in C++). This is the approach taken in GJ. Hence, the problem with virtual template member functions is less serious than the other problems – it is a consequence of the macro-like nature of the C++ template mechanism.

As described in [7], a somewhat more involved technique than the one used in Fig. 2 must be employed in order to express families of mutually recursive classes with genericity based on F-bounds, but it is still possible.

Note, however, that the C++ approach where each template instantiation is statically analyzed separately is in a sense the maximally flexible approach. Any kind of constraints that could be specified on type parameters of a generic entity in order to enable type checking of the entity as such (and not per instantiation) would only be able to reduce the flexibility, compared to the C++ approach. This is because constraints on the type arguments will only be sufficiently strict if every possible choice of type arguments will actually make the implementation type safe, and in those cases the C++ style per-instantiation checking would also succeed. In other words, there is no hope that constrained type arguments could afford us greater flexibility at instantiation sites than what we have already seen in C++.

On the other hand, it is possible in very advanced type systems such as the one used in Cecil [15] to explicitly declare that a given parameterized class is, e.g., co-variant in a given type argument and contra-variant in another type argument. The problem is, however, that this would not help us. For instance, `EdgeF<N,E>` is in-variant in `N`, and `NodeF<N,E>` is contra-variant in `E`. Hence, any attempt to declare that `NodeF<N,E>` and `EdgeF<N,E>` are co-variant in `N` and `E` would simply make their implementations type incorrect. So any approach based on (possibly constrained) type parameterization of individual classes and methods will not allow us to obtain polymorphism at the level of class families.

This should not be a surprise, since any mechanism in a type system that would establish a memberwise subtyping relation between the members of class families would also allow us to mix classes from different families, as it was done in Fig. 1, in the last invocation of `build`. In other words, if we could do such a thing, the type system in question would be unsound.

In summary, the approach taken in Fig. 2 can be used in other languages with genericity mechanisms based on type parameterization, but it does not solve the problems associated with: excessive propagation of templatization; the lack of type safe data structures for class family member instances (except for data structures statically bound to *one* particular family); the widespread propaga-

tion of static dependencies on implementation details; and the lack of dynamic polymorphism.

Hence, in order to achieve a safe and flexible mechanism, we must strive for something other than memberwise subtyping. In the next section we shall see how the notion of classes as attributes makes it possible to establish a safe and useful kind of family polymorphism.

4 Handling Graphs with Family Polymorphism

The main problem in the approaches considered so far is that the family itself is not represented explicitly. As long as the family is only implicitly present, it is hard to conceive of any other kind of polymorphism for families of classes than the one based on a memberwise subtype relationship.

However, if we introduce the notion of classes as attributes of objects then it is suddenly possible to use an object as a repository of classes – a class family. If we moreover introduce the notion of late binding of such class attributes then it becomes possible to specify a number of families of classes by means of an ordinary inheritance network describing variants of the enclosing object, the *family object*. For each such family object it is statically known that it is a repository for *some* variant of the class family declared in the statically known type of the family object, but it is not statically known *which* class family it is.

This is the approach taken in `gbeta`. The `gbeta` type system is consistent with the type system design for BETA[3] that is described informally in [19], but it is stricter than the actual implementation of type checking in the Mjølner implementation of BETA [29]. In the languages `gbeta` and BETA, classes and methods (and more) have been unified into the single abstraction called a *pattern*. This means that we may use words like 'class' and 'method', but the denoted entities will in both cases be patterns, so these words are simply synonyms for the word 'pattern' with an added hint to the reader about how to understand the role played by that pattern in the given context. Consequently, class attributes are really pattern attributes and late binding of class attributes is late binding of pattern attributes, normally designated as *virtual patterns*.

To make this concrete, we will present and discuss a version of our class family example written in `gbeta`, as shown in Fig. 3.

In the `gbeta` version of the class family example, the two class families are declared explicitly as the pattern `Graph` and the subpattern (think 'subclass') `OnOffGraph`. Each instance of `Graph` or a subpattern of `Graph` will have two attributes named `Node` and `Edge`. These two attributes will be patterns ('classes'), and they are known to belong together, forming a family of mutually recursive patterns ('classes'). That is, an object `myGraph` is known to contain a class family whose members are accessible as `myGraph.Node` and `myGraph.Edge`, respectively.

