gbeta

– a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance

Ph.D. Thesis

Erik Ernst
DEVISE, Department of Computer Science
University of Århus, Denmark

Abstract

A language design development process is presented which leads to a language, gbeta, with a tight integration of virtual classes, general block structure, and a multiple inheritance mechanism based on coarse-grained structural type equivalence. From this emerges the concept of propagating specialization. The power lies in the fact that a simple expression can have far reaching but well-organized consequences, e.g., in one step causing the combination of families of classes, then by propagation the members of those families, and finally by propagation the methods of the members. Moreover, classes are first class values which can constructed at run-time, and it is possible to inherit from classes whether or not they are compile-time constants, and whether or not they were created dynamically. It is also possible to change the class and structure of an existing object at run-time, preserving object identity. Even though such dynamism is normally not seen in statically type-checked languages, these constructs have been integrated without compromising the static type safety of the language.

Contents

Chapter 1

Introduction

This thesis is about the design of object-oriented programming languages, viewed through the development of a particular language, gbeta, as a generalization of the language BETA. This generalization provides benefits in two main areas: the abstraction mechanisms are made even more expressive, and the run-time flexibility is improved without compromising the static type safety.

A recurring topic in this thesis is how the activity of programming is managed by human beings, and how the technical programming language design decisions can be put into perspective as being motivated, ultimately, by the effects they have on human beings who work with programs. This aspect of the thesis is of course quite subjective since it is concerned with matters which are far too complex to formalize, and it is therefore a personal message which is useful only to the extent that it fits into some other persons subjective view of these matters—in particular if it is not a perfect fit but rather a nagging partial fit that spurs rethinking of some otherwise unquestioned assumptions. However, this perspective is applied in context of a lot of technical content which constitutes the actual language design, so we will elaborate some more on that.

BETA already offers very strong abstraction mechanisms via the support for virtual classes in the context of general block structure; gbeta generalizes the foundation, building on a coarse-grained structural type equivalence and supporting multiple inheritance. With the tight integration of these features in gbeta, a concept of *propagating specialization* emerges. With propagating specialization it is possible to, e.g., combine aspects of families of classes: One class family aspect, `Conc`, might deal with concurrency control, and another, `Impl`, with implementation. The expression `Conc&Impl` would then combine the two class family aspects, first the families `Conc` and `Impl`, then by propagation the members of the class family, and finally by propagation the methods of the members. As a result, each method in each member of the class family can be equipped with several aspects using just one simple top-level expression.

One common trend in the development of gbeta from BETA is the support for many new possiblities normally associated with languages without static type checking, and doing this without destroying the safety guarantees provided by static type checking. Firstly, it is possible to create classes at run-time and use them just like other classes. Secondly, objects can be specialized dynamically, i.e., an existing instance of a class `C` can be morphed into an instance of a more specialized class `C'` without disrupting the object identity. Thirdly, it is possible to inherit from virtual classes and to specify specialization relations between virtual classes such that classes which are not known at compile-time are still known to have certain well-defined *relations*. Finally, by means of inheritance from class variables it is also possible to inherit from a class which is constructed dynamically.

All these dynamic constructs are unusual in the context of a language with static type checking, but they are integrated in such a way that they do not disrupt the type safety of other constructs. As an analogy, consider a simple functional language supporting only multiplication of integers modulo some prime number p, having no run-time errors. Now add a division operator. With the enhanced functionality comes a new run-time error, "`Divide by zero!`", but the multiplication operator is still safe, and even though division may fail it will not produce ill-defined results—it will either fail immediately or produce results which are every bit as sound and safe as all other numbers. Returning to `gbeta`, an example would be that the creation of a new class may fail due to a well-formedness criterion which cannot always be checked during static analysis, but when creating instances of a class it is equally safe whether or not that class was created dynamically. Moreover, all constructs—including the new, dynamic ones—are statically checked with respect to name lookup, so `MessageNotUnderstood` errors cannot occur.

1.1 Readers Targeted

An obvious purpose of writing a PhD dissertation is to obtain the degree, hence it would be conceivable to target only the few, selected specialists in the topic area who are involved in the graduation process. However, my motivation for doing research is to improve the state of the art, in order to obtain profoundly improved solutions to known problems, to go beyond the realm of known problems into the realm of new possibilities, and—last but not least—to experience the joy of creation and collaboration around creation. For this, the natural target is the computer science world at large.

To mediate between such a highly specialized group of readers and the world at large I decided to describe the expected reader of this dissertation as follows: A computer science professional or student who ...

- is interested in programming language design and implementation.
- does not necessarily know BETA, but knows some object-oriented language at least from a user's point of view.
- has some common knowledge about the object-oriented tradition, such that slogans like "code reuse is good" or "conceptual modeling is good,

code reuse is just a derived benefit" make sense and possibly generate some arguments against or in favor.

This description outlines the background topics assumed to be well-known, hence it describes information deliberately missing from this dissertation. From a positive point of view, the contributions of this work would be of interest for specialists who are working with the design, implementation, or specification of statically typed object-oriented languages, specifically ...

- inheritance mechanisms, virtual classes, genericity.
- general block-structure (inner classes), advanced name-binding (scoping).
- method combination, class combination, systematic propagation of such.
- classes and methods as more-than-usual first class entities.
- dynamic classes and dynamic object specialization (extension).
- type analysis for such systems.
- the trade-off between name equivalence and structural equivalence.

Moreover, crossing the border of "just" language design into a broader topic area, the results presented here are also closely related to the following:

- aspect-oriented programming, subject-orientation.
- activities, object collaborations.

1.2 Organization

The chapters of this thesis are quite different; some are concerned with the conceptual framework around gbeta and Beta—at this level there is little difference between the two—and others present and motivate technical details of certain language constructs or briefly survey the approaches to specific topics in other programming languages; yet others focus on software engineering aspects or on the implementation of gbeta. Finally, some chapters present partial formalizations of the semantics and static analysis.

Chapter 2 establishes the basic concepts such as objects and patterns, with an emphasis on the underlying conceptual framework which puts these concepts into perspective. It then goes into a brief presentation of the concrete language constructs which support these basic concepts, deferring a large amount of detail to later chapters.

In Chap. 3, patterns are treated in great detail. Since the concepts are so tightly integrated, this chapter also introduces mixins which are the building blocks that patterns are made of, and it introduces objects, since they are to such a large extent determined by their associated patterns. Moreover, the mechanism which is used throughout to create new patterns from existing ones, the C3 linearization algorithm, is introduced, formalized, and some soundness properties about it are proved. Inheritance of attributes and specialization of behavior is covered, and the rule for name lookup within an object is presented. Finally, the notion of qualifications, which is similar to declared types of references in other languages, is presented.

Some pattern attributes are bound to pattern values by means of a unification process which actually drives the entire propagation machinery, namely virtual pattern attributes, and they are covered in Chap. 4.

The next chapter, Chap. 5, discusses the notion of block structure and the associated notion of context dependency, and motivates why that might actually be considered the essence of object-orientation even though block structure has had an unstable and often rather low popularity in the object-oriented community and elsewhere.

The interplay between block structure and virtual pattern attributes is the essential basis on which the capability for propagating specialization relies. How this works is covered in Chap. 6.

Chapter 7 goes on to another unusual feature in a statically type checked language, namely the capability of gbeta to support the creation of new classes and methods at run-time. The first part of this chapter argues that it is indeed justified to call these features 'dynamic' even though they are kept under strict control by the static analysis.

Finally, Chap. 8 presents a number of miscellaneous functionality related enhancements in gbeta, some of which are not backward compatible with BETA; and Chap. 9 presents a few mechanisms which were added to gbeta in order to solve some problems with the expressiveness and/or safety properties of the BETA static analysis. Programs using the former enhancements can generally be rewritten to an equivalent form that do not use these enhancements, by means of local changes to the source code. In contrast, the latter improvements, the ones related to the static analysis, allow the expression of type safe designs which could not expressed safely in any similar design without these improvements.

This concludes the direct treatment of the language gbeta. After that, in Chap. 10, the modularization system in gbeta is presented. This is the fragment language, and the only difference between the fragment language in gbeta and in BETA is that the gbeta implementation is more general—because it skips over some hard problems in the area of separate compilation. Nevertheless, the more general implementation of the fragment language may work as an illustration of how important it is to try to implement it more completely than is the case with BETA today.

The two next chapters present two more formally strict descriptions of two small languages which exhibit the core properties of gbeta, but avoid a large amount of complications from all the non-core constructs which exist in order to make gbeta a practical language. It was for a long time an important goal to formalize gbeta as a whole, but it seems that concrete language design bears a rich motivation in itself whereas a rigid formalization may appear to be a more mundane task of cleaning up the results achieved elsewhere.

The short Chap. 14 concludes, and thereby marks the end of the main part of this thesis. The appendices which follow after Chap. 14 give additional details about certain topics which have been covered more briefly throughout the earlier chapters. Appendix A gives the complete, context-free grammar for gbeta. The proofs for some properties of the linearization algorithm appear in App. B. The next appendix, App. C, contains the original presentation of the Expression Problem, as it was given by Philip Wadler on the Java genericity mailing list in the autumn of 1998; this is used in Chap. 9.4. Appendix D presents each of the instructions in the special virtual machine for execution of gbeta programs, making it easier to find an estimated upper bound of the detailed time and space complexity of the execution of gbeta programs.

Since the chapters of this thesis are so different in content, it might be beneficial to separate the different kinds of topics and create, say, several selective tables of content which simply omit references to all the parts of the thesis which are concerned with all other perspectives than the one in focus for that particular table of content. This would make it possible to read some parts of the thesis in order to learn about the concrete language syntax and semantics, skipping all the more philosophical considerations about the conceptual framework etc. However, this would not be an easy task, exactly because it is one of the main points of the thesis that all these perspectives must be brought together in order to do serious language design, so to the extent that this has actually been achieved it will be almost impossible to read about one aspect in isolation because there will be cross-references to all the other aspects which would then be hard to understand.

Part I

The Language **gbeta**

Chapter 2

Basic Concepts

Coming from the Scandinavian tradition of object-orientation, and in particular having its roots in Beta, gbeta has a terminology which is in some ways non-standard. It might seem that the unusual terminology is an unjustified added difficulty, making it harder for the general public to understand and judge the value of the contributions of this community. However, the unusual words often denote unusual concepts (e.g. pattern), and in these cases a non-standard word is obviously called for. Moreover, some words which are used everywhere (e.g. object) have a different meaning. Hence, these basic concepts need to be introduced carefully; the next section introduces objects and patterns at an abstract level, and the following sections of this chapter introduce concrete language constructs for patterns, objects, and other basic aspects of gbeta.

2.1 Objects and Patterns

The following discussion introduces the conceptual foundation for gbeta, complementing the conceptual framework for Beta, of which a detailed presentation can be found in [74, ch. 18]. This treatment should be self-contained, though, such that [74] need only be consulted for additional depth. The discussion applies to Beta as well as gbeta. The concrete language constructs arising from the considerations in this section are described in Sect. 2.2 and on.

2.1.1 Modeling

Program executions are considered to be similar to simulations, having a *mod-*
eling relation to a "topic": the structure and dynamics of the program execution ◇
(the *model system*) should reflect the structure and dynamics of a selected part ◇
of the real world, as viewed from a given perspective (the *referent system*). ◇
The choice of a perspective is essential—for instance, a bus may be considered a complex system of interconnected and interacting physical components in a CAD/CAM system used by a bus manufacturer, or the same bus may be

represented by a few data items like name and price in an accounting system. Different perspectives on “the same thing” may give rise to entirely different computerized representations.

Many other approaches to object-orientation also emphasize the modeling relation between the real world and object-oriented programs, e.g. [24], even though conceptual modeling may not be given the first priority, and the choice of perspectives is treated only implicitly.

It might seem that this framework only applies to simple-minded mirroring of physical phenomena like train schedules or payrolls, leaving many well-known computer programs unexplained, e.g., word processors. To counter this objection we must expand on the importance of a peculiar circularity, namely that
◇ *models are themselves phenomena*. For example, the contents of a file used in an accounting system may be considered a *model* of the real-world state at some point, but at the same time it may be managed (e.g., copied) as a *phenomenon* in its own right by an operating system utility (such as ‘`cp`’ or a ‘File Manager’). Similarly, a word processor supports the presentation and manipulation of a computerized phenomenon—a piece of written, natural language—which, considered as a model, might be concerned with the description of objects and patterns in a computer language called `gbeta` . . .

The perspectives on computerized material as model or as phenomenon will always be intertwined because the main benefit of a computerized modeling system compared to, e.g., a book is the dynamic manipulability of the model. To manipulate a model, it must be treated as a phenomenon.

2.1.2 Modeling Phenomena with Objects

A program execution cannot reflect the development in a part of the real world in an amorphous, holistic way. A divide and conquer strategy must be applied, by dividing the referent system into phenomena, each less complex than the referent system as a whole. The programming language must then provide a
◇ representation of real-world phenomena; *objects* play this rôle.

This rôle is dual, since phenomena may be things as well as behavior. Consequently, the object concept corresponds to both “objects” and “method invocations” in more traditional languages. Note that a conventional method is not the same as an invocation of that method; in most languages there is no explicit access to invocations.

2.1.3 Modeling Dynamics—a Non-solution

Supporting phenomena is not sufficient—the world is not static. A development in the real world may bring phenomena into view or otherwise change their status from irrelevant or inexistent to present and relevant, e.g., when a house is built or a tornado emerges. Because of the vast complexity of the real world, it is not feasible to rebuild it in all details inside the computer. Hence, the emergence of phenomena in the execution of a computerized model (a program) can not be expected to be an automatic consequence of the immanent properties of the model. In other words, we cannot model the society and nature in such faithful detail that the "house" will be built inside the computer and then destroyed by the "tornado" inside the computer, for reasons which in details parallel a similar development in the real world, down to the stroke of the wings of a butterfly on Sri Lanka which originally made the difference between "tornado" and "no tornado". The conclusion is:

- We do not want to copy the world, it is too complex.

2.1.4 Complexity Management

Natural language provides a wonderful wealth of accumulated knowledge about modeling and complexity management.

To take a simple case first, consider fixed references to phenomena, like names of persons or places, or signals. A *signal* unconditionally signals the state of the sender, like saying "Ouch!" when it hurts, hence referring to a fixed phenomenon within the sender. A particular sound might be used by some birds to signal fear, and those birds would not need any language capability beyond signals to make use of them. Fixed references may efficiently direct the attention to known phenomena and hence work in a complex world, but they rely entirely on previous knowledge and do not provide intra-lingual complexity management.

Luckily, language is not only fixed references to known phenomena. For example, we can talk about the house and the tornado from the previous section without having experienced them ourselves, and without reconstructing the events in every detail. We are using a model of the event which includes a *purposeful* level of complexity, and this is only possible because we give up built-in *causality* (i.e., the inevitable link from a cause to its effect). The words that describe the emergence of the tornado have no built-in mechanism which forces the production of words that describe the effects of the tornado sweeping over the landscape, even if that were the actual development. Someone telling a story about the tornado could just as well tell about the miraculous change which suddenly made the tornado weaken and dissolve, turning into a peaceful breeze and leaving the house untouched. The loss of causality is at the same time a liberation from necessity, giving the freedom to describe non-existing phenomena as well as existing ones, thus enabling dreams, lies, hypotheses, theories, etc.

To explain how natural language obtains this liberation from real-world causality, but still retains the ability to go beyond simple references to existing knowledge, we must deal with the concept of *concepts*. There are several different philosophical views of concepts, including the Aristotelian and the prototypical view [74, ch. 18]. All of them recognize the ability of concepts to denote a collection of phenomena, called the *extension* of the concept, by means of some kind of decision procedure, called the *intension* of the concept. E.g., for a "hard", Aristotelian concept like 'prime number' we might use a quite rigorous procedure to determine whether a given phenomenon already considered a number could also be considered a prime number, and for a "soft", prototypical concept like 'nice weather' we might have a long discussion about it. For concrete system development, Aristotelian concepts are much more manageable (implementable) than prototypical ones, but in the discussion here, the choice of concept of concepts is unimportant.

The ability of a concept to denote a set of phenomena determined by a decision procedure contrasts with the more primitive language entities like proper names or signals. They can be learned directly by repeated experiences of the connections, whereas concepts are unavoidably dependent on descriptions or other specifications of the intension. This introduces a circularity in that 'concept' can only be defined using concepts; luckily this is no problem when explaining it post-hoc.

The extension of a concept is not arbitrary, the members of the set of phenomena in the extension are in certain ways similar. These similarities make it possible to use existing experience to estimate the properties of a situation described in terms of concepts. Hence, a concept based description will provide a useful model, avoiding both the restriction to simple, fixed references to phenomena, and the complexity of (in any sense) copying the mechanisms of the referent system. We may now expand on the conclusion made in the previous section:

- We cannot copy the world, but we can *describe* it.
- Natural language avoids the complexity explosion by not supporting real-world causality.
- Natural language then gains the ability to go beyond fixed references to known phenomena by means of concepts.

2.1.5 Understanding

This section presents a simplistic view of the human mind. It is of course not supposed to overthrow all the efforts made by psychologists, philosophers, and others over the centuries; it is only supposed to help leveraging the richness of the human mind as a source of inspiration when doing programming language design. Moreover, it focuses on the activity of consuming and understanding language, only mentioning sensory experiences and production of language briefly at the end.

Think of the human consciousness as a universe, capable of supporting dynamic processes by means of entities. We will make no attempt to explain the physiology which supports such entities on the basis of networks of neurons, nor to detail the nature of those entities; but note that they imply that *understanding* at a very fundamental level consists of *dividing* the world into parts, ◇ phenomena, and then reconstructing an image of the world in terms of images of those parts. This is an *analytical* approach to understanding, based on breaking ◇ down and reconstructing. It is very suitable for our purposes, oriented towards language, whereas a *holistic* approach would be more oriented towards word-free ◇ exploration of fine details of total, undivided mental states. Since programs are just (extremely regular) language, the analytical approach has the right bias for us.

The mental entities can be described from a functional point of view. Their basic responsibility is to be images of real world phenomena, thus enabling the human carrier to make reasonable predictions and thence useful decisions when interacting with that real world. The human understanding of the surrounding world is thus an active reconstruction of the world in terms of such mental images. The reconstruction may shift rapidly, as if several potential versions of mental universes are available and more or less activated, corresponding to changes of attention and of mode of thinking.

The mental reconstruction need not be "perfect". For example, ghosts are[1] simply mental images which do not correspond to real world phenomena; conversely, walking right into a glass door is usually the consequence of having failed to build a mental image of that glass door at the right time. More importantly, different perspectives—chosen according to different basic understandings of the world and different desires and goals in different situations—radically influence the contents and structure of the mental shadow world.

Furthermore, human beings are capable of detaching the mental image of the world from the actual surroundings, for instance when being intensely engaged in reading a book. With this we arrive at the core topic of this section: listening, reading, or otherwise consuming language. Language consumption corresponds to using the language as abstract (world-detached) directions as to what developments to induce into the mental universe. Each sentence is actively being interpreted by the listener's mind, and the meaning of the sentence *is* the set of changes induced into the mind of the listener.

It may be illustrative to think of this process as the insertion of a piece of code into an interpreter which will then execute that code in the given context. Note that the execution happens in a debugger! E.g., considering a given statement a lie corresponds to rejecting to "run" that statement.[2]

The existence of numerous near-activated mental universes makes the picture very complex, since the reaction to any stimulus might include a shift in the priorities of mental universes. In any case, all the mental universes are constructed within a general framework of understanding, the *world view*, which ◇ contains basic assumptions, and outlines the limits of acceptable images of the world. The twelve categories of Immanuel Kant is one famous attempt to outline inevitable basic assumptions on which understanding must be built. For example, time and space are fundamental modes of organization of perception—we do not know by experience that the world exists in space and develops in time, because experience is only possible when time and space are already in place. On top of such basic infrastructure, but otherwise at the foundation of the framework of understanding, we find sensory experiences. Any suggestion of developments in a mental universe which violate the huge base of sensory experience will generally be rejected, or cause the mental universe in question

[1] Probably

[2] Of course, a meta-statement like 'He is a liar!' may be executed instead

to be labeled as 'phantasy'.

Finally, the perceiving mind may set out from a mental development, arising from experience or phantasy or both, and reconstruct a "program" which would give rise to a similar development when received and "executed" in a similar mental context, and then "transmit" that program to others. This is called 'talking' or 'writing'.

It might be interesting to try to use this description of the human mind to build more "intelligent" and robust computer systems, but the discussion about holistic approaches, multiple mental universes, attention, and more are only included here in order to make the picture broad enough to make sense. What we will use directly in the following sections is only the following core:

- A relevant view of human understanding is as construction and development of models, based on mental images of phenomena.
- Natural language can be received and "executed" by the mind, thus building or modifying mental models.
- Simple references just allow natural language to redirect attention, but concepts allow the construction of mental models which are liberated from the "truth". Human understanding of computer programs takes this independence and self-drivenness of models to an extreme.

2.1.6 Concept Based Modeling Using Patterns

We need a mechanism in the programming language to play the rôle of concepts; ◇ *patterns* play this rôle. As is the case with concepts, patterns may have (images of) things as well as (images of) behaviors in their extension, so patterns correspond to both 'class' and 'method' in most other languages.

Similar to the intension of a concept, a pattern is associated with a specification of the extension, but since a programming language must be machine executable there is no room for vagueness or discussion. Hence, patterns are at the extreme Aristotelian end of the spectrum of concept views.

The concrete syntactic constructs used to specify patterns are presented in Sect. 2.2; more details are given in Chap. 3.

At this point we are ready to defend the class based approach to OO language design as opposed to the seemingly cleaner and simpler ◇ *prototype based*, classless designs. The argument is that a classless approach will need complexity management just as much as a class based one, and the hundreds of generations of experience embedded in the structure of natural language is simply too good to ignore; when classes are not supported directly, essentially the same concepts will inevitably turn up under other names, or as more or less elegant programming conventions; for example, Cecil distinguishes between abstract/template objects and concrete objects—the former work just like classes and the latter work like objects; and the convention of putting all methods in a Self object into a separate 'traits' object which holds all methods is actually very similar to an implementation of classes.

2.1.7 Modeling Dynamics—Object Creation

The relation between patterns and objects is similar to the relation between concepts and *mental images* of phenomena, not the relation between concepts and phenomena. The main difference is that the real world with all its details supports a causality which is neither supported by natural language descriptions nor by computer programs. As a result, objects do not just emerge including all the needed properties during program execution, they must be explicitly created according to some description, which is in fact a pattern. Note that object creation in a program does *not* directly correspond to a similar event in the real world; but it does correspond to the change in a mental model when a phenomenon is discovered, or when other changes make a previously ignored phenomenon relevant.

When a pattern is used to *create* a new object we say that the pattern is used *prescriptively*, in contrast to the descriptive use of patterns which is introduced in the next section.

Natural language actually does have a similar "object creation" mechanism, although it is of course much more subtle than in a programming language: If a story starts with 'Once upon a time there was a king whose daughter ... ' then the mental image of the king and his daughter are induced in a listener hearing about them for the first time. Certain syntactic constructs ('*there was*'), modes of articles ('*a* king', not '*the* king') or explicitness of relations ('*whose* daughter') serve to mark the introductory references as such. It depends heavily on the linear structure of language which makes it quite well-defined when a phenomenon is mentioned for the first time; that implies creation.

Creation of mental images of phenomena is the core of dynamics of mind, and creation of objects in a program execution is the core of dynamics for such an execution.[3] Objects which become irrelevant at some point may simply be ignored, so object destruction plays a much smaller rôle than creation at this level.

2.1.8 Navigating in a Model

Concepts and patterns are not only used prescriptively. In fact, presentations of the conceptual foundation for classes and similar concepts usually emphasize *descriptive* usages, providing information about an already existing (considered-as-relevant) phenomenon or object.

The need for information is obvious; conceivably we could ramble around in the world with closed eyes and plugged ears etc., but usually it is safest to interact with phenomena only when they have known properties, to some extent. Descriptive uses of concepts supply the listener with a similar property enrichment of the imagined world as the sensory input does for the real world. Similarly in a program execution, the knowledge that a particular object is in the extension of a known pattern improves the safety of interacting with that object. In fact, with strict type-systems as in gbeta and in BETA, no property

[3]Measurable properties may also change, see Sect. 2.1.10

of an object is ever assumed to exist without a static proof of its existence; this is covered in more detail in Sect. 2.2.5, Sect. 3.11, and Sect. 13.

