Dynamic Inheritance and Static Analysis can be Reconciled

Erik Ernst[1]

DEVISE – Center for Experimental Computer Science
Department of Computer Science, University of Århus, Denmark

Abstract. In the area of object-orientation there is a long-standing schism between the rigid but safe statically typed languages, and the expressive and flexible but less safe "typeless" languages. Many efforts have aimed at combining the best of both. This paper presents a language mechanism which enhances the flexibility and expressivity of static languages while preserving the safety properties. It is an inheritance mechanism, with standard single inheritance as a special case. It allows both compile-time and run-time construction of new classes. Moreover, it supports specialization of existing objects at run-time. This helps avoiding the combinatorial explosion in the number of classes associated with multiple inheritance, and it supports a better separation of concerns in large systems. Pre-methoding—inheritance applied to behavioral descriptors—has been used for the construction of control structures for many years, in BETA. With dynamic inheritance, pre-methoding becomes more expressive, supporting control structures as first class values which may be constructed and combined dynamically. Even though the concept of pre-methoding is missing from most other languages, the basic idea could be applied to any statically typed object-oriented language.

1 Introduction

The dilemma has always been there: should a programming language protect programmers from themselves by means of a static type checking system, or should it provide greater flexibility in a more brittle, dynamically typed setting? The two choices appear to be incompatible—either safety or flexibility gets the highest priority, and the unchosen goal suffers. Our work takes a step towards reconciliation of the two goals, by enhancing flexibility in the context of static typing.

The results presented here were achieved during the design and implementation of a generalized version of the language BETA [18, 15], but the basic ideas could as well be applied to other statically typed object-oriented languages.

We present an inheritance mechanism which is more dynamic than usual statically typed inheritance, and whose expressive power is not immediately paralleled by dynamic languages such as Smalltalk or CLOS. Standard, single inheritance comes out as a special case. Classes are genuine run-time values which may be transferred as parameters, stored in variables, and computed from

other class values. Moreover, the structure and type of existing objects may be enhanced at run-time.

One specialty of BETA is the support for behavioral inheritance, or *pre-methoding*. This does not mean that one entity inherits the behavior of another entity, but rather that one behavior is specified as an incremental modification of another behavior. A typical and useful way to use pre-methoding is to create special purpose control structures, such as collection iterators, or GUI framework callbacks. Dynamic behavioral inheritance can be used to create first class control-structures, which can be transferred as arguments to methods and applied to a "body" of code dynamically. This could also be applied to other languages, but they would have to benefit from the introduction of behavioral inheritance first...

The rest of this paper is organized as follows: A very brief introduction to the BETA language is given in Sect. 2, along with an outline of the enhancements we added to BETA to support dynamic inheritance. Section 3 gives some important examples. At a first reading, the examples provide an intuition about how the dynamic inheritance mechanism works, and motivate why it is useful. Hereafter, the mechanism which supports dynamic inheritance is described in more detail in Sect. 4. Using this it should be possible to gain a deeper understand of the examples in Sect. 3. A discussion about related work follows in Sect. 5. Finally, Sect. 6 describes future work, and Sect. 7 concludes.

2 BETA Basics

If you already know BETA you might want to go directly to Sect. 2.1, to review the enhancements we made to the language in order to support dynamic inheritance.

```
Point:                          (* a class declaration *)
  (# x,y: @integer;             (* attributes of Point *)

     Move:                      (* a method of Point   *)
       (# dx,dy: @integer       (* declare attributes  *)
       enter (dx,dy)            (* specify arg. list   *)
       do x+dx->x;              (* assign "x:=x+dx"    *)
          y+dy->y
       exit (x,y)               (* return new position *)
       #);

  (* enter/do/exit is optional; Point has none *)
  #)
```

Fig. 1. A pattern used as an class (Point) or as a method (Move)

In BETA, methods and classes are unified into the very powerful *pattern* concept. A pattern may be described as a class with a default method—the *do-part*—and with specifications of input and output properties—the *enter-part* and *exit-part*. All these "parts" can be recognized by the keywords `do`, `enter`, and `exit`. See Fig. 1.

Since classes and methods are technically unified into patterns, the distinction between them is just a convention: "class" and "method" are words used by programmers to hint at intended usage. A class is then a pattern with some features but with no `enter`-part, no `exit`-part, and no `do`-part.[1] A method or procedure or function is a pattern which declares the arguments (if any) in the `enter`-part, the behavior in the `do`-part, and returned values (if any) in the `exit`-part. In all cases, the pattern is a *descriptor*, and it is used when creating *substance*. The substance may then be used as an object (i.e. it is long-lived and its state is "interesting") or as an activation record for a method/procedure/function (i.e. it is short-lived, usually anonymous, and its side-effects or returned results are "interesting"). Since the distinctions are purely conventional all combinations are allowed, providing for a very expressive and flexible language. For instance, a method invocation can be stored away for later execution.

BETA syntax is unusual in that the arrow operator, `->`, is used for assignment and argument transfer. The data flow has the same direction as the arrow, i.e. the traditional notation `y:=x` corresponds to `x->y` in BETA. Invocation of methods is similar—what is typically written as `x:=obj.m(a,b,f(2+3))` becomes `(a,b,(2+3)->f)->obj.m->x`, where `f` would be a function, `m` a method, and `x` a variable. In fact, `f` and `m` could as well be variables, and `x` could be a method. This serves to illustrate that it is a homogeneous syntax, abstracting away the difference between access to stored and computed values.