As we shall see below, the type system does *not* allow us to mix members of different class families – in other words, when `myGraph` and `yourGraph` are

[3] The `gbeta` type system is considerably more expressive than the BETA type system, but the BETA type system comes out as a special case.

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

Fig. 3. Reuse: Yes – Safety: Yes

not statically known to be the *exact same object*, the patterns `myGraph.Node` and `yourGraph.Node` are considered to be unrelated (unless of course they are statically known, e.g., because of a type exact reference to the enclosing object, and those statically known patterns are related). Also note that the type system will distinguish between an unbounded number of class families because they are associated with *instances* (e.g., `myGraph`) and not with *classes* (e.g., `Graph`). If they had been associated with classes then the type system would at most have been able to distinguish between a fixed number of families, determined at compile-time – increasing the danger that objects could be mixed inappropriately, because conceptually separate families would have to be treated as one family by the type system.

To continue with the example, `Node` and `Edge` are specified with the same attributes as they were in Fig. 1 and Fig. 2. The further-binding of `Node` and `Edge` in `OnOffGraph`, corresponding to the classes `OnOffNode` and `OnOffEdge`, are

also incrementally specified in a similar manner as previously. The expressions `exit this(···)[]` specify that the result of evaluating a `Node` or an `Edge` is a reference to that object itself (in BETA and `gbeta` the evaluation semantics of a class must be specified explicitly).

Finally, a method `build` is defined, one instance of each kind of graph is declared, and `build` is invoked twice, once with members of the `Graph` family and once with members of the `OnOffGraph` family. The two last statements are commented out; they demonstrate mixing of families, and if they are included the type system detects that they are not type safe.

We have to clarify a few points about the example. First, argument passing to methods and functions, assignment, and other evaluations are expressed in BETA and `gbeta` with the '`->`' operator, and the direction of the dataflow is left-to-right (where most other languages employ a right-to-left direction, opposite to the reading direction). It might help to think of the '`->`' as similar to the pipe symbol used on the command line in many operating systems.

Second, BETA and `gbeta` provide a kind of *transparency*: it is invisible in many places whether a result is stored or computed. Thus, `g1.Node` denotes a pattern, but when it is used in an evaluation context it gives rise to an object instantiation, and the new object is the result of the expression; in other words, a '`new`' operator is implicitly added to the expression.

Third, `build` accepts four arguments, namely `g`, `n`, `e`, and `b`; `n` and `e` are received by reference, `b` is received by value, and `g` is a constant attribute of each invocation of `build`.

There are many reasons why the different argument modes are specified syntactically the way they are (some of them historic), but for the purposes of this discussion we will just mention that a syntactic form like the following might work better to communicate the actual semantics of the invocations of `build`; note that this is for illustration, it is not valid `gbeta` syntax:

```
build(g1, new g1.Node(), new g1.Edge(), true);
build(g2, new g2.Node(), new g2.Edge(), false);
```

It is essential to ensure that the first argument to `build` (`g1` and `g2`, respectively) is constant throughout the evaluation of the arguments and the execution of the method. Only then is it known for sure that we are not mixing different families. If we were to provide this argument as an ordinary (assignable) by-reference argument, then the `gbeta` type analysis would not accept the implementation of `build` as type safe.

On the other hand, it makes no difference whether the graph given as an argument to `build` is an instance of `Graph`, of `OnOffGraph`, or of any other subpattern of `Graph`. We just need to know that it is *some* kind of a repository for a family consisting of `Node` and `Edge`, i.e., that it is an instance of a pattern that is less than or equal to `Graph`. This means that `build` can be reused with an unbounded number of different subfamilies of `Graph`, and it means that each invocation is guaranteed to not mix up different families. That amounts to the conclusion that the class family example has now been expressed with both safety and reuse opportunities preserved.

4.1 Revisiting the Problems

Let us reconsider the issues described near the end of Sect. 3.2, associated with the template method based approach:

1. Type checking with family polymorphism is based on an ordinary subtype constraint on the family object, so there is no need for exact static knowledge about any of the involved classes. The *relations* between the involved classes must be captured, but that may be expressed by means of the identity of the family object.
2. There are no special considerations about the methods of other classes – `build` could as well have been a virtual method. As mentioned, this problem can also be solved in other ways.
3. Data structures may be constructed to hold nodes and edges from a family whose family object is an instance of an arbitrary (not statically known) subpattern of `Graph`. Such data structures are 'family polymorphic'.
4. Since it is easy to hide the actual class of the family object by ordinary dynamic polymorphism, there is no need to propagate static knowledge about every subfamily of `Graph` to all usage points in a large system.