2.1.9 Causality After All

It may seem reasonable to describe natural language as purely a vehicle for transport and (complex, receiver dependent) manipulation of states of mind, being a passive entity driven entirely by extra-lingual forces like sensory input and desires. However, the existence of logical reasoning demonstrates that causality also *does* occur intra-lingually. The concept of language used here is broad enough to include formal logic and mathematics as special cases.

◇ Formal logic inference rules treat language entities *as phenomena* (like chess pieces which can be moved around according to rules) independently of their modeling rôles, and that aspect has been driven to extreme prominence in the case of programming languages. An implementation of a programming language, or a complete formal semantics for it, establishes a complete formality. Such a complete formality enables a program to control the actions taken by a machine, introducing a whole new world of possibilities for intra-lingual causality. In other words, the execution of programs contain mechanisms which with necessity produce certain effects from certain causes, thus making the program execution dynamic in a sense which used to be reserved for the real world only. This enables automatization at a level of sophistication which has changed the world.

Hence, causality in programs certainly makes a difference. However, we should always remember that computerized causality is of the same kind as logical reasoning, which has basically nothing to do with the causality of nature that causes the universe to behave as it does.

2.1.10 Modelling Dynamics with Measurable Properties

Natural language has developed another mechanism which helps making language based models useful even though they have vastly less complexity than
◇ the world they model. This mechanism induces *measurable properties* into mental images of phenomena as a postulate, not by mirroring the causal basis for those properties. As an example, saying that 'the house is *red*' provides the mental image of a house with the property that visible light is reflected mostly for wavelengths near 700 `nm`, without giving any details about why that would be the case.

As with concepts, the benefit is a complexity reduction through absence of faithfulness in the modeling relation. If postulates were not available and we had to establish the redness of the house by reconstructing the mechanism, then we would need to model every single atom on the surface of the house and every photon hitting it. At the macroscopic level there is *no* mechanism which causes color.

So we want to support measurable properties. A measurement yields a
◇ *value* [69], which is a simpler concept than that of a phenomenon or an object, because values do not have identity. For example, if we count the number of

people in a room twice and get 17 both times, it does not make sense to ask whether it is the *same* 17. Semantically, a value may simply be represented as a member of a set, for instance the set of natural numbers; 17 is 17 and that's it. In contrast, a semantic representation of an object requires a notion of identity such that two distinct objects will still be considered distinct even if they happen to be in the same state. Typically, the semantic representation would be an object identity (perhaps a natural number) which could be used to look up the *current* state of the object in the store (the semantic notion of memory) of the program anytime during the execution. To support measurable properties, we need values.

A very entrenched point of view is that it is characteristic of a clean object-oriented language design that "*everything is an object*". In particular this view ◇ is represented by the Smalltalk [50] community, but it is rarely even challenged. If that goal were to be reconciled with the other (commonly accepted) goal of maintaining a modeling relation to the real world, then we would need to represent, e.g., 'red' in the above example using objects, and there is no reasonable way to do this—'red' and 'house' are inseparable from an object point of view.

The usual solution in languages like Smalltalk is to introduce *unique* objects. ◇ For a unique object, object identity is made irrelevant because there is exactly one of each kind, none of them disappear, and no new ones can be created. For example, there is *the* '1' object and *the* '2' object and so on, at all times.

If everything is an object then unique objects are needed in practice: for example, most practical programs will break if somebody introduces an extra boolean object (or an extra class inheriting from boolean) besides `true` and `false`. Furthermore, unique objects solve the problems with disturbing object identity: for example, with unique integers $3 + 4$ will always yield "the same" 7 as $5 + 2$, which is of course desired in case somebody wants to compare the results.

Now if unique objects work so well, what is the problem? The problem is simply that the unique objects have *exactly* the same properties as pure values would have had, so claiming that "everything is an object" and then making some objects unique is just a cover-up for the fact that integers, booleans, etc. actually *are* values and not objects. They should not have that distinguishing feature of objects which is object identity, and they do not have it either.

After this vendetta, values are safely incorporated as a useful and well-justified element of an object-oriented programming language design, and we may reveal that `gbeta` provides a small, predefined set of value domains. They are described in Sect. 2.2.1.

2.1.11 Relation to the BETA Conceptual Framework

As mentioned already in the beginning of this chapter on page 9, the conceptual framework associated with BETA has been an all-important source of inspiration for the presentation given in this chapter. However, the presentation here differs in some ways; especially by emphasizing that the transition from a part of the real world to a computerized model of it necessarily is accompanied by a vastly reduced complexity; then by introducing patterns as the parallel of concepts *because* concepts are an age-old, well-tested solution to the problem of providing such a vast but meaningful complexity reduction; and finally by motivating measurable properties and values as yet another well-tried natural language device for obtaining useful models without excessive amounts of detail. Some highlights are the following claims:

- an object should *not* resemble the real world, it should resemble a useful natural language *description*, hence ...
- a prescriptive use of a pattern, that is object creation, is not an ugly, unexplained corner of object-oriented languages, it is a natural consequence (parallelled in human thinking) of the loss of those details which lets the real world generate phenomena "automatically"
- even a model is a phenomenon for some purposes

2.2 Language Constructs for the Basic Concepts

After having motivated the choice of fundamental concepts in gbeta at length, we can introduce the concrete details. This section just introduces the core syntax and a very brief explanation of the informal semantics of gbeta, to give an overview of the language. Many aspects are covered in more detail in later chapters.

2.2.1 Value Domains

◇ As mentioned in Sect.2.1.10, gbeta offers a set of *value domains*:

- boolean values (true and false)
- char values ({a–z, A–Z, 0–9 ... }.)
- integer values ({ ... -2,-1,0,1,2 ... })
- real values ({1.0, -3.14159, 1.2e38 ... })
- string values ({ '', 'x', 'Readme, please' ... })
- the set pattern of patterns
- the set oid of object identities (think "pointers" to objects)

Except for the fact that control structures depend on booleans and integers, the set of domains and the exact set of values in each domain is not essential for the language design, even though it would have to be characterized precisely for language *standardization*. But the fact that the set of value domains is predefined is not satisfactory. We considered adding a complete functional language at the "bottom" of gbeta (i.e., for expression evaluation) but that has not yet been worked out and may bring more confusion than benefits.

Compared to BETA, the string values have been added; string values are immutable sequences of characters. The motivation for adding strings is that the language (Mjølner) BETA depends on hundreds of lines of code by having built-in knowledge about the declarations of large patterns like `stream` and `text`, especially in order to be able to handle implicit coercions from literal strings to full-fledged `text` objects. In the design of gbeta, such dependencies were considered inappropriate, and the `string` basic pattern (introduced in Sect. 2.2.4) enables an alternative implementation of `text` which preserves the functionality and relieves the language as such from the dependencies on concrete source code.

This even improves the performance—compared to the Mjølner BETA implementation, which also to some extent defines the language BETA, in cases where [74] is ambiguous. Evaluation of literal strings in BETA implies the creation of a `text` object, hence a literal string should not be evaluated unless the value will actually be used; with string values, evaluation of literal strings can be as cheap as integers. Moreover, in today's BETA programs lots of `text` objects are copied, because it is too hard to avoid the combination of aliasing and updating which makes "my" `text` change just because "somebody else" needed to change "his" `text`, and the two `text`s happened to be the same object. The string values can be shared freely; on the other hand, a programmer may need to go back to mutable strings for the special cases where many small changes must be made to a large string (again, by avoiding excessive copying). The design and the performance implications associated with gbeta strings are well-known from, e.g., Python built-in dictionaries [115].

In BETA [74, p. 45], `true` and `false` are patterns, inheriting from the basic pattern `boolean` (see Sect. 2.2.4). We find it hard to see how that could be specified in a manner which is consistent with the rest of the language; it would be hard to give a satisfactory definition of what `&true[]->aBoolean[]; false->aBoolean;` should mean—and such usages should be allowed with the given description.

2.2.2 Values and Immutability

The purpose of this section is to explain why it is sometimes necessary to mention that values are "immutable". It is included because the concept of 'value' and the concept of 'object' sometimes seem to be hard to keep clear of each other. The reader who thinks that this is a trivial problem may want to skip the rest of this section. The concept of values and its relation to objects has been analyzed in [69], but we are not aware of a treatment that deals with the specific source of confusion which is the topic of this section.

In a typical computer hardware design, as seen from the machine code level and ignoring details about caches etc., there are a few CPUs and a store which is realized as an array of cells of 8-bit binary values, normally augmented with processor instructions for accessing 2, 4, or 8 of those bytes as a unit.

This design provides those values which may be represented in a few bytes with a special status, since they may be stored and loaded directly with built-in processor instructions, atomically. The view of the value as the *state* of a small group of atomically accessed memory cells, and of the group of memory cells themselves as a *container* for such a value is easy to grasp. Two different groups of cells may hold the same value and still be distinct groups, and two different values may be loaded from the same group of cells at different times. Sofar, there is no need to talk about values being immutable, just like there is no need to emphasize that a value like '123' is immutable; '123' is '123', and any attempt to change it would be considered a silly and counter-productive exercise; for example, we don't want to worry about things like "Do you mean the '123' of today or the '123' of yesterday?"

However, values may be arbitrarily complex, so built-in hardware instructions for the retrieval, storage, and management of values in general cannot (realistically) be implemented. As a simple case, a character string of arbitrary length may be kept in a contiguous area of memory cells and transferred to a similar area of cells by a loop which copies the contents of a few cells per iteration. The fact that retrieving and storing this value is not atomic in terms of machine code actions *does* make a difference.

In context of concurrency, the abovementioned implementation of string value transfers might be considered wrong, because another thread might change single cells in the source area during the transfer, such that the value produced at the destination is not equal to the value present at the source at any point in time. If the source holds the value `'Hello, world!'` and is later assigned the value `'Veni, vidi, vixi!'`, then the destination might receive the value `'Hello, wor, vixi!'`. As this shows, not even when considering the whole series of intermediate states at the source can it be explained what the resulting state is at the destination, because the state of the source was *never* `'Hello, wor, vixi!'`; in other words, this behavior cannot be explained using string values, it can *only* be explained in terms of `char` values kept in separate, mutable, `char`-sized chunks of memory.

Even in the strictly single-threaded case, aliasing invalidates any attempt to explain the behavior using string values only. An access path to any subset of the string storage area, e.g., access to a single memory cell via a (simple, `char` typed) variable, will enable changes to the composite value, the string, indirectly by changing the value of the simple variable. Again, since no computation, intermediate or not, in the execution of the program produced the resulting value of the string, there is no way to explain this resulting value without describing the string as a composite, mutable entity containing a number of separate memory cells.

If we cannot use the concept of a composite value when explaining the behavior of program executions, then composite values are simply not implemented correctly.

Hence, in order to be able to explain the semantics using a notion of composite values, we must restrict the possible behaviors by imposing a certain discipline on the use of the individual memory cells. The discipline includes the following:

- The entire set of memory cells used to hold a composite value must be updated only as a whole (no subparts of it can be updated via other access paths, e.g., using variables with simpler types)
- Any change to the value by means of a sequence of changes to parts of the representation must happen atomically (in a critical region), such that no retrieval of the composite value will ever obtain an intermediate, "half-updated" result
- Any retrieval of the composite value must happen atomically, perhaps overlapped with other retrievals, but not overlapped with updates

A simple solution which satisfies this is to allocate fresh memory to hold a given composite value and then never change it afterwards; any change to a variable holding that value would require allocation of more fresh memory and construction of the new value in there; on the other hand, it is safe to let many variables refer to that storage, to represent the fact that they hold that particular composite value. This is a typical implementation in functional programming languages, where composite values play a very important role. The unlimited aliasing may avoid a lot of copying, but on the other hand the computation of many similar composite values is expensive (e.g., when a long string is being edited interactively it will be copied with every change).

This discipline on the usage of memory cells establishes a closer connection between the groups of memory cells themselves and the contents, the composite value. This is because that area of memory is used for nothing but holding the value, during the entire period from allocation to garbage. That connection probably causes the confusion which it is the purpose of this section to remove.

A given area of memory may be used to represent an object—a mutable entity with object identity—or it may be allocated, filled in with a value, and then kept unchanged, in order to hold the given value. In the first case we might very well find two areas of storage with the same bit pattern, thus representing two *different* objects which happen to be in the same state; but in the second case, two areas of storage with the same bit pattern would generally represent the *same* value,[4] and even though they would have different memory addresses, it should *not* be possible within the (high-level) language to distinguish between them, e.g., to discover whether they were allocated in the same or in two different areas of memory. As a consequence, such a composite value representation is *not* the same as an object, not even the same as an object whose state is declared immutable ("`const`"), if the language supports such a concept.

In summary, because composite values cannot be handled atomically at the machine code level, it is impossible to obtain the correct semantics in the management of a composite value without a special discipline on the usage of memory

[4]Value equality may of course be more complex; the values might, e.g., be graphs which include pointers representing internal edges, and they would of course not have the same bit pattern, but rather describe the same graph

cells containing such a composite value. A safe and simple discipline is to ensure that memory containing a composite value is never changed (until it is garbage collected, at least). This discipline makes the representation of composite values resemble the representation of composite, mutable entities (objects), and hence it is tempting to use the term "immutable" for the former, to distinguish it from the latter. This is purely an implementation concern; semantically, a mutable value is an absurdity, so values are not immutable, they are just values.

2.2.3 Literals

The basic value domains boolean, char, integer, real, and string were already introduced in the previous section. The only syntax associated with values is the *literal notation* of those values, which was also exemplified in the previous section. For example, the syntax `true` is a literal; each evaluation of `true` delivers the boolean value true. No other operations than evaluation are supported for literals—it is not possible to assign to a literal, or to obtain the pattern of a literal (a value doesn't have a pattern), or a reference to it (a value doesn't have object identity), etc. The precise syntax of literals is given in App. A.

2.2.4 Patterns and Objects

For each basic value domain there is a *basic pattern*—`boolean`, `char`, `integer`, `real`, and `string`—whose instances are capable of carrying a state which is a value from the corresponding value domain. For example, we may use the pre-defined pattern `integer` to create an (`integer`) object which functions as a container for a value from the domain of integer values. Instances of basic patterns are called *basic objects*.

Measurable properties of objects are supported through *object state*. Simple object state is supported by basic objects,[5] and more complex object state is supported inductively, by composition. Basic objects as well as variable attributes (introduced later in this section) are the atomic building blocks, and more complex entities can be composed from less complex entities using a syntactic construct called a MainPart.

A simplified version of the MainPart syntax is given in Fig. 2.1 on page 23. The full grammar can be found in App. A. The grammar here is only concerned with the aspects of MainPart which are used to declare class-like patterns. The support for behavior (for method-like patterns) is covered in Sect. 2.2.6.

The declaration syntax is unusual but consistent. Every declared name is placed on the left hand side of a colon ('`:`'), and the ObjectSpec syntax which specifies what that name means is placed on the right hand side. Between the colon and the ObjectSpec is a short string, Kind, which determines what kind of attribute is being declared.

In the *notation* used in the grammar in Fig. 2.1, non-terminals are written like this: 'MainPart'; terminals are single quoted (like `':'`); alternatives are

[5] Actually we should say part objects because they may be parts of larger objects, but part objects are not presented before Sect. 3.3.

```
     MainPart ::= ‘(#’ AttributeDecl* ‘#)’
AttributeDecl ::= Name ‘:’ Kind ObjectSpec ‘;’
         Kind ::| ‘’   | ‘<’  | ‘:<’  | ‘:’ | ‘##’
                | ‘@’  | ‘^’  | ‘^=’
                | ‘@|’ | ‘^|’ | ‘^|=’
   ObjectSpec ::| Name | Descriptor
   Descriptor ::= Name? MainPart
```

Figure 2.1: Simplified (class) syntax of MainPart

separated by bars (‘|’); and standard regular expression syntax is used to mark optional and repeated elements (X* derives zero or more repetitions of X, and $Y^?$ derives the empty string or Y). Name is used for names (identifiers); it is specified at the lexical level by the regular expression `[a-zA-Z_][a-zA-Z0-9_]*`, which allows for a commonly used set of strings for names. Names are case insensitive.[6]

2.2.5 Attributes

The MainPart syntax allows for several different Kinds of attributes, presented in Fig. 2.2 on page 24. Compared to BETA, the exact variants (marked by ‘=’) have been added. Compared with other languages, gbeta attributes correspond to both methods, fields, and inner classes in Java; to features in Eiffel (both routines and attributes); and to members in C++ (both data members and member functions). Of course, there are many differences in the details.

The kinds of attributes are divided into four groups in Fig. 2.2. The four groups arise from two choices, between pattern and object and between simple and variable. An attribute may provide a pattern, as in the groups ‘pattern’ and ‘variable pattern’, or an object, as in the groups ‘object’ and ‘variable object’. An attribute may also denote an entity directly, as in the groups ‘pattern’ and ‘object’, or it may denote a variable which in turn provides an entity, as in the groups ‘variable pattern’ and ‘variable object’. A variable pattern is just a variable whose values are patterns. A variable object, however, holds the identity of an object, thus supporting not only different objects at different times, but also aliasing.[7]

Note that even though pointers may be used in an implementation (object identity may simply be implemented as memory addresses), the terminology emphasizes the mutability (if present), not the indirectness. As we shall see in Sect. 2.3, the concrete representation of an attribute is transparent at many usage points, such that the usage does not depend on whether the attribute

[6] gbeta would have been case sensitive if it had not violated BETA compatibility.

[7] Actually a declaration like `x: @y` may also introduce aliasing, as described near the end of Sect. 2.3.4.

Attribute	Kind	Description	Example
Pattern	(none)	Pattern	`X: string(#..#)`
	<	Virtual pattern	`X:< Point`
	:<	Virtual further-binding	`X::< ColorPoint`
	:	Virtual final-binding	`X:: ColorPoint`
Variable pattern	##	Variable pattern	`X: ##object`
Object	@	Object	`X: @integer`
	@|	Component	`X: @|task`
Variable object	^	Variable object	`X: ^string`
	^=	Variable exact object	`X: ^=string`
	^|	Variable component	`X: ^|task`
	^|=	Variable exact component	`X: ^|=task`

Figure 2.2: The different kinds of attributes

denotes an object, a pattern, or a variable object or pattern.

Members of class types in C++ and attributes of expanded types in Eiffel are similar to object attributes; instance variables in Smalltalk, fields in Java, members of pointer types in C++, and attributes of non-expanded types in Eiffel are similar to variable object attributes. In BETA, object attributes are called 'static references', and variable object attributes are called 'dynamic references'. We feel that the BETA terminology for patterns ('pattern' and 'variable pattern') should be followed for objects, too, both for simplicity, and because the word 'object' should not be missing, and because 'static reference' suggests the use of a constant pointer. It adds unnecessary complexity (in the mind of a programmer, or in a formal semantics) to introduce a pointer and then require that it is never changed. Hence the use of 'object' and 'variable object'.

When describing the individual variants of attribute kinds we need to use specific terms for the right hand side of the declaration. For the pattern group it is called the *value* of the attribute; for the object group it is called the *specification* of the attribute; and for the two variable attribute groups it is called the *qualification* of the attribute. In the traditional BETA terminology the right hand side of an object attribute would also be its 'qualification', but there are two reasons why this is not used in gbeta. First, the new terminology ensures that 'qualification' always refers to a pattern which is used as a constraint on the allowable entities referred by a variable attribute; for an object attribute this is not an issue. Second, in gbeta the specification of an object attribute need not be a pattern at all. This is detailed in Sect. 2.3.

Three of the four groups of attribute kinds have variants. These variants do not invalidate the description of attribute kinds given sofar, but they affect the structure of the declared entity or the constraints made on its use.

In the pattern group, a missing kind (marked with '(none)' in the figure) specifies that the declared entity is simply the value. The three virtual kinds specify that the declared entity is a pattern which depends on the context—i.e., the object of which it is an attribute, the *enclosing object*. To determine the ◇ precise pattern denoted by such a virtual attribute, the precise pattern of the enclosing object must (generally) be known, but the declaration itself at least gives an upper bound for the pattern. The partial order between patterns which determines the meaning of 'upper bound' is presented in Chap. 3, and virtual patterns are treated in detail in Chap. 4.

All variants whose Kind includes bar ('|') differ in the same way from the corresponding Kinds where that bar has been deleted: When the bar is present, the pattern associated with the attribute is guaranteed to be a specialization of `component`. This has to do with concurrency and is treated in Sect. 9.5

For those who know BETA this might be surprising. In BETA, objects and components are different kinds of entities; in gbeta, however, a component is just an object whose pattern is a specialization of the pre-defined pattern `component`. This simplifies and regularizes the language without sacrificing functionality, and actually improves type safety and expressive power. Again, details can be found in Sect. 9.5

Finally, the only variants not yet covered are the variable object variants containing '='. Normally, a variable object attribute may refer to any object which is an instance of a pattern which is less than or equal to the qualification. With '=', the variable may only refer to objects which are instances of exactly the qualification. An example where this proves valuable is given at the end of Sect. 8.1.2.

2.2.6 Methods and Behavior

Like the previous section, this section is also about patterns and objects; but the focus is on the behavioral aspects, so the patterns will be similar to methods, and the instances of the patterns will often be implicit, unnoticed, not unlike activation records for method invocations in traditional object-oriented languages. This presentation serves to introduce the reader who does not know BETA to the somewhat unusual syntax used for expressions such as assignment and parameter transfers. Since gbeta and BETA are identical at this level of detail, the reader who knows BETA might wish to skip to the next section.

A simplified grammar for MainPart with focus on the method-like aspects of patterns is given in Fig. 2.3 on page 26. The simplification mainly affects Evaluation, which just includes names, lists, and addition here. Of course, the full grammar in App. A will be needed in order to write useful programs, but the rather drastic simplification is appropriate here since the semantics of subtraction, multiplication and other expressions and control structures is generally

```
  MainPart ::= '(#' AttributeDecl* EnterPart? DoPart? ExitPart? '#)'
 EnterPart ::= 'enter' Evaluation
    DoPart ::= 'do' Imp*
  ExitPart ::= 'exit' Evaluation
Evaluation ::| Name | Evaluation '+' Evaluation
             | '(' Evaluation <',' Evaluation>* ')'
       Imp ::| Name | Assignment
Assignment ::= Evaluation '->' Name
```

Figure 2.3: Simplified (method) syntax of MainPart

unsurprising; moreover, later sections, e.g., Sect. 9.1, will cover some control structures and other aspects neglected here.

◇ The derivation of lists from Evaluations uses some new *notation*, namely angle brackets ('<' and '>'). They are only used for grouping, such that the repetition operator ('*') is applied to the comma and the Evaluation together. Hence, that alternative allows an Evaluation to be a parenthesized, comma separated, non-empty list of Evaluations, for example `(x,(y,z))`.

The rest of this section introduces the various parts of pattern specifications associated with behavior, as well as the interconnections between those parts which may be constructed using Assignments.

◇ The simplest part is the *DoPart*, marked by the keyword `do` and containing ◇ a sequence of imperatives. *Imperatives* are often called 'statements' in other languages, but the word imperative is standard BETA terminology, and moreover it specifies unambiguously that we are talking about commands given to the computer, not, e.g., about questions or assertions.

◇ The informal semantics of *executing an imperative* which is a Name is to look up the attribute denoted by that Name, obtain an object from it, and execute that object. How an object is obtained from different kinds of attributes is explained in Sect. 2.3, in particular in 2.3.4. The rules for name lookup are described in two phases, in Chap. 3 for the so-called local case and in Chap. 5 for the general case.

The informal semantics of executing an Assignment is to look up the attribute denoted by the Name subterm, obtain an object from it, evaluate the Evaluation subterm, insert the obtained value into the object, and finally execute the object. In order to give a description which aligns better with main-stream terminology we might phrase it like this: To execute an Assignment is to look up the method denoted by the Name subterm, create an activation record for it, evaluate the arguments, transfer them into the parameters in the activation record, and finally execute the method with that activation record. Or, alternatively: To execute an Assignment is to look up the object specified by the Name subterm, evaluate the Evaluation subterm, assign the result to the object, and finally execute the default method of the object, to let it integrate the received values correctly. Those translations reveal that the Assignment imperative covers both expression evaluation and function call for the left hand side, as well as method invocation and value assignment for the right hand side.