Inheritance applies to methods as well as classes in BETA, because they are both patterns. So what does it mean that a method inherits from another method? Think of it as a process of syntax tree completion. A pattern may contain placeholders—the `INNER` statement—in its `do`-part, and the `do`-part provided in a derived pattern gets executed when the `INNER` statement is reached. Except for name lookup, it works as if the sub-pattern `do`-part is inserted into the super-pattern `do`-part in place of the `INNER` statements, i.e. as if inheritance allows you to fill in the placeholder nodes (`INNER`) in the syntax tree. Figure 2 illustrates this by the pattern `E` whose do-part is the manually created equivalent to the do-part created by inheritance in `D`.

As noted, that is except for name lookup. Each name application must be looked up starting from its actual position in the source code, searching among the statically visible names at that position. The visible names are those of declarations in the surrounding block `(# .. #)`, those of inherited declarations, and the visible names in enclosing blocks, in that order. Since each block describes a part of an object which is situated in an environment, an object consists of a number of part objects (one for each block, i.e., one for each level of inheritance), and each part object has a pointer to its enclosing part object, an *origin* pointer.

[1] This is really simplistic, but not entirely wrong.

```
B: (# (* INNER is a placeholder *)
   do (for i:3 repeat INNER for)
   #);

D: B (* inherit from B *)
   (# (* fill in the INNER in B *)
   do 'Hello, world!'->putline
   #);

E: (# (* print 'Hello, world!' three times *)
   do (for i:3 repeat
         'Hello, world!'->putline
      for)
   #)
```

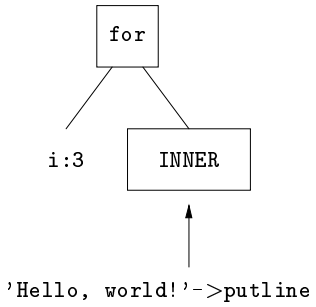

Fig. 2. Inheritance and behavior: `D` and `E` are equivalent

The static analysis has to take the origins into consideration, and it does so using *run-time paths* that describe how to find the enclosing part object.

Run-time paths must have a more general form in **gbeta** than that used to analyse BETA. Apart from that, all this is standard BETA. The following section outlines the enhancements we introduced into BETA, including the dynamic inheritance.

2.1 Enhancements

Standard BETA does not support dynamic inheritance, and it does not support explicit nor implicit type combination mechanisms, such as multiple inheritance. Our enhanced version of BETA, **gbeta**,[2] supports all this and more. Correct BETA programs are also correct **gbeta** programs, and they have the same behavior as in BETA. There are a few minor exceptions where the **gbeta** semantics is different from BETA, but these are deliberate attempts to improve on murky corners of the BETA semantics. They are not essential for this discussion and could easily be reverted to the BETA semantics.

Syntactically there is almost no difference between BETA and **gbeta**. The most visible difference is that patterns may be combined using the "`&`" operator, as in `P&Q`. This combination is defined semantically by the merging rule given in Sect. 4.2. Merging lies behind all the deep differences between BETA and **gbeta**, but in this paper we will only refer to merging in connection with dynamic inheritance.

[2] An implementation can be downloaded from `http://www.daimi.aau.dk/~eernst/gbeta`.

The precise meaning of merging is described in Sect. 4.2, but the intuition is that `P&Q` is a pattern which inherits from both `P` and `Q`, yielding a pattern with a combination of their attributes and with a combination of their behavior (`do`-parts) and interaction properties (`enter`- and `exit`-parts). The chosen meaning of the "`&`" operator is non-commutative, i.e. `P&Q` is not the same thing as `Q&P`, and any one of them might be the right choice in a concrete situation. It would be nice to have a commutative operator, but the current choice is well-defined and useful in practical programming, and no commutative alternative with useful semantics has been found.

Seemingly ordinary inheritance may be dynamic if the pattern being inherited from (the super-pattern) is not a compile-time constant. For instance, `P(# .. #)` denotes a pattern inheriting from `P`, and such an expression occurs many times in any BETA program. However, if `P` is not a constant pattern but, e.g., a pattern variable,[3] it is dynamic inheritance. Both BETA and **gbeta** support pattern variables, but only **gbeta** allows using a pattern variable as a super-pattern.

Another example of dynamic inheritance is the dynamic modification of the structure of an existing object, preserving object identity. This is accomplished by statements like `somePtn##->anObject##`, and the intuitive semantics of this is that the structure of `anObject` is enhanced until it is an instance of the pattern denoted by `somePtn`. Syntactically this is an assignment, and the `##` markers are used in BETA to focus on "the-pattern-of-something," so a BETA programmer might describe it like "put `somePtn` into the pattern of `anObject`." One probably has to be a BETA aficionado to think that this syntax is obvious, but it is reasonably consistent with the rest of the language. Note that in BETA, it is a compile-time error to have an object on the receiving side of such an assignment, so this construct does not alter the meaning of existing BETA code. Again, the merging rule defines the exact meaning of this dynamic enhancement of the structure of objects, and the precise description will be given after the examples, in Sect. 4.