For instance, if we wish to operate on a list of nodes and a list of edges belonging together in the same subfamily of `Graph`, then we may use the following data structure:

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

This pattern is parameterized by the immutable object reference `g`, and it contains the list `nodes` with elements of type `g.Node`, and the list `edges` with elements of type `g.Edge`. In essence, it is a package containing two lists holding instances of members of a class family.

We can create a subpattern of this data structure to hold some nodes and edges belonging to a family object `myGraph` which is an instance of a subpattern of `Graph`, say `LabelledGraph`:

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

This declares `myNodesAndEdges` to be an object which is an instance of a subpattern of `NodesAndEdges` where the attribute `g` is immutably bound to `myGraph`. At this point we can pass `myNodesAndEdges` as an argument to methods such as this one:

```
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#)
#)
```

This method receives `ne` as a constant argument and thereby provides access to a class family object – namely `ne.g` – and a list of nodes belonging to that family – `ne.nodes` – and finally a list of edges belonging to the same family – `ne.edges`. The method starts by calling `head` twice, extracting the first element of the two lists (and omitting the check for an empty list...) and then invokes `build`. Note that we could have threaded `ne` through any number of method invocations as an ordinary by-reference argument, known only as an instance of a pattern that is less than or equal to `NodesAndEdges`. For example:

```
m1: (# x:^NodesAndEdges enter x[] do listBuild(# ne::@x #)#);
m2: (# x:^NodesAndEdges enter x[] do x[]->m1 #);
m3: (# x:^NodesAndEdges enter x[] do x[]->m2 #)
```

In this example, `m3` calls `m2` which calls `m1`, each time passing `x` – known as an instance of `NodesAndEdges` or a subpattern – to the next method. None of these methods depend on the exact classes in the class family and, of course, neither does `listBuild` nor `build`. We could invoke it with `ne[]->m3`, where `ne` is any reference whose declared type is `NodesAndEdges` or a subpattern thereof, e.g., `myNodesAndEdges[]->m3`.

This shows that we can package and re-package a family of classes and some instances of those classes, and we can statically ensure that the classes belong to the same family and the objects belong to the classes – without knowing statically *what* classes the family contains.

4.2 Revisiting the Alternative

In an approach based on parametric polymorphism, i.e., type parameterization, type safety in the management of class families is achieved by avoiding subtyping relationships between families. This implies that every individual piece of code dealing with a class family is either monomorphic (statically tied to one particular class family) or it is inside a generic entity with the family members as type parameters. In the first case, reuse opportunities are obviously lost. Let us consider the second case more closely.

Any execution of code inside a type parameterized entity corresponds to a ground instantiation of that entity – a direct or indirect instantiation having actual type parameters all of which are types not containing type variables. This is enforced by the design of such type parameterization mechanisms: (1) a parameterized class is not a class, it is a function from types to classes, and it is only possible to create objects as instances of classes, such as a parameterized class *applied to some type arguments*; (2) a type parameterized method may be called from another type parameterized method, but the call stack has finite depth and it does not start with a type parameterized method, so at some point

the type parameters to the method are received from some other source than the caller-method, i.e., as ground types or depending on type parameters of an enclosing parameterized class – which brings us back to the first case. Note that if the types are type variables, they must be the type parameters of the *same* enclosing generic entity for *all* members of the class family; otherwise they cannot be mutually recursive.

This strict discipline is necessary for the soundness of the static analysis; if it were possible to have a mutable entity (an object) at run-time which is parametrically polymorphic (i.e., an instance of a type parameterized class which has *not* received all of its type arguments as ground types), then it would be possible to interpret the "free type variable" differently at different times and thereby destroy the overall type correctness of the program. This is well-known from functional languages with mutable references, such as Standard ML [20] and Caml [30].

This means that *every* run-time call-chain of methods passing instances of members of a class family as arguments or looking them up in their receiver object includes a call-site which is monomorphic in the class family, and any method which is type parameterized by the family is eventually called from such a monomorphic site. In other words, a call chain can only access a class family polymorphically after a certain point where the access is monomorphic.