Note that the general support for (possibly nested) lists allows combining expressions into lists and thereby returning more than one value from a "function"; or accepting a list of values and thereby supporting argument lists for "procedures" or "methods" without introducing a separate concept or syntax for argument lists; or specifying logical, user-defined notions of value assignment (similar to user defined assignment methods in C++) by using EnterParts with objects.

To execute an object means to execute its DoPart, which again means to execute the imperatives of the DoPart sequentially.[8]

The insertion of a value into an object has an inductive definition. The basic cases are associated with the basic patterns, and with variable attributes. Value insertion and evaluation involving variable attributes is described in Sect. 2.3.4, but the semantics is similar to the semantics of basic objects in Assignments. With basic objects, e.g., inserting an integer value v into an `integer` object o_i means changing the state of o_i such that, until its next state change, an evaluation of o_i will deliver v; similarly for other basic values and objects. Correspondingly, the evaluation of object state is defined inductively with basic objects providing the basic cases, as implied in the description of value insertion. For example, a `boolean` object o_b will deliver either the value true or false, depending on the value last inserted into o_b. The composite (non-basic) cases of value insertion and evaluation are described below, after the description of the EnterParts and ExitParts.

The *inital values* of basic objects are false (for `boolean`), `'\0'` (the nul char, for `char`), 0 (for `integer`), 0.0 (for `real`), and '' (the empty string, for `string`). ◇

When executing an object o, instance of a pattern whose syntax contains an EnterPart N and/or an ExitPart X, both N and X are ignored; when o is evaluated, X specifies how to obtain the value of o and what structure that value has; and when a value is being inserted into o, N specifies what kind of value is accepted, and how to insert it.

As promised above, the composite case in the inductive definition of the informal semantics of object state evaluation and value insertion will now be explained. Given an object o whose pattern p is associated with syntax containing the EnterPart N and ExitPart X. Then value insertion proceeds as follows:

- If the Evaluation in N is a Name then inserting a value into o is the same as inserting it into the entity denoted by that Name, with lookup starting from o.
- If the Evaluation in N is a list L of evaluations, then the value accepted for insertion must have the structure obtained recursively from the structure of values accepted by the elements of L; the effect of inserting such a value

[8]The complete explanation depends on inheritance and `INNER`; see Chap. 3

is the same as the combined effect of inserting the elements of the value into the elements of L.

- If the Evaluation in N is on the form Evaluation+Evaluation then the program is rejected with a static semantic error (similarly for other expressions which do not specify a pattern or object, including all binary expressions).

The explanations about evaluation are parallel:

- If the Evaluation in X is a Name then evaluating the value of *o* is the same as evaluating the value of the entity denoted by that Name, with lookup starting from *o*.
- If the Evaluation in X is a list L of evaluations, then the value delivered by *o* is obtained recursively by evaluating each element of L.
- If the Evaluation in X is on the form Evaluation+Evaluation then each of the operands must deliver a single integer value when evaluated; they are evaluated, left to right, and the (integer) sum of the obtained values is delivered; otherwise, each of the operands must deliver a single string value when evaluated; they are evaluated, left to right, and the (string) concatenation of the obtained values is delivered.

The integer addition which provides the abovementioned sum happens in a monoid (integer, +) whose properties are not specified exactly here. It may, e.g., raise overflow errors, be non-commutative, and/or compute the sum modulo 2^{31}. It is apparently nice for a language to conform to a mathematically beautiful definition of, e.g., integer addition, but not all users of a language may want to pay for it in terms of lower performance, higher space usage, or similar. Specialized versions of a language might be very beautiful in this respect.

Of course, the full language gbeta has more expressions than just addition. There is also the topic of value coercions, e.g., '1+2.5' will coerce 1 into 1.0 and then perform an addition of real values. Since the gbeta approach to these issues is non-innovative we skip over the details.

One topic has been silently skipped over in the entire description above, namely the effects of a variable object attribute providing an object which is an instance of another pattern than the qualification. Of course, this only occurs when the variable object is not exact (see Sect. 2.2.5 about attributes in general and exact ones in particular), and then only with patterns which are specializations of the qualification. The general rule is that the statically known pattern unconditionally determines what part of an object is taken into consideration for evaluation and value insertion, whereas all parts of an object contribute with DoParts to the behavior of the object. More details can be found in Sect. 3.

Notice that the full generality of the MainPart is needed for patterns used as methods, since the DoPart is the body, the EnterPart specifies incoming arguments, the ExitPart specifies the returned results, and the attributes are used for arguments, results, local variables, and for local methods, classes, etc. It would be technically messy and wasteful to define two separate language constructs for method invocations and for objects now that the latter will do the job of the former just fine.

Moreover, *procedure activation records* (in Algol), rather than records or `structs` (in C terminology) were the original source of inspiration which gave rise to objects. Hence, objects were active and had behavior from the very beginning, even though many OO language designers still hesitate when it comes to introducing concurrency and active objects. ◇

Finally, classes may also need all parts of the MainPart construct, using attributes for local state, methods, nested classes and so on, using the EnterPart and ExitPart to provide a user-defined notion of value assignment (i.e., transfer of the logical object state), and using the DoPart to maintain state invariants (the DoPart is executed after every value insertion and before every evaluation).

Even though it would be messy and wasteful to have separate constructs for classes and methods, it might be valuable to be able to specify that a given pattern should be used only in certain ways, hence supporting programmer assumptions about the intended usage. An example where this is important is those cases where a (procedure) pattern is programmed under the assumption that it will never be executed twice. If a user of that pattern creates an instance, stores it, and executes it several times, then only comments might have helped the user avoid errors caused by this sequence of actions. The ability to say "not storable", "cannot be provided by a variable object", and other similar things might be valuable, though not currently supported in gbeta.

2.3 Transparency and Coercion

From a characterization of a subtle but ubiquitous phenomenon in natural language we derive the concept of transparency in the realm of programming languages. Transparency is a widely accepted goal within programming language design, and some approaches to it are presented. Finally the gbeta approach is detailed and compared with the BETA approach.

2.3.1 Natural Language and Transparency

"I was at the Jones'es today. Suddenly a big dog came running into the room. The dog *jumped* onto the table and started eating everything!"

This little story may not seem to demonstrate any particularly interesting properties of natural language, but that is just because we are so used to it. The following paragraphs focus on different relations between phenomena and words, thus paving the way for the claim that natural language provides good inspiration for transparency.

Consider the name of a person, like 'Jones'. In a given context it would often be unambiguous, such that this name is a fixed reference to the real world phenomenon which is that person. Personal pronouns, e.g., *I* or *you*, have a similar fixed meaning relative to a given colloquial situation. Fixed references have the nice property of allowing the listener to add in all the experience (s)he has with the given phenomenon, i.e., they provide much information, concisely.

However, in many cases words are used without having such a fixed association with one, known phenomenon. For example, at the beginning of the story there is no dog. The dog is introduced into the mind of the listener by the first reference to it, 'a big dog', and the indefinite article '*a*' confirms that this is an introductory occurrence. Later, the phrase 'The dog ... ' uses the definite article '*The*' to emphasize that 'dog' refers to an already introduced phenomenon. Articles, along with several other means including the pronouns 'this' and 'that', are used to coerce words designating concepts into simple references to fixed phenomena. Those phenomena are rarely given their own names, they are just recognized by the generic word ('dog') because the situation as described does not happen to have more than one phenomenon covered by that concept. Naturally, using a concept as a local name of a concrete phenomenon is so ubiquitous that almost no non-trivial statements could be expressed without it.

In particular, events or behaviors which occur and are then gone are commonly denoted in this manner, e.g., "After having said this, he made a very illustrative movement with his arm." The movement occurs and is gone, hence it can be introduced, live, and disappear in one go. Verbs as a word class is another device of natural language which introduces a phenomenon (an action or a development) transiently.

Actually this mechanism is so common that the concepts tend to disappear altogether. The word 'dog' designates a concept, not a simple reference to a fixed phenomenon, but we have to put quotes around to refer to it as a concept. The previous sentence, and entries in dictionaries, and some other specialized usages of language really treat a concept as a concept, but it usually requires special markers to do this. For instance, verbs have a special form (the infinitive) which is used to refer to the concept, all other forms inevitably denote an otherwise anomymous phenomenon covered by that concept.

After dealing with fixed references to phenomena, concepts used as fixed references, and concepts used as concepts, consider the possibility of using fixed references to phenomena as concepts. This is not rare either; generalizations, metaphors, word explanations, and questions often do that. For example, consider the exchange "Mom, what is a car? A car is just like our Morris, only a lot bigger and faster!". Here, the Morris (which references a concrete phenomenon) is used to define the concept of a car.

Of course, the quoted remarks have more, and more subtle meanings than just "everything = nice dinner", but that just serves to remind us that any attempt to exhaust the meaning of a piece of natural language is likely to fail. What we *do* want to extract from this discussion is the principle of using an entity of another kind than the contextually appropriate one, and then implicitly coercing it into the right kind. For example, using a word that really designates a concept in a position where a simple reference to a phenomenon is expected, and then implicitly solving the problem by effectively redefining that word to be a locally defined fixed name.

2.3.2 Transparency in Programming Languages

Transparency is a further development of the natural language mechanism which was treated in the previous section, emphasizing the complete absence of local clues to the transformations. When using one entity as if it were of another kind, the statically or dynamically available information about its kind is used to provide the appropriate implicit transformations.

For example, if the retrieval of the state of a variable has the same syntactic appearance as the invocation of a function then the two could not be distinguished at usage points. This is covered in detail for Eiffel on p. 57 and p. 175 in [79], and motivated with the improved freedom to change the implementation without affecting usages. In CLOS [56, p. 72], the freedom to change implementation is again given as the primary reason for this transparency, which is in this case provided by means of accessor methods. An *accessor method* simply ◇ retrieves or updates the value of a variable; exclusively using accessor methods for access to variables ensures that it is indistinguishable from having two methods with whatever implementation. Self uses this approach [2], by letting the name of the slot be the getter (e.g. '`x`'), and the name with a colon appended be the setter ('`x:`'). For Cecil, the accessor method based approach is presented on p. 13 of [21], this time also mentioning variables overridden by methods and vice-versa as a benefit. Dylan introduces slots along with 'getter' and 'setter' methods on p. 57 of [97], and Sather [102] also consistently defines accessor methods. In Java, accessor methods are described as a useful programmer convention on p. 41 of [6], and in C++ [31, 104] the use of accessor methods to hide the implementation is considered well-known, e.g., on USENET news groups.

In other words, there is overwhelming concensus on the benefits of the kind of transparency provided by accessor methods. However, as the Eiffel presentation mentions explicitly, only functions with no arguments delivering one result may appear the same as a variable. In those cases (as in Cecil) where variables may be assigned remotely (from "outside" the object, like `obj.x:=5`), also procedures taking one argument may appear the same as a variable—since assignment is syntactic sugar for a call of the setter method, both method call syntax and "`:=`" may be used in both cases.

Summing up, this transparency is achieved by forcing all accesses to variables to be method invocations, which will then (of course) be indistinguishable from other method invocations with similar arguments and returned results.

Another kind of transparency hides dereferencing of pointers, as for example C++ reference types which are used like direct object denotations, contrasting with traditional pointer types where the dereferencing operation is explicit at every usage point. This kind of transparency is not as frequent as the previous one, which is quite natural since accessor methods hide the difference between direct access or dereferenced access anyway. Moreover, many languages including Simula, Smalltalk, and Cecil only support indirect attributes, preempting that kind of transparency, too.

2.3.3 Transparency in BETA

In BETA, a more general approach is taken, thanks to the unified evaluation syntax. The transparency of stored vs. computed values is not achieved by enforcing accessor methods for all variable accesses, but by implicitly transforming the entity at the usage point, somewhat like the situation in natural language as described in Sect. 2.3.1. For example, when a term is a denotation of a pattern (e.g., '`p2.move`' below), the semantics of executing that term is to create an object as an instance of the pattern, and then to execute that object. The traditional BETA terminology for this is to call the syntax denoting that pattern an *inserted item*. Consider the following example:

```
(# point:
     (# x,y: @integer;
        move:
          (# dx,dy: @integer;
          enter (dx,dy)
          do x+dx->x; y+dy->y
          #)
     enter (x,y)
     exit (x,y)
     #);
   p1,p2: @point
do
   (3,4)->p1->p2.move
#)
```
Ex. 2-1

In the outermost DoPart, `p1` is an object which is being assigned the state `(3,4)` and then evaluated, providing the argument list `(x,y)`, which also yields `(3,4)`, to the method invocation `p2.move`.

Hence, the usage of objects and patterns appear the same, with object creation happening implicitly. The unification of syntax for argument transfer and assignment, along with general support for tuples of values, ensure that function calls and method invocations with any number of arguments and returned results can appear the same as evaluation and assignment with (possibly variable) objects.

The approach which uses accessor methods only makes different entities appear the same insofar as they may always meaningfully be treated in the same way. With variable patterns, for example, it does not make sense to force them into being accessed through accessor methods, because changing the variable pattern and using it as a method (which might take one argument which might be a pattern) would compete for the meaning of a 'setter' method.

So instead of enforcing just one access path to entities, we provide different contexts which declaratively require different entities. This will be detailed in the next section which explains the approach taken in gbeta. Since the gbeta approach is a generalization of the BETA approach, the BETA rules will be characterized as restrictions of the gbeta rules at the end of the next section.

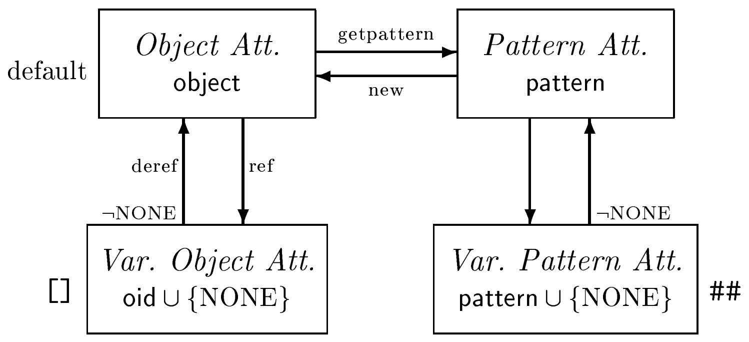

Figure 2.4: Coercion in execution, evaluation, and assignment

2.3.4 Transparency in gbeta

This section was written to be read and forgotten! It exposes all those coercions between different kinds of semantic entities (run-time entities) that transparency is there to hide. These coercions will be inserted by the compiler in the right places, such that names can be used in the same way even though they may be defined as different kinds of attributes, e.g., as an object or as a pattern. However, the details must be described at some point, and that point is this section.

As explained in Sect. 2.2.5, gbeta provides four basic kinds of attributes, namely object, variable object, pattern, and variable pattern. In figure 2.4, the coercion mechanism for execution, assignment, and evaluation (i.e., everything except declarations) is specified. The figure contains much information, so it will be described in details in the following.

The four kinds of attributes are shown, one in each box, using an *italic* typeface, along with the associated semantic entities, such as object or pattern. The arrows between the boxes represent coercions. For example, with a given variable object attribute, an object identifier (member of oid) or NONE can be obtained—that's what a variable object attribute contains; this object identifier can then be coerced into an object unless it is NONE, as indicated by the ¬NONE annotation on the leftmost upward arrow. If object identities are represented simply as memory addresses, the coercion would be a dereferencing operation, but other operations might be used with other representations of object identities. The important thing is that we can get hold of an object.

Coercion happens when a piece of syntax which denotes an attribute is used in some syntactic context. In the general case, that piece of syntax is an AttributeDenotation, see the full grammar in App. A, but in the simplified grammar used sofar, it is just a Name. There are three different *syntactic contexts*, namely ◇
'Name[]', 'Name##', and the *default context*. The default context applies in all ◇
other cases, i.e., for all names not followed by one of those two *coercion markers*. ◇

The coercion is a journey from one of the boxes to another one, consisting of the actions associated with the arrows on the path. The starting point is determined by the kind of attribute which the Name is declared to be. For example, it would start in the top left box for an object attribute. The destination of the journey is the box which is marked with the syntactic context, i.e., with '`[]`', '`##`', or 'default'. Given a starting point and a destination, the path of arrows is fully determined, and the coercion can be described. The following piece of code uses all paths through a number of examples:

```
(# i: @integer;
   s: ^string;
   p: (# #);
   pv: ##object
do
   i; s; p; pv;
   i[]; s[]; p[]; pv[];
   i##; s##; p##; pv##;
#)
```

Ex. 2-2

The attributes `i`, `s`, `p`, and `pv` are of all kinds, namely object, variable object, pattern, and variable pattern, respectively. The DoPart then systematically puts them into the three different syntactic contexts, hence causing coercions from any of the four starting points to any of the three destinations.

The first imperative, '`i`', causes the empty coercion, because an object is needed and that is exactly what the attribute already denotes. In contrast, the imperative '`pv[]`' in the next line of the program causes a coercion with several steps. The starting point is the bottom right box since the attribute is a variable pattern. The destination is the bottom left box, since the coercion marker is '`[]`'. As a consequence, the following actions are taken: It is checked whether `pv` is NONE; if it is NONE then a run-time error is raised, otherwise the pattern is obtained. Then the pattern is instantiated, yielding a new object. Finally the identity of the object is obtained, and that is the result.

Note that an imperative like '`s[]`' does not do anything, since the object identity is obtained and then immediately discarded, but in order to explain coercion these more or less silly imperatives are the simplest possible examples. A more useful imperative could be like '`pv[]->m`', which would perform the same coercion on `pv` as above and then give the obtained object identity as an argument to the invocation of the method `m`.

◇ There is one *anomaly* in this system, namely that assignment to syntax in a '`[]`' or a '`##`' context requires that this syntax denote a variable attribute of the destination kind, i.e., variable object or variable pattern, respectively. In other words, no coercions are allowed when using one of the two markers on the right hand side of an assignment. For example, with '`m->s[]`' it is required that `s` denote a variable object attribute.

The problem is that assignment with non-trivial coercion to, e.g., the lower left box would have unwanted semantics. Consider the case where the attribute denotes a pattern, `p`, the imperative is '`m->p[]`', and the evaluation of `m` delivers the object identity ϑ. If this were to be allowed, then the assignment should change `p` in such a way that future evaluations of `p[]` would deliver ϑ, at least until the next change.

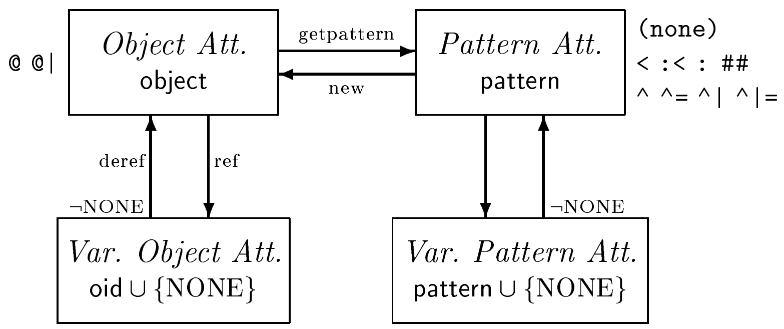

Figure 2.5: Coercion in declarations

However, this would not match well with the rest of the language. Whenever an object or object identity is requested from syntax denoting a pattern, the object will be instantiated afresh. So an object identity can be obtained from a pattern, but we cannot change *what* object it will deliver the next time. In this respect the pattern attribute is similar to an object attribute—the object attribute invariably denotes one fixed object, and the pattern attribute delivers a different object at every request, but both of them have so strict semantic constraints on what object to provide that the semantics of variable assignment is incompatible.

Hence, the situation is similar to that of assigning to a constant attribute in other languages, like '`const int i=1; i=2;`' in C++, which is of course also rejected at compile-time. Note that languages with accessor methods, e.g. Cecil, have a similar behavior: A constant variable or field has only a reader accessor, no writer. As a consequence, any attempt to assign to such an attribute will lead to an error because the required method is missing. In those languages it is possible to add a user-defined method with the signature expected of a writer accessor, thus allowing for a user-defined resolution of the conflict. A similar approach could be used in gbeta, but such a feature has not yet been designed in detail.

Hence, transparency is very complete for evaluation and for value assignment, but the coercion markers '`[]`' and '`##`' used on the receiving side of an assignment break the transparency with the current language design.

We have described the coercions associated with execution of code; the rest of this section deals with coercions in declarations. This is simple as there are solely two groups of attributes: The pattern attributes and the variable attributes, both variable objects and patterns, require a pattern on the right hand side of the declaration, and this pattern is obtained by coercion. The object attributes, being the only attributes not yet covered, require an object on the right hand side. Figure 2.5 describes the mechanism. The only difference between Fig. 2.4 and Fig. 2.5 is the annotation which defines the destination of coercions in different contexts; for declarations the destination is determined by the Kind of declaration, as introduced in Fig. 2.2 on page 24.

Especially two consequences of this are interesting. The syntax this(Name) is used to denote the nearest enclosing object which is statically known to be an instance of a specialization of the pattern obtained from the given Name; it is similar to this in C++ and self in Smalltalk, except for the usage of a Name to select the right one out of the potentially many enclosing objects. Consider the following example:

```
link: (# next: ^this(link); value: @integer #)
```
Ex. 2-3

The link pattern represents a basic singly linked list where each link in the list may hold an integer value and a reference to the next element in the list. Since the qualification of next is obtained from the denotation of the link object itself, it denotes the pattern of that object. This might be link, but in a subpattern of link it would be that subpattern. In other words, this is a genuine 'SelfType' or 'MyType' [15]. In the BETA community there has been some discussion about defining a special construct to be able to provide genuine self-types [73], but this has not yet been implemented nor completely designed.

There is a well-known workaround which uses a virtual pattern that the programmer must manually redeclare in all subpatterns, as in the following example:

```
link:
  (# selfType:< link;
     next: ^selfType;
     value: @integer
  #)
```
Ex. 2-4

The workaround is error-prone, and it does not have the right typing properties: The type system cannot assume that the selfType virtual is the pattern of the enclosing object, because there is no guarantee that it will actually be that pattern. If the programmer forgets to further-bind the virtual in a new subpattern, it will not anymore be a correct self-type.

Another case where the coercion in connection with declarations is useful, if perhaps not beautiful, is the case where an object attribute has a specification which is a variable object. An example is the following:

```
(# X: ^somePattern
enter X[]
do (# constX: @X do ... constX ...  #)
#)
```
Ex. 2-5

In the outermost MainPart, the variable object X can be used for many things, but sometimes it must be assured for application specific reasons that different usages of a name actually refer to the very same object. Moreover, this is also valuable in the type analysis, because it may prove that certain patterns are the same even though it is not known what pattern it is, thus proving, e.g., an assignment type safe without exact knowledge about the involved patterns.

To obtain such an immutable object name, we can declare an object attribute which denotes the object available from X at some point. To do this we need a place to put the new declaration. This is achieved by creating an anonymous object, using the inserted item syntax (which denotes a pattern which is then by coercion instantiated and executed), as with `(# constX: @X ... #)` in the DoPart. The name `constX` can be used in the DoPart of the inserted item, and that name will invariably denote the object which was available from `X` when the inserted item was created. This is an example of the 'snapshot' semantics which is presented and motivated in Sect. 3.9.

Note that a similar semantics with the same syntax is obtained with a different approach in [12], which is based on a generalization of the notion of qualifications.

Finally, the relation between gbeta coercion and BETA coercion can be described. The entity transformations presented in this section have not traditionally been described in terms of a consistent coercion scheme in BETA; indeed, not everybody in the BETA community accept this as a natural explanation of the BETA semantics. However, the actual behavior of BETA programs can be described exactly by the figures and explanations in this section, except for a few cases which are prohibited in BETA, namely:

- Coercions cannot have variable object as the destination except when the starting point is object. For coercions from pattern and variable pattern, the *new* operator, '`&`', must be added in front of the (variable) pattern denotation, like in `&p[]`.

- In declarations, only one kind of entity can be used—both the specification of an object attribute and the qualification of a variable attribute must be a pattern.

Chapter 3

Patterns

This chapter deals with patterns in gbeta, and since all the concepts are so tightly integrated this tends to touch on everything; so this is a long chapter with many different topics covered in various sections. The overall outline of the chapter is as follows: First there is a presentation of the basic premises and the building blocks from which patterns are constructed. Then properties of patterns as a whole are discussed, and then composition of several patterns into new patterns. Finally there is a discussion of a few satellite topics.

Section 3.1 explains that patterns are values and not objects, and why it is so. It is followed by a presentation of the concept of mixins in general in Sect. 3.2, and Sect. 3.3 gives a presentation of mixins in gbeta, along with the entities which are built out of mixins, namely patterns and their brethren, the objects.