3 Dynamic Inheritance at Work

The following three subsections contain examples of dynamic inheritance used in practice.

3.1 Incremental Object Creation

In large projects, it is important to apply "divide & conquer" strategies whenever possible, e.g. by modularizing the solution, and keeping module dependencies at a minimum. Global knowledge goes against this strategy by creating many dependencies. The ability to create objects incrementally gives a new opportunity to remove global knowledge about the total structure of highly visible objects. For example, when ordinary, static multiple inheritance is used to combine a set

[3] A variable whose value is a pattern; a genuine "type variable."

```
(# car: (# (* minimal, common car pattern *) #);
   ...
   (* Here: declarations of views on cars, e.g. 'realCar'; they
    * would be placed in different subsystems of a large system *)
   ...
   myCar: ^car; (* a reference to a car *)
do
   (* create a new car and make 'myCar' refer to it *)
   &myCar[];

   (* build the structure of 'myCar' dynamically;
    * this would also happen in different subsystems *)

   (* the driver needs a real car *)
   realCar## -> myCar##;

   (* accounting cares about money *)
   property## -> myCar##;
#)
```

Fig. 3. Incremental Object Creation

of largely independent aspects in order to describe the structure of a complex object used by many subsystems, this combined description depends on many subsystems, and it is probably used in many places, too. Incremental object creation avoids that problem since the combination classes need not be declared explicitly.

As an example, illustrated in Fig. 3, consider the several different views on a car in a large organization. It must be bought, paid, registered, written off now and then, reserved for daily use and for maintenance and repair, and the overall computerized representation of the car should be kept consistent. By creating a basic car object and letting the different departments specialize the object dynamically, the system avoids expressing static knowledge about the total car object.

Let's consider the alternatives. If we create a class "`BeAllEndAllCar`" which inherits (by ordinary static multiple inheritance) from many classes used in different subsystems, then this class will depend on all those subsystems and it will have to be changed often. This costs time and resources, because many changes presumably means many bugs, and because of more complex collaboration where many programmers change the same files, but also because of many recompilations, not to mention the design distortions introduced to avoid them. Moreover, many subsystems might in turn depend on `BeAllEndAllCar` because they need to create instances of this class. The whole system ends up very unstable an unmanageable.

We can improve on this by using abstract classes and interfaces. They presumably change less often than classes with an implementation. So each subsys-

tem workgroup delivers a "requirement specification" in the shape of an abstract class or an interface. It's a dilemma to decide whether each subsystem workgroup should also deliver an implementation. If they do, the implementation level `ConcreteBeAllEndAllCar` will depend on all these and be just as vulnerable as before, and if `ConcreteBeAllEndAllCar` is implemented centrally, the people who implement it will have to have almost universal knowledge about the system. But at least we can use an abstract factory design pattern with an `AbstractBeAllEndAllCar` class to ensure that the subsystems will not have to recompile every time the implementation changes. This removes a very important ripple effect.

But there are still advantages of dynamic, incremental object creation which are not parallelled in a static approach. Firstly, if we do not really need the `BeAllEndAllCar` but would rather prefer to use any of the 2^n different selections from a set of n aspects, then the static approach leads to the tedious definition of up to 2^n combination classes, where incremental object creation only needs the n aspects. Secondly, the ability to add aspects on demand may be important when there are many objects and only some of them need some "expensive" aspect. E.g., we may keep lots of cars around but only few of them need to carry information about the legal aspects of a serious traffic accident. When we need to use this aspect, we test to see if it is already there, otherwise we add it.

Until now the focus has been on independent aspects, so let us consider dependencies. If we have one aspect, e.g. `Limousine`, which can only be defined in terms of another one, e.g. `Sedan`, then `Limousine` would inherit from `Sedan` such that the features of `Sedan` would be available for the specification and implementation of `Limousine`. When incrementally building an object `obj`, the statement `Limousine##->obj##` would enhance `obj` with whatever is *missing* in order to make `obj` a `Limousine`. This might be only the `Limousine` part or it might be both the `Sedan` and the `Limousine` part. If it is important for application specific reasons that `obj` is indeed a `Sedan` when we make it a `Limousine` then we can always check,

```
(if obj##<=Sedan## then
   Limousine##->obj##
 else
   ...
if)
```

but this check is not necessary to ensure internal consistency of objects (an aspect of guaranteeing that "message-not-understood" errors never occur). That consistency is an inherent property of **gbeta**.[4] Such correctness guarantees are a main reason for having type systems—the interesting twist here is that they extend to run-time in a new way.

[4] It is an ongoing effort to prove this! Practical experience with **gbeta** suggests it holds.

3.2 Breakpoints

```
(# (* an example procedure to put breakpoints into *)
   proc: (# do <<proc-things>> #);

   (* to manipulate an invocation of 'proc' we need a reference *)
   procRef: ^proc;

   (* the "breakpoint" prints a message *)
   break: (# do 'proc called!'->putline #);
do
   (* create new instance of 'proc' and bind 'procRef' to it *)
   &proc[]->procRef[]; (* not executed *)

   (* create a pattern which runs 'break' before executing 'proc' *)
   (# do break; INNER; #) & proc ##
   -> procRef##;

   (* run the modified 'proc' invocation *)
   procRef; (* prints "proc called!", then
             * does <<proc-things>> *)
#)
```

Fig. 4. Adding a "breakpoint" to a procedure call

The concept of a breakpoint is usually associated with programmer tools such as debuggers, but here it is used to describe a language-internal construct which might be used in many different ways. The word "breakpoint" just seems to be the most straightforward vehicle for evoking the right intuition about what is going on.

