Method Mixins

Erik Ernst

Department of Computer Science
University of Aarhus, Denmark
eernst@daimi.au.dk

Abstract. The procedure call mechanism has conquered the world of programming, with object-oriented method invocation being a procedure call in context of an object. This paper presents an alternative, *method mixin* invocations, that is optimized for flexible creation of composite behavior, where traditional invocation is optimized for as-is reuse of existing behavior. Tight coupling reduces flexibility, and traditional invocation tightly couples transfer of information and transfer of control. Method mixins decouple these two kinds of transfer, thereby opening the doors for new kinds of abstraction and reuse. Method mixins use shared name spaces to transfer information between caller and callee, as opposed to traditional invocation which uses parameters and returned results. This relieves a caller from dependencies on the callee, and it allows direct transfer of information further down the call stack, e.g., to a callee's callee. The mechanism has been implemented in the programming language gbeta. Variants of the mechanism could be added to almost any programming language with mutable state.

1 Introduction

Mixins are well-known as a device that allows for decoupling of subclasses from their superclasses [7, 11, 6]. This paper presents a related concept, *method mixins*, allowing for a novel and in some ways more profound decoupling of behavior specifications than traditional procedure or method calls.

The basic idea is that a method need not be a monolithic entity, it may be constructed from smaller parts, method mixins, which may be reused to construct many different methods. A procedure call or method invocation may also be used to create a more complex behavior from existing, simpler behaviors, but method mixins represent a very different set of trade-offs.

There are several reasons for having method mixins in a language. With a traditional invocation, the caller depends on multiple characteristics of the callee, including the number and order of arguments. With a method mixin, the caller is completely independent of the callee. With a traditional invocation, the caller can provide extra information to a callee by adding an extra argument to the call; this means that *all* callees (e.g., a given method in many classes) must be modified to accept that extra argument. With method mixins, the caller may provide extra information freely; if there are a hundred callees and only one of them need the extra information, then the other 99 callees need not be changed. Traditional methods communicate with each other using arguments and shared state in objects or globally. Mixin methods support a novel scope for shared state which improves encapsulation and makes programs more robust in context of recursion or multiple threads. Moreover, mixin methods have the potential for better performance than traditional invocation. Finally, a novel and useful feature of method mixins is that they are able to create customized methods at run-time.

The contributions of this paper are: The basic concept of method mixins and their properties, a detailed comparison with traditional invocation, and a presentation of mixin methods as they have been implemented in gbeta. Moreover, we present a unified descriptive framework for variables and related concepts as channels; this is used to characterize the qualities of method mixins compared to other mechanisms, and to support the basic premises behind this direction of development.

The rest of this paper is organized as follows: Section 2 introduces the 'communication channel' point of view for a number of concepts including variables. Section 3 presents the basic principles for method mixins and compares them with traditional invocation mechanisms. The realization of method mixins in gbeta is presented in Sec. 4. Related work is discussed in Sec. 5, Sec. 6 describes the implementation status, and Sec. 7 concludes.

2 Variables as Channels

It is quite common to describe languages with mutable state as machine-oriented, and to describe functional, logical, and other kinds of 'declarative' languages as more abstract, liberated from the old-fashioned attachment to bits and memory cells. This opinion was brought to prominence with the 1977 Turing Award speech by John Backus [3], but see also, e.g., [16, p.45–47], [25, p.2–4], [12, p.36–37], and [13, p.8–10]. Note that Backus' language FP is restricted almost completely to the communication topology of Fig. 1, as described in Sec. 2.2.

We must recognize that the functional and other paradigms have have produced deep and useful results. However, it is our opinion that imperative languages, especially object-oriented ones, are being widely used because of their inherent power and flexibility, not because programmers are nostalgic about writing programs in assembly language. It is not a question of abstraction or hardware concreteness, it is about safety and freedom. Restrictive communication topologies bring safety, and flexible topologies bring freedom.

To substantiate this, we need to consider variables and similar concepts as communication channels, thereby making them comparable. Based on this, we venture into the project of *expanding* the role played by imperative variables, and exploiting their flexible communication topologies in a new way, namely in method mixins.

Variables and related concepts are value-carrying entities in the semantics of a language. They may or may not be mutable, and they may or may not have a name. Typical examples would be global variables in C [19] or Pascal [17], static variables in a C++ [28] class or method, instance variables in a Java [14] object, local variables in a Python [22] method invocation, but also function arguments and return values in Haskell [18], patterns in BETA [23], and structures in SML [24].

The ability of all these entities to carry information is *not* directly useful. It only becomes useful when information produced by one part of a program execution is *communicated* to another part, thereby influencing actions taken or values produced. These entities are only useful as communication channels.

They can work as communication channels because they support *sending* and *receiving* values. For an imperative variable, sending is assignment and receiving is evaluation. For a class in C++, the value is sent (defined) at compile time or link time, and received (used) at run-time whenever the name of that class is mentioned in the currently running code. SML structures may be created at run-time, but only at top-level, so as channels they behave like the C++ class. For more interesting channel behaviors we will discuss three important topologies below. However, we first need to describe the contexts in which communication topologies may exist.

2.1 The Topological Space

To have communication topologies we must establish a space[1] where they can exist. The first dimension we need is *time.* We also need to locate *places in source code.* We will assume that abstract syntax trees (ASTs) are available. Every interesting selection of source code will then be a set of AST nodes. A *scope* denotes a systematic selection of AST nodes, according to the rules of the language. Finally, a single piece of syntax may correspond to many run-time entities, e.g., one Java class may have many instances. Thus, the concept of *multiplicity*: The multiplicity of a global variable declaration is 1; the multiplicity of an instance variable declaration in a class is 'many'. The notion of an *enclosing* instance allows us to select an entity where the multiplicity is greater than one; for instance, the enclosing object for a Java method invocation is the object denoted by `this`. In summary, a communication takes place at a certain time, to/from a certain source code area, and within a certain instance. Our topologies will be characterized by means of constraints on these coordinates.

2.2 A Simple Communication Topology

Probably the simplest communication topology is the one in Fig. 1, namely the topology of returning a value from a function call.

The diagram to the right in the figure illustrates the source code dimension. Communication is illustrated by lines; a small bullet (•—) indicates a send operation, and an arrowhead ($\rightarrow$) indicates a receive operation. The operations are

[1] We do not define what it means to be an open set here. The notion of topology is closer to that which is commonly used in connection with computer networking.