Now that we have the value domain of patterns available the question about equality in that domain arises, and Sect. 3.4 covers both various kinds of equivalence which is used for classes and similar entities in other languages, and the very strict equivalence criterion which is used for patterns in gbeta (and in BETA).

Patterns in gbeta are organized into specialization networks, and this is a significantly more densely populated universe than the corresponding strictly tree-shaped specialization hierarchies in BETA. The relation between these two is covered in Sect. 3.5. With patterns organized into specialization networks it becomes possible to talk about superpatterns and therefore also about inheritance of attributes, as it happens in Sect. 3.6.

This establishes patterns as standalone entities, and Sect. 3.7 builds on this by describing how patterns can be composed into new patterns by means of merging. Any construction of a new pattern, including pattern merging and "plain, old inheritance" like in BETA, will affect the behavior associated with that pattern when it is used as a method. The topic of Sect. 3.8 is how such composite behavior can be created and explained in terms of the mixins and their do-parts in a pattern, and how that gives rise to a broader notion of specialization of behavior than that of BETA, but one that grows out of the

BETA tradition.

Finally there is a treatment of a few additional topics in association with patterns. The first topic is the notion of object creation by instantiation of patterns, which is covered in Sect. 3.9. After that, Sect. 3.10 goes into more detail about how attributes in objects can be accessed, specifying the local name lookup rules which are the basic elements of all name binding in gbeta. The last section in this chapter, Sect. 3.11 deals with the notion of qualifications, similar to such a notion as the declared type of references.

Hence, this chapter covers not only patterns, but also the building blocks from which patterns are built, namely mixins, and the entities which are created according to patterns, namely objects. It should be obvious that patterns are absolutely essential in the design of gbeta—a trait that gbeta has inherited from BETA.

3.1 Patterns are Values

In BETA and gbeta, patterns are values. This corresponds to the situation in natural language, where concepts are also in a sense values. Of course, in these matters there will never be absolute truths. However, a given concept, as designated by a spoken or written word, should be available from the appearance (sound or graphical shape) of the designation, such that understanding can proceed. The alternative view, being that the word denotes an entity with a unique identity and some internal state, does not really make sense, since there is no place to go and look up what that state is; moreover, in case of a change in the alleged state of the concept, should all the "instances" (mental images of phenomena in the extension) be "updated" to reflect the changes?

From a technical point of view, the fact that patterns are values in BETA makes it natural that (variable pattern) attributes may have them as values. These values are exclusively taken from the set of patterns known at compile-time. The patterns are organized into a *partial* order (the specialization order, see Sect. 3.5) which is also completely known at compile time. Because of this, there is no support for computation on patterns in BETA. Patterns may be compared for equality or inequality, but there is no way to compute a pattern by giving one or more other patterns as arguments to an operation.

However, there are cross-domain operations, like inheritance (see Sect. 3.6) which produces a pattern from another pattern and a mixin (see the next section), or like instantiation which produces an object from a pattern (see Sect. 3.9). The fact that BETA has attributes whose values range over patterns makes (the equivalent of) classes and methods first class entities. They may become even more first class, though.

In gbeta, a genuine operation on patterns is introduced, taking a number of patterns as operands and producing a new pattern. This introduces the concept of pattern computations. As an analogy, assume that we started out with a set of natural numbers, such that comparison would be almost the only supported use, and then enhanced it with an operation like 'addition', such that numbers not in the original set could be constructed. The operation in question, pattern merging, is presented in Sect. 3.7.

Since patterns are values and the merging operation is defined entirely in terms of those values, such pattern computations can be, and are, allowed at run-time in `gbeta`. This means that some patterns in a program execution may not be available for static analysis at compile-time, they are genuine run-time values. This makes patterns (classes and methods) even more first class than they are in BETA. Note that the type analysis of a dynamically constructed pattern is subsumed by an already existing case, namely the case where a given pattern is not known at compile-time but it is known to be a specialization of a given pattern; this is, e.g., typically the case with virtual patterns. Run-time construction of classes is covered in more detail both in other sections of this chapter and in Sect. 7.2.

Please note that this concept is significantly different from the ability in certain dynamic languages (like Self, CLOS and Smalltalk) and recently also in BETA to compile classes or other entities during a program execution, and then integrate the resulting compiled entity into the execution and continue running it. The Java dynamic class loader plays a similar role.

One important difference between dynamic compilation and real class computations is that a new, dynamically compiled class would have no particular relation to existing classes in the program, though it could be created as a subpattern of a class which is already known in the program, and integrated by means of subpattern polymorphism [75]. In contrast, the value of a pattern merging operation is fully determined by the (existing) patterns being merged, and the type analysis takes this knowledge into consideration (using whatever information is known about the operands). Dynamic compilation or loading yields results which are not so well integrated into the language, in particular in connection with static type systems. In Java [6, p. 315++], for example, a dynamically loaded class is *not* a first class entity similar to all all other classes; it is an object, instance of the class `Class`, and instances of such a class are created using the method `newInstance` instead of using the built-in operator `new`, and the new object is *not* recognized as having a type which is associated with the dynamically loaded class, it is just an `Object`, which requires a dynamic cast in order to be useful. In contrast, dynamically created classes in `gbeta` are fully integrated; the type system does not know the complete type, but whatever is known is used, and the type-checking situation is no different from the case with virtual patterns.

On the other hand, only certain patterns may be computed using pattern merging. An analogy would again be the addition of natural numbers. If all the numbers available from the beginning are even numbers, then there is no way to add them up to an odd number. Similarly, if there is no pattern in a program which has, say, a `print` method then pattern computations cannot be used to obtain one. See Sect. 3.2 and 3.7 for more details.

In other languages, classes and methods are not generally considered values. In Smalltalk where "everything is an object", the consistent choice is to make classes objects, too, and this is indeed the case. Consequently, new classes are newer computed, only compiled. A well-known result of letting classes be objects as well is that it is hard to find a simple, understandable way to terminate the chain of objects from a given object to its class, which is then also an object, to the class of the class object, which is then also an object, to ...

This infinite regress problem was actually one of the motivations for developing classless languages like Self and Cecil.

One special topic, which is linked to the question of whether classes are values or (more like) objects, is the topic of *shared state*. In many languages, a class may have some state which is accessible from every instance of the class by means of special attributes. These attributes are called static members in C++, static fields in Java, class variables in Smalltalk, shared slots in CLOS, and slots with class allocation in Dylan. 'Once functions' in Eiffel offer the functionality of on-demand initialization and (thereafter) shared state, all in all similar to the others. The motivation for this mechanism is that some tasks need to deal with all the instances of a class, e.g., counting all instances, keeping lists of them, or ensuring that at most one of them is in a special state.

It seems that this notion forces a class to be object-like, having its own state and hence having identity, such that aliasing is non-transparent and implicit copying generally not allowed. If classes are actually somewhat object-like, the natural question is "why not make them into real objects like in Smalltalk?"

The answer could be that those classes are *not* objects to any significant degree. For instance, if the core properties of a class—the description of the structure of instances—were actually mutable, then it would be possible to add, remove, or change attributes described by the class. That immediately raises the question already mentioned above: When the class changes, should all the existing instances "automagically" be updated? In CLOS it *is* possible to change a class in a running program, and objects *will* be updated by a user-defined or automatically generated version updating procedure; but this is again dynamic compilation as opposed to an integrated language feature. It would not be very easy to reconcile with static type-checking, either, as is demonstrated in the Orm system [58, 53] where this topic has been explored—only a quite limited class of special cases can be handled without running the entire type-analysis again, and the running program execution is lost when the type-analysis must be re-done.

In other words, the identity and mutability aspect of classes associated with shared state are marginal to the class as such. Since classes in those languages are constant, global entities, a more suitable explanation of the shared attributes is that they are simply global variables whose names are made available in a special name space which is identified with the class name. This is actually also a common way to describe what static members are, in the C++ community. Without shared state, classes in C++, Java, and Eiffel become similar to Beta patterns (used as classes), because they are values, but there is no support for computation with those values.

By the way, the natural way to obtain shared state in languages with general block structure like Beta and gbeta is simply to move the declaration one level out in the block structure [71, 72].

3.2 Mixins

In the CLOS community there is a very old and well-established programming technique associated with the mixin concept [56, p. 46]. A *mixin class* in CLOS is technically just a class like other classes, but it is intended to be used in a special way. It should be one of several super-classes in an occurrence of multiple inheritance, and it will thus be "mixed in" with the other super-classes. Other classes must be present for the mixin class to function correctly, because methods of the mixin typically use attributes—*ghost attributes*—which are not defined in the mixin class itself but are *expected* to be provided by the other classes. When looking at a mixin class in isolation it looks just like a run-time error waiting to happen, but this is acceptable since there is no static type check, and with the right companion classes, the properties needed by the mixin class will actually be available.

The mixin class mechanism depends on the linearization used in multiple inheritance in CLOS (and similarly for LOOPS, Dylan, and others). A *linearization* is an algorithm or a specification which reshapes a given directed acyclic graph into a list—it linearizes the graph. The list must be a topological sorting of the graph. The precise details are described in Sect. 3.7. The graph in question is the set of superclasses of a given class, connected with edges from every class to each of its direct superclasses; this is usually called the inheritance graph.

For an attribute which is used in a mixin class but seemingly not defined—a ghost attribute—there is no path in the inheritance graph from the mixin class to any class which declares that attribute. Accordingly, a direct translation of the technique into C++ or another statically type checked language will just cause a compile-time error, since the ghost attribute is 'not defined'. However, the linearization process generally changes the set of reachable classes from any given class in the inheritance graph, and this may add a class which actually declares the ghost attribute to the set of classes reachable from the mixin class. In other words, the reorganization of the inheritance graph gives the mixin class one or more *new superclasses* which may provide declarations for ghosts in the mixin class. This motivates an alternative name for mixin classes, namely *abstract subclasses*.

Since this technique is not type safe, and because it is so intertwined with linearizing multiple inheritance, a further development of the idea has lead to a separate concept of *mixin*s, different from the concept of classes, but related to it. The first, ground-breaking paper which introduced the mixin as a separate concept was [10]. The connection between mixins and classes lies in the inheritance mechanism. Inheritance allows for the creation of one new class based on zero or more superclasses and a specification of a *class increment* (often a block enclosed by braces, `{ ... }`, and containing a list of declarations; in BETA and gbeta the syntax for the increment is the MainPart, see Sect. 2.2.4 and 2.2.5).

Mixins liberate the incremental specification entity such that it can be applied to several different superclasses, instead of being an inseparable part of one particular occurrence of inheritance. The benefit derived from this generalization is that one increment can be reused in several different contexts. Otherwise, in a language without mixins, it would have had to be textually copied for each usage, with the well-known adverse consequences for maintainability, flexibility, readability, etc.

The incremental specification entity may be a function from classes to classes as in [45], or it may be a class-like entity which can be composed using special mixin-composition operators [10, 9], or it may even be a method whose execution enhances the structure of the enclosing object [99]. In any case, the application of a mixin to an actual superclass resembles inheritance, hence the alternative name *abstract subclass* for mixins. Note that this terminology might be confusing because a mixin is not a class; the term might actually be more appropriate for the usage of mixin classes, as in CLOS.

In a statically typed language, the not-yet-known superclass of a mixin must be characterized somehow, before the usage of inherited (ghost) attributes in the mixin can be type checked. In [45]—which deals with a subset of Java enhanced with mixin support—this is achieved by requiring that mixins specify
◇ an *inheritance interface*. This is an interface which is *assumed* of the formal superclass during checking of the mixin, and *required* of the actual superclass at mixin application. With an inheritance interface it is made explicit exactly what parts of an actual superclass the mixin depends on, and it is made possible to type-check the mixin (and generate code for it) once and for all. It will then be a robust, reusable abstraction.

Finally, we should mention raw, text pre-processing approaches to providing mixin-like functionality—to explain why such solutions are insufficient. In C++, there is a well-known technique which parameterizes a "class" with a superclass, thus enabling the creation of several different classes using the same piece of syntax as a class increment. The technique is to create a template class which inherits from one of its formal parameters; different instantiations of the template with different classes as arguments for that parameter will then work like applications of the "mixin".

This may seem to be genuine support for mixins, but there are some significant drawbacks. Firstly, C++ templates are essentially textual macros, since implementations do not, and cannot, analyze templates statically nor generate code for them; instead, analysis and code generation must run from scratch for every different instantiation, as if the code had just been put into a new context with copy/paste. This takes time and space—in particular the space usage is known to be a serious problem in practice. More importantly, the analysis may reveal errors deep inside the implementation because, e.g., a class given as a template argument does not happen to declare a specific method. Nothing less than the entire implementation suffices to determine whether a given template argument is appropriate, so there is no encapsulation robustness, no abstraction.

Secondly, since any two instantiations with different arguments are analyzed independently, the meaning of names used in the implementation of the template can vary freely. In all other parts of C++, any given name application can be annotated with information about the kind of entity denoted by that name as well as the statically known type; but for names used in a template, neither is

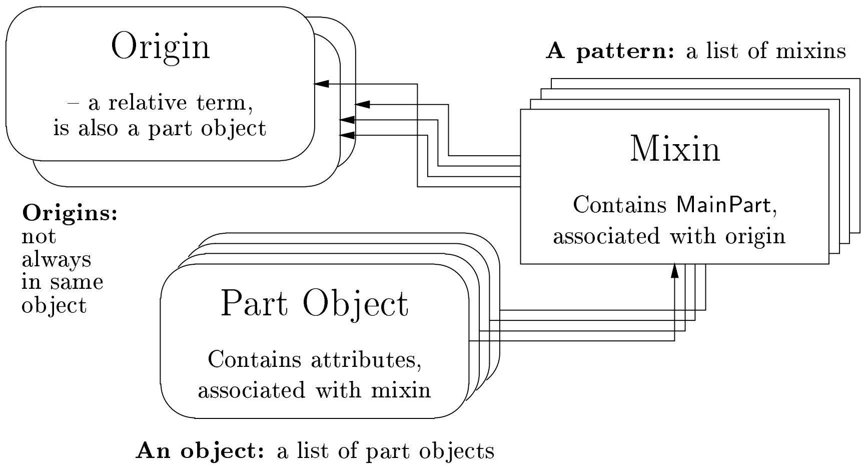
Origin
– a relative term, is also a part object
Origins: not always in same object
A pattern: a list of mixins
Mixin
Contains MainPart, associated with origin
Part Object
Contains attributes, associated with mixin
An object: a list of part objects

Figure 3.1: Overview of mixins, patterns, part objects, and objects

known. Even though the binding is per-template-instantiation, i.e. at compile-time, this is similar to having *dynamic* name binding because the information about kind and type is not available at the template declaration. Such a subtle change in semantics is confusing and error-prone.

3.3 Mixins and Derived Entities in gbeta

This section presents gbeta mixins along with the derived entities, patterns and objects. At the conceptual level, mixins in gbeta are best viewed as aspects of concepts, i.e., as entities similar to concepts but so intimately dependent on something else (such as other mixins) that they are simply unthinkable in isolation. Compare this to the relation between the concepts of 'person' and 'musician'; there *is* a difference in the amount of knowledge we have about a phenomenon which can be described as a 'person' and one which can be described as a 'musician', but it would not make sense to try to isolate this difference such that it could be used without any reference to the underlying 'personality'.

However, the presentation in this section is concerned with patterns and objects as semantic entities, i.e., as analyzable, constructible phenomena within a computer or in a formal semantic specification (at this level of detail it fits both). Using the general knowledge of mixins from the previous section, mixins in gbeta can now be characterized in a rather terse manner.

In gbeta, the specialization relation, the inheritance mechanism, and the pattern merging operation are all best explained in terms of building blocks which function much like the above described mixins; first it will be described what they are, then the other entities are described in terms of mixins, and finally the gbeta mixins will be compared with other kinds of mixins.

◇ The syntactic representation of a gbeta *mixin* is a MainPart; all mixins are either basic or associated with a MainPart, and basic mixins come in the usual variants `boolean`, `char`, `integer`, `real`, and `string`.[1] Each MainPart may be part of zero or more mixins in a given program execution; a MainPart in a gbeta program may *only* affect program executions by being used in the creation of mixins; patterns are created entirely from mixins; and each objects is created according to a pattern.

◇ A gbeta *pattern* is a list of gbeta mixins. Each gbeta *object* is a list of part
◇ objects, and each part object is associated with one mixin. This associates the
◇ object as a whole with a list of mixins, i.e., a pattern. That is the *pattern of* which the *object* is an instance ("its" pattern). We may also consider each part object an instance of the associated mixin.

A mixin consists of two components, a MainPart and an enclosing part object,
◇ called the *origin* of the mixin. The origin of a mixin is used as the origin of any part objects which are instances of the mixin. The origin of a part object is the execution context for the part object: Whenever a name lookup process starting in the part object needs to search the enclosing environment, the origin is used. A lookup process proceeds to the enclosing environment when the local environment does not provide a declaration of a given name. This is covered in more detail in Sect. 3.10 and in Chap. 5.

Each part object has originally been created according to some mixin and thereby becomes (and will forever be) an instance of that mixin. In this creation process, the mixin is used as a "blueprint", so the attributes available in the part object are the ones described by the declarations in the MainPart associated with the mixin.

All this describes what a gbeta mixin is, and how it is situated at the core of the structures in terms of which gbeta programs are executed. The details of how to build patterns from mixins are given in Sect. 3.6 and 3.7, but at this point we may still compare some more high-level properties of mixins with the other variants of mixins, descibed in the previous section.

Like other genuine kinds of mixins, gbeta mixins are not based on macro expansion; they are analyzed statically (and code can be generated) once and for all. Similar to mixins in [10] and [9], gbeta mixins are entities which are used as building blocks for the creation of patterns. Like mixins in [45] which must declare an inheritance interface, gbeta mixins only interact with actual superclasses in ways which can be detected statically at the declaration point—there are no ghost attributes.

However, the interaction specification is a standard pattern, not a special purpose inheritance interface. The inheritance interface only allows the mixin to depend on some method signatures in the actual superclass, not for instance to access state (fields). This corresponds to the special case of using a pattern with no state and no implemented methods as the interaction specification. Even

[1]There are actually a few more basic mixins, including `component` and `semaphore` which are associated with non-sequential execution and mentioned in Sect. 9.5

though this special case may often be a desirable choice, the general case where the interaction specification is a pattern of any kind yields greater flexibility.

Note that the development of mixins has almost closed a circle. The starting point was a special usage of technically ordinary classes in CLOS and similar languages; mixins were then separated out as a new concept, different from a class; later, the interactions between a mixin and its actual superclass were made explicit by means of inheritance interfaces; and finally, in gbeta, inheritance interfaces were generalized to ordinary patterns.

The situation is then again similar to the starting point, because mixins in gbeta may only be specified together with a pattern on which they depend (or, if that pattern is missing then the mixin must not depend on the actual superclass at all), and this specification of a mixin together with its dependency looks just like an ordinary declaration of a pattern, enhancing a given superpattern with a new increment using inheritance. Actually, the same declaration may be viewed as a declaration of a new pattern, or as a declaration of a mixin together with a specification of its dependencies—the difference lies in the usage.

This is consistent with the rest of the language where, e.g., patterns unify methods and classes, leaving it open for the programmer to decide whether to view a given pattern as a method or as a class. There is a subtle richness in having an entire spectrum between method and class, or between mixin and pattern, instead of just having the end points of the spectrum.

However, in order to explain the structure and construction of patterns, and in order to define the specialization ("isa") relation, the only reasonable approach is to consider the mixins individually. That is the way it was designed, and that is the reason why these sections on mixins precede the sections about inheritance and merging.

3.4 Equivalence

This section is about *qualities*, not in the sense of "good quality" or "bad quality", ◇ but in the sense of immanent differences, such as the difference between tomatoes and oranges. An *immanent difference* is an unexplained difference; two things ◇ are considered "just different" without having a common deconstruction in terms of which the difference is accounted for. Conversely, e.g., a paper bag containing two tomatoes and a paper bag containing three tomatoes are different in terms of an analysis which describes them as composite entities built out of the *same* kinds of building blocks. That is an explained difference, a *completely modeled* difference, not a difference in qualities.

The notion of qualities is a mental device which is needed because models are not faithful. As usual, instead of copying the humongous complexity of the real world and hence recreating any difference in behavior or properties by means of copying the mechanisms, we use qualities as a mental device to obtain useful models with much less complex structure. Tomatoes and oranges are "just different", axiomatically different, and then we may enrich our knowledge about tomatoes and about oranges independently, by adding typical properties to the concepts. At no point are those concepts brought into a common, commensurable state, where all the differences are completely explained within the model. To emphasize the point: qualities are concerned with differences which are taken for granted, as opposed to differences which are modeled; it is a property of a model and not a property of the real world.

Our claim in this section is that support for qualities is obviously needed in a programming language, and that we may choose between different degrees of support for immanent difference. Moreover, these different degrees of support correspond to a technical topic which is usually called type equivalence.

◇ In the programming language universe, the *absence of qualities* corresponds to machine code, where the bit is the only kind of matter and everything else is just collections of bits and manipulation of bits. All differences are structural. Similarly, untyped lamba calculus is also a world of pure structure. Such a smooth and homogeneous world may be considered "clean" and ideal, but for modeling purposes it does not suffice, since the recreation of differences by mechanism is too heavy-handed; some things are "just different" and we should be able to postulate that. However, support for immanent (postulated) differences must generally be provided on top of the native, smooth, quality-less universe of raw machine code execution. This is done by imposing a certain discipline on the usage of the raw bits.

The first step in the direction of support for qualities could be the separation of differently sized chunks of memory, e.g. distinguishing between an 8-bit `byte`, a 16-bit `word` and a 32-bit `dword`. The idea is that there should be a discipline on the usage of memory which ensures that a given bit is treated consistently as being a member of just one of those types of chunks of memory. Assembler language typically helps enforcing such a policy.

Memory chunks of the same size may be considered different, as when a language defines the types `int` and `float`—both 32-bit entities, but intended to be manipulated by means of different procedures (integer vs. floating point operations). This is an example of support for built-in qualities. Let us call ◇ built-in qualities *primitive types*.

◇ At this level we might also introduce a concept of *record types*—composite units of storage which are defined inductively as containing entities of primitive types, or of other record types. Using C syntax we can illustrate a couple of record types:

```
struct point { int x,y; };
struct position { int longitude, latitude; };
```
Ex. 3-1

The question about type equivalence arises as soon as there is any mechanism available through which programmers may define "similar" types. Respecting the built-in qualities and the user-defined composite structure, it is only natural to consider `point` and `position` equivalent. Whether a usage of a given entity occurs based on the `point` or on the `position` declaration, the two contained `word` entities will be used according to the discipline we decided to enforce at this level. This kind of type equivalence might be designated *pure structural*

◇ *equivalence*. Note that it is not too practical, because the equivalence may be ambiguous. In the above example we might identify `x` and `longitude`, but we might just as well identify `x` and `latitude`. Both choices would ensure the required discipline on the underlying usage of memory.

To describe the next level of support for qualities, we must introduce the concept of a *record path*. With a given record type, an entity of that type is ◇ known to contain entities which may be accessed by name, according to the list of declared names in the record type definition. If such a name is used to access a part of the entity which is again of record type, a similar step may be taken from there. Thus, a list of names corresponds to a process of repeated subentity selection. Let us call such a list of names a record path.

Now, the next level of support for qualities takes record paths into consideration, by considering those types equivalent which support the same record paths leading to entities of the same primitive type. Note that recursion, along with pointers with a special NULL value or disjoint sum types (tagged unions), would introduce infinite sets of record paths, making the type equivalence check more complicated. With the requirement that each record must define unique names (i.e. no name can be declared twice in the same record), this may ensure unambiguous mappings between equivalent types. In such a type system, the following two record types would be equivalent:

```
struct mypoint { int x,y; };
struct yourpoint { int y,x; };
```

Ex. 3-2

An example of a mapping which would ensure unambiguous access to the underlying memory would be to sort the declared names alphabetically and store the entities in that order (`x` before `y` in both `mypoint` and `yourpoint`). Note that even though we have referred to stored values only, and referred to them as if the representation should be trivial (like mapping high-level language names directly to offsets into contiguous areas of memory), the notion of type equivalence is independent of the representation. The important point is whether or not two given type definitions specify the same type.