Consider Fig. 4. The (pattern used as a) procedure `proc` is given. The code for its behavior is represented by the placeholder `<<proc-things>>`. Assume that we do not want to change the code for `proc`, but still we want to intercept the execution of it, and add some behavior at the time, say, when this execution starts. For this example the `break` behavior is simply to print the string `"proc called!"`.

An instance of the pattern `proc` is created and bound to the reference `procRef`. This may be described as the creation, but not execution, of an invocation of the procedure. This is usually considered a reflective capability, but in BETA it is a simple consequence of the unification of methods and classes. The news lie in the subsequent modification of it.

The unexecuted invocation is modified by inserting an invocation of `break` at the beginning. The expression

```
(# do break; INNER #) & proc ##
```

denotes a new pattern, constructed by merging `(# do break; INNER #)` and `proc`, at run-time. This new pattern is used to dynamically enhance the structure of `procRef`. The merging rule defined in Sect. 4.2 defines the exact outcome.[5] The combination of `do`-parts gives the resulting object denoted by `procRef` a behavior equivalent to

```
do break; <<proc-things>>
```

This example is not very useful as it is, in particular because we do not usually have the opportunity to manipulate all the "interesting" invocations of `proc`. To make it more realistic it would be necessary to introduce virtual patterns, because an existing object with a virtual method could be modified such that *all* subsequent invocations of that method on that particular object would lead to the execution of such a "breakpoint." This could be used to set up triggers, e.g. to keep an eye on the state of the object and execute some callback in case it violates given rules.

```
(# abstractP: (# m:< object #);
   concreteP: abstractP
     (# m::< (# do <<m-impl>> #)#);
   aP: ^abstractP;
   break: (# do .. #);
do
   &concreteP[]->aP[];
   abstractP(# m::< (# do break; INNER #)#) & aP ## -> aP##;

   (* now all invocations of 'm' on the object denoted
    * by 'aP' will 'break' before executing <<m-impl>> *)
   aP.m; aP.m; ...
#)
```

Fig. 5. Adding a breakpoint to a virtual method

Since, for simplicity, virtual patterns are not mentioned elsewhere in this paper, a small example of this is given as Fig. 5 without further explanation.

3.3 Dynamic Control Structures

A control structure is a language entity (builtin or user-defined) which is parameterized with one or more pieces of code, *bodies*, immersed into a name space. A standard example is an `if`-statement in almost any language, where the bodies are the `then`-part and the `else`-part; in this case the name space is empty (no

[5] The merge has the form $[b]\&[proc] = [proc, b]$ which is then used to enhance an object whose structure is $[proc]$. This adds a b-slice to that object as the last/outermost, such that its original behavior is enclosed in the `do`-part of the new slice.

```
(# (* just execute the body two times *)
   twice: (# do INNER; INNER #);
do
   (* prints two lines of text *)
   twice(# do 'Hi again, world!'->putline #)
#)
```

Fig. 6. A user-defined control structure

declared names are provided by the `if`-statement). A standard control structure which provides a non-empty name space is a `for`-statement, which typically allows the body to refer to an index variable which is incremented with each execution of the body.

The typical way to create a control structure in BETA is to define a pattern in which an `INNER` statement is placed in the position where the body should be executed. See Fig. 6. An expression like `twice(# .. #)` denotes an anonymous pattern inheriting from `twice`; when executing it as a statement, an instance of the anonymous pattern is created and executed. Such "immediate" patterns are used very much in BETA, in particular with user-defined control structures.

In **gbeta** it is possible to have dynamic control structures, using inheritance from a pattern variable. This makes it possible to parameterize a method with a control structure, delaying the decision about what control structure to use until call-time. The example given in Fig. 7 does just this. It had to be a little more involved than the previous examples, but we hope that the chosen names, formatting etc. makes it understandable even though it uses aspects of BETA that we haven't described, including some standard libraries. The statement `anIterator(# do theLine[]->putline #)` requires **gbeta**, because a pattern variable is being used as a super-pattern. The rest would compile and run as-is in BETA.