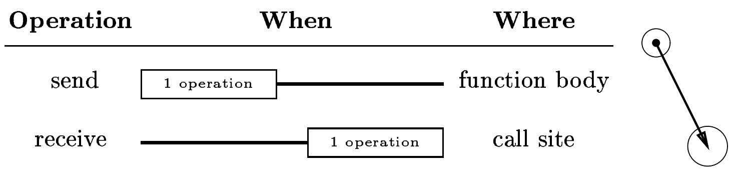


Fig. 1. Communication topology for a function return value

enclosed in circles, representing the source code area where the operations can occur. This area is described in the 'Where' column of the table; both 'function body' and 'call site' are single AST nodes. I.e., the channel can only be used for sending resp. receiving in one particular syntactic location. Finally, the time constraints are illustrated in the 'When' column. A bar indicates that the operation is allowed, and a line indicates that it is not. Time progresses from left to right, so there will first be a phase with exactly one send operation, and then a phase with exactly one receive operation.

In summary, Fig. 1 illustrates that a function return considered as a channel has one send operation, sending the value of the function body, and then one receive operation, used in expression evaluation at the call-site. There is one instance (invocation) associated with each operation, although the figure does not make this explicit. Note that the channel has no name, which forces each operation to occur in one particular AST node. We could say that this channel has an extremely narrow scope.

This kind of channel is also used for temporary values, e.g., to communicate the value of subexpressions into the evaluation of a composite expression. It is syntactically very lightweight and ensures very disciplined communication patterns. In other words, programmers may ignore it, and use it constantly.

2.3 A Safe Topology

Figure 2 illustrates a communication topology that is widely used and even enforced in some situations or languages. A good example is binding a function argument to a value for a function invocation in Haskell.

In this case the channel is a function argument—so it has a name—and it is subject to a strict discipline. Again, there is exactly one send operation. It takes place after evaluation of the actual argument, just before the function call.[2] Later there will be zero or more receive operations, one for each use of the formal argument name during evaluation of the function body.

The diagram to the right contains one bullet, indicating the send operation at the call site, enclosed in a narrow scope consisting of a single AST node, namely the actual argument expression at the call site.[2] The large oval represents

[2] Currying and pattern matching complicate matters, but there will still be one point in time and one AST node where the argument is bound to a value.

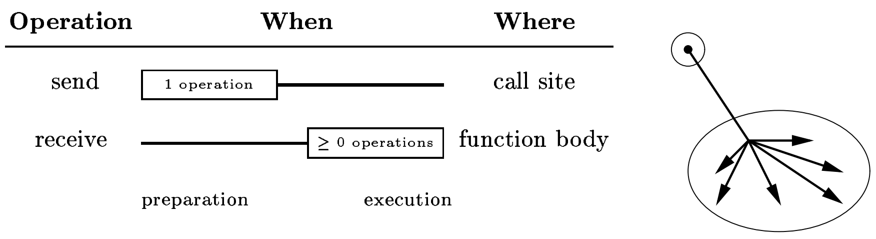


Fig. 2. Communication topology for a functional argument

the function body, i.e., the scope of the formal arguments. Several arrowheads illustrate that the argument may be evaluated anywhere in the function body.

This kind of channel is very useful because we may interpret it as follows: Each receive operation provides exactly the message that was sent from the call site to the current function invocation. Reasoning about function arguments becomes easier if programmers maintain this discipline. If the language enforces the discipline, as in Haskell and other functional languages, it allows for lazy evaluation, as well as Hindley-Milner style parametrically polymorphic type systems with type inference.

However, there is a non-trivial cost that programmers must pay when they accept such a strict discipline, namely the loss of communication flexibility. We believe that the very general and flexible communication topology described in the next section is so useful for programmers that this alone largely explains why pure functional languages have not conquered the world of programming.

2.4 A General Communication Topology

The communication topology supported by an imperative variable is much more general and flexible than the previous two. Figure 3 shows this topology.

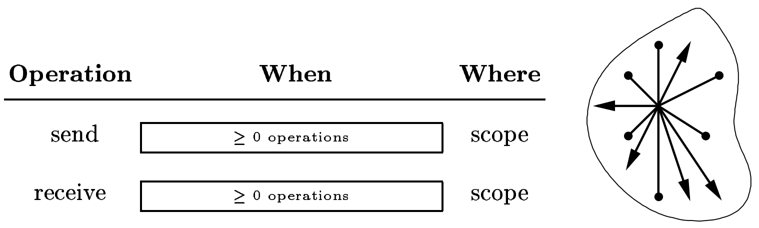


Fig. 3. Communication topology for an imperative variable

This topology is quite simple to describe. The bars in the 'When' column indicate that both send and receive operations are allowed, anytime. The diagram to the right illustrates that every location in the scope of the variable can perform both operations. Typically there is also an instance constraint. E.g., assignment to an instance variable in Smalltalk only affects the current object.

This topology is very flexible: A channel may be used to send several messages; it may be used to send messages from several places in the scope; and it may be used to send messages to different receivers. There is no need to change the declaration of the channel in order to extend or modify the topology. It also supports shifting between multiple topologies dynamically.

However, imperative variables are broken channels! Every send operation may potentially destroy a previously sent value that should have been received somewhere. And there is no built-in support to ensure that the first receive comes after the first send. Data-flow analysis, along with default initial values or initialization expressions improve the situation, but it is not perfect. It would be very interesting to improve on imperative variables as channels, but for now we just conclude that the power and flexibility of a general communication topology is so useful that programmers in large numbers choose to handle these problems. However, there are two other problems that programmers may actually manage by judicious program design:

- An imperative variable is a resource. The larger the scope and the lower the multiplicity, the more likely it is that there will be conflicts. To avoid them, coordination is needed among usage locations.
- The power and flexibility that lets us express sophisticated communication topologies when we need them may trick us into creation of unnecessary complexity.

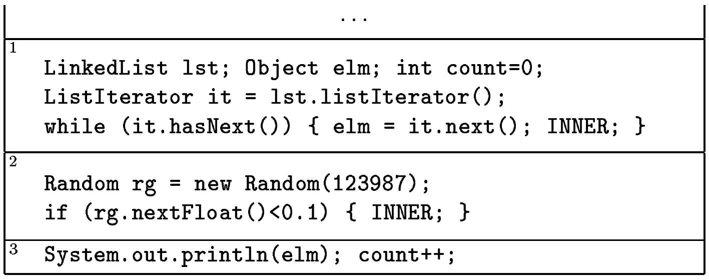
The entire effort called object-orientation could be described as a solution to these problems—not a perfect one, but a hugely useful improvement.