This type equivalence criterion would normally be designated *structural equivalence*, even though it depends on both structure and naming. This kind ◇ of equivalence automatically equips composite entities with different qualities, when their internal structures are defined using different sets of names; but when the same set of names is chosen for both, then the types may be considered equivalent, depending on the types of the part entities.

It seems that the indirect derivation of qualities from the set of names inside a definition is of an accidental nature. In particular, when systems grow very large and complex, it is inconvenient to have to make a *global* search in order to ensure that some new type which is being defined will not by accident be confused with an existing type. The need to inspect the entire system to avoid accidental clashes (which will silently allow unforeseen actions) is not normally considered desirable in software engineering.

The other side of the coin is that types may be constructed in different places and still be the same, by using similar definitions. This is a typical correctness/convenience trade-off.

An example of a language where the underlying record type system is of exactly this kind is Objective CAML [94], which is an object-oriented extension of CAML, which is again a functional language in the ML family.

Note that the specialization ("isa") relation in Objective CAML further intensifies the problem of accidentally confusing entities which should be immanently different. For example, consider a `door` class for which the `open` method would send signals to some hardware in order to actually open a physical door; and a `window` class which is used to control a rectangular area of a bitmapped computer screen, and where `open` means 'initialize the internal data structures and show the window on the screen'. A type which just lists an `open` method might very well be a supertype of both `door` and `window`,[2] and an invocation of one `open` where the other was expected could be a disaster—imagine that somebody in the staff of a nuclear power plant wants to open the window showing the current state of a particular nuclear reactor, and that action in fact opens the door to that reactor and lets radioactive material flow out ...

The moral of this is that even a theoretically very well-founded and strict type system, like the one in Objective CAML, may actually exhibit correctness faults because it does not have a sufficiently thorough support for distinguishing different qualities. It is our opinion that it is naïve to assume that models will be complete and faithful, and structural equivalence seems to build on the assumption that there cannot be significant real-world differences between phenomena whose descriptions are formally identical—the structure of the model is everything there is to know.

Structural type equivalence takes the record paths into consideration, but ignores the *first* name, the name of the record type itself. A more strict version of structural equivalence could be defined by simply requiring that the declared names of types should also be the same, in addition to the record paths etc. as before. However, this would still build on a derived notion of qualities which would require global checks in order to avoid accidental confusion of similarly declared types. A difference in degree, only.

◇ Nevertheless, this kind of type equivalence, known as *name equivalence*, is quite widespread. That is because it often coincides with the next kind of type equivalence, as explained below. In fact, the presentation of name equivalence in [4] assumes this coincidence; it is unsound unless all names used in any two type expressions are defined in the same (flat, global) environment. It says, on p. 356:

> *Name equivalence* views each type name as a distinct type, so two type expressions are name equivalent if and only if they are identical.

The combination of this definition and the associated examples (in Pascal) clearly indicate that, e.g., two variables declared to have type '↑`cell`' are considered to have the same type under name equivalence, with no mention of the

[2]Assuming that all the involved `open` methods take no arguments and deliver no results, it *is* a supertype in Objective CAML

lookup process that determines the meaning of the identifier '`cell`'. If '`cell`' is looked up to mean '`integer`' for one of the declarations and '`string[30]`' for the other then the type equivalence obviously confuses two different types, and hence the existence of more than one scope for type names must be implicitly excluded from consideration. This is natural for Pascal but less natural for a, supposedly, generally applicable definition of name equivalence.

Another presentation is given in [44] on the pages 333–334. Type equivalence, which is here called type compatibility, is presented by comparing a notion of structural type equivalence with a so-called 'strict definition of type equivalence' associated with Ada, Pascal, and Modula-2. The strict definition considers two types equivalent iff they are defined in (syntactically) the same declaration, as tested in an implementation by comparing pointers to describing structures generated during a traversal of the program syntax tree.

These presentations, along with the actual semantics of Pascal, support the view that name equivalence is generally considered the same as the next kind of equivalence, based on program positions. However, since name equivalence does not seem to be very well-defined, and since the definition in [4] is unsound in any language which does not have one global name-space for types, the definition given here was created as a generalization which would make sense in a non-flat type name space, and which actually depends on names.

The next approach to type equivalence, *locational equivalence*, takes a radical step away from the smooth, homogeneous world of bits and lambdas. This kind of type equivalence is incompatible with the basic lambda calculus execution model, because it presupposes that the expression of the program, the source code, remains unchanged during the entire program execution. This deviation from the lambda calculus makes it harder to formalize the semantics using a direct translation of programs to lambda expressions, and this seems to have alienated many mathematically oriented researchers to the concept. It is as if they consider locational equivalence unacceptable, just because they cannot readily use the traditional approaches to formalization of the semantics. For example, it is stated by Abadi and Cardelli in [1, p. 27] that: ◇

> Structural subtyping [. . .] has desirable properties, such as supporting type matching [. . .] A disadvantage [of structural subtyping] is the possibility of accidental matching of unrelated types. In contrast, subtyping based on type names is hard to define precisely, and does not support structural subtyping.

After this, they exclusively use structural notions of types, with or without involving subtypes, in the rest of the book.

In lambda calculus, the ability to transform the program itself (β-reduction) is the only computational tool. The simplicity of the semantic model—only the program text itself is needed for a small-step operational semantics—is traditionally viewed as desirable, especially when the emphasis is on proving formal properties of executions. However, another basic model has proved useful in practical programming, possibly because it matches the way people think better than β-reduction does. In this model, programs are constants, and the execution happens in terms of a run-time system whose actions are *directed* by the program. Object-oriented languages generally fall in this category. Presumably, the human mind is capable of building an understanding of a static program in a similar way as it may get to know a physical landscape; there are "locations", each location has its own, characteristic properties, and the "distance" between different parts of the program may have been designed to reflect degrees of relatedness—when being in a particular location, the most relevant parts of the program are "nearby".

Taking it seriously that the program is immutable,[3] it becomes meaningful to refer to locations in a program. This is the basic mechanism behind *locational* ◇ *equivalence*. With this kind of equivalence, two types are considered the same iff they are constructed at the same location in the program, e.g., by the same declaration. This criterion has the nice property that any system, however large, will let a programmer define a new type and rest assured that it will be destinct from all other types in the program. In other words, this equivalence criterion takes the clean approach of providing syntactically distinct type introductions with distinct qualities.

The other side of the coin is that two similar types will be distinct, even if they should be considered the same; a separate mechanism may then be provided to allow explicit identification of two given types. Note that this does *not* introduce the need for global checks in order to secure against unexpected semantic effects. The failure to view two types as being equivalent may give compile-time errors, but it will not cause run-time confusion. The former is a much less serious problem since it is obvious, whereas the latter is silent at compile-time and probably subtle at run-time.

Revisiting the relation between name equivalence and locational equivalence, the two happen to coincide in the case where the type names are all defined at top-level. Since the top-level names are required to be unique, two types have the same name iff they are defined in the same location (otherwise they would have to have different names). With a partitioned global name space, as with Java packages and sub-packages, and with C++ name spaces, name equivalence is again effectively changed into location equivalence by considering the path of names of (sub)packages or name spaces as a part of the name of the type.

Turning to BETA, the notion of name equivalence or locational equivalence does not suffice for a characterization of the equivalence of patterns; location equivalence is the correct starting point, but the run-time environment must be taken into consideration, too. Note that location equivalence is defined on basis of the *construction* of an entity, in this case a mixin, not on the occurrence of a declaration. This makes a difference because BETA and gbeta support the specification of patterns in many places outside of pattern declarations; for example, a descriptor (see Fig. 2.1 on page 23) may be used in a pattern declaration, or directly as a statement. When used as a statement it constructs an anonymous pattern which is then by coercion instantiated and the resulting

[3]It may actually be extended, but not changed in ways which invalidate existing location specifications

object executed (see Sect. 2.3.4).

Even though it is not the tradition, patterns in BETA may be characterized in exactly the same way as patterns in gbeta, namely as lists of mixins, where each mixin is a pair, consisting of a MainPart and an origin, which is a part object that provides the mixin with an environment. See Sect. 3.3 for the introduction of these concepts.

Two patterns in BETA are equivalent iff they are equivalent as lists of mixins, i.e., if they have the same number of mixins and the mixins are pairwise equivalent. Two mixins are equivalent iff they are associated with the same MainPart and the same origin. Note that the definition of a mixin implies that patterns do not exist before run-time; they are genuine run-time values. Also note that this means that patterns are different just because they have different origins in one or more mixins; it is not enough that they are associated with exactly the same syntax (same MainParts at same positions in the list) and hence have the same set of defined names, according to the same attribute declarations. They must really be situated in the very same run-time context.

This very strict notion of pattern equivalence also ensures that a pattern is a *complete* generator of instances (objects); with a pattern alone it is possible to create a new object, and that object will be situated correctly in a run-time context. If patterns had only included MainParts and no origins, then an object created from a pattern would not have an environment unless it were specified explicitly, and that means that a pattern could not, for instance, work as a method: a method is generally expected to produce side-effects on “its object” (origin), and to do this it must have information about what object it “lives” in. Of course, the static analysis must deal with patterns in terms of a compile-time representation; this is covered in more detail in Chap. 13.

So, BETA supports an even finer distinction between patterns than ordinary locational equivalence. A consequence of this notion of pattern equivalence is that the number of different patterns in a program is *unbounded*, even though ◇ the number of syntactic occurrences of pattern declarations is of course fixed and finite. This makes it possible to distinguish between an unbounded number of different qualities associated with a given structure; all it takes to obtain N different patterns with a given declaration is to dynamically create N instances of some pattern associated with the MainPart which lexically encloses that declaration. As an example of how natural and useful this may be, consider the following:

```
university:
  (# student: (# ...  #);
     course:
       (# register: (# s: ^student enter s[] do ...  #);
       #);
     ...
  #);
Aarhus, UW, VUB: @university;
```
Ex. 3-3

With these definitions, the student pattern is nested in university. Consequently, the type system will distinguish between students at different univer-

```
(* first example: horizontal *)
a: $^1$(# #);  b: $^2$(# #);   c: $^3$(# #);

(* second example: vertical *)
p: $^4$(# #);  q: p$^5$(# #);  r: q$^6$(# #);
```

Figure 3.2: Two small BETA or gbeta examples, with numbered MainParts

sities, in particular the three different university objects `Aarhus`, `UW`, and `VUB` have distinct `student` patterns inside them. This supports correctness in the cases where students at different universities should actually be considered different. In the case where students at different universities should be considered equivalent, the simple change needed is to move the declaration of `student` out of the `university` pattern, placing it as a sibling to `university`, not as nested in and dependent on `university`.

Now that the description of equivalence of patterns in BETA has been adapted to use exactly the concepts which are suitable in gbeta, there is only one thing to add: Pattern equivalence in gbeta is exactly the same as pattern equivalence in BETA. Then note that gbeta has a much more flexible way to combine mixins than BETA, and hence there are many new ways to obtain equivalent patterns in gbeta.

3.5 The Pattern Space

As mentioned before, e.g. in Sect. 3.3, patterns in BETA and in gbeta are lists of mixins. This section describes *what* lists of mixins can be constructed. Section 3.6 and 3.7 describe *how* they can be constructed using inheritance and merging. Moreover, the organization of the set of patterns into a partial order relation is described and compared for BETA and gbeta.

For any given BETA program, the patterns may be organized into a tree, namely the well-known inheritance hierarchy which is associated with any single inheritance system. The root of the tree is `object`, and each node in the tree has its direct subpatterns as children. The `object` pattern is the empty list of mixins. The 'direct subpattern' relation is established by inheritance. Inheritance is treated in the next section, but for now it suffices to say that it works like the traditional `cons` function, taking a list L (the prefix, or superpattern) and an element m (the new increment, a mixin) and returning a list whose head is the element and tail is the list, written as m`::`L in, e.g., Standard ML.

Two small sets of pattern declarations are given in Fig. 3.2, and the corresponding BETA pattern spaces are shown in Fig. 3.3 on page 55. The *pattern* ◇ *space* of a program is the set of patterns which may occur during an execution of that program, organized into a partial order which defines the specialization ("isa") relation. In this case we ignore the origins, for simplicity and because

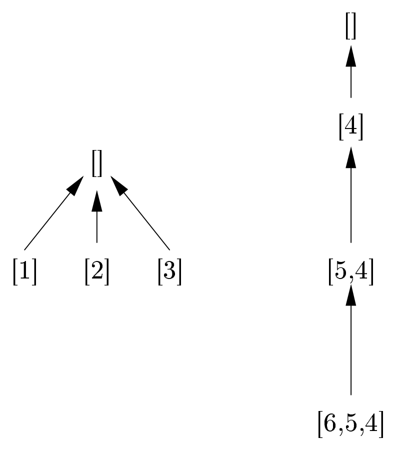

Figure 3.3: The BETA pattern spaces of the examples in Fig. 3.2, using the MainPart numbers

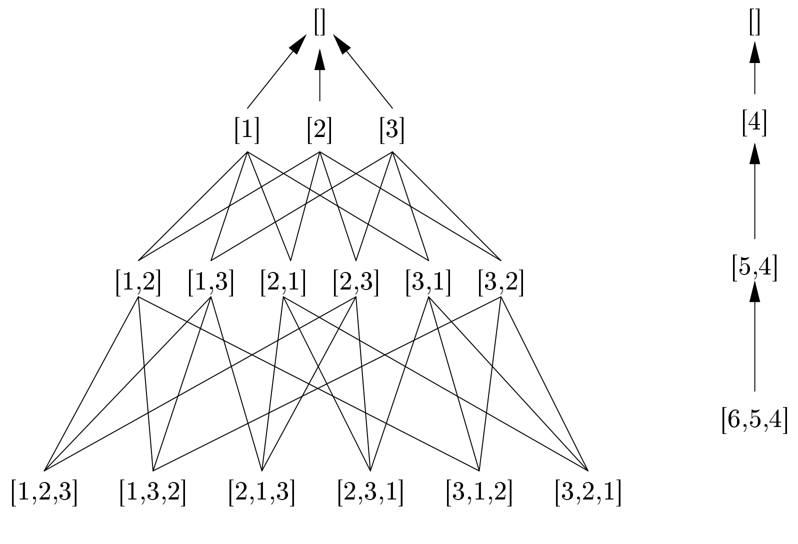

Figure 3.4: The gbeta pattern spaces of the examples in Fig. 3.2, using the MainPart numbers

the origins would be the same for all mixins anyway.

When looking at Fig. 3.3, there is no direct reference to the names of patterns, such as a or q. This is because the patterns cannot be specified in terms of the declared names—they denote entire patterns, and we need to explicitly show the construction of each pattern as a list of mixins. To be able to name the individual mixins, every MainPart has been annotated with its own, unique number in Fig. 3.2; since the origins would all be identical, each number denotes exactly one mixin.

In BETA, in these examples and generally, every pattern in the pattern space is directly associated with a pattern constructing piece of syntax (mainly pattern declarations and inserted items). By disregarding the origins, the pattern is fully specified somewhere in the program by syntax and name lookup rules alone. Note that this even holds for virtual patterns: Even though the value of a virtual pattern may not be known at compile time, there is always *some* declaration in the program which defines the most specific, actual value of it. Virtual patterns will be treated in Chap. 4.

Moreover, since the BETA pattern space is always organized into a tree with every mixin occurring in only one node of the tree, it is actually possible to identify mixins and patterns in BETA: for any given mixin, there is one and only one pattern which has this mixin as its most specific, i.e., as the head of the list. Also, when starting from any node in the tree, each step towards the root removes one mixin from the pattern, namely the head of the list, thus producing *the* direct superpattern.

Consequently, the set of patterns in a BETA program is actually a very sparse selection of lists of mixins, considering how many combinations there are with a given set of different mixins.

In contrast, the pattern spaces of the two examples when considered as gbeta are quite different, as shown in Fig. 3.4. The gbeta pattern spaces contain the BETA pattern spaces because the BETA semantics is a special case of the gbeta semantics. But in the first, horizontal, example, the gbeta pattern space is much larger than the corresponding BETA pattern space. This is because the mixins can be combined freely, using the merging operator (see Sect. 3.7).

When using pattern merging like multiple inheritance to bring together entirely unrelated patterns, there is no difference between, e.g., $[1, 2, 3]$ and $[3, 1, 2]$. But when behavior is involved it makes a big difference, and any one of the combinations may be the right choice in a concrete situation. Section 3.8 gives more details about behavior in connection with patterns containing more than one mixin.

So the horizontal example was quite different in BETA and in gbeta. On the other hand, as the figure shows, there is no difference between the BETA and gbeta pattern spaces with the second, vertical example. The situation is different because the mixins '4', '5', and '6' are introduced differently. In Fig. 3.2, the mixin '5' is introduced in a Descriptor where p is used as a superpattern. This makes a difference, because all the p mixins are then guaranteed to be present in *every* pattern where mixin '5' occurs, and hence the declarations provided by the p mixins may freely be used when binding names and checking types in the '5' mixin. This is a general phenomenon: A Descriptor binds mixins associated with the contained MainPart to the list of mixins denoted by the specified superpattern, thus guaranteeing that the superpattern attributes will always be available for that mixin. With these guarantees, there are no other allowable patterns than those patterns which are already available with BETA semantics. The horizontal and the vertical examples are extreme cases, and it is a question of good software engineering practice to design inheritance graphs such that mixin dependencies are introduced just when they are needed, preserving flexibility as much as possible. At the modeling level it corresponds to building clear and crisp concepts, since the independence corresponds to a separation of concerns. However, it is always possible to make entities independent in the superficial sense that they do not technically depend on each other—for example by tediously changing all the connections into other mechanisms such as parameter transfers—and this may be an inappropriate move. The criterion is whether or not the separate parts are still both meaningful and usable.

The general observation is that each mixin in BETA is rigidly associated with one fixed tail of mixins, whereas mixins can be combined freely in gbeta, subject to the restrictions which are necessary to ensure type safety in access to inherited attributes (to avoid `MessageNotUnderstood` errors, one might say). Furthermore, unrelated patterns, associated with flat, horizontal inheritance graphs, more freely allow combinations of mixins, whereas tall, vertical inheritance graphs allow only few different combinations. In any case, the programmer does not need to worry about this, because the constraints are managed entirely by the language, simply because there is no way to specify a pattern containing a mixin for which the depended-upon mixins are not present.

The specialization ("isa") relation may now be defined. The edges in the pattern spaces show individual elements of the specialization relation, and the relation may be obtained as the reflexive and transitive closure of these elements. However, a characterization in terms of lists of mixins is easier to make precise, and it is correct for both BETA and gbeta:

Definition 1 *Given two patterns $P = [p_1 \ldots p_n]$ and $Q = [q_1 \ldots q_m]$, we say that P is a* specialization *of Q iff P may be obtained from Q by adding zero or more mixins, i.e., if there is an injective, increasing function $\varphi : \{1 \ldots m\} \to \{1 \ldots n\}$ such that $p_{\varphi(i)} = q_i$, for all $i \in \{1 \ldots m\}$.*

In other words, a superpattern is a sublist, e.g., $[1, 2, 3] \leq Q$ when Q is any of [], [1], [2], [3], [1, 2], [1, 3], [2, 3], or [1, 2, 3]. In the case where $m = 0$ (so $Q =$ `object` and $\{1 \ldots m\} = \emptyset$), the requirement is trivially satisfied, so every pattern is a specialization of `object`, as expected.

For BETA, only one special case of this specialization relation is exploited, because mixins are inseparable from their tails (the declared superpattern), namely the case where the deletions always remove the frontmost element. For example, $[3, 2, 1]$ could be an actual specialization of $[2, 1]$ in BETA, but $[3, 1]$ or $[2]$ could never exist if $[3, 2, 1]$ exists. A consequence is that a specialization test in BETA can be made simply by inspecting the elements of a given pattern. P is a specialization of Q iff the frontmost mixin of Q occurs in P.

Traditionally in the BETA community, this relation is called the 'specialization' relation, and the reverse relation is called the 'generalization' relation. In other communities it may be called the 'isa' relation or the 'inheritance' relation. Using 'isa' would be fine, but using 'inheritance' would redistribute the emphasis from the relation between concepts which may be more or less general/special, to the detailed, implementation-oriented phenomenon of 'inheriting' attributes from a superclass. Since such reuse of attributes is definitely subordinate to the soundness of the relation between the modeled concepts, the term 'inheritance' will be reserved for purposes which actually depend on attribute specifics.

3.6 Inheritance and Available Attributes

◇ The *inheritance* mechanism is the mechanism which allows a mixin which is the head of a pattern to use attributes declared in the tail of that pattern; since the tail of a pattern is itself a pattern, this description applies recursively to all the mixins in patterns. The word 'inheritance' reflects the fact that those attributes are in a way given to the mixin at the head by the "ancestors" (superpatterns), because all sublists of the tail are superpatterns.

The syntactic representation of the inheritance mechanism is the ObjectDescriptor, which is the full version of the simplified syntactic construct Descriptor which was specified in Fig. 2.1 on page 23 and Fig. 2.3 on page 26. The main difference is that the Name in front of the Mainpart in a Descriptor is more flexible in the real ObjectDescriptor, allowing more different kinds of superpattern specifications. However, the Descriptor will suffice for the presentation of the mechanism here, so we will stick with the simplified syntax. The full syntax can be found in App. A.

Like in Sect. 2.2, the class-like aspects and the method-like aspects of the semantics of Descriptors are treated separately, with this section describing inheritance of attributes and Sect. 3.8 describing specialization of behavior.

A description of inheritance would typically talk about the attributes which a given object or class "has". To be able to do this we must make it reasonably precise what 'has' means in that connection. This will be done in three phases: First the notion of attributes in part objects and mixins is treated, then the attributes of an object or pattern as a whole can be dealt with, and finally the notion of a view on a pattern or object is introduced. Since the view determines what attributes can actually be used, all these concepts need to be brought together in order to describe what a given mixin inherits from its superpattern.

For a given part object, the available attributes accurately reflect the attributes specified in the mixin from which the part object was originally created. So we only have to consider the attributes specified in a mixin, and they accurately reflect the syntactic declarations given in the MainPart associated with the mixin, with all name applications interpreted according to the run-time environment which is provided by the origin of the mixin.

◇ This *interpretation of names* is fixed at compile-time, where the static analysis annotates every name application in the program with a specification of how to find the entity denoted by that particular name application. This specification is called a *run-time path*. For any given run-time environment, the ◇ denoted entity can be accessed by mechanically following the instructions in the run-time path. For example, an attribute used as a parameter for a (pattern used as a) method would often be located in the same part object as the one which contains the name application that refers to it. In that case, the static analysis would annotate the name application with the empty run-time path, meaning "it is right here!". If the method accesses an attribute of its enclosing object (which per definition includes a part object which is the origin), then the run-time path would start with one step outwards, and then possibly one more step in order to go to the right part object.

In summary, each part object contains attributes as specified in its mixin and using the static analysis annotations to give semantic meaning to the syntax on the right hand side of declarations. In this sense, a part object and a mixin never inherits anything! However, attributes in tail mixins are available according to the static analysis of them, and for that we must consider the set of attributes offered by complete patterns.

The fact that every name application during static analysis is annotated with a fixed run-time path implies that every name application is statically associated with one specific name declaration. This is called static name binding, and it has some important implications, which are discussed near the end of this section.

The notion of inheritance only makes sense when applied to complete objects and patterns. Since the set of attributes of an object is determined fully by its pattern we only need to consider attributes as specified in patterns. However, we need to remember the connection between objects and patterns because an object may be specialized dynamically, hence becoming an instance of a more specialized pattern. More about this in Sect. 7.3.

The attributes of a pattern are simply the attributes of all the mixins in the list which is that pattern. However, some of these attributes may have the same name, causing a *name clash*. This is a classical problem with multiple ◇ inheritance [59], but it also occurs when a mixin declaring a given name N is applied to a class which already declares N. It is our opinion that name clashes must be handled gracefully, since it is inappropriate to require that separately developed code which is brought together (by merging, e.g.) must use disjoint sets of names. Separately developed patterns should be capable of being merged and then used as each of the contributors with exactly those interactions that are appropriate for the modeling task. E.g., if a pattern is created by a combination of `Voter` and `Employee`, then it should be possible to use instances of it *as* `Voter` or *as* `Employee`, without confusing the contributions from each aspect, but also with unification of those contributions which should actually be considered the same. This ideal might not be fully achievable, but gbeta does much to approach it.

There are two possible reactions to name clashes. One is to consider different declarations of the same name as the same attribute, another is to consider them as different attributes. In gbeta (and BETA) these possibilities are both supported, and the choice is made explicitly by the programmer. However, not all kinds of attributes can be unified (considered as the same).