To create a similar example (with just one iterator, for brevity) in a language which supports behavioral arguments, e.g. function pointers in C/C++, we could let `anIterator` be a pointer to a function which takes a pointer to another function as an argument, and then make this other function execute the print statement, as in Fig. 8. But this would not suffice. The extra expressive power here lies in the fact that the specialization of `anIterator` in `LinePrinter` has access to the statically known name-space of `anIterator`, e.g. it can use the name `theLine`. In the function pointer approach this name space is provided by the argument list to the function `body`. For example, adding a name to the name-space has only local effects (in `iterator`) in the **gbeta** approach; in the function pointer approach, we must also change the `typedef` of `callback`, as well all usage points (all `iterators`). In both cases, of course, all places where the new name is actually *used* would have to be changed, too.

```
(# myFile: @file; (* an interface to a disk file *)
   myList: @list(# element::text #); (* a list of texts *)

   (* the iterator interface *)
   iterator: (# theLine: ^text do INNER #);

   (* concrete iterators *)
   fileIterator: iterator
     (# (* iterate through the file, and make the
         * current line available in 'theLine' *)
     do cycle
        (# while::(# do (not myFile.eof)->value #)
        do myFile.getline->theLine[];
           INNER fileIterator
        #)
     #);
   listIterator: iterator
     (# (* iterate over the list elements, and make
         * the current text available in 'theLine' *)
     do myList.scan
        (# do current[]->theLine[]; INNER listIterator #)
     #);
   inputIterator: iterator
     (# (* read lines from std.in until empty line *)
     do cycle
        (# while::(# do ((getline->theLine[]).length>0)->value #)
        do INNER inputIterator
        #)
     #);

   (* a method which takes an iterator as argument *)
   LinePrinter:
     (# anIter: ##iterator (* a pattern variable *)
     enter anIter##
     do (* iterate and print each text *)
        anIter(# do theLine[]->putline #)
     #)
do
   (* initialize *)
   'somename'->myFile.name; myFile.openread; myList.init;
   (* iterate over the file, the list, and std.in *)
   fileIterator## -> LinePrinter;
   listIterator## -> LinePrinter;
   inputIterator## -> LinePrinter
#)
```

Fig. 7. `LinePrinter` is parameterized with a dynamic control structure, `anIterator`

```
typedef void (*callback)(char *);
typedef void (*control_structure)(callback);
FILE myFile;

void fileIterator(callback cb)
{
  char buffer[1000];
  while (! feof(myFile)) {
    fgets(buffer,999,myFile);
    (*cb)(buffer);
  }
}

void body(char *theLine) { printf("%s\n",theLine); }

void LinePrinter(control_structure anIterator)
{
  (*anIterator)(&body);
}

int main(int argc, char *argv[])
{
  myFile=fopen("somename","r");
  LinePrinter(&fileIterator);
  return 0;
}
```

Fig. 8. Using function pointers to simulate a dynamic control structure

4 The Dynamic Inheritance Mechanism

Now we describe the mechanism behind dynamic inheritance in more detail.

4.1 Inheritance, and Linearization

First we briefly motivate the idea that patterns (and classes) can be viewed as lists of blocks of declarations. In a declaration such as

```
D: B(# .. #);
```

we have a derived pattern `D`, a base pattern `B` and a block of declarations, `(# .. #)`. This syntactic notion of a block corresponds to a semantic entity that we will call a *pattern slice* henceforth. It includes an origin which specifies the environment. Looking at an inheritance hierarchy like the following:

```
B:  (# (*1*) .. #);
D: B(# (*2*) .. #);
```

we may say that `D` is created from `B` according to `(# (*2*) .. #)`, or that `D` is created according to `(# (*2*) .. #)` and `(# (*1*) .. #)`, in that order. The difference is that we consider all slices separately in the second description (ignoring the names of intermediate patterns). The slices are organized into a simple list as long as we only use single inheritance, but with a type combination mechanism they would in general be organized into a directed acyclic graph (DAG). Name lookup rules are very straightforward in the case of simple lists, so linearization is a way to resolve name lookup ambiguities.

Linearizing multiple inheritance has a bad reputation, primarily because it can give rise to confusing name lookup processes. Especially in CLOS there has been work on achieving a "monotonicity" property [13, 11]. In the absence of this property, it can happen that a lookup process finds the same declaration when searching for a name, say `x`, in two classes, $\texttt{C}_1$ and $\texttt{C}_2$, but in the combined class $\texttt{C}_1\,\texttt{\&}\,\texttt{C}_2$, a lookup for `x` will find some *other* declaration. In other words, $\texttt{C}_1$ and $\texttt{C}_2$ agree on the "meaning of `x`," but their combination doesn't agree with them on this. No wonder people get confused!

However, this problem is not a characteristic of linearization as such, and in particular the problem is trivially solved in a language like **gbeta** with static name binding. In general, since static name binding ensures that each name application is associated with one particular name declaration already at compile-time, the lookup process is quite perspicuous for programmers. For any given name application, a tool[6] can show exactly what declaration it refers.

Consequently, our approach is to linearize the slice graph using the merge rule defined below. The static analysis of **gbeta** takes the origin of each pattern slice into account, and other things. The description given here is a simplification which suffices to explain the merging process. Moreover, even though subpatterns do not play an important role in this paper, we should mention that a pattern P is a sub-pattern of another pattern Q iff Q is a sub-list of P, considered as lists of pattern slices.[7]

There are several good reasons to choose a linearizing approach. First, with the new possibilities for type combination in **gbeta**, linearization gives a very useful and expressive semantics for the combination of behavior. The difficulty in finding a useful and understandable semantics for combination of behavior is a prime reason why it hasn't been implemented for BETA before our project. Second, name clashes are automatically resolved (the declaration in the most specific slice[8] hides the other declarations with the same name). And since name lookup always uses the compile-time information, hiding only affects those who use a static type which includes both/all conflicting names. Use a type which knows the declaration you want, and you are guaranteed to get it!

---

[6] Including the earlier mentioned implementation of **gbeta**

[7] $[]$, $[a]$, $[b]$, $[a, c]$, and $[a, b, c]$ are all sublists of $[a, b, c]$, but $[b, a]$ and $[d]$ are not.