First, by supporting expression of smaller entities than a program, namely objects, it allows programmers to concentrate on a manageable chunk of complexity. It lets a group of conceptually related procedures communicate with each other, based on a set of imperative variables that are explicitly reserved for this purpose. Encapsulation of the internal complexities by means of a simple and useful interface ensures that this step reduces the global complexity, and subtype polymorphism greatly enhances this effect. Second, by separating the notion of class and object (or by providing a `clone` primitive as in Self [1] and other classless languages) it supports the creation of multiple instances of those sets of variables. Along with this goes the transition from procedures to *methods*, where a method is a pair consisting of a procedure and an object. These two features let object-oriented programmers use the power of imperative variables, and still keep the complexity and the resource conflicts under control. This is the approach that gives priority to freedom of communication; 'declarative' languages generally give priority to the safety, and object-oriented programmers must work a little harder to get that.

In this paper, we seek to improve the safety while preserving the freedom. We do this by introducing a new and even more narrow scoping mechanism than the object, thereby letting several behavioral entities communicate with each other by means of shared state, created for each individual invocation of that group of behaviors. This makes the approach deeply object-oriented in nature! The next section explains how it works.

3 Method Mixins

A method mixin (MM) is equipped with behavior and local state, just like a procedure. Moreover, it is associated with an enclosing object, just like a method. However, it is not invoked by name, and it does not accept arguments or return results. It communicates with its caller by means of access to shared state. Methods of the same object may communicate with each other by means of shared state in that object, but the shared state of a group of MMs is not declared in an enclosing object, it is declared directly in the MMs. We will use a graphical notation to explain the basic principles of MMs, as shown in Fig. 4.



```
...
1
   LinkedList lst; Object elm; int count=0;
   ListIterator it = lst.listIterator();
   while (it.hasNext()) { elm = it.next(); INNER; }
2
   Random rg = new Random(123987);
   if (rg.nextFloat()<0.1) { INNER; }
3  System.out.println(elm); count++;
```

Fig. 4. Three method mixin invocations $\mu_1 \ldots \mu_3$ on the stack

We assume a run-time context, i.e., a running program capable of composing and invoking MMs. The graphical notation then shows a run-time execution stack, growing downwards, with a number of currently executing MMs on it.

Each mixin method invocation (MMI) is separated from neighboring MMIs by horizontal lines. An MMI is actually very similar to an activation record, but for readability we show declarations and source code where storage cells containing values of the declared variables and pointers to compiled code would have been more precise. The example code is Java-like for readability, we just added an "`INNER`" statement. In this section we describe MM principles. Section 4 describes MMs as they have actually been implemented in `gbeta`.

Figure 4 shows three MMIs marked 1, 2, and 3; we will use the notation μ_1, μ_2, and μ_3 when referring to them. Most of the code executes like in standard Java. The most visible difference is that we may now and then execute an `INNER` statement; the effect of this is to call the next MMI further down the stack. For instance, μ_1 will call μ_2 whenever it executes `INNER`. This 'call' operation denotes a jump to the beginning of the body of μ_2 and, when μ_2 terminates, a return to just after the `INNER` statement in μ_1. However, no transfer of arguments takes place, nor does μ_2 return a result. If `INNER` is executed from the bottommost MMI, it does nothing.

Apart from the `INNER` statement, the crucial difference between standard Java semantics and mixin method execution is that the body of an MM can be included in the scope of declarations in other MMs. For instance, μ_3 uses the names `elm` and `count`, which are declared in μ_1. Note that μ_2 does not use any names declared in μ_1 or μ_3, but it declares local state.

We may now describe the effect of executing these MMIs: μ_1 contains generic iteration code that accesses the elements in the list, and calls μ_2 once for each element. In μ_2, the `if`-statement will randomly execute its body about 10% of the time; if so, μ_2 calls μ_3. In μ_3 the current list element is printed and `count` incremented. In summary, this MMI group randomly selects about 10% of the elements in the list and prints and counts them.

When designing MMs it is necessary to decide on the scoping rules—exactly when is a given MMI μ_j included in the scope of the declarations in another μ_i? One possible choice is to use dynamic scoping rules. Whenever an MMI uses a name N that is not locally defined, the entire call stack is searched for an MMI that defines the name N. This design would be very hard to make type safe, and we believe that it would have rather obscure and surprising semantics. For this reason, MMIs are grouped and every scope among MMIs is limited to be inside each group. For instance, μ_1, μ_2, and μ_3 could be a group, and no lookup operation from any of these could then look into an MMI outside this group. Note that each MMI may also be in scope of enclosing objects, it is just never in scope of any MMIs beyond its group.

Moreover, in order to enable static type checking, it must be ensured that each MMI μ_j will only be a member of such groups that every name used by μ_j is defined by some other member μ_i of the group. In gbeta we have chosen to express MMs in such a way that every MM at the declaration site is statically associated with exactly those other MMs from which it may use declared names. It is ensured that every MMI group at runtime contains the necessary fellow members for each of its members; this is actually trivial, because it is impossible to express an MM isolated from those that it depends on.

Another design decision is about the life-times of MMIs. Obviously, if μ_j depends on state in μ_i then μ_i must exist at least at every moment where μ_j exists. It would be sound to allow any kind of life-times that respect this criterion. In gbeta we have chosen a more restrictive approach: an MMI group can be extended, but no members can be taken out once they are in. This is because other properties of the language—in particular virtual patterns and reification of method invocations—would make the type analysis unsound if it were possible to remove an MMI from a group.

At this point we can compare MMs with ordinary methods from several different points of view, and that is what we will do in the next section.

3.1 Method Mixins vs. Ordinary Methods

MMs are a supplement to—not a replacement for—ordinary methods. These two devices represent so different trade-offs that it makes sense to allow programmers to make the trade-offs explicitly. To be able to characterize the purposes for which each device is most suitable we must consider two different points of view:

External Once we have written an entity that solves a particular problem, it may be reused many times by being accessed from many different contexts. This point of view emphasizes the use of a *preexisting* entity; we could call it *passive* reuse.