Now compare this to the well-known case of traditional dynamic polymorphism used with single objects (not families). Consider for example the case where we have an inheritance hierarchy rooted in `GraphicalObject`, containing subclasses such as `Circle` and `Rectangle`, and supporting a (virtual) method `draw`. With this design it is possible to create a number of instances of various subclasses of `GraphicalObject`, and to store them all in a `List` whose elements are typed as `GraphicalObject`. Now we may traverse the list and execute `draw` on each element. Note that the call-stack in this case does *not* include a monomorphic access before the polymorphic access. In fact, there may not exist *any* pointers typed by the actual class to each object in the list in the entire program execution state (a `this` pointer typed by the actual class may be created *later*, in the execution of the `draw` method).

This makes a big difference.

The big difference is not unlike the effects of manual memory management – it is a global phenomemon. In systems without garbage collection, it is necessary to design intricate, global management schemes such that the following question can be answered correctly at certain points: "Is it possible that there exists another live pointer to this object?" If the answer is incorrectly "No!" there will be dangling pointers, and if the answer is incorrectly "Yes!" there may be memory leaks. In a similar fashion, to be able to perform an operation on a group of objects which are instances of some members of a class family, it is necessary to design management schemes to ensure that there is at least one monomorphic pointer to each of those objects somewhere in the system, and we must be able to find that pointer in order to initiate a (possibly parametrically polymorphic) call-chain that will perform the operation.

In the single-object case, we can collect `GraphicalObject`s in a polymorphic data structure and then forget about their precise classes, and the definition and usage of the data structure is strictly isolated from static dependencies on the individual subclasses such as `Circle` etc.

But in the multi-object case, we cannot create a similar polymorphic collection of nodes and edges and perform operations on them without creating dependencies on their actual classes. This means that we will have to change our collection every time we want to put objects from a new sub-family into it.

One possible approach would be to use wrapper classes like `NodesAndEdges` – the difference is that, with parametric polymorphism, creation of these objects and insertion of nodes and edges would have to happen monomorphically. We could then have lists of these wrapper objects etc. However, it would be necessary to *rediscover* the exact actual subclass of `NodesAndEdges` for each wrapper in such a list we intend to use, because the contained nodes and edges can *only* be made accessible with monomorphic access. The rediscovery could be achieved with `instanceof` or similar means, but the rediscovery site would depend specifically on each class family that it is capable of rediscovering. Add a new subfamily, and this piece of source code must be changed.

Hence, even though there seems to be only a subtle difference between the approach based on parametric polymorphism and the approach based on family polymorphism, we claim that the difference has far-reaching consequences, especially for large scale systems where the propagation of static dependencies have the most devastating effects.

As mentioned, in the approach based on family polymorphism we exploit the features of *virtual patterns* in `gbeta`, which are a generalization of virtual patterns in Beta [18,19]. The next section discusses some properties of the underlying type system.

4.3 Aspects of the gbeta Static Analysis

It has been claimed that virtual types are inherently not type safe [7]. The reason why this opinion has emerged is probably that the community behind virtual patterns has not expressed with sufficient clarity that virtual patterns are attributes of *objects*, not attributes of classes. Consequently, virtual types are not attributes of types. In particular, this point was not emphasized in [27], where a design of virtual types in Java is proposed, inspired by the notion of virtual patterns in Beta. Also, virtual patterns may have been confused with unchecked covariance. However, virtual patterns have a kind of existential type, so potential covariance – in the type of a method argument, say – is always known statically, at all call-sites.

Let us briefly outline why it would be unsound to let virtual types be attributes of types. Assume that a type system for a language with virtual attributes (be it virtual classes or virtual patterns) would have the following property: If an object x is known to have type T and V is a virtual attribute associated with T and declared to have type T_V, then $x.V$ has the type $T.V$; $T.V$ would be an existential type such as $\exists\, T'_V \leq T_V . T'_V$, i.e., a type T'_V that is a characteristic

of T, but only known by its upper bound T_V. If this type T'_V is assumed to be a property of the type of the enclosing object, T, then two different objects x and y both having type T would have the *same* virtual type, i.e., $x.V$ and $y.V$ would have the same type. That would obviously be unsound in a type system with subsumption, since x could be an instance of a class having most specific type T, and y could be an instance of a subclass whose virtual V could be furtherbound to a strict subtype T_V^y of T_V. An assignment from a reference $x.r$ to a reference $y.r$ referring to the same declaration of r, having the type of V, would then be an assignment from a reference of type T_V to a reference of type T_V^y (a strict subtype of T_V), i.e., the assignment would be type incorrect – but such a type system would consider it to be an assignment between references having the *same* type.