[8] The slice which comes first in the list defined by merging, see Sect. 4.2

4.2 Merging

In **gbeta**, patterns can be combined at compile-time and at run-time, and the linearization used is defined in terms of a *merge rule*. Syntactically, the "&" operator is used for pattern merging, as we have already seen in the examples. One important property of the merge rule is that it has the traditional single inheritance (of BETA, and of almost all object-oriented languages) as a special case. The merge rule applies to finite sequences of distinct elements in general, and it can be characterized in several different but equivalent ways. First, in Fig. 9, we give an algorithm for it by an SML function `merge` which merges two lists into one.

```
fun merge xs [] = xs
  | merge [] (y::ys) = y::ys
  | merge (xxs as x::xs) (yys as y::ys) =
    if x=y then y::(merge xs ys)
    else if not (member y xs)
         then y::(merge xxs ys)
    else if not (member x ys)
         then x::(merge xs yys)
    else raise Contradiction;
```

Fig. 9. The merge rule as an algorithm

The algorithm uses the following standard function to search in the two lists being merged:

```
fun member x [] = false
  | member x (y::ys) =
    if x=y then true else member x ys;
```

A formal definition of the merge rule can be given when the sequences are considered as total order relations, i.e., an element x in a sequence is "$\leq$" than another element y iff x occurs earlier than y in the sequence. It is easy to see that a total order determines exactly one sequence of distinct elements, and a sequence of distinct elements determines exactly one total order.

Definition 1 (Merging Rule). Given two finite sets A and B, the *merge* $R_A \& R_B$ of a total order relation $R_A \subseteq A \times A$ with another $R_B \subseteq B \times B$ is

$$R_A \& R_B \triangleq R \cup (B \times A \setminus \overline{R})$$

where $R \triangleq (R_A \cup R_B)^*$ is the transitive closure of the union of the relations R_A and R_B, and $\overline{R} \triangleq \{(y,x) | (x,y) \in R\}$ is the inverse of R.

The intuition behind this definition is that the ordering of A and the ordering of B are respected, but we need to take the transitive closure to include the indirect consequences of the union; this yields the relation R. Then R is enhanced with all those pairs from $B \times A$ that do not contradict R, i.e., all elements from B are considered smaller than elements from A, unless R already says the opposite. The short version of the intuition is "respect R_A and R_B, and make elements from B the smallest by default."

We have recently discovered that the merging rule specifies not only the algorithm used in **gbeta** but also an algorithm considered for linearization in Dylan in [2]. Because it was "unusual" compared to CLOS and LOOPS, it was not used in Dylan, in spite of the fact that it is the only known linearization which is both monotonic and respects two other consistency requirements related to inheritance graphs (local precedence order and extended precendence graph, see [2] for details).

Note that this linearization, like all others, do not handle all possible combinations of sequences. The problem is that two sequences may list two elements a and b in different order. The merged sequence cannot list them in any order without contradicting one of the sequences being merged, so the merging process fails. In this case, the exception `Contradiction` is raised in the algorithm, and the relation $R_A \& R_B$ will not be a total order in definition 1.

Now to a few examples. As a special case, the merge rule just concatenates two sequences if they have no elements in common, e.g. $[a,b,c]\&[x,y] = [x,y,a,b,c]$. This is pure "disjoint" multiple inheritance. If the second sequence has length one, the merge rule puts the new slice in front of the rest, e.g. $[a,b]\&[x] = [x,a,b]$. This is exactly the semantics of traditional single inheritance. Finally, Fig. 10 shows a "non-special-case" example.

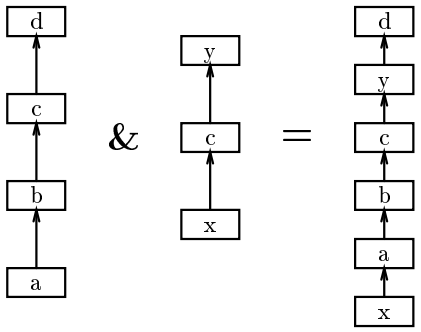

Fig. 10. Merge rule example; each box stands for a pattern slice

As we know, dynamic specialization (structure enhancement) of objects is also possible: a pattern can be applied to an object as a *constraint* on the structure of the object. It is like saying to the object "you must enhance your structure

until you are an instance of this pattern!" If the object is currently an instance of a pattern `P` and we apply a pattern `Q` to it as a constraint, then the object will be enriched with new slices until it is an instance of `P&Q`. The identity of the growing object is preserved, it *is* the same object changing structure. The syntax for this was also mentioned in Sect. 2; if `q` has a pattern (it may be a pattern, an object, or a reference to one of those), and `x` is a reference to an object, then `q## -> x##` will dynamically specialize `x` according to the pattern of `q`.

5 Related Work

"Dynamic inheritance? No problem! Just write your program in Self [23] and build and rebuild whatever inheritance hierarchy you want at any time by assigning to the parent slots." Admittedly, dynamic inheritance exists, and you could hardly envision a greater flexibility than what is already supported in prototype based languages like Self. There is also support for type inference in Self [1], but this builds on a closed world assumption, so the entire program must be available for the inference algorithm, and changing one single line of the code would potentially invalidate any of the inferred types. Nevertheless, Self serves as an ideal with respect to the flexibility.