Internal An entity may have a general structure that allows it to solve many different problems, if only we are able to express the variants. This point of view emphasizes *creation* of variants of a flexible entity; we could call it *active* reuse.

To apply this to behaviors, consider a method or MM 'm'. Passive reuse focuses on being able to call m flexibly; active reuse focuses on flexibly controlling what methods m calls.

The two kinds of reuse complement each other. Passive reuse is governed by an explicit interface and is often easy. However, it does not help much if a suitable entity is not available. Active reuse may then allow us to *build* the kind of entity that is needed. Active reuse can be more difficult than passive reuse, in the sense that it usually requires more knowledge about the internal structure of an entity to create a specialized variant of it. In return for this investment, active reuse gives us flexibility. It is then very important that the active reuse mechanisms are flexible.

It is also reasonable to say that passive reuse is a large scale mechanism, because a reusable entity could be reused from anywhere in a large scope, e.g., an entire program. On the other hand, active reuse is focused on the internals of the entity that is being specialized.

We believe that ordinary methods are the most suitable choice for passive reuse, and MMs are more powerful for active reuse. In the following we will compare MMs and ordinary methods from various different points of view, thereby also exposing their weak and strong points.

Communication. Ordinary methods may communicate with each other by means of arguments and returned results, or by means of shared access to mutable state in some reachable entity, e.g., global variables or instance variables in an object. Note that usage of shared state in an object or globally is error-prone because of the resource conflicts that may arise if recursion or multiple threads causes more than one invocation of those methods to exist simultaneously.

Arguments may be used to transfer information from a caller to an immediate callee, and return values may be used to transfer information back from the callee to the caller. If an argument is of reference or pointer type then it may be used to establish aliased access to mutable state. Aliases may be used in a similar way as other kinds of shared state. They may have fewer resource conflicts than object state or globals, but they require declaration of arguments in all methods on a contiguous part of the stack, even if some methods do not use that shared state (for an example, consider `g` below).

MMIs communicate with each other by means of shared state in the MMIs themselves; they may of course also use shared state elsewhere. Communication via local shared state is considerably more flexible than communication via arguments. In particular, communication may happen across any number of possibly oblivious MMIs. An example of this, in Fig. 4, is the ability of μ_3 to use entities declared in μ_1 across the μ_2 that does not depend on the other two.

If we had used ordinary method invocation then we would have had a method `f` (corresponding to μ_1) calling another method `g` (μ_2) which would again call method `h` (μ_3). All information delivered from `f` to `h` would then have to be delivered either as an argument to `g` that `g` would just pass on to `h`, or via shared state in an object or globally. In particular, if we were already using `f` to call `h` somewhere then we could not introduce `g` between `f` and `h` without changing both `f` to call `g`, and `g` to transfer all the information that `h` needs.

Callee dependencies. Consider the situation where an ordinary method `foo` is called. This is achieved by writing the name `foo` at the call site, along with an explicit argument list. The call may also be used as an expression if `foo` returns a result. This means that the caller depends on the number, order, and types of arguments; on the name of the callee; and on the result type of the callee. Methods are not first class entities in many object-oriented languages, including Java and Eiffel, so the only way in which the implementation of `foo` may vary is by late binding. In other languages such as C++ we may also use function pointers or similar devices, but the dependency on the signature of the callee is common to all cases.

MMs are very different in this respect. An invocation of an MM is simply the keyword `INNER`, and the caller is entirely independent of the callee. There is no mention of a name for the callee; there are no arguments; and there is no returned result. It is even possible that there *is* no callee, in which case `INNER` will do nothing. This means that an MM is in a sense maximally flexible with respect to the callees that can be attached to it.

Caller dependencies. Consider the task of writing an ordinary method `bar`; part of the job is to specify the exact arguments and their types, as well as a return type. This works well as an interface to the callers because it is explicitly defined and it allows the body of the method to use information received from callers without knowing anything about them. Hence, an ordinary method does not depend on its callers.

On the other hand, if `bar` needs more information in order to carry out the prescribed task then there is no way to get more information out of the callers except for changing the interface of `bar` and then changing all call sites, e.g., to add some new arguments. This is especially laborious if the `bar` we are working on is one of many implementations of `bar` in many different classes—

all those method implementations must then have the new arguments added to their declarations, even if few of them use it.

An MM is again very different. It may depend on declarations in one or more of its callers. This means that an MM may not be called in nearly so flexible ways as an ordinary method—it must be called directly or indirectly from the MMs that it depends on. It would provide greater flexibility if we used structural equivalence to determine whether an MM could or could not be called from some other MMs, but it could also easily become hard to understand the effect of the body of an MM if it might be used in many different contexts that just happen to agree on some declared names.

On the other hand, the MM uses names in its callers one by one, and it is not affected by changes applied to the names that it does not use. For instance, we could add a new declaration to a caller and then use that name in one or more callees; all the other callees could remain unchanged.

In summary, an MM depends on an inferred (partial) interface of some of its callers, and different MMs using the same callers may use them via different inferred interfaces. In contrast, an ordinary method uses information exclusively from the immediate caller, and via an explicitly defined interface. Different methods called from the same call sites must agree precisely on the interface that they present to those call sites.

Encapsulation. Ordinary methods are encapsulated in relation to each other, in the sense that a caller knows nothing about the callee except for its name and signature. However, the encapsulation of a group of methods that have a need to communicate with each other may be broken by the reliance on external shared state. When a method changes the state of its enclosing object it may be conceptually well-motivated, but when object state or global state is used mainly for the transfer of information between method invocations then it becomes error-prone.

For inter-method communication, MMs support an exact match between the involved methods and the life-time and scope of the state that is used for this communication, hence they enable an improved encapsulation in this area. On the other hand, MMs are able to use the internal structure of each other in a very unencapsulated manner. This is because they are optimized for gray-box reuse of each other. This also supports the claim that MMs are appropriate for active reuse, whereas ordinary methods are oriented towards passive reuse.

Note that it is possible for MMIs to contain state that is used throughout the life-time of the MMI group, but not depended upon by all members. An example is `rg` in μ_2 in Fig. 4, which influences the overall behavior of the group but is neither in scope of μ_1 nor of μ_3. As in the paragraph about communication, assume that we were to achieve a similar effect with ordinary methods. Then `rg` should either have been declared in `f` (corresponding to μ_1 as earlier) and delivered to every invocation of `g` (corresponding to μ_2), or `g` should have relied on external state, e.g., in an object. In C++ we could have used a `static` variable inside a method, but due to the multiplicity of 1 that would give rise to resource conflicts in cases where more than one instance of `g` is executing at the same time. In this sense the MMI group is also better encapsulated than the ordinary methods, because it is possible for μ_1 and μ_3 to depend dynamically on state that matches them in life-time and multiplicity, but to be oblivious of that state.