Conversely, if a virtual V declared in a class having type T should be an existential type $\exists\, T'_V \leq T_V.\, T'_V$ that is treated is such a way in the type analysis that *no* assignments between references of type $T.V$ were allowed – thus avoiding the abovementioned type hole – then it would be impossible to write useful implementations involving virtual types. For instance, a method accepting an argument of type V would not be able to invoke another method accepting an argument of type V as an invocation on the current receiver object (a "self-send").

Of course, neither of these approaches is used in gbeta. In fact, as it was already stated very clearly for BETA in [19, p. 133], a pattern declaration Q inside another pattern declaration P declares a distinct Q pattern for *each instance* of P. This means that the static analysis of BETA and gbeta must consider pattern attributes, including virtual pattern attributes, as having composite types, consisting of two kinds of information. The space constraints do not permit a detailed description of the gbeta type system here; please refer to Chap. 13 and App. E of [13] for more details. We will however extract some salient features of this type system, in order to support the claims made about its properties.

The first kind of information in a gbeta type is the usual kind of static representation of object types: maps from names to types, indicating that any instance having the given type will have attributes with some specified names having specified types. The second kind of information is a relative representation of an enclosing object of a pattern, represented as a path leading from the current object to that enclosing object of the pattern. Moreover, every gbeta type is characterized as being exact, or known by upper bound only, or known by upper and lower bound. Types which are known by upper bound could be characterized as existential types, but it should be noted that they are also dependent types, depending on their enclosing objects.

We should mention that if Q is a pattern attribute of two objects a and b, it is often the case that $a.Q$ and $b.Q$ are indeed statically known to be the same pattern – gbeta and BETA would hardly be practically useful otherwise. But $a.Q$ and $b.Q$ generally cannot be assumed to be the same pattern if any of them are only known by upper bound, not even if a and b are known to be instances of the same pattern.

Both patterns and objects have types in gbeta. Two pieces of syntax denoting patterns have different types if they are not known to be associated with the exact same maps from names to types *and* the same enclosing objects, and two pieces of syntax denoting objects have different types unless they are guaranteed to denote the exact same object at run-time. It is *not* sufficient to know that two objects are exactly an instance of the same pattern, they would still have different types if they might be different objects.

To put this into context of the examples given above, the virtuals $x.V$ and $y.V$ discussed above would be known to have certain attributes (declared in the statically known maps from names to types), and they would moreover be known to be the V virtual of exactly the object denoted by x and the object denoted by y, respectively. In the (typical) case where x and y are not guaranteed to be the exact same object, $x.V$ and $y.V$ will generally have assignment incompatible types – no subtyping relation exists between them, they are just possibly different.

Note that this means that a virtual pattern known only by upper bound which is reached via a mutable reference is "not even equal to itself"; for instance, if z is a mutable reference then two different occurrences of z may refer to two different objects – not even flow analysis could have guaranteed that no assignments to z will happen between two usages of z, because there could be other threads having access to the current object.

In practical BETA and gbeta programming it is very often the case that a virtual pattern occurs as an attribute of an object that is accessed via an immutable reference. As described in [28], virtual types can be changed into ordinary types (whose structure is known completely at compile-time) by means of so-called final bindings. This is possible in BETA and gbeta, but an immutable reference to the enclosing object is an equally valid and more common way to make references with virtual types assignable. Note that the approach based on an immutable reference works both when the virtual in question is known exactly and when it is known only by upper bound. Actually, an example of the latter is the `element` types of the `list`s in `NodesAndEdges`.

A special case is the source code in a pattern declaration P containing a virtual pattern declaration V, i.e., the code executed in a context where V is an attribute of an enclosing object (think: V is an attribute of '`this`'). An enclosing object is accessed via an immutable reference, usually implicit at the source code level, but available as `this(X)` for an appropriate identifier `X`. This means that the name V used on its own has a type that is the same everywhere in the declaration of P.[4] This in turn means that it is both dynamically safe and recognized as type safe by the static analysis to assign between different references having the type of V.

Hence, a virtual attribute V of a pattern P can inside P be treated in much the same way as a constrained type argument inside a type parameterized class: The statically known upper bound of the virtual yields a certain available interface and allows for assignment to all non-existentially typed references having

[4] For those who know that this isn't quite true: In the enclosing `MainPart`.

supertypes of the upper bound of the virtual, and the virtual is known to be "equal to itself" such that assignments between references with the type of V are also allowed. This makes it safe and convenient to program patterns containing virtuals.