Smalltalk-80 [12] is class-based, and every object is an instance of one particular class. However, the `become:` primitive will swap the identities of two objects and thus supports arbitrary structural changes to any given object identity. It remains a low-level task for the programmer to transfer any shared state to make a group of objects seem like one object with varying structure. In a very sophisticated approach [19], Mira Mezini uses a so-called *metaCombiner* object in a reflective middle layer between objects and classes. Each "object" (let's call it complete) corresponds to one core object and a number of adjustment objects (let's call them internal), as well as the metaCombiner object which manages information about the methods of the complete object. It is possibile to add and remove adjustments. When two or more internal objects implement a given method they can be treated as aspects of the same and executed sequentially, or they can be treated as unrelated and made available in separate scopes. This enables a programmer (who noticed the danger) to avoid accidental identification of methods that are conceptually different but have the same name.

The language Sina embeds the concept of *composition filters* [3] which also allow for very flexible and expressive control over the method dispatch and state distribution within a collection of objects. The composition filters may reject or redirect message sends depending of dynamically evaluated conditions, and it is possible to simulate a standard inheritance mechanism, which may then select varying "parents" dynamically. No static type system here either.

All these systems are very flexible. The flexibility goes along with a very rich universe of potential program executions, and this makes it difficult to prove that any specific properties hold about individual program elements—in other words, they are not designed for static type checking.

The CLOS [14] convention of using some classes for "mixin" inheritance has been developed [4, 10, 22] into a separate concept of mixin-based inheritance. A *mixin* is a class modifying device, supporting inheritance directly as an incremental modification that may be applied to several different classes. In [22, 17, 16] it is described how mixins can support dynamic inheritance and how it can be statically type checked. The catch is that each object must contain specifications of all its potential enhancements which makes practical software engineering and reuse hard.

In Cecil [8, 9], predicate objects have been introduced to support dynamic changes of object structure and method implementations. Cecil is prototype based like Self, but with a slightly different object model. An object has a fixed position in the ordinary, static inheritance hierarchy, but it may also inherit from a number of predicate objects, depending on its state. Since predicate inheritance is determined by general boolean expressions any non-trivial questions are undecidable; but if the programmer manually proves (or claims) some disjointness and completeness properties and annotate the code, a type check can be made. However, the program is only accepted as type safe if the dynamic inheritance provably has no effect on the interfaces, i.e. if the dynamics may simply be ignored for type checking purposes.

Hence, in general, dynamic inheritance and genuine strict, static type checking have not been reconciled. This paper presents support for dynamic creation of classes, inheritance from classes known only at run-time, and dynamic evolution of the structure of objects, without compromising the static type checking. The starting point for this was the programming language BETA and some inspiring work [6, 5, 7] on it. We have designed and implemented **gbeta**, a backward compatible, generalized version of BETA supporting the above.

Compared to Self, the Smalltalk approaches, and composition filters, **gbeta** is less flexible: The structure of objects may be enriched, not reduced. This means that an object may be specialized to an instance of a more derived class, not generalized to an instance of a superclass or to an unrelated class. Compared to the metaCombiner approach, adjustments can be expressed naturally, and the method combination of standard BETA is more static but also more expressive. Moreover, the Smalltalk problem of accidental identification of methods is irrelevant in our language which has static name binding. Compared to mixin-based inheritance, it is not required that each object foresees its potential for structure development. Compared to predicate based inheritance, there is no support for changing the structure of an object automatically, an explicit statement must be executed. But the structure enhancements in **gbeta** certainly go beyond such ones that leave the interface unchanged.[9]

[9] Since **gbeta** is statically typed, new names only become accessible after using a typecase to obtain a reference to the enhanced object whose statically known type includes those new names

6 Future Work

The current implementation places a couple of restrictions on the usage of the facilities described in the previous sections, and this gives rise to some obvious future work.

Firstly, patterns containing two or more slices associated with the same syntax are not considered well-formed. When this is detected at compile-time, the program is rejected. Dynamic type combinations cannot always be checked at compile time, and run-time checking is performed for those cases. We could remove this restriction, but the ambiguity would most likely give a confusing semantics. On the other hand, there should be a test for well-formedness of dynamic type combination, such that run-time errors associated with well-formedness can be avoided—just like `(if y<>0 then ..x/y.. if)` can be used to avoid a "Divide by zero error" at run-time. There is currently no generally applicable test of that kind, so failure must be handled as an exception.

Secondly, only dynamically allocated objects may be dynamically specialized. An object which is statically allocated (i.e. declared with an at-sign like `x: @integer`) is a very valuable resource for the type analysis, because its type is usually a compile-time constant. Since BETA allows and extensively uses covariance, type exact references is *the* most important resource which allows the type checker to determine that a covariant type constraint is in some context actually strengthened to an invariant one. It is not very likely that this restriction will be lifted totally, but we might lift it partially by detecting more cases where the restriction does not help type checking anyway, such as when the statically allocated object is an instance of a pattern variable.

Turning to future work which is not motivated by current restrictions, it would be interesting to explore the possibilities for relaxing the strict linearization of pattern slices into a partial ordering of the slices. It would make two slices unordered if swapping them would have no visible semantic effects. In particular, two consecutive slices without a `do`-part whose declared names are disjoint sets could be unordered. This would rescue some type combinations where the merging would otherwise fail.