Life-time. Procedure and method invocations have nested life-times: A caller lives at least as long as any of its callees, and possibly much longer. The number of invocations on the run-time stack is unbounded. This makes it possible to organize an entire program execution in terms of traditional invocations.

MMs are composed into groups and then invoked as a unit. Different design decisions could be made with respect to the ability to add or remove members from this group during the execution; gbeta supports addition of new members, but execution must then be restarted, so it is recommended to finish building the MMI group before executing it. When the topmost member of the group terminates, the whole MMI group is considered terminated.

Hence, MMs can not be the behavior structuring principle for an entire program execution. MMI groups are expected to consist of a few members, perhaps up to a few tens of members. MMs allow for a deeper and more flexible composition than ordinary methods; on the other hand it is probably best to require that they align their life-times. As we saw above, this may improve encapsulation.

Call topology. An ordinary method may call any number of methods. An MM may only (directly) call one other MM by means of the `INNER` statement. We have considered ways to relax this restriction, but not yet found a good design. As a work-around in gbeta, we may create any number of nested virtual methods in a given MM, and they may then invoke `INNER` and allow for addition of further MMs for more than one "`INNER` site" (virtual method call site). Moreover, MMs may of course use ordinary method calls in their bodies.

Kinds of entities communicated. A method argument may only be an entity of certain kinds in some languages, usually known as "first-class" entities. For instance, a Java method cannot receive a method as an argument, and a class can only be given as an argument if it is accessed via the reflection system, which means that type safety is lost and special syntax must be used to create instances of that class, etc. With MMs, the communication takes place by means of shared name spaces. Hence, anything that can be declared can also be communicated.

Name spaces. An ordinary method provides a name space by means of the formal argument list, and it is up to the caller to initialize this name space. It is possible in some languages to define default values for some of the arguments (argument number N and upwards for some N). This makes it possible to call a method with fewer arguments than the argument list contains, corresponding to the situation where the caller "does not say enough" about the arguments. Note that this only provides information to the callee that the callee already had *statically* in its declaration.

An MMI cannot be called from a group of callers that "do not say enough", it is simply prohibited to access a name that is not known to exist in a caller (unless the name is declared in an enclosing entity). It is not easy to relax this restriction and preserve type safety. However, the fact that an MM may choose to use only some of the declared names in statically known callers allows another phenomenon, namely that the callers "say too much". This enables the callee to expand the usage of the *dynamically* provided information at the call site without having to synchronize with all the other callees. As usual, MMs emphasize the ability to flexibly enhance the effect of its callers, whereas ordinary methods emphasize the ability to be called from arbitrary call sites.

Invocation. An ordinary method is invoked by calling it. It does not involve other methods, except if it takes one or more arguments of type 'function pointer' (or whatever it is called in the given language) and calls that.

MMs must be composed into an MM group which is then invoked. One of the MMIs is positioned as the topmost one, and that one is called when the group as a whole is called; the others may or may not be called using `INNER`. MMs must be composed from the outside, because otherwise none of them have any way to determine which MM to call when `INNER` is executed.

Because it makes a given MM call another one, the operation of composing MMs may be compared to a higher-order function in a functional language. However, such a higher-order function would behave similarly to a traditional call chain as described earlier: it could *not* make a function `f` call another one `g` that again calls a third one `h` such that `f` transfers information to `h` without explicitly passing this information through `g`.

Performance. Invocation of a routine (procedure or method) will of course include a jump-subroutine instruction as well as a return instruction at the end, but that is just one of many tasks associated with standard invocation. Significantly, it also requires setting up an activation record, including setting up actual argument values in that activation record. It also includes saving registers in memory, with caller-save registers saved just before the jump and callee-save registers saved just after the jump; the registers must also be restored at the end of the invocation. Finally, there may be other tasks to perform according to the standard calling conventions of the platform.

Leaf routines (routines that do not call other routines) may sometimes be invoked without setting up an invocation record, i.e., the invocation may be reduced to essentially a jump and a return instruction. However, this can only be done if it is statically known at the call-site that the callee is a leaf-routine, that does not use (much) local state. Even then it may be necessary to save/restore some registers.

Moreover, if the callee is statically known then it may be possible to inline the call, i.e., to compile the routine directly into the call site and then avoid the transfer of control entirely. The main difficulty with this is that it is incorrect to simply transfer source code from one binding environment to another, so one must generally either restrict inlining to be used with a certain specific set of method shapes (e.g., a forwarding method, just containing one statement that calls another method with arguments that satisfy some constraints, and other special cases) or one must apply some kind of source code transformation that compensates for the change in binding environment, which is then restricted to cases where such transformations are known to be correct. Moreover, it is often inappropriate to inline everywhere possible, because this may cause a program size explosion. If inlining is possible and desirable, it usually improves the opportunities for various other optimizations, including avoiding some save/restore cycles for registers and allowing better register-allocation, but also other things like loop reorganizations or common subexpression elimination.

For MMs, the situation is different. If the MMIs in a group have the same life-time, it is possible to set up an activation record for the group as a whole. If the group is statically known (which might be rather common because of the nature of MMs) then the size of this combined activation record would be known statically, including all the offsets of local state used in all member MMIs, and it is then possible to compile the entire MM group as a whole. This is a multi-level inlining operation. This is often appropriate, because MMs are optimized for flexible customization of specific callers, i.e., they are not expected to be called from very many places. The reason why inlining is easier with MMs than it is with ordinary routines is that they generally share the same binding environment. It is possible that some members of an MMI group are located in different objects, but this just means that there will be several '`this`' pointers in the combined activation record; it is statically known which one to use.

However, it is of course not always known statically what members there will be in an MMI group. The process of setting up a common activation record still happens only once, even though some MMIs in the group may be called many times. If members are added to the group it must be extensively reorganized; this is not a cheap operation, but probably occurring only rarely. Since each MMI may have to be able to access the state of some other MMIs, it is generally necessary—in this dynamic case, not in the static—to equip each MMI with pointers or offsets to the MMIs that it depends on. Note that these inter-MMI pointers or offsets are similar to static links in languages that support nested procedures, e.g., Pascal. Since we only need these pointers when the MMI group is not known statically, it is generally necessary to generate different code for the two cases. The dynamic code is generated per MM, and the static code is generated per MM group. In order to minimize code size, it would be possible to use the dynamic code in the static case, at a reduced performance because of indirect cross-MM state accesses.