Finally, we can apply this knowledge about the typing of `gbeta` in general and `gbeta` virtuals in particular to the example shown in Fig. 3 and the method `listBuild` shown near the end of Sect. 4.1. Whenever an immutable reference to an object is established (e.g., with a constant argument like `g:<@Graph`), all references to virtual attributes in that object are then known to be the virtuals of exactly that object. This means that references declared to have the same virtual type, i.e., the type of the same virtual pattern, are assignment compatible. For instance, the elements of `ne.nodes` in the method `listBuild` are known to have the type of `ne.g.Node`, exactly like the local attribute `n` of `listBuild`. Hence, it is safe to assign an element from `ne.nodes` to `n`, even though we have no static knowledge about the exact pattern of which `ne` is an instance. Similarly, `n` may safely be given to `build` as an argument, because that argument is declared to have type `g.Node` – and `g` is known to be the same object as `ne.g`, because of the binding `g::@ne.g` in the invocation of `build`.

In this description we have used the term 'type' to denote the knowledge established by static analysis about each of the entities – patterns and objects – accessible in the run-time environment (patterns are, at least conceptually, available at run-time).

In particular, the type of a virtual pattern is a compile-time description that restricts the possible actual patterns denoted by a given virtual attribute to a well-defined (but generally unbounded) set of patterns. This description is parameterized by a run-time context; in other words, it is a function that maps a run-time context into a run-time entity, in this case a pattern.

From this notion of the type of a virtual pattern it might be possible to derive a notion of virtual types, defined without referring to virtual patterns or similar concepts. There is an ongoing debate as to whether 'virtual X' should be 'virtual types' or 'virtual classes', also touched upon in [7]. The approach taken in `gbeta` is a kind of 'virtual classes' approach, because patterns may (also) be considered as classes.

The main difference between virtual patterns and (pattern-less) virtual types, considered from a practical point of view, would be that virtual types can not be used to create new instances, whereas virtual classes/patterns can be used just like other classes/patterns to create objects. As a result it is, e.g., possible to create nodes and edges in a given subfamily of `Graph`, and to compose them into a graph, again without having any static dependency links between the graph creation code and the exact `Graph` subfamily being used. It is our experience that the constructive use of virtual patterns is extremely useful. It is also yet another example of a situation where it is possible to use (in this case enlarge or create) a `Graph` without creating monomorphic dependencies; with an approach based on type parameterization or even virtual types, it would be necessary to

refer to the exact classes of one particular class family in order to create new nodes and edges.

5 Related Work

The language gbeta has been developed as a generalized version of BETA, so the design of BETA is an immensely important starting point for gbeta, and the community around BETA has provided lots of valuable feed-back. Moreover, as mentioned in Sect. 4, the informal understanding of types in BETA as described in [19] matches the actual type system of gbeta quite well, apart from the fact that the basic concepts are more general in gbeta. However, the implementation of gbeta is very different from the implementation of BETA. In particular, the static analysis of virtual patterns in BETA – as described in [17] – does actually not suffice to handle family polymorphism correctly. The problem is that this static analysis in too many cases considers a virtual pattern in two different objects to be the same pattern. Even though the author had used BETA for years at this point, this problem with the static analysis of BETA only became apparent after a close inspection of [17]. This underscores the importance of formalizing the semantics and static analysis – a task which has unfortunately not yet been completed. However, the gbeta static analysis is the first one to solve this problem in the static analysis of BETA, and moreover it handles the added generality of gbeta.

In Sect. 3.3 and 4.2 it has already been discussed in what ways and to what extent parametric genericity can provide both type safety and reuse opportunities with class families. Our general conclusion is that either safety or reuse opportunities must be compromised, and in particular the almost-solution based on type parameterized methods will cause widespread static dependencies on the exact class families being managed. We should mention that the proposal in [7] is based on having type exact references to the members of a class family, thus making family polymorphism impossible at the outset.

In [22] it is described how families of mutually recursive classes may be expressed in OCaml, and how subfamilies may be created by inheritance. The structural type equivalence and the sophisticated support for type inference in this language makes it possible to decouple the classes in families, and in some cases to avoid the heavy notation for type arguments associated with some other approaches based on parametric polymorphism. However, this is only possible when the types of the members of the family are known in one type checking context, such as a single `let` statement. If we were to create one member of a family and store it in a variable and later create another member of that family, the types would have to be expressed explicitly. Moreover, this approach has the same problem as all the other approaches based on parametric polymorphism, namely that there must be a monomorphic call site on the call stack whenever a polymorphic piece of code is working with members of any class family.