Attributes with the same name in different MainParts are generally considered distinct, so an object may contain, e.g., two attributes named `x`, one being an `integer` object, and the other being a variable pattern, as in this example:

```
p: (# x: @integer #);
q: (# x: ##object #);
r: p & q; (* brings together two attributes named 'x' *)
```
Ex. 3-4

Attributes of different kinds will never be considered the same, and for attributes of the same kind, only pattern attributes may be declared in multiple MainParts and be considered the same.

This is marked by making the pattern attribute virtual. As shown in Fig. 2.2 on page 24, virtual pattern attributes are associated with three different kinds ◇ of attribute declarations. One of them, the *virtual pattern declaration* (with kind `'<'` as in `X:< Point`) introduces the attribute. Every virtual further-binding and final-binding is statically associated with one particular virtual pattern declaration, so there are exactly as many pattern attributes associated with virtual declarations of all kinds in a pattern as there are virtual pattern declarations. The virtual pattern declaration "*is*" that pattern attribute, and the further- and final-bindings modify the value of it. Virtual patterns are covered in more detail in Chap. 4.

An object attribute, or a variable (pattern or object) attribute can *not* be unified across mixins, but the obvious semantics of unifying such attributes is exactly what is achieved by introducing an auxiliary virtual pattern V, and then exploit the unification of V; the object specification or variable attribute qualification should then be V. Since unification must in all cases be declared explicitly, this is only slightly less convenient than having attribute unification available for all kinds. However, unification of different kinds of attributes (e.g., a pattern and an object) is not supported, cannot easily be obtained by a re-write, and does not have an obvious, meaningful semantics. As an example of the by-virtual-pattern unification of variable object attributes, consider the following:

```
link: 1(# v:< object; value: ^v; next: ^=this(link) #);
intlink: link 2(# v::integer #);
```
Ex. 3-5

Details of virtuals are deferred to Chap. 4, but the example is hopefully understandable with the following explanation. In this example, the `link` pattern implements a basic singly linked list, which uses the virtual `v` as the qualification of the variable object attribute `value`. In `intlink`, which is $[2, 1]$ using the given numbering of MainParts, the unification of the attribute `v` in the two mixins ensures that the qualification of `value` in an `intlink` is `integer`. This is actually a useful semantics which might very well have been the meaning of unifying variable object attributes. However, since this is an easy re-write, there is no need for such unification.

```
(# cell: (# v:< object; value: ^v #);
   integerCell: @cell(# v::< integer #);
   touchyInteger: integer
     (# enter (# 'Somebody changed me!'->stdio #)#);
   myCell: ^cell
do
   touchyInteger[]->integerCell.value[];
   integerCell[]->myCell[];
   (* for myCell.value we now have different view,
    * qualification, and pattern of referred object *)
#)
```

Figure 3.5: Views, qualifications, and actual patterns

The result is that a pattern has exactly those attributes which are declared in the MainParts of the mixins in it, except for the virtual further- and final-binding declarations (which are statically associated with some virtual pattern declaration but do not "count" themselves). Since some declarations may have the same name without being unified, there is still the problem of ambiguity when looking up such a name. To handle this we need different views on objects.

Actually, for backward compatibility with BETA, an attempt to access an attribute of name N where two or more attributes of the name N are declared *will* succeed, and it will access that N attribute which is declared in the front-most (most specific) mixin. However, this is deprecated and a warning will be generated during static analysis. The right solution is to use a view which selects a suitable superpattern that does not contain the name clash. This will probably improve the readability of the code anyway.

A *view* on an object is a statically known pattern which is used to access the object.[4] Such a pattern may be derived from the specification of an object attribute, it may be the qualification of a variable object or pattern, or it may be denoted in a qualified attribute denotation (an "upward cast", with syntax QualifiedAttrDen which can be found in the grammar, App. A, and which is presented in Sect. 8.2.2). In all cases the view is the contract between the object being accessed and the "client" or "user" of the object which performs the access. The view determines what attributes may be accessed and what properties those attributes have themselves. ◇

In many cases, for instance with a virtual, the pattern on which a view is based is not known exactly at compile-time, but there is always a statically known upper bound. This upper bound is then used for the view. It is noted in the static analysis that the view is possibly incomplete, i.e., that the pattern may be strictly more specialized than what is statically known. This means that

[4]Since patterns do not exist before run-time there is no such thing as a compile-time constant pattern, but often a pattern is completely known statically *relative to the current object*

a certain list of mixins is statically guaranteed to be present, but the actual run-time qualification may contain additional mixins.

Note that this is *not* the same as subtype polymorphism. Where subtype polymorphism is concerned with the relation between, e.g., the qualification of a variable object attribute and the pattern of the object referred by it, this is about the relation between the qualification and the static knowledge about the qualification. In a language where classes are always compile-time constants, there may be subtype polymorphism, but still each qualification will be known exactly during static analysis.

Figure 3.5 illustrates the different patterns involved in this situation. It uses virtual patterns which will be explained in detail in Chap. 4, but at this point we just describe the properties of the concrete example. The `cell` pattern describes objects which may hold a `value` qualified by the virtual attribute `v` which is `object` in `cell`, and the `integerCell` object further-binds `v` to `integer` such that the `value` in `integerCell` must be qualified by `integer`. A `touchyInteger` is a specialization of `integer` which will complain whenever it is being assigned a new value. Line 7 and 8 in the figure then initialize the `value` of `integerCell` to an instance of `touchyInteger`, and makes `myCell` refer to `integerCell`. As a result, in line 9, the variable object attribute `myCell.value` has `object` as the view pattern (because the view of `myCell` is `cell`); it has `integer` as the qualification, because `myCell` refers to `integerCell` whose `v` attribute has the value `integer`; and the object referred by `myCell.value` is an instance of `touchyInteger`. This makes the view, the qualification, and the pattern of the referred object different, so these concepts are obviously distinct.

The static analysis and semantics of `gbeta` ensures that certain invariants hold. For instance with a variable object attribute, the view pattern will at all times be a superpattern of the qualification, and the qualification will at all times be a superpattern of the pattern of the referred object. These invariants are necessary and sufficient to ensure that attribute access will never fail (there are never any `MessageNotUnderstood` errors). However, note that if the view is different from the qualification, reference assignment (like ... `->myCell.value[]` in the example) is and must be detected as type unsafe, unless relative information otherwise guarantees the type safety.

Finally, as promised, we will discuss some important implications of the fact ◇ that `gbeta` has *static name binding*. It allows programmers to look up (probably by double-clicking ...) what declaration any given usage of a name refers to, and hence matches very well with locational (or stricter) equivalence: The name application may unconditionally "take over" whatever information can be obtained about the associated declaration, including comments and general knowledge about the intended usage and properties of entities near that location; the programmer may then rest assured that this knowledge can not be invalidated by accidental name clashes or similar phenomena which can only be detected using global knowledge.

Moreover, the static name binding ensures that certain dynamic operations are safe, such as dynamically adding new part objects to an existing object, thereby giving it new attributes. Luigi Liquori says in [63, p. 150] that this is well-known to be unsound:

> As clearly stated in [13, 4], subtyping is unsound when we allow objects to be extended.

The demonstration of this problem by means of an example right after the above quote relies on changing the meaning of a name application by introducing a new declaration of the same name but with another type. This is an example of unifying two attribute declarations in such a way that the type checker will never know about both of them at the same time. Now, with static name binding and explicit unification that problem is trivially solved, since an attribute declaration can only override another when the two are recognized by the static analysis as being one, unified attribute, and that means that the necessary type conformance checks between the two declarations can be made during static analysis. However, even though such a result is nice, it should be viewed as one of many examples of improved analyzability and understandability associated with a language where programmers can reason about programs in terms of properties of declarations, without having to worry about *what* declarations to look at. In other words, it is a benefit of static name binding.

3.7 Pattern Merging

> *This chapter shares material with our paper* Propagating Class and Method Combination*, which was accepted for publication and presentation at the ECOOP'99 conference.*

The *merging operation*, '`&`', is based on a linearization, as mentioned already in Sect. 3.2. This section specifies precisely how the merging works, giving a formal definition of the linearization based on operations on total order relations which are again just a formal view on lists. ◇

3.7.1 Linearization

Until now the class combination mechanism '`&`' used in `gbeta` has only been presented by example. This section motivates the mechanism, specifies precisely what the mechanism is, and proves some properties about it. The concrete algorithm is shown in Fig. 3.6. This algorithm uses the standard `member` function which determines whether or not a value (first argument) is a member of the given list (second argument). As is evident, merging may fail. This corresponds to the situation where the compiler rejects a `gbeta` program because it contains a "bad merge". It occurs when the contributing patterns give conflicting directions as to the order of two or more elements, e.g., where one contributor requires the order $[\dots a \dots b \dots]$ and another requires $[\dots b \dots a \dots]$ for two mixins a and b. Where behavior is combined, it certainly makes sense for the programmer to decide exactly in what order contributing behaviors are composed, but for the case where the combination deals exclusively with state it would be nice to have

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

Figure 3.6: The algorithm used in merging

a symmetric, unordered mechanism. That is future work. The details about behavior combination is given in Sect. 3.8.

A small oddity: In object-oriented languages it is a tradition to write the most specific part *last*, like in `aSuper(# .. #)`, not `(# .. #)aSuper`, and hence the class combination operator '`&`' was designed to make the last argument the most specific. In this section it is more convenient to reverse the order, so the programming language syntax `A&B` corresponds to the mathematical notation $B \lhd A$ below. Among other things, it matches better with the standard list notation which also puts the most accessible ("head") element at the front (left) end of the notation.

The class combination mechanism is a graph linearization, i.e., a procedure which from an oriented graph constructs a list which determines a topological sorting of the nodes. Obviously a cyclic graph does not allow this, hence the potential for rejection of a merge. Since there are many possibilities for typical graphs, some systematic choices must be made in order to arrive at a well-defined result. Existing linearizations [29, 30, 7] are described in terms of such systematic choices of "what node to take next", and this makes it hard to understand their outcome and to reason about their properties.

Luckily, the 'C3' linearization [7], which is the one used in gbeta, can be characterized in a much more declarative way, and it can even be generalized in a way that makes it a proper operation on a suitable set M:

$$\forall x, y \in M.\ x \lhd y \in M$$

The name C3 reflects three consistencies exhibited by this linearization, namely consistency with the local precedence order,[5] consistency with the extended precedence graph,[6] and monotonicity.[7] The other known linearizations (includ-

[5]The programmer-chosen ordering of direct superclasses, or, in gbeta, the ordering of the operands of the merging operator

[6]The extended precedence graph additionally orders classes according to the local precedence order from the most general common subclass

[7]Avoidance of the phenomenon that an inherited feature is looked up in a class that none of the direct superclasses would have chosen.

The C3 merging principle:
The merge of two orders A and B is
the union of A and B together with
all non-contradictory edges from A to B

Figure 3.7: The intuitive principle behind merging

ing the ones used in LOOPS, CLOS, and Dylan) do not have all three consistencies. Moreover, the C3 looks even better with the simple characterization given below. The remaining problem with linearization is that *no* linearization can handle all inheritance hierarchies, some will be rejected as inconsistent. There is simply no way the lists $[1, 2]$ and $[2, 1]$ could be merged into a new list of distinct values which would preserve the order of 1 and 2 for both of those lists.

To reach a declarative characterization we must make a shift in mindset and terminology from the 'list' and 'graph' based thinking. If we regard the edges in a given acyclic oriented graph as a relation and take the reflexive and transitive closure of that, we get a partial order. Similarly, a list determines a total order. Hence, a linearization is a construction of a total order by adding elements to a partial order. C3 actually constructs a total order from a number of given total orders, namely the linearizations of the superclass hierarchies. The C3 principle for two orders is as shown in Fig. 3.7. To elaborate on this principle, it says that "the merge is everything A says and everything B says and then, by default, elements from A are smaller than elements from B." This is formalized straightforwardly in section 3.7.1. First we have to establish a few facts.

Total preorders

We need to consider total preorders:

Definition 2 *A* total preorder *is a relation which is reflexive, transitive, and total. A* total order *is a total preorder which is also anti-symmetric.*

It is easy to prove that:

Lemma 1 *Assume $\preceq$ is a total preorder. The relation $\sim$ defined by $a \sim b \Leftrightarrow a \preceq b \wedge b \preceq a$ is an equivalence relation, and the relation $\leq$ on equivalence classes defined by $[a] \leq [b]$ iff $a \preceq b$ is well-defined and a total order.*[8]

Given an equivalence relation $\sim$ and a total order on the equivalence classes $\leq$, then the relation $\preceq$ defined by $a \preceq b \Leftrightarrow [a] \leq [b]$ is a total preorder.

In other words, a total preorder corresponds to a list of equivalence classes of elements, rather than a list of individual elements.

This is the desired generalization: to construct a *list of groups* of mixins, each group consisting of mixins considered equally specific.

[8] $[a]$ denotes the equivalence class containing a.

In such a setting, clashing names are not always disambiguated. This might at first seem to be a step backwards; it is in fact an improvement. When the ordinary C3 linearization (Fig. 3.6) would succeed, the generalization delivers the same result (all groups have size one). When the hierarchy would be rejected, the resulting non-trivial groups would in many cases work well enough. For example, as long as a name is only declared by one of the mixins in a given group, there will be no clashes on that name. In fact, a number of inheritance hierarchies would be better described by making certain mixins equally specific, since the commitment to one order causes unnecessary restrictions on future usage. However, the lack of ordering does not blend well with combination of behavior.

Merging

We need a couple of tools before C3 merging can be formalized:

Definition 3 *When R is a relation, its* domain $\mathrm{dom}(R)$*, its* inversion $\overline{R}$*, its* one-step transitive closure R^+*, and its* transitive closure R^**, are defined by:*

$$\begin{aligned}
\mathrm{dom}(R) &\triangleq \{y | (\exists z.\ (y,z) \in R)\ \vee\ (\exists x.\ (x,y) \in R)\} \\
\overline{R} &\triangleq \{(y,x) | (x,y) \in R\} \\
R^+ &\triangleq R \cup \{(x,z) | \exists y.(x,y),(y,z) \in R\ \} \\
R^* &\triangleq \bigcup_{i \in \omega} R_i
\end{aligned}$$

where $R_0 \triangleq R$, $\forall i \in \omega.\ R_{i+1} \triangleq R_i^+$.

The following lemma is immediate from the definitions:

Lemma 2 *Let R and S be relations. Then R^* is transitive. The domain is additive:* $\mathrm{dom}(R \cup S) = \mathrm{dom}(R) \cup \mathrm{dom}(S)$*. The domain is preserved by transitive closure and inversion:* $\mathrm{dom}(R^*) = \mathrm{dom}(\overline{R}) = \mathrm{dom}(R)$*. Reflexivity is preserved by transitive closure, inversion, and union: if* $\forall x \in \mathrm{dom}(R).\ x \preceq_R x$ *then* $\forall x \in \mathrm{dom}(S).\ x \preceq_S x$*, $S \in \{R^*, \overline{R}\}$, and if* $\forall x \in \mathrm{dom}(T).\ x \preceq_T x$*, $T \in \{R, S\}$ then* $\forall x \in \mathrm{dom}(R \cup S).\ x \preceq_{R \cup S} x$*.*

The formalization of the C3 merging principle is:

Definition 4 (C3 Merging) *Let R_1 and R_2 be relations. The* C3 merge *of R_1 and R_2 is*

$$R_1 \lhd R_2 \ \triangleq\ R \cup (\mathrm{dom}(R_1) \times \mathrm{dom}(R_2) \setminus \overline{R})$$

where $R \triangleq (R_1 \cup R_2)^*$.

Intuitively, the merge combines the two given relations R_1 and R_2 into $(R_1 \cup R_2)$ which is then "repaired" to be a transitive relation R by taking the transitive closure. R is complemented with everything from $\mathrm{dom}(R_1)\times\mathrm{dom}(R_2)$ which does not contradict R. In other words, R_1 elements are smaller than R_2 elements, unless something is known to the contrary.

As an example of a general merging, $\{a \le b, a \le c, b \le c\} \lhd \{c \le b\}$ is $\{a \le b, a \le c, b \le c, c \le b\}$, or as lists: $[a, b, c] \lhd [c, b] = [a, \{b, c\}]$. The elements $\{b, c\}$ for which there are conflicting ordering requirements—a cycle, as defined below—are collected into a group and thus made equally specific.

Now we can state the closure property that makes total preorders interesting:

Proposition 1 *Assume R_1 and R_2 are total preorders. Then $R_1 \lhd R_2$ is a total preorder.*

Proof: (See appendix B) □

The ordinary C3 merge (Fig. 3.6) fails precisely when the generalized C3 merge (Def. 4) produces a total preorder which is not a total order. A total preorder is a total order if and only if there are no cycles, so we need to consider them:

Definition 5 *Let R be a relation. A sequence of distinct elements $d_1 \ldots d_n \in \mathrm{dom}(R)$, $n \ge 2$, is a* cycle *in R iff*

$$(\forall i \in 1 \ldots n{-}1.\ (d_i, d_{i+1}) \in R) \wedge (d_n, d_1) \in R$$

R is acyclic *iff there are no cycles in R.*

Lemma 3 *Let R be a acyclic relation. Then R^* is acyclic.*

Proof: (See appendix B) □

We can now state and prove the “non-pre” equivalent of proposition 1:

Proposition 2 *Assume R_1 and R_2 are total orders and $R_1 \cup R_2$ does not have cycles. Then $R_1 \lhd R_2$ is a total order.*

Proof: (See appendix B) □

We have seen that the C3 merging principle can be formalized in a rather obvious manner and proved that the formalization has the nice stability property of proposition 1 and the incomplete stability property of proposition 2. It seems to be worthwhile to try to develop the strict linearization of various languages into the more wholesome total preorder model, going from class precedence lists to class *group* precedence lists. This has not yet happened for gbeta, and as mentioned the main problem is the fact that behaviour can hardly be combined in a symmetric way—a non-deterministic choice of ordering of the behavior composition or generally having concurrent execution of equally specific behaviors would surely be a nightmare to debug.

It is not hard to see that the algorithmic version of C3 actually implements the formalization presented here—the algorithm each time selects the least remaining element according to our formalization of C3. Of course, the algorithm just terminates with an error message if the result would not be a total order.

As an aside it is interesting to note that the gbeta implementation actually used a quite different algorithm for merging during a period of more than a year. Only after the above formalization was created did it become clear that the C3 algoritm (which was simpler and therefore attractive) solved the exact same problem, because both algorithms clearly implement the formal characterization of the linearization. Algorithms *are* generally harder to compare and reason about than declarative formalizations like Def. 4.

3.8 Specialization of Behavior

In most object-oriented languages, behavior cannot be specialized. Let us first consider a non-example. Typically, the behavior of a method m (in some languages it must be 'virtual') in a given class C can be made different from what ◇ it is in the superclass(es) of C by *overriding* the implementation of m with a different one. The new implementation is generally written from scratch. Only ordinary method invocations may be used to produce a new behavior built on the existing behaviors, for instance by invoking the implementation of m in a specific superclass (as with `superOfC::m()` in C++) or by sending a message to a special object designator like '`super`' (as in Smalltalk) which will invoke the implementation available in the immediate superclass.

It may seem natural to include method overriding into the concept of behavior specialization, but it is *not* the same. Method overriding will make an invocation of m exhibit different behavior when invoked on instances of different classes, but the implementations of m will not be a family of related methods which are defined incrementally; the different implementations of m are entirely independent methods which are only brought together by the common name. In contrast, a family of methods related by behavioral specialization is more like a class hierarchy, where commonalities in behavior are factored out into the general members of the family, and more special members are created incrementally from more general ones. As a special case, such a family of related methods may be used as the implementations of a virtual method, but it may also be used in other ways.

Specialization of behavior is available in BETA, as presented already in in [60], and it is available in a generalized form in gbeta. In both BETA and gbeta, specialization of behavior is based on the INNER imperative. INNER works the same in BETA and gbeta, at least when describing patterns as lists of mixins, as it was done in the previous sections. The only difference is that gbeta allows for much more flexible ways to create patterns from given mixins. As shown in Sect. 3.8.2, this makes a real difference in expressive power.

Behavior specialization is the creation of a more refined, detailed, complex behavior from a less refined one by means of an incremental specification. For example, yelling "di-bah-ba-doo-dah-dosh" is a specialized version of the behavior of yelling "ba-dooh-dah", but also a specialized version of yelling "di-bah-[]-dosh", where "[]" is a special placeholder which by the rules of the game may be replaced by anything, and which defaults to nothing if it does not get replaced.

Actually, both of the less special behaviors may be viewed as the general basis from which the special behavior is derived, using the *other*, less special behavior as an incremental specification. However, the behaviors cannot just be "added" symmetrically, there must be a specification of *how* to combine them.

Kristine Stougaard Thomsen created a proposal for how behavior could actually be combined in a mostly symmetrical manner by means of non-sequential execution (similar to coroutines) already in 1987 [108]. The central concept in this proposal is that the INNER imperative is replaced with another construct which would transfer the control to another do-part, thereby allowing several equally specific mixins to coexist without having to decide on an ordering. This idea might well be applied in context of the generalized merging mechanism. However, we believe that the programmer in the general case would have too much trouble determining exactly what would happen at run-time with such a cross-over mechanism instead of INNER, so it is our impression that it should preferably be used only in those cases where the do-parts in question could really be executed in an arbitrarily interleaved order.

Together, the notion of patterns as lists of mixins (not sets of mixins), inheritance, the pattern merging operation, and the INNER imperative provide a rich framework for creating and using combinations of behavior specifications in gbeta. The specialization of behavior which this gives rise to will now be described in two phases, first from above considering patterns and behaviors as a whole, then from below considering the individual mixins and their DoParts.

3.8.1 Specialization of Behavior—a Top-Down View

The behavior associated with a pattern will be present as an aspect of the behavior of any of its subpatterns; it can never be discarded, only refined. This ensures that a subpattern will always have a behavior which is in some sense a completion and enhancement of the behavior of each of its superpatterns.