If an MMI μ_j only uses state that is present in another MMI μ_i (same enclosing object and no local state in μ_j), then μ_j may be executed with the stack pointer pointing to μ_i; μ_3 and μ_1 in Fig. 4 could be an example of this. No matter what or how many intervening MMIs there are, this will give the correct semantics, given that the code for μ_j was compiled under this assumption. This is actually a case where the static performance can be achieved without static knowledge about the entire MMI group. Note that this situation is similar to the situation with a leaf routine because invocation is reduced to essentially a jump-subroutine and a return, and possibly some register save/restore cycles.

The important difference is that this μ_j need *not* be a leaf, and the μ_i need *not* be compiled with knowledge about whether a callee uses this optimization.

The current implementation of gbeta generates and executes byte code, and the released gbeta only supports the dynamic (per MM) kind of code generation. However, there are good reasons to expect MM invocation to have better performance than ordinary invocation:

– An MM may reuse the same activation record in many calls, and there is never the cost of transferring arguments or returning results.
– An MM is a very good target for inlining, because of the largely shared name spaces and the fact that MMs are by nature not likely to be called from very many places.
– Even if an MM can not be inlined, it may often be possible to call it in a similar manner as a leaf routine.

In summary, the nature of an MMI group—somewhere between a single method invocation and a set of n method invocations where n is the size of the group—makes it more flexible than a traditional method invocation, but not as expensive as n methods calling each other.

4 Method Mixins in gbeta

The language gbeta [8, 10] supports a version of MMs that is very tightly integrated into the language: Every method is created by composing MMs. Even the signature (arguments and return values) can be specified incrementally; this may happen if more than one MM specifies an argument list or a list of returned results. However, we will only use single-MM signatures in this paper.

The MMs in gbeta always occur in lists of MMs, and each MM can never exist without being accompanied by at least those MMs that it depends on. These lists can be composed by means of the operator '&'. When two lists A and B are merged, as in $C = A \,\&\, B$, the merged list C will contain all the elements of each argument, preserving the order. This is achieved by means of a linearization algorithm [4, 9]. Since linearization is not the topic of this paper we will refer to [9] for details, and specify the result of each merging below.

To improve the readability for readers who are not so familiar with the language gbeta, we have adjusted the syntax in examples such that it becomes more Java-like. In particular, argument lists are written outside method bodies, assignment statements are written like `x=y` for an assignment to `x`, and method invocations are written with argument lists after the name of the method, just like Java and other main-stream languages.

Fig. 5 shows a few 'archetypes' of MMs which are useful in many different contexts. In the figure, $\mathcal{A}$ and $\mathcal{A}'$ stand for pieces of code that do not contain `INNER`. Similarly, $\mathcal{B}$ is some boolean expression that does not contain `INNER`.

The first archetype is very common. It is suitable for sequential composition of behavior. If $M_1, M_2 \ldots M_n$ are method mixins of this type then the merged result $M_1 \& M_2 \& \ldots M_n$ will contain all the MMs in the order M_1 to M_n, and

```
1  { 𝒜; INNER; }
2  { INNER; 𝒜; }
3  { 𝒜; INNER; 𝒜'; }
4  { if (ℬ) INNER; }
5  { while (ℬ) INNER; }
```

Fig. 5. Five mixin method archetypes

they will execute the actions in M_1, let us call them $\mathcal{A}_1$, followed by the actions $\mathcal{A}_2$ in M_2, etc., up to $\mathcal{A}_n$. In other words, this archetype allows programmers to compose sequences of operations on whatever shared state these MMs may have, simply by composing the MMs in the same order. The MMIs on the stack would look as follows:

...
1 $\mathcal{A}_1$; INNER;
2 $\mathcal{A}_2$; INNER;
· ...
n $\mathcal{A}_n$; INNER;

For the second archetype, a similar merging $M_1\&M_2\&\ldots M_n$ will execute the actions $\mathcal{A}_n, \mathcal{A}_{n-1}\ldots\mathcal{A}_1$, i.e., in *reverse* order compared to the order of the MMs.

Note that the merging operation does not depend on whether it is composing MMs of type 1, type 2, a mixture, or something entirely different. This allows the MM writer to specify composition order rules that the MM composer may later use without having to know about them.

The third archetype combines the two first types by having actions both before and after the `INNER` statement, and the result is that the actions are executed in a "parenthesized" order: $\mathcal{A}_1$, $\mathcal{A}_2$,$\ldots\mathcal{A}_n$, $\mathcal{A}'_n$, $\mathcal{A}'_{n-1}$,$\ldots\mathcal{A}'_1$. In other words, each MM is a wrapper around the rest of the MMs. A wrapper may be used to manage resources, e.g., by acquiring a lock on a database table before `INNER`, and releasing it afterwards.

The fourth and fifth type of MM can be used to control the execution of other MMs. The MMI marked with the number 2 in Fig. 4 was of this type. A variant of the fifth type is used in Fig. 6.

Figure 6 shows an example that demonstrates the usefulness of being able to compose methods from method mixins. The auxiliary method `uneq` returns true iff the arguments given to it are "too different". The next method, `ns`, plays the role as a *name space* for a range of computations; it accepts a `float` argument `a`, calls `INNER`, and returns a `float` result in `r`. The idea is that various functions can be implemented by adding MMs using `a` and producing a result in `r`.

The following three declarations specify MMs that depend on `ns` and may be used to implement such functions. The MM `iterate` depends directly on `ns`; it provides a new local variable `g` and it executes the following MMs repeatedly as long as `g` is too different from `r`. The loop invariant is that `g` is the old guess,

```
float uneq(float a, float b):
  { return abs(a-b) > abs(0.0001*b); };
float ns(float a):
  { float r=a; INNER; return r; };
iterate: ns
  { float g=1; INNER; while (uneq(r,g)) { g=r; INNER; }};
step: iterate
  { r = g-(g*g-a)/(2*g) }
peek: iterate
  { print("\nGuess: "+g); INNER; };
```

Fig. 6. Computing square roots using Newton's method

and `r` is the new guess. When the two guesses are "equal", the result is assumed to be sufficiently precise and the iteration stops. Note that `iterate` may be used with many different MMs, they just need to be able to produce a better guess in `r` than the one currently in `g`.