In the area of functional languages there is a large body of work concerned with dependent types (see, e.g. [31]). A dependent type is a type that is allowed

to depend on run-time values in program executions, and it is typically used to express and prove detailed properties of the outcome of computations, such as "`reverse` is a function that accepts an argument of type `'a list(n)` and returns a result of type `'a list(n)`", meaning that it returns a list of the same type *and length* (`n`) as the argument. Often, dependent types are made less useful because support for general usage of program values in types makes type checking undecidable (as in Cayenne [3]), and often it is required that programmers provide proofs manually, using some kind of theorem prover.

The gbeta type system has not yet been proved correct, but the implementation certainly does not require manual intervention. This type system employs dependent types in that it is part of each pattern type that this pattern is defined in one particular run-time context, and the type system only accepts two pattern types as being equivalent if they are associated with the same run-time context, in addition to having the same attributes with the same types, of course. No flow analysis is made to discover what expressions will denote the same object – we do not consider flow analysis to be an acceptable tool as part of type checking – so object 'sameness' is only detected in the case where the object is accessed via equivalent paths of immutable references. This approach seems to work very well in practice, so there are no immediate plans to extend the analysis in order to discover further occurrences of object sameness.

Finally, it is instructive to compare the usage of objects in gbeta as class repositories with the usage of structures, signatures, and functors in SML [16] to provide packages of types and values. A *structure* in SML is a package of types and values which may be created at top-level and referred to by means of structure names (they are not first class values). A *signature* is a structure specification, listing required names and kinds of types, and names and types of values. By applying a signature to a structure, it may be ensured that the structure conforms to the given specification, and all parts of the structure not specified in the signature will thereafter be invisible outside the structure. Finally, a functor is a function from structures to structures (again: not a first class function). It may take a structure constrained by a signature as an argument, and it will itself have a signature. An application of the functor to a structure which matches the required signature will then produce a structure with the promised resulting signature.

Tentatively, the following concepts are related: A gbeta object is similar to an SML structure; subtype polymorphism performs a similar role in gbeta as signatures in SML; and a gbeta mixin may play a role similar to the one played by a functor in SML.

The first difference between gbeta objects and the SML module system is that gbeta objects are (partially) mutable, first-class entities, whereas SML structures are immutable entities that may only be used at top-level, in their own, separate name space. Moreover, it causes differences at many levels that the SML type system is oriented toward structural equivalence, whereas the gbeta type system distinguishes between two different declarations of the same name, except where these two declarations are explicitly declared to be related.

On the other hand, subsumption (subtype polymorphism) makes it possible for a gbeta object to present a subset of the actually implemented interface, similar to a structure with a declared signature. A mixin may be used to enhance a pattern which may then be instantiated, yielding an object which is an enhanced version of the object that the original pattern would have produced; when the object is used as a class repository, this is similar to the effect of applying a functor to a structure. Note that this may happen at run-time in gbeta.

In summary, the basic concepts in the SML module system may be useful as a starting point for the understanding of the usage of gbeta objects in the management of class families. However, there are so many and so deep differences that the analogy should not be taken too far.

6 Conclusion

This paper has presented the notion of family polymorphism. It has been demonstrated that traditional notions of polymorphism – dynamic, single-object subsumption and parametric polymorphism, with or without F-bounds – do not allow us to treat groups of objects belonging to mutually recursive families of classes in a safe manner without causing widespread dependencies on the exact classes involved, thereby prohibiting reuse with other families of classes.

The virtual pattern mechanism in gbeta supports polymorphic access to such groups of objects based on a notion of types depending on the identity of objects used as class repositories. This solves the abovementioned problems with safety and loss of reuse opportunities, and it only requires the explicit passing of the class family repository object together with the instances of members of that class family.

We believe that the correct but polymorphic management of multiple related objects is a natural and inevitable development in the area of object-orientation, on top of the well-established polymorphic usage of single objects. In particular, we expect various approaches to systematic production of variants of more than one class, including systems for advanced separation of concerns, to become more and more pervasive. Consequently, variants of groups of mutually dependent classes will also become more and more important. Family polymorphism is needed to ensure the traditional benefits of object-orientation, also when using these class families.