The behavioral specialization relation is based on *syntactical* criteria, so there is no consistent, bullet-proof semantic meaning behind the claim that 'behavior cannot be discarded'; in contrast, it is actually possible to add a mixin to a pattern P such that the resulting pattern will do what the new mixin DoPart specifies, ignoring the behavior of P entirely. An example of this kind of distorted behavioral specialization is the following:

```
(* original behavior *)
do 'Hello, world!'->stdio;

(* "refined" behavior *)
do (if false then 'Hello, world!'->stdio if);
   'Good day, earth!'->stdio;
```

Ex. 3-6

The refined version of the code is in fact built from the existing behavior and a new contribution, but the semantics of the execution ensures that only the new behavior will be observed. As a general rule, though, a subpattern will have a behavior which can reasonably be described as a specialization of the

```
p: (# do (for i:3 repeat INNER for)#);

q: p(# (* fill in the INNER of p *)
    do 'Hello, world!'->stdio
    #);

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

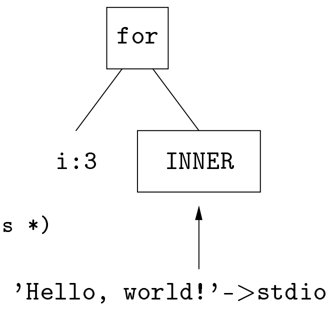

Figure 3.8: Specialization of behavior: `q` and `r` are equivalent

behavior of any of its superpatterns. In particular, this is a meaningful goal which programmers may strive for and generally achieve.

Like gbeta, BETA also supports a syntactic—not a semantic—behavioral specialization relation; but it would not be easy to improve that into a genuine semantic refinement relation. It would be hard to formalize 'behavioral specialization', and it would surely be an undecidable problem to verify it statically for a non-trivial language. Hence, a syntactic approximation is probably the best we can hope for. Of course, run-time checks of assertions like the pre- and post-conditions of Eiffel can be added, and there has been a framework for doing this in BETA, to a certain extent, for several years. We should note that pre- and post-conditions and invariants in Eiffel express these correctness criteria as part of the *interface* and thereby elevate them to a more visible position than they could have as part of method implementations. This is likely to strengthen the consciousness of programmers about these criteria and thereby support the creation of high quality systems.

However, support for provable behavioral conformance is not the basic idea behind the BETA and gbeta behavioral specialization mechanisms, the basic idea is rather to enable incremental specification of behavior. That is an obvious goal given the relation between concepts and patterns, and given the existence of specialization relations between concepts. Admittedly, the mechanical specialization relation supported by BETA and gbeta are very crude imitations of the specialization relations in natural language. But they are indeed useful!

3.8.2 Specialization of Behavior—a Bottom-Up View

The behavior specialization process can be described as a syntax tree completion process, i.e., as a source code transformation which inserts entire DoParts from more specific mixins into special positions (`INNER` imperatives) in less specific ones. This transformation process correctly shows which imperatives will be executed in what order, and it promotes the right idea about specialization as a process of filling-in missing pieces; however, the transformations generally change the name binding environments of names, so they could not be performed on real programs without changing their semantics or—more likely—introducing static semantic errors.

Nevertheless, these transformations will be used to illustrate the behavior combination mechanism, starting with the example in Fig. 3.8 which happens to work correctly also after the transformation. In the figure, the pattern p is the general basis whose behavior is to repeat INNER three times, and the definition of q adds an increment to the behavior of p, thereby determining what the meaning of INNER in p is. As a result, the behavior of q is equivalent to the behavior of r.

From the concrete example we turn to an informal but general, recursive definition: The behavior of object, the empty list of mixins, is to do nothing. The behavior of a pattern containing just one mixin is the DoPart of that mixin. The behavior of a pattern with at least two mixins is derived from the behavior of its tail (which is the same as the immediate superpattern in BETA) by inserting the behavior of the head into each occurrence of an INNER imperative in the behavior of the tail.

The behavior of a DoPart is the effect of executing the imperatives in that DoPart in context of a part object which is an instance of a mixin associated with the enclosing MainPart. The insertion of a behavior into an occurrence of INNER is a transfer of control similar to a sub-routine call, i.e., a jump to the DoPart in question, followed by the execution of the imperatives there, in the environment of the corresponding part object, and returning from that DoPart upon termination.

An alternative explanation of the semantics of the INNER is that it is similar to a message send to super in Smalltalk, but it calls the "subclass" (the next more *specific* mixin, i.e., the preceding element in the pattern) instead of the "superclass" (the next more *general* mixin, i.e., the successor element). Thinking of INNER as an 'inverted super send' might seem to be the most direct approach at first, but the view of the syntax tree being completed gradually by filling in the INNER placeholders with the DoPart from the next more special mixin is closer to the intention behind the mechanism. Again, BETA and gbeta have exactly the same semantics of INNER for any given pattern, but gbeta will allow a more flexible construction of patterns from existing mixins.

Figure 3.9 on page 72 is an example where combinations of a few mixins into several different patterns show different behavior as determined by the semantics of INNER (note that the boolean doit is false at the beginning of each imperative). For example, the merging c&b&a gives rise to the equivalent of the following DoPart:

```
do (for 4 repeat
        'c'->stdio;
        (not doit)->doit;
        (if doit then  'b'->stdio
         else true->doit; 'a'->stdio; false->doit
        if)
    for)
```

Ex. 3-7

```
(# doit: @boolean;
   a: (# do true->doit; 'a'->stdio; INNER; false->doit #);
   b: (# do (if doit then 'b'->stdio else INNER if)#);
   c: (# do (for 4 repeat
                 'c'->stdio; (not doit)->doit; INNER
             for)
      #)
do a;           (* prints 'a'            *)
   b;           (* prints ''             *)
   c;           (* prints 'cccc'         *)
   a&b;         (* prints 'ab'           *)
   b&a;         (* prints 'a'            *)
   a&c;         (* prints 'acccc'        *)
   c&a;         (* prints 'cacacaca'     *)
   b&c;         (* prints 'cccc'         *)
   c&b;         (* prints 'cbccbc'       *)
   a&b&c;       (* prints 'ab'           *)
   a&c&b;       (* prints 'accbccb'      *)
   b&a&c;       (* prints 'acccc'        *)
   b&c&a;       (* prints 'cacacaca'     *)
   c&a&b;       (* prints 'cabcabcabcab' *)
   c&b&a;       (* prints 'cbcacbca'     *)
#)
```

Figure 3.9: Many examples of specialization of behavior

Of course, a systematic enumeration of the possible combinations like this example does not make sense in a real usage context, but some of the combinations are examples of more generally applicable principles. For example, try to compare `c&a` with `a`, `c&a&b` with `a&b`, and `c&b` with `b`; this shows the usage of `c` as a behavioral aspect which will repeat whatever it is combined with, and modify the environment in which the repeated behavior is executed (by changing `doit`).

Similarly, `b` is a "conditionalizing" behavioral aspect which chooses whether or not to execute whatever behavior it is combined with. This is demonstrated by `c&b&a`, where `b` uses `a` differently for each iteration of the `for`-imperative, depending on the environment—which is controlled by `c`. Mixins similar to `b` can be used to switch on and off the behavior of given patterns, according to whatever criteria are appropriate. Clearly, a given basic behavior enhanced with the ability to be switched on and off may be described as a refined, specialized version of the basic behavior itself—although the conditionalizing mixin arguably plays the role of a post-hoc added *superpattern* when considered from a BETA point of view.

These kinds of behavioral specialization which include the placement of a given, comparatively simple "super" behavior in the context of some other "incremental" behavior are not supported in BETA, only in gbeta. The reason is that a pattern on the form $[m_k, m_{k-1} \dots m_1]$ in BETA only has the patterns $[m_j, m_{j-1} \dots m_1]$ for $j \in \{0 \dots k\}$ as superpatterns, no other sublists of $[m_k, m_{k-1} \dots m_1]$ can even exist, and hence a given behavior can never be "inserted into" another behavior by specialization in BETA, it can only "insert" the incremental behavior as leaves in the syntax tree completion.

Finally, by comparing a with b&a, c&a, b&c&a, and c&b&a we can observe the traditional specialization of the behaviors of b, c, b&c, and c&b with a as a new leaf behavior. Of course, even though this kind of behavior specialization is possible in BETA, it would require the textual copying of the definition of a into every position where it should be used to specialize a behavior.

3.9 Object Creation

Objects do not just magically come into existence in gbeta. Actually, the process of creating an object is complex, even though there is no notion of 'constructors' in gbeta. The following description details what should happen at the conceptual level in any gbeta implementation. It also describes the current (slow, but general) method used in the existing implementation of gbeta. However, a high quality implementation of gbeta would exploit the information from static analysis to pre-compute or avoid the need for many entities which are actually created at run-time in the current gbeta implementation.

As described in Sect. 3.3, an object is a list of part objects, each part object is associated with the mixin from which it was originally created, and each mixin is associated with a MainPart which contains a list of attribute declarations. The part object must have the attributes defined in the MainPart, and the run-time paths (presented in Sect. 3.6), which have been initialized during static analysis, are used to find the entities (objects or patterns) denoted by the right hand sides of declarations, and that often makes it possible to initialize the attributes of a given part object.

However, it is not that easy in the presence of recursion. The problem is that an object may contain an attribute which denotes an entity which must be looked up in another attribute of the same object or even in a directly or indirectly nested object. In other words, the construction of the object cannot be completed without using (parts of) the object itself. The approach used to break this cycle in the current gbeta implementation is to compute attributes on demand, annotating each attribute which is being computed as "half-done". Whenever the object creation algorithm looks up an attribute and finds that it is "half-done", a circularity must have been encountered and a run-time error is raised which kills the thread in which this failing object creation was going on. A traversal of all attributes ensures that the demand occurs for every attribute during the object creation, whether or not it is used in the initialization of other attributes.

In many cases it will be possible to determine statically whether or not such an on-demand computation will give rise to circularities, but with the generality of BETA (hence also gbeta) it is possible to create circularities which cannot be detected without flow analysis, i.e., it is in general an undecidable problem to detect such cyclic dependencies. It would still be nice to detect a large number of easy, safe cases and then give warnings for the few, hard cases. An example in BETA containing a cycle which requires flow analysis to detect is the following:

```
p: (# r1: [(# exit r2.range #)] @integer;
      r2: [(# exit r1.range #)] @integer
   #)
```
Ex. 3-8

This example contains some constructs which have not yet been presented, but they will be explained for this special case here. The pattern p contains two repetition attributes, i.e. two arrays of objects. The lengths of the repetitions are determined by the evaluations `(# exit r2.range #)` and `(# exit r1.range #)`, and these expressions must be evaluated before each of the repetition attributes can be created. Since the length of `r1` can only be computed when `r2` has been initialized and vice-versa, there is no way to create an instance of `p`. It would be possible to "solve the equation" by choosing an arbitrary non-negative integer and initialize both `r1` and `r2` with that length, but an implementation of gbeta is not required to discover such possibilities. The cycle should be detected and rejected, preferably at compile time, even though that is not possible in the general case. In the current implementation of gbeta it is never detected at compile time, so in particular this example gives rise to a run-time error.

Cyclic dependencies are not the only things to consider when creating new objects. In gbeta, the approach was taken to allow many things which are prohibited in BETA, in order to try out whether or not the increased generality and flexibility would be worthwhile.

As an example, all kinds of entities may be used on the right hand side of declarations, so for instance a variable object have a variable pattern as its qualification ("declared type"). Since a variable pattern may be NONE, there may arise a run-time error already during initialization of the attribute. However, even if the variable pattern is not NONE, some decisions must be made with respect to the semantics of this variable object later. The problem is that the variable pattern may change during the life time of the variable object. If we have the following situation, what would be the correct reaction?

```
1  (# vp: ##object
2  do string##->vp##;
3     (# vo: ^vp
4     do &string[]->vo[];
5        integer##->vp##;
6        ..
7     #)
8  #)
```
Ex. 3-9

In this example, `vp` is a variable pattern with qualification `object` which happens to have the value `string` at the point where the inserted item is created, i.e., when line 3 is reached. In line 4, the assignment makes the variable object `vo` refer to a `string` object. Since the value of the qualification `vp` is `string`

The snapshot principle:
The right hand side of an attribute declaration
is evaluated when the enclosing object is created,
and it is never changed after that

Figure 3.10: The principle behind variable entities used in declarations

that would be fine. In line 5, however, `pv` is changed by the assignment to have the value `integer`, and at that point the variable object `vo` will no longer refer to an object which is qualified by the value of `pv`.

It is obviously not manageable to have a qualification of a variable object which may change like that during the life time of that variable object. In general, it is not desirable that attributes may silently become invalid, so qualifications should be immutable. One rule which will make them immutable is the conservative rule that qualifications must be compile-time constants (relative to the current object), but this was exactly the limited universe from which we wanted to escape. Another rule which is very simple and which handles all the new, dynamic cases consistently is the snapshot principle which is shown in Fig. 3.10. This principle is used in `gbeta` for dynamic entities used on the right hand side of declarations. Note that it applies equally well to the well-known static cases, even though it makes no difference there.

Also note that the semantics where only compile-time constants are allowed on the right hand side of declarations is a special case of the snapshot semantics, which makes it possible to prohibit any or all of the dynamic enhancements provided by the snapshot principle as desired. One aspect of traditional BETA semantics which points in the direction of the snapshot principle is the semantics of length specifications in declarations of repetitions ("arrays"). Such an expression, like x in '`r: [`x`] ^boolean`', is evaluated when the enclosing object (of which '`r`' is an attribute) is created, and the length of the repetition is of course not adjusted every time the current value of the expression x changes. That is an example of a snapshot of the state of the program being used to determine the meaning of a declaration.

Another example that we have already seen earlier is the support for constant references by coercion, as described in Sect. 2.3.4. An example which will be covered later, in Sect. 7.2, is the usage of a pattern variable as a superpattern, to create a dynamic control structure.

Reconsidering the example above with the declaration `vo: ^vp`, it may actually be useful: The declaration gives `vo` a qualification which is only known by upper bound—since `vp` is less-equal than its own qualification, the qualification of `vo` is also less-equal than that bound. This means that an object can safely be obtained from `vo` with the qualification of `pv` as the statically known pattern, but any attempt to reference-assign to `vo`, e.g. `m->vo[]`, will cause a message from static analysis that this is not type safe. On the other hand, it

Local lookup of a name N in a view P:
If P is `Object` then the lookup fails; otherwise
if the head of P contains a declaration of N,
then that declaration is the result; otherwise
the result is the lookup of N in the tail of P

Figure 3.11: The rule for local lookup

is possible (i.e. statically type safe) to renew `vo`, i.e., to create a new instance of the qualification of `vo` and make `vo` refer to this new object. The syntax for this is `&vo` (or `&vo[]` if we want to prevent execution of the new object). As a result we have obtained a variable object attribute which is read-renew, as opposed to read-only or read-write. Client code may use `vo` as an instance of the qualification of `vp` (we might call this the "official" qualification of `vo`), and every instance of the enclosing object can have its own "secret" qualification for `vo`, determined by the value of `vp` at snapshot time. By the way, this is an example of a context dependency; see in Chap. 5 why that is actually a natural and justifiable thing to use, when used right.

3.10 Local Lookup

As mentioned in Sect. 3.6, name lookup is performed entirely during static analysis, and at run-time the entity denoted by a name is obtained by following the instructions in the run-time paths which were constructed by static analysis and attached to each name application. Hence, no searching procedures are performed at run-time in order to find any declarations with a given name, that has already happened during static analysis. This section is about the searching procedure which is used when the run-time paths are constructed. It is only concerned with the case where the name being looked up is actually available in the current object; when it is not available in the current object, i.e., when the lookup with the rule given in this section fails, the search may continue with the enclosing objects, as described in more detail in Chap. 5. If the global search also fails then the program is rejected with the static analysis error 'name is undefined'.

The local lookup rule is based on the view pattern, which is not surprising since the view is the information which is available during static analysis when ◇ the local lookup is performed. So, given a view of an object, the *lookup rule* simply says that the mixins are searched one by one, starting with the most specific; the result of the lookup is the first declaration encountered which declares the requested name. A recursive formulation of this rule is given in Fig. 3.11.

A few examples of local lookup are given in Fig. 3.12 on page 77. In the figure, three applications of names in the context of the object referred by `myQ`

```
1  (# p: [1](# x,y: @integer #);
2     q: p [2](# x: ##object #);
3     r: q [3](# y: @string; z: ^object #);
4     myQ: ^q
5  do
6     r[]->myQ[];
7     myQ.x; (* refers to the variable pattern *)
8     myQ.y; (* refers to the integer object *)
9     myQ.z; (* compile-time error! no 'z' in view *)
10 #)
```

Figure 3.12: Local lookup examples

are performed in the lines 7–9. The name `myQ` is also looked up, but this is a trivial case, so we will concentrate on the usages of `x`, `y`, and `z`.

In line 6 `myQ` is made to refer to a new object created as an instance of the pattern `r`, which is $[3, 2, 1]$ using the usual notation for patterns and the given numbering of MainParts in the example. This means that the object referred by `myQ` in the lines 7–9 actually contains two attributes named `x`, two attributes named `y`, and one attribute named `z`. However, the view on `myQ` is in all cases the pattern `q`, i.e., $[2, 1]$, which contains two attributes named `x` and one attribute named `y`. Since the MainPart 2 is more specific than 1 in `q`, the result of the lookup of `x` in line 7 is the variable pattern attribute in 2, not the `integer` object in 1. In line 8, there is only one `y` attribute available in the view of `myQ`, namely the `integer` object declared in 1, so that is the result. Finally, the access to `z` in line 9 is rejected at compile-time because the view of `myQ` does not contain any declarations named `z`.

Note that the local lookup is the only applicable rule for lookup in the context of a given object (i.e., with syntax including a dot like `myQ.x`, and generally with the Remote syntax as specified in the grammar, App. A). Only lookup which does not occur explicitly in the context of an object, i.e., stand-alone name applications like `myQ`, are eligible for global lookup.

3.11 Qualifications

A *qualification* is a pattern associated with a variable attribute which regulates ◇ what values the variable attribute is allowed to take on at run-time. It is the pattern obtained by coercion from the run-time entity denoted by the right hand side of the declaration of a variable attribute. Other terms for similar concepts in other languages would be 'declared type', 'type constraint' or simply 'type'. Note that 'types' mostly are a compile-time phenomenon whereas qualifications in gbeta (and BETA) are genuine run-time concepts, even though the static analysis is all about discovering what the run-time values will be, wherever possible. The intuition is that a qualification describes a certain set of properties, and the value of the variable attribute is guaranteed to be such that this set of properties is actually available.

The exception which has a parallel in most languages is that a variable attribute generally may be “disabled” in some sense, by being NONE in gbeta or BETA, by being in the `void` state in Eiffel, by being a `NULL` pointer in C++, or by being `nil` in CLOS and related languages. In this state, the variable attribute does not refer to an entity, and access to any property of the (missing) entity is a run-time error.

In Cecil this problem has been removed by not allowing variable attributes to attain any disabled state [21]; instead, variable attributes will initially refer to a special ‘default’ object. However, the practical consequence seems to be that the default object cannot do anything but respond to any attempted accesses by raising a run-time error, hence recreating the situation with disabled attributes.

Otherwise, for a non-disabled variable attribute A, the qualification Q of A constrains the entities which are allowed to be referred by A. If A is a variable pattern then the value of A must be a pattern which is less-equal than Q. If A is a variable object then the value of A must be the identity of an object which is an instance of a pattern which is less-equal than Q (exactly Q for the exact variants).

With the snapshot semantics of declarations, presented in Sect. 3.9, Q will immutably be the same pattern during the entire life time of A, and that ensures that the qualification constraint just needs to be assured for each assignment to the variable attribute itself (i.e., for assignments like `... ->`A`[]` when A is a variable object, and `... ->`A`##` when A is a variable pattern).

A simple way to ensure this is to dynamically check that the constraint is satisfied at every assignment to the variable attribute; but that would not be appropriate, because it would make every such assignment a potential run-time error. With the level of static analysis in gbeta and BETA, we can do better.

For the cases where the qualification is known exactly at compile-time, these assignments may be checked statically: We must ensure that the view on the entity being assigned is less-equal than the qualification of the variable attribute which is the target of the assignment. The entity being assigned unto the variable attribute may be more special than the qualification, but that does not violate the constraint for ordinary variable attributes. The exact ones are covered below.

When the view on the entity being assigned is *not* less-equal than the qualification of the variable attribute, then the assignment might actually be acceptable, but the static knowledge does not guarantee this. In BETA, the tradition is to accept this case with a compile-time warning, leaving it to the programmer to consider the case and take the responsibility, perhaps by making a proof of type safety based on ad-hoc techniques, perhaps just by guessing or hoping. In gbeta, the intention has been to get rid of such warnings and simply prohibit the unsafe cases. Sofar, for backward compatibility, the situation is still handled with a warning and a run-time check, but gbeta contains a construct which can be used to remove the need for all such dangerous assignments. This construct, the `when` imperative, is presented in Sect. 9.1.

For the cases where the qualification of a variable attribute is *not* known exactly at compile-time, assignments to the attribute are generally not type safe, but in some cases the static analysis may establish knowledge about the *relation* between some patterns, without knowing exactly what patterns they are. A typical example is the case where two variable object attributes `x` and `y` have the same virtual, `v`, as qualification. An assignment like `x[]->y[]` is then type safe because the object referred by `x` must be an instance of a pattern less-equal than `v` no matter what pattern `v` is, and that is sufficient to ensure that `y` is allowed to refer to the same object. Other examples arise with the `when` imperative mentioned above.

For variable object attributes whose Kind include '=', see Fig. 2.2 on page 24, the qualifications are exact. This means that the object referred by such a variable attribute A must be an instance of exactly the qualification Q, and this can be assured with similar techniques as the normal case covered above. When the exact pattern is known for an object whose identity is being assigned to a given exact variable object attribute, it can be checked whether those two patterns are equal; a similar case occurs when the patterns are not known exactly, but are nevertheless known to be equal; this is the case when assignment happens between two exact variable objects whose qualification is the same virtual pattern. The case where an object referred by a normal, inexact, variable object attribute is assigned onto an exact one corresponds to other unsafe cases: A warning is given, and the unsafe assignment can be avoided entirely by using the `when` imperative.

The last topic of this section is the need for sets of patterns as qualifications, similar to the concept of union types in some languages (e.g., `type-union` in Dylan [97]). Such sets of patterns are convenient, because the pattern space is so fine-grained, and sometimes some patterns are not "significantly" different. However, they are not (yet) available in gbeta.

Since patterns depend on their run-time environment, because each mixin is associated with its enclosing part object, there is a potential for an unbounded number of distinct patterns in any non-trivial BETA or gbeta program. Generally, other OO languages have a number of classes which is fixed at compile-time. (In CLOS it is possible to invoke the compiler at run-time and add new classes, but there is no type system which keeps track of all the types/classes potentially associated with the value of each variable. In Java new classes may be loaded, but with a limit on the size of the source code there is also a limit on the number of distinct classes.) Hence, the need for a concept similar to union types is especially relevant in BETA and gbeta.

In (Mjølner) BETA, a special kind of qualifications which are not patterns but sets of patterns are allowed. The syntactic appearance of those qualifications is included in the syntax which is also used in gbeta, but the difference is that BETA allows them whereas they are static semantic errors in gbeta. An example is as follows:

```
window: (# item: (# #)#);
aWindowItem: ^window.item;
```

Ex. 3-10

In this example, a pattern `window` is declared, and the pattern `item` is declared as an attribute of `window`. This means that each object which is an instance of `window` will contain a pattern attribute `item`, and the `item` from two different `windows` will be different patterns. Nevertheless, the variable object `aWindowItem` is declared to be able to refer to any object which is an instance of `window.item`. In gbeta, the very expression `window.item` causes a compile-time error, because there is no such thing as an attribute of a pattern—only objects have attributes, patterns *describe* what attributes their instances will have. However, in (Mjølner) BETA this is taken to mean that `aWindowItem` is allowed to refer to any instance of an `item` pattern in any instance of `window`, i.e., the qualification is a set of patterns with an unbounded number of members. There will be as many patterns in this set as there are instances of `window`.

This kind of generalization of qualifications has been proposed for BETA in 1996 [12]. In this proposal it is possible to use arbitrary AttributeDenotations as qualifications, such that a pattern anywhere in a construct like `a.b.c...x` would indicate that there is *some* object in the program execution which could be chosen to take that position in the chain. E.g., if `j` were a pattern attribute in a qualification `h.j.k` for a variable object `obj` then it must at all times be possible to find an instance of `j` such that the `k` attribute in this instance yields a pattern which is greater-equal than the actual pattern of the object referred by `obj`.

However, this design has not been used in gbeta. There are some problems with it; consider the following example:

```
a: (# b:< object #);
theB: ^a.b;
```
Ex. 3-11

With these declarations it should be possible to let `theB` refer to an `object`—if and only if there is an instance of `a` in the program execution in question. Now, what should the response be if the last instance of `a` were removed by the garbage collector? Moreover, what if there is an instance of `c`:

```
c: a(# b:: integer #);
d: a(# b::< string #);
e: d(# b::< (#..#)#)
```
Ex. 3-12

In this case it would presumably still be OK to let `theB` refer to its object *if* it happened to be an `integer`. It would be acceptable for `theB` to refer to a `string` as long as some instance of `d` were in existence, but in this case it would be necessary to check that this were not an instance of a subpattern of `d` which further-binds `b`, such as `e` does.

In other words, it seems to be a highly confusing semantics in the general case. We have not been able to come up with restrictions which would be understandable and which would allow a "sane" subset of all the possibilities with these generalized qualifications, but it appears obvious that there should be added some constraints on the allowable expressions. For example, we might require that each non-last pattern element in the AttributeDenotation (these are the “ambiguous” objects) should be uniquely determined as standing for one particular object by the immediately following element in the AttributeDenotation. In the patterns `a`–`e` above there is no way to choose “the” `a` object for `theB`, because `theB` does not in any way depend on the `a`.

In other words, attempts to understand or explain the semantics of this construct in general seem to run into complications, not to mention implementing a correct type check for the usage of it. Consequently, no such construct has been included in gbeta, even though it remains an attractive goal to support qualifications which denote non-trivial sets of patterns.

Chapter 4

Virtual Patterns

This chapter introduces virtual pattern attributes. They have already been mentioned several times before because they play such an important rôle, but this section is the place to look for a comprehensive presentation of them. We will start with a slightly extended summary of the information about virtuals already given earlier.

A virtual pattern attribute is, as the only kind of attribute, unified across multiple mixins. This means that an object having several part objects whose mixins specify declarations of a certain virtual V will only have one such virtual attribute V, but all the declarations of V will be considered as a group when creating that attribute.

The unification of declarations belonging to a given virtual attribute is a combination process which puts together the contributing patterns from all the declarations, thus arriving at a combined pattern. This pattern is a specialization of each of the patterns denoted by the right hand side of the unified declarations, so each declaration may be considered as a constraint which specifies a minimum structure which the resulting pattern must support. Technically, the combination is defined in terms of the merging rule, presented in Sect. 3.7.

The remaining sections in this chapter will give the technical details of the attribute unification process and then compare the resulting notion of virtual patterns with the virtual patterns in BETA. After that, they will be compared to virtual methods in other object-oriented languages, and then to class or type parameters in languages with parameterized classes or types.

4.1 The Construction of a Virtual Pattern

For a given object, the value of a virtual pattern V for which there are some declarations in the mixins of the object is obtained by merging the patterns denoted by the right hand sides of those declarations. To be able to perform this merging the declarations must be selected, and the merging process must be specified in details. This will be covered in the following.

First, we need to mention a fine point of terminology—the term *virtual declaration* covers both a virtual pattern declaration (whose Kind is '`<`'), a virtual further-binding declaration (whose Kind is '`:<`'), and a virtual final-binding declaration (whose Kind is '`:`'). The various kinds of declarations were introduced in Sect. 2.2.5.

To specify the selection of virtual declarations which are considered as belonging to a given virtual attribute, we must introduce the concept of a *virtual chain*. A virtual chain is a list of virtual declarations of the following kinds: exactly one virtual pattern declaration, then zero or more virtual further-binding declarations, and finally zero or one final-binding declaration. Specified as a regular expression, the list of Kinds must be on the form < :<* :$^?$. The meaning of the (optional) final-binding of a virtual is that it cannot be further-bound after that; in other words, it changes the virtual pattern from a pattern being known only by upper bound into a completely known pattern.[1]

The virtual chain of a virtual attribute named N is determined by traversing the pattern backwards, starting with the most general mixin and ending with the most specific mixin—the head of the pattern. The virtual is identified with one particular virtual pattern declaration of the name N, the *identity* of the virtual relative to the enclosing object, and that must be selected. Note that a given pattern may actually contain more than one such declaration, so we must choose one particular virtual attribute named N among zero or more choices.

With a given choice of identity declaration for N, D_{id}, the virtual chain is fully determined. We need to introduce one more concept to describe it, though, namely the *identity of* a virtual further- or final-*binding* declaration D. The MainPart containing D is syntactically a part of a Descriptor (in the full grammar: an ObjectDescriptor) which also specifies a superpattern, and by local lookup in that superpattern a certain virtual pattern declaration is chosen as the identity of D. If no such virtual pattern declaration can be found, the program will be rejected with a static semantic error.

Now, the mixins can be scanned, starting from the mixin containing D_{id} and going through more and more specific mixins until the head of the pattern is reached. Whenever a virtual further-binding or final-binding declaration D' of the name N is found, it is included into the virtual chain if and only if the identity of D' is D_{id}. The intuition is that we include those virtual declarations which themselves claim to belong to D_{id}.

The end result is a list of virtual declarations, which is then checked to make sure that it has the right shape, < :<* :$^?$. If it has another shape, the program is rejected with a static semantic error.

Now, the pattern which is the value of the virtual attribute may be constructed. The virtual chain is consulted, and for each element in the chain, the right hand side is looked up and a pattern is obtained from it. This gives a list of patterns which are merged, again starting with the contribution from the most general mixin and going towards more specific mixins. This produces one

[1] A virtual may be final-bound and still only be known by upper bound; more about this at the end of this section.