The next MM, `step`, can be used as the algorithm core, producing a new guess in `r` from the old one in `g`. We may execute the whole algorithm as follows:

```
float result = (ns&iterate&step) (1000);
```

However, since `step` depends on both `iterate` and `ns` it cannot exist without the two, so we get the same effect by simply executing `result=step(1000)`. Here, `ns` and `iterate` are implicitly included. It may be a matter of personal preference whether to use one or the other style, but it seems that at least an explicit declaration and a clear choice of naming is appropriate:

```
sqrt = ns & iterate & step;
```

We may then execute `result=sqrt(1000)`. Note that even if `sqrt` is used as a method of a class whose instances are accessed polymorphically, the code for `sqrt` may still be *generated* with static knowledge about all the included MMs, i.e., with best-case performance. The dynamic case is when the MMs that we *compose* into a group are not statically known. Finally, we may create a variant of `sqrt` that prints the current guess for every step in the iteration:

```
debug_sqrt = peek & sqrt;
```

An execution of this method would have the following MMIs on the stack:

	...
1	`float r=a; INNER; return r;`
2	`float g=1; INNER; while (uneq(r,g)) { g=r; INNER; }`
3	`print("\nGuess: "+g); INNER;`
4	`r = g-(g*g-a)/(2*g)`

As we can see, the guess in g is printed just before the new guess in r is computed, because of the code in μ_3. It is important for getting the right semantics that the MMI from peek is located as number 3. This happens because peek depends on number 2, iterate, and because we composed debug_sqrt with the arguments in the given order. So, in order to control behavior composition in more sophisticated ways it is necessary for programmers to take a look at how linearization works—at least in languages where MMs are composed using linearization. However, we think that it is rather easy with a little bit of practice.

We could have applied the method modification to an invocation via some polymorphically accessed object someObject. It would look as follows:

```
float result = (peek & someObject.sqrt) (1000);
```

This would be the dynamic case, where it is not known during code generation exactly what MMs will be in the resulting method at runtime. Consequently, we should expect to pay a certain penalty in performance when executing this dynamically created method, because access from the core algorithm to a and g would have to happen indirectly.

5 Related Work

Since the comparison between MMs and ordinary methods was the main topic of Sec. 3.1, a kind of related work has already played an important role. However, some other mechanisms still need to be considered.

First, gbeta is a generalization of the language BETA [23], and gbeta owes much to BETA. The unification of classes and methods into the more general concept of a *pattern* originated in BETA and has been upheld in gbeta. The INNER mechanism also originated in BETA, and the ability to access state in another MM in gbeta originates in the BETA inheritance mechanism. There are even very well established traditions for using BETA patterns in a way that is similar to some examples in this paper:

```
(# lst: @list(# element::text #);
do lst.scan(# do current[]->putline #)
#)
```

In this piece of BETA (and gbeta) code, the elements in the list lst are visited one by one and printed; lst.scan and the (#..#) block containing putline work in a similar way as the MMI groups that we have described.

However, the relation between the behaviors of a pattern and a subpattern has always been described as a *behavior specialization* relation in the BETA community. To our knowledge it has never been analyzed as a subroutine invocation mechanism. This is understandable, because BETA's equivalent of a gbeta MM is inaccessible on its own. Analogously, (class) mixins correspond to the difference between a class and a superclass in many languages, but it is still makes a difference whether or not a language actually supports mixins.

The result is that an MM (a pattern) in BETA is inescapably tied to the exact same group of MM callers (its superpatterns), in all contexts. In contrast, gbeta supports a flexible pattern composition mechanism, '&', that actually makes it possible to choose to *call* an MM in many different contexts. Suddenly it is much more like a method invocation.

In summary, several crucial elements of the framework of MMs have already been available in BETA for many years. However, we consider it a contribution in its own right to analyze the INNER mechanism as a method invocation mechanism, thereby showing that it represents a very different set of trade-offs than ordinary procedure or method invocations. Moreover, the generalization process that lead to the design and implementation of gbeta also contains a significant contribution to the notion of MMs, because much of the expressive potential of MMs is only available with the generalized concepts of gbeta.

A language mechanism that seems to be usable for similar purposes as MMs is the *macro* mechanism in Common Lisp [15]. Such macros are capable of manipulating source code (not individual characters, but syntax trees) in an extremely flexible manner, because it is based on program transformations by means of arbitrary user-defined functions. This very general transformation capability makes it hard to ensure that the output from a macro has known properties. E.g., in a host language that otherwise supports static type checking, such as gbeta, there would hardly be any other way to check typing properties of a macro call than to actually expand the macro. This indicates that macros are unlikely to be able to coexist with static typing, and at the same time be able to be expanded dynamically. MMs in gbeta can be composed dynamically, and they do not conflict with static type checking.

Dylan [26] macros are similar, but *hygienic*, which means that they do not allow names used in macro calls to be captured in a new binding environment associated with the macro definition, or vice versa. It is possible to 'intentionally violate' the hygiene, e.g., to let a binding of a name in a macro definition be used by expressions in macro calls, but this is considered to be a dangerous practice. In other words, macros are not intended to combine behaviors having shared access to a name space, which is a core idea for MMs.

The *composition filters* [2, 5] approach allows for very flexible adaptation of the effect of sending messages. Composition filters can be used to redirect a message to another receiver, they can be used to change the selector (method name), and they can be used to modify arguments in a message. There are also some special filters for other purposes, e.g., for delaying a message until some condition is satisfied. This approach is largely orthogonal to MMs, because method mixins are concerned with the *construction* of methods, whereas composition filters are concerned with *selection* of which method to call, and how.

Aspect Oriented Programming [21] is also an approach that aims to provide new abstraction mechanisms in programming languages. This approach generally emphasizes the ability to modify the meaning of an entity, e.g., a class or a method, without having to modify the syntax of that entity. This is called *non-invasive* modification, and the main benefit is that the modified entity may be used in client code without changing that client code. However, it is an inherent property of non-invasive modifications that they modify an existing entity and do not create a new one. This means that there is no way to express several different variants of a given entity and have them coexist in the same program. In other words, aspects do not support active reuse. Conversely, MMs are only invisible for client code if they are hidden by ordinary subtype polymorphism. Other than that, there are several areas of overlap. For instance, an MM of archetype 3 in Fig. 5 may be used as a wrapper around an existing method; that is similar to an `around` advice in AspectJ [20], except for the fact that an around advice cannot extend the signature of the target method. Archetypes 1 and 2 correspond similarly to `before` and `after` advices.