Acknowledgments. Thanks to the anonymous referees for their valuable and detailed comments, and to participants in various workshops on separation of concerns, aspect orientation, and related topics for the inspiration to focus on these ideas.

References

1. M. Aksit, K. Wakita, J. Bosch, and L. Bergmans. Abstracting object interactions using composition filters. *Lecture Notes in Computer Science*, 791:152++, 1994.

2. Ken Arnold and James Gosling. *The Java™ Programming Language*. The Java™ Series. Addison-Wesley, Reading, MA, USA, 1998.
3. L. Augustsson. Cayenne – a language with dependent types. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, 1998.
4. Don Batory, Rich Cardone, and Yannis Smaragdakis. Object-oriented frameworks and product lines. In P. Donohoe, editor, *Proceedings of the First Software Product Line Conference*, pages 227–247, August 2000.
5. B. Bobrow, D. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. *Common Lisp Object System Specification*. Document 88-002R. X3J13, June 1988.
6. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *Proceedings OOPSLA'98, ACM SIGPLAN Notices*, volume 33, 10, pages 183–200, Vancouver, BC, October 1998.
7. K. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. *Lecture Notes in Computer Science*, 1445:523–549, 1998.
8. Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *Fourth International Conference on Functional Programming and Computer Architecture*. ACM, September 1989. Also technical report STL-89-5, from Software Technology Laboratory, Hewlett-Packard Laboratories.
9. Robert Cartwright. Compatible genericity with run-time types for the Java™ programming language. In Craig Chambers, editor, *Proceedings OOPSLA'98, ACM SIGPLAN Notices*, volume 33, 10, Vancouver, October 1998. ACM Press.
10. Craig Chambers. Object-oriented multi-methods in Cecil. In O. Lehrmann Madsen, editor, *Proceedings ECOOP'92*, LNCS 615, pages 33–56, Utrecht, The Netherlands, June 1992. Springer-Verlag.
11. Craig Chambers. *The Cecil Language, Specification and Rationale*. Dept. of Comp.Sci. and Eng., Univ. of Washington, Seattle, Washington, 1997.
12. Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 1st edition, 2000.
13. Erik Ernst. *gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, DEVISE, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1999.
14. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP'97*, LNCS 1241, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer.
15. Vassily Litvinov. Constraint-based polymorphism in Cecil: Towards a practical and static type system. In Craig Chambers, editor, *Proceedings OOPSLA'98, ACM SIGPLAN Notices*, volume 33, 10, Vancouver, October 1998. ACM Press.
16. D. MacQueen. Modules for standard ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 198–207, New York, August 1984. ACM Press.
17. Ole Lehrmann Madsen. Semantic analysis of virtual classes and nested classes. In Linda M. Northrop, editor, *Proceedings OOPSLA'99, ACM SIGPLAN Notices*, volume 34, 10, Denver, October 1999. ACM Press.
18. Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings OOPSLA'89, ACM SIGPLAN Notices*, volume 24, 10, pages 397–406, October 1989.

19. Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, MA, USA, 1993.
20. R. Milner, M. Tofte, R. W. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, 1997.
21. Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 15–17 January 1997.
22. Didier Rémy and Jérôme Vouillon. On the (un)reality of virtual types. Work in progress, available from `http://pauillac.inria.fr/~remy/`, 2001.
23. Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass., 1997.
24. Jose H. Solorzano and Suad Alagić. Parametric polymorphism for Java$^{\text{tm}}$: A. In Craig Chambers, editor, *Proceedings OOPSLA'98, ACM SIGPLAN Notices*, volume 33, 10, Vancouver, October 1998. ACM Press.
25. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.
26. Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, pages 107–119, Los Angeles, May 1999. Association for Computing Machinery.
27. Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings ECOOP'97*, LNCS 1241, pages 444–471, Jyväskylä, June 1997. Springer-Verlag.
28. Mads Torgersen. Virtual types *are* statically safe. In *5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, at `http://pauillac.inria.fr/~remy/fool/program.html`, January 1998.
29. Mjølner Informatics, Århus, Denmark: `http://www.mjolner.dk/`.
30. Pierre Weis, María-Virginia Aponte, Alain Laville, Michel Mauny, and Ascánder Suárez. The CAML reference manual, Version 2.6. Technical report, Projet Formel, INRIA-ENS, 1989.
31. Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, Pittsburgh, PA 15213, September 1998.