Finally, a formal semantic specification of **gbeta** is under construction, using the formalism of Action Semantics [20, 21].

7 Conclusions

A flexible inheritance mechanism which supports static as well as dynamic class construction and method combination has been presented. Moreover, the class (and hence structure) of existing objects may be incrementally enhanced according to any given class. The mechanism works in the context of a language with strict, static type checking and static name binding. The basic strategy of viewing a class as a graph of class slices applies to any statically typed object-oriented language, but the type combination rule would have to be adjusted to be able to combine general DAGs. There is a working implementation of the language **gbeta** which embodies the dynamic inheritance mechanism. Some usage

examples have been given demonstrating advanced management of control flow and name spaces. All in all, one step has been taken in the direction of making static languages more flexible and expressive.

Acknowledgments

The design and implementation of **gbeta** has taken years, and during that period lots of discussions have taken place which helped in the process. In particular, Mads Torgersen and Kresten Krab Thorup and also Henry Michael Lassen and Søren Brandt gave valuable input. Of course, the whole project would be unthinkable without the BETA tradition and community which includes many more people at Århus University, first of all my advisor Ole Lehrmann Madsen.

References

1. Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, LNCS 707, pages 247–267, Kaiserslautern, Germany, July 1993. Springer-Verlag.
2. Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, Andrew L. M. Shalit, and P. Tucker Withington. A monotonic superclass linearization for Dylan. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31, 10 of *ACM SIGPLAN Notices*, pages 69–82, New York, October6–10 1996. ACM Press.
3. Lodewijk Bergmans, Mehmet Aksit, and Ken Wakita. An object-oriented model for extensible COncurrent systems: The composition-filters approach. *IEEE Tr. on Parallel Distributed Systems*, 1993.
4. Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, pages 303–311, October 1990. Published as Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices, volume 25, number 10.
5. Søren Brandt. Reflection in a statically typed and object oriented language – a meta-level interface for BETA. Technical Report DAIMI IR-126, Computer Science Department, Aarhus University, Aarhus, Denmark, 1996.
6. Søren Brandt and Jørgen Lindskov Knudsen. Generalising the BETA type system. In P. Cointe, editor, *Proceedings ECOOP '96*, LNCS 1098, pages 421–448, Linz, Austria, July 1996. Springer-Verlag.
7. Søren Brandt and Rene Wenzel Schmidt. Dynamic reflection for a statically typed language. Technical Report DAIMI PB-505, Computer Science Department, Aarhus University, Aarhus, Denmark, 1996.
8. 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.
9. Craig Chambers. Predicate classes. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, LNCS 707, pages 268–296, Kaiserslautern, Germany, July 1993. Springer-Verlag.
10. Wim Codenie, Koen De Hondt, Theo D'Hondt, and Patrick Steyaert. Agora: Message passing as a foundation for exploring OO language concepts. *SIGPLAN Notices*, 1995.
11. R. Ducournau, M. Habib, and M.L. Mugnier M. Huchard. *Proposal for a Monotonic Multiple Inheritance Linearization*, volume 29, no 10, pages 164–175. ACM SIGPLAN Notices, Oregon, USA, October 1994.
12. Adele Goldberg and David Robson. *Smalltalk–80: The Language*. Addison-Wesley, Reading, MA, USA, 1989.
13. M. Huchard, M.L. Mugnier, M. Habib, and R. Ducournau. Towards a unique multiple inheritance linearisation. In *Proceedings of the Conference East EurOOPe'91*, pages 48–61, Bratislava, September 1991.
14. Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, Reading, MA, USA, 1989.
15. J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, and B. Magnusson. *Object-Oriented Environments – The Mjølner Approach*. Prentice Hall, Hertfordshire, GB, 1993.

16. Carine Lucas, Kim Mens, and Patrick Steyaert. Static Typing of Dynamic Inheritance. Technical Report vub-prog-tr-95-04, Programming Technology Lab, Vrije Universiteit Brussel, 1995.
17. Carine Lucas, Kim Mens, and Patrick Steyaert. Typing dynamic inheritance. Technical report, Programming Technology Lab, Vrije Universiteit Brussel, 1995. (Poster session at OOPSLA'95 Conference, October 15-19, 1995, paper available from ftp://progftp.vub.ac.be/ftp/tech_report/1995/vub-prog-tr-95-03.ps.Z).
18. Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley, Reading, MA, USA, 1993.
19. Mira Mezini. Dynamic object evolution without name collisions. In *Proceedings ECOOP '97*, LNCS 1241, pages 190–219, Jyväskylä, June 1997. Springer-Verlag.
20. Peter D. Mosses. *Action Semantics.* Cambridge University Press, Cambridge, GB, 1992.
21. Peter D. Mosses. *Action Semantics Home Page.* WWW, http://www.daimi.aau.dk/BRICS/FormalMethods/AS/index.html, 1997.
22. Patrick Steyaert and Wolfgang De Meuter. A marriage of class- and object-based inheritance without unwanted children. In W. Olthoff, editor, *Proceedings ECOOP '95*, LNCS 952, pages 127–144, Aarhus, Denmark, August 1995. Springer-Verlag.
23. David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings of the 1987 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 227–242, Orlando, FL, October 1987.