There has been much work on (class) mixins since the seminal paper [7] where mixins were established as a separate concept. We do not know of any other language than gbeta where mixins are used as composable method building blocks, so the connection between gbeta mixins and other mixins is not very close in context of this paper. However, we should mention a few points in this area.

In [27] the notion of a *mixin method* is introduced. Execution of a mixin method makes the receiver object change class and, e.g., obtain some new instance variables. In other words, mixin method invocation is the mechanism that is otherwise known as mixin application, and mixin methods are therefore unrelated to our concept of MMs (method mixins). On another related point, [11] presents an extension of a subset of Java with mixins, called MixedJava. In this paper they introduce the notion of an *inheritance interface*, specifying the requirements of a mixin on potential superclasses. In gbeta, the inheritance interface is expressed via the set of required mixins that each mixin must have available. In both cases the mechanism ensures statically that name lookup will indeed succeed at run-time. An (optional) explicit inheritance interface might be useful in gbeta, especially because it would not introduce a requirement for all callees to agree upon it, such as ordinary methods must.

6 Implementation Status

The language gbeta implements method mixins, and the source code is available[3] under the GPL license. The syntax in this paper has been adjusted to be Java like and we omitted some details like module import statements, but apart from that the examples correspond to actual running code in gbeta. We have experimented with inlining of entire MMI groups, and that is expected to be part of the next release. However, since gbeta generates and executes byte code (though not Java byte code), we have not implemented the machine code level optimizations that were discussed near the end of Sec. 3.1. This is exciting future work!

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

7 Conclusion

We have presented the notion of method mixins; compared their characteristics extensively with those of traditional subroutine invocation mechanisms such as procedure calls and method invocations; and argued that method mixins represent a substantially different trade-off, compared to the traditional mechanisms. In particular, method mixins are optimized for active reuse, i.e., the creation of variants of behaviors based on flexible building blocks; traditional invocation is optimized for passive reuse, i.e., invoking existing functionality as-is, from many different places. We believe that these two mechanisms supplement each other well. Some special characteristics that make method mixins attractive are as follows: With method mixin invocation, the caller is completely independent of the callee; with traditional invocation, the caller depends at least on the signature of the callee. With method mixins, communication is based on an inferred signature that allows callees to use different subsets of the available information; with traditional invocation the communication via arguments and returned results must be identical on all callees. With mixin methods, communication is free to occur across oblivious intermediate mixin method activations; traditional arguments only support communication between a caller and its immediate callee. Moreover, mixin methods have excellent performance perspectives. Mixin methods have been implemented in the language gbeta, where they are tightly integrated with other features of the language, but the basic ideas could be used in many other languages.

References

1. Ole Agesen, Lars Bak, Craig Chambers, , Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. *The Self 4.0 Programmer's Reference Manual.* Sun Microsystems, Inc., Mountain View, CA, 1995.
2. M. Aksit, K. Wakita, J. Bosch, and L. Bergmans. Abstracting object interactions using composition filters. *Lecture Notes in Computer Science*, 791:152++, 1994.
3. J. Backus. Can programming be liberated from the von Neumann style? *Communications of the ACM*, 8:613–641, 1978.
4. 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 OOPSLA'96, ACM SIGPLAN Notices*, volume 31, 10, pages 69–82, New York, October6–10 1996. ACM Press.
5. Lodewijk Bergmans. *Composing Concurrent Objects.* PhD thesis, Faculty of Computer Science, University of Twente, 1994.
6. Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In Rachid Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628 of *Lecture Notes in Computer Science*, pages 43–66. Springer-Verlag, New York, NY, June 1999.
7. Gilad Bracha and William Cook. Mixin-based inheritance. *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, 25(10):303–311, October 1990.
8. 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.

9. Erik Ernst. Propagating class and method combination. In Rachid Guerraoui, editor, *Proceedings ECOOP'99*, LNCS 1628, pages 67–91, Lisboa, Portugal, June 1999. Springer-Verlag.
10. Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *Proceedings ECOOP'01*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.
11. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, San Diego, California, 19–21 January 1998.
12. Linda Weiser Friedman. *Comparative Programming Languages – Generalizing the Programming Function.* Prentice-Hall International, Inc., 1991.
13. Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts.* John Wiley & Sons, Inc., 3rd edition, 1998.
14. James Gosling, Bill Joy, and Guy Steele. *The Java^{TM} Language Specification.* Addison-Wesley, Reading, MA, USA, 1996.
15. Jr. Guy L. Steele. *Common Lisp – The Language.* Digital Press, Digital Equipment Corporation, 2nd edition, 1990.
16. Ellis Horowitz. *Fundamentals of Programming Languages.* Springer-Verlag, 1983.
17. K. Jensen and N. Wirth. *Pascal User Manual and Report.* Springer-Verlag, 1978.
18. Simon Peyton Jones and John Hughes (eds.). Haskell 98: A non-strict, purely functional language. Electronic form, `http://www.haskell.org/onlinereport/`, 1998.
19. Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 1988.
20. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings ECOOP'01*, LNCS 2072, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.
21. 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.
22. Mark Lutz. *Programming Python – Object-Oriented Scripting.* O'Reilly & Associates, Inc., 2nd edition, March 2001.
23. Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley, Reading, MA, USA, 1993.
24. R. Milner, M. Tofte, R. W. Harper, and D. MacQueen. *The Definition of Standard ML.* MIT Press, 1997.
25. Chris Reade. *Elements of Functional Programming.* International Computer Science Series. Addison-Wesley, 1989.
26. Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language.* Addison-Wesley, Reading, Mass., 1997.
27. Patrick Steyaert, Wim Codenie, Theo D'Hondt, Koen De Hondt, Carine Lucas, and Marc Van Limberghen. Nested Mixin-Methods in Agora. In Oscar M. Nierstrasz, editor, *Proceedings ECOOP'93*, LNCS 707, pages 197–219. Springer-Verlag, 1993.
28. Bjarne Stroustrup. *The C++ Programming Language (Second Edition).* Addison-Wesley, Reading, MA, USA, 1991.