```
1  (# p: ^1(# v:< object #);
2     q: p^2(# v::< integer #);
3     r: q^3(# v:: integer #);
4
5     a: ^4(# v:< ^5(# #)#);
6     b: a^6(# v::< ^7(# #)#);
7
8     mix: a & r & b; (* two 'v' virtuals *)
9  #)
```

Figure 4.1: Examples of virtual attributes

pattern which is then the value of the virtual attribute.

Consider a few, simple examples of virtual chains, as shown in Fig. 4.1. The specialization chain `p`, `q`, and `r` contains a virtual chain named `v` which give the following pattern values: In an instance of `p`, the virtual chain only contains the virtual pattern declaration in the MainPart 1, so `v` in `p` is `object`. In an instance of `q`, the virtual chain contains the identity declaration of `v` in 1 and a further-binding in 2, so `v` in `q` is `object&integer`, i.e., `integer` (remember that `object` is the empty list of mixins). Finally, in an instance of `r`, the virtual chain contains the identity declaration, the further-binding in 2, and a final binding in 3, so `v` in `r` is `object&integer&integer`, i.e., `integer`.

Even though the value is `integer` in both cases, there is a difference between `v` in `q` and in `r` since `v` is only known by upper bound in `q` whereas the statically visible bound in `r`, `integer`, is guaranteed to be the exact value. The difference is that assignment to a variable object attribute qualified by `v` in context of an object qualified (inexactly) by `q` would be type unsafe, but in context of an object qualified by `r` it would be type safe. Of course, the static analysis determines this, it is not something that a programmer is expected to analyze manually.

Now consider the combination of `a`, `r`, and `b` in line 8 of Fig. 4.1. There is nothing new with `a` and `b`, so we just mention that `v` in `a` is [5] and `v` in `b` is [7, 5]. However, when combining patterns from the $\{\mathtt{p}, \mathtt{q}, \mathtt{r}\}$ family with patterns from the $\{\mathtt{a}, \mathtt{b}\}$ family, two distinct virtual identities named `v` are brought together. Consequently, the `mix` pattern contains two separate virtual attributes named `v`. The reason why the virtual chains are kept separate is that each virtual declaration in the $\{\mathtt{p}, \mathtt{q}, \mathtt{r}\}$ family is statically associated with `v` in the MainPart 1 as its identity, and similarly for the $\{\mathtt{a}, \mathtt{b}\}$ family and the `v` in the MainPart 4.

For an instance of `mix`, an access to `v` using a view from $\{\mathtt{p}, \mathtt{q}, \mathtt{r}\}$ will select the virtual with the former identity, and an access to `v` from $\{\mathtt{a}, \mathtt{b}\}$ will select the latter. An access to `v` using `mix` as the view will cause a compile-time warning about the ambiguity, but then (for backward compatibility with BETA) it will proceed and choose the identity in MainPart 4, because the mixin associated with MainPart 6 is the most specific choice in the merge `a&r&b`, and the declaration in 6 is associated with the identity from 4.

Note that the merge a&r&b is $[6, 3, 2, 1, 4]$, so the contributions to the MainPart 4 identity of v are collected from the very ends of the pattern (mixins 4 and 6), without being "disturbed" by the intervening mixins (1,2,3), even though they contain declarations of the same name.

The result, which is very important for correctness in context of large scale projects where dissimilar code is brought together, is that virtual declarations are unified if and only if they belong to the same identity, and the identity is determined statically, such that a programmer can check out exactly what virtual (s)he is working on or using. Here is a small example—whose basic idea is commonly attributed to Boris Magnusson—which illustrates why this is important:

```
1  (# cowboy: (# draw: (# do 'BANG!'->stdio #)#);
2     graphicalObject:
3       (# draw: (# ... linepointrectfill ...  #)#);
4     graphicalCowboy: @
5       cowboy(# get_your_gun: draw #) &
6       graphicalObject(# show_yourself: draw #);
7     aCowboy: ^cowboy;
8     aFigure: ^graphicalObject;
9  do
10    graphicalCowboy.get_your_gun; (* BANG! *)
11    graphicalCowboy.show_yourself; (* creates drawing *)
12    graphicalCowboy[]->aCowboy[];
13    aCowboy.draw; (* BANG! *)
14    graphicalCowboy[]->aFigure[];
15    aFigure.draw; (* creates picture *)
16 #)
```
Ex. 4-1

In line 1 and 2 of the example the patterns cowboy and graphicalObject are declared, and line 4 declares an object which is an instance of a pattern combining both. The name clash is that draw was chosen as a method name in both cowboy and graphicalObject, so graphicalCowboy has two virtual methods named draw, with entirely different meanings and intended usages.

However, these methods are kept cleanly separate, and they can be selected individually using intuitive methods. First, line 5 and 6 show how to define additional names for the methods formerly named draw, and line 10 and 11 show how these new names can be used to disambiguate the two draw methods. Finally, lines 12–15 show how more restricted (focused) views on the graphicalCowboy object will recover the separate meanings of draw according to the chosen view. "When treated as a cowboy, the object will respond as a cowboy".

The chain based semantics of virtual attributes may sound convoluted, so we will outline some consequences which hopefully make the picture a little bit more self-evident. Firstly, the merging process has the semantics of virtuals in BETA as a special case. The BETA rule about virtuals states that a virtual further-binding or final-binding may (only) declare a more special value for the virtual than the one which holds in the immediate superpattern. This is exactly the effect of the merging semantics when the contributions form a decreasing sequence (i.e., when the right hand sides of the virtual chain denotes a sequence of increasingly specialized patterns).

We have this result (which is trivial to prove):

Lemma 4 *For any sequence of patterns $p_1, p_2 \ldots p_k$ which can be merged, the following holds:*

$$(\forall i \in \{1 \ldots k-1\} \ . \ p_i \geq p_{i+1}) \quad \Rightarrow \quad p_1 \& p_2 \& \ldots \& p_k = p_k$$

Moreover, for any two mergeable patterns p and q the following holds:

$$p \& q \leq p \quad \wedge \quad p \& q \leq q$$

The first part of the lemma says that an increasingly specialized chain of patterns simply merges to the last element in the chain. Actually, any sequence of pairwise comparable patterns merges to the most specialized pattern in the sequence, but only the decreasing chain is possible in BETA. Hence, the gbeta merging rule has the BETA specialization rule as a special case.

A consequence of the second part of lemma 4 is that the value of any virtual is a specialization of each of the contributions from the virtual chain, so each virtual declaration may be considered a *constraint* which requires that a certain list of mixins must be present among the mixins which constitute the value of the virtual as a whole. ◇

As an example of the *declarative* nature of this *mechanism*, it makes no difference if the same requirement is stated twice, as with the repeated `integer` requirement in the example in Fig. 4.1 on page 85. Since the merging process will add in the mixins which are missing it works similarly to a constraint solver which ensures that the required mixins are available, and does nothing for those mixins which are already present, just adds the new ones in suitable places. Similar to the transparency mechanism, Sect. 2.3, the action by the programmer is to request a particular end result. What is at hand and what must be done to end up with that result is implicit and is handled automatically. ◇

Note that some merging operations fail, but also note that all mergings of patterns known at compile-time (which is still the typical case) can be and are checked statically, such that failing merge operations only occur when combining patterns which are not known at compile time. A warning is given for all those cases where the merging must be carried out at run-time and a run-time error can not be ruled out. These very dynamic cases are the topic of Chap. 7.

Even though a virtual declaration in gbeta does not have to specify a more specialized pattern than the "inherited value" of that virtual, the result is the same as in BETA, in the sense that the merging operation will indeed produce a pattern which is more special than the value of the virtual in any of the superpatterns. This is another application of the second half of lemma 4. It means that a virtual attribute in a sequence of increasingly specialized patterns will itself be an increasingly specialized sequence of patterns. In other words, a virtual always "grows along with the enclosing pattern". This is also necessary to ensure type safety.

```
directedGraph:
  (# node:< (# incidence: @edgeSet; label:< string #);
     edge:< (# from,to: ^node #);
     edgeSet: set(# element::edge #);
     addNode: (# ...  #);
     ...
  #);
person: (# name: @string; ...  #);
family: @directedGraph
  (# node::person(# label::(# do name->value #)#)#);
```

Figure 4.2: An example of further-binding to an unrelated pattern

So BETA virtuals are a special case of gbeta virtuals, but that is of course not the only interesting property of gbeta virtuals. The generalized semantics allows for profound enhancements in the practical usages of virtual patterns. One seemingly small difference which will undoubtedly unfold to radically different designs in programs of non-trivial size is the fact that virtuals may be further-bound to unrelated patterns, not just to specializations of the value in the superpattern.

Figure 4.2 shows a simple example of how this can be used. The pattern `directedGraph` would be a generally useful implementation of support for directed graphs, with creation, traversal, searching, and so on. The `person` pattern would also be generally useful, and `directedGraph` and `person` would not have been written with each other in mind. However, the `family` pattern brings them together by specializing on `directedGraph` and further-binding the `node` virtual to `person` (and adding a new mixin to make the `label` of the `node` deliver the `name` of the `person`, thus integrating `person` into the new context).

The important observation is that `node` and `edge` in `directedGraph` depend on each other, and by further-binding `node` with `person` in `family`, a pattern is created which has all the functionality of `person`, enhanced with the rôle of being a `node` in a `directedGraph`. If a virtual cannot be further-bound to an unrelated pattern then it is not possible to make a separately developed pattern play a rôle which entails mutual dependencies, as in this example. This ability to create contexts in which independently developed patterns can be integrated could change the entire style of writing programs and libraries in gbeta, as compared to BETA. We will return to this example again in Sect. 4.4, when comparing virtuals with type parameters in other languages.

Another area where gbeta virtuals are more flexible than BETA virtuals is the choice of entities used for further-binding; in BETA they must be patterns which are completely known at compile-time,[2] in gbeta they can also be virtuals (or variable patterns, actually). The effect of further-binding a virtual V to another virtual W is that those two virtuals are connected in a network of constraints—

[2]As a special case, BETA does actually allow a virtual to be *final-bound* to an open virtual.

with such a declaration it is always ensured that $V \leq W$. Arbitrarily complex graphs of specialization dependencies like that can be defined, and the resulting system of patterns will behave similarly to a constraint graph; when a change is introduced somewhere in the system, the semantics of virtuals will ensure that the changes propagate all the way through the system until the constraints are again satisfied. This is an unusual and powerful mechanism, since it may change the value of many pattern attributes in a regular and statically inspectable manner in response to one or a few declarations, but it probably takes some time to develop a design and programming tradition which exploits the new possibilities. Here is a small example of such a virtual-to-virtual dependency:

```
stack:
  (# element:< object;
     common:< object;
     push:< common(# e: ^element enter e[] do INNER #);
     pop:< common(# e: ^element do INNER exit e[] #);
  #);
noisyStack: stack
  (# common::< (# do 'Working!'->stdio; INNER #)#);
```
Ex. 4-2

The example defines an interface pattern called `stack` which would have subpatterns supplying an implementation. The specialty is the "hook" virtual pattern, `common`, which has been used as the superpattern of the virtual methods `push` and `pop`. In `stack` the hook is empty, `object`, but in `noisyStack` it is further-bound to print a message before the rest of `push` resp. `pop` is executed. Such a hook can of course be used to invoke common behavior before/after each execution of the methods as it is done here, but is should be noted that it is more powerful than that. It can be used to add more arguments or returned results or attributes to all the methods in one operation, for example an extra integer input argument used to give a timeout to the operations in a distributed setting with unreliable network connections.

Finally, there are some new issues with the static and dynamic semantics of final-bindings in gbeta compared to BETA. When contributions to virtuals may be patterns which are not known exactly at compile-time, such as other virtuals, then a final binding with the semantics from BETA (which makes the virtual a compile-time constant) does not have acceptable properties. The problem is that a final-binding of a virtual which depends on another virtual (which has not been final bound) will affect that other virtual in a way which is unacceptable, namely by prohibiting further-binding of it. As an example, consider this:

```
p: (# v:< object; w:< v #);
q: (# w:: integer #);
r: q(# v::< boolean #); (* PROBLEM! *)
```
Ex. 4-3

The problem is that the final bound on `w` in `q` requires that `w` in `q` and in all subpatterns of `q` must be exactly `integer`, since that is the statically known value of `w` as seen from the final-binding declaration itself. This conflicts with the further-binding of `v` in `r`, because the requirement that `w` $\leq$ `v` would then only be satisfied if `w` contained a `boolean` mixin; `w` cannot be exactly [`integer`] and contain a `boolean` mixin at the same time.

The reason why this is an unacceptable situation is that the restriction on the further-binding of `v` has been created by a *usage* of `v`, namely the usage in the identity declaration for `w`. Since a declared name may be used arbitrarily far away from the declaration and in source code written by various different people, it is in general not acceptable to have restrictions which are caused by usage. The only way to avoid putting restrictions on `v` is to give up on the fixedness of `w`. If the final bound really does not mean that `w` must be exactly `integer`, then we can just make it `integer&boolean` in `r`, and all requirements are satisfied.

The approach used in gbeta is to detect the two situations and treat them differently. When the virtual is actually fixed by a final bound, the resulting pattern is registered and all bindings which change it are rejected. When the virtual depends on non-fixed patterns (typically other virtuals) the requirement imposed on the virtual is that no new contributions to the virtual chain can be added. Hence, the virtual may become more specialized because of already known contributions, but no new contributions can be added, not even by merging; as a result, the virtual will be fixed as soon as all nonfixed contributions become fixed—this could happen when some other virtual gets final bound.

It would be nicer to unify the two treatments by simply rejecting all bindings which would change a virtual which is already final-bound, but we have not been able to implement a correct static analysis of this rule. The current rules allow a subset of the programs which would be accepted under this "nice but hard to analyze" rule, so if it is at some point implemented then the only change would be that additional programs would be accepted by the static analysis, no programs acceptable under the current analysis would become illegal.

4.2 Why Non-Virtuals?

One of the rôles which are played by virtual attributes in Beta and gbeta is that of providing method implementations which depend on the run-time pattern of the object on which the method is executed, often designated as ◇ *late binding* of methods (look at the actual object, *then* choose the method implementation). In many object-oriented languages including Smalltalk, Eiffel, and CLOS, methods have late binding, either always or by default. Without static analysis late binding is the only option, and in many contexts including the Eiffel community it is considered the only justifiable choice [79, p. 514]. In other languages, including Simula and C++, methods must have a special annotation, like the keyword '`virtual`', to obtain late binding. Hence the term 'virtual method'.

With late binding, the object (or class) is so-to-speak allowed to decide for itself what action to take in response to an incoming message, and this is generally—rightfully—considered to be the most useful and manageable semantics for method invocations. This attitude is even embedded in the (originally

Smalltalk but now widespread) terminology which uses 'message send' instead of 'method invocation'. The alleged alternative is that of having several methods with the same name in a given class and then choosing one of those methods at compile-time, called *early binding*, based on the static knowledge about the ◇ class of the object, not its actual run-time class. It is obvious that the latter approach is error-prone because it may impose a treatment upon an object which does not fit the object—such a group of methods would presumably have been written to fit the instances of the class in which they are declared, and unless the static knowledge about the class of an object is exact, the method from a "wrong" class can be chosen. Hence, we agree that late binding is generally the right choice.

Then why is early binding the easy, default, "normal" choice in gbeta? The reason is that the alternative to late binding need not be the error-prone early binding semantics described above, it may just as well be a special kind of late binding whose outcome can be determined statically ... The idea is *not* that the choice of a method implementation should be made on basis of (incomplete) static knowledge, the idea is that there should not be a choice!

So in gbeta, the interpretation of an ordinary pattern declaration named N (intended for method-like usage) is "here's the method N". It specifies the interface and implementation[3] of the method, and it declares that the method is specified entirely in this declaration. Because of this understanding, the existence of several methods with the same name in a given view is considered a problem—bad design—and a warning is given if that method is used. The existence of several methods of the same name in an *object* is irrelevant, because it is the view on the object that determines what names are available and what they mean.

In contrast to an ordinary pattern declaration, a virtual declaration named N (intended as a method) is interpreted to mean "here's a contribution to the method N", explicitly allowing for other contributions which may affect both the implementation and the signature of the method. Note that there is no notion of 'binding' or 'choice' between different attributes named N, only the notion of having exactly one or at least one contribution to the *single* attribute named N.

Since it offers more immediate flexibility for programmers to allow for several contributions to a given attribute than to require exactly one contribution, it might be argued that only the former case, virtuals, should be supported, while the latter case, ordinary pattern declarations, should just be a special usage of virtuals where the flexibility is not fully exploited. In the same vein, the support for final-binding declarations could be removed, they are just further-bindings with some added restrictions.

However, the flexibility comes at a price. The price is zero as long as no static analysis is attempted, but for languages with static analysis including gbeta and BETA, very valuable extra information becomes available when some of the flexibility is explicitly discarded. In particular, a pattern which is only known

[3]The implementation may be hidden in another file, this is a separate issue, see Chap. 10

by upper bound is not fully useful as a qualification, because an assignment to a variable (object or pattern) attribute with such a qualification is not type-safe. Evaluation is no problem, but assignment is.

This problem is well-known from many contexts, but we have to introduce a concept before discussing them, namely covariance [1, p. 20]. In type systems for lambda calculi with subtyping, the following would be a typical type inference rule [1, p. 94]:

$$\frac{\Gamma \vdash \alpha \geq \alpha' \quad \Gamma \vdash \beta \leq \beta'}{\Gamma \vdash \alpha \to \beta \ \leq \ \alpha' \to \beta'}$$

The rule states that a function type ($\alpha \to \beta$) is a subtype ($\leq$) of another ($\alpha' \to \beta'$) if the result types have that relation ($\beta \leq \beta'$) and the argument types have the *opposite* relation ($\alpha \geq \alpha'$). This is the rule which must be used in order to obtain a sound (correct) type system, the "obvious" rule with $\alpha \leq \alpha'$ would conflict with the subsumption rule (subtype polymorphism). As a result, the argument position and the result position of such a function type are often designated as contravariant resp. covariant positions—the type in a *covariant*
◇ *position* is allowed to vary *with* the composite type it is a part of, but a type
◇ in a *contravariant position* must vary *against* the composite type. The problem with the upper-bound-only qualification mentioned above is that a "covariant type" occurs at a contravariant position, typically when the variable occurs in an EnterPart so the assignment happens during argument transfer. For this reason it is commonly known as the "covariance" problem—the qualification is covariant, but should have been contravariant.

Now we can return to the question of why the restricted cases (such as ordinary pattern declarations) are at all supported in gbeta. Since the design of gbeta (and BETA) rests on the assumption that static type safety is a valuable property of a programming language, the flexibility trade-off is actually redistributed considerably: It is true that a programmer who writes a specialization of a given pattern P with a virtual attribute named N has greater freedom than (s)he would have had with an ordinary pattern attribute named N, everything else unchanged. But since the ordinary pattern attribute can be used (safely) in many more places than the virtual pattern, the flexibility of *changing* N must be weighed against the flexibility of *using* N.

Hence, the situation is not so simple that virtual patterns are "better" and should be the only choice; on the contrary, it does make very good sense to allow programmers to commit to certain restrictions.

4.3 Comparison with Virtual Methods

Virtual patterns used as methods are not in themselves very different from virtual methods in other languages. The big differences are associated with patterns, and they have already been covered elsewhere. So at this point we will just give a summary of the consequences of using patterns for this purpose.

A virtual pattern is specialized covariantly along with the patterns of the enclosing object, so behavior cannot be discarded, only refined. The precise meaning and consequences of this have been discussed at length in Sect. 3.8. The fact that behavior cannot be overriden may be viewed as a needless loss of flexibility, but it is also possible to consider it as a sobering device which teaches programmers to commit only to those details which are actually going to be relevant and useful in all subpatterns. The resulting designs would thus be "cleaner". Nevertheless, gbeta allows programmers to effectively discard behavior by using the merging semantics to put a mixin "on top of" the inherited value of the virtual, e.g., by further-binding $[y, x]$ to $[y, x, z]$ where the new z may choose to ignore y and x by not executing INNER.

A widely debated issue is that of argument variance. C++ provides no argument variance [31]; Eiffel allows unrestricted argument covariance [79, p. 621++] and uses global analysis to compensate for the loss of type safety; theoretical object calculi may have explicit variance declarations, e.g. [1, p. 223]; Cecil [21] also allows for explicit variance by means of type inequality declarations. In gbeta there is no separate notion of support for 'covariant' or 'contravariant' arguments. Arguments are just attributes which happen to be used in an EnterPart, and the effect of a covariant attribute may be achieved by using a virtual qualification (virtuals *are* covariant), as in the following example where push has a covariant argument e:

```
stack:
  (# element:< object;
     push:< (# e: ^element enter e[] ... #);
     ...
  #);
```
Ex. 4-4

An invocation of push can only be type safe if the exact pattern is known for the stack object on which push is invoked; this may be achieved in various ways, very often by accessing the stack via an object attribute (not variable). A subpattern of stack might also final-bind element and thus remove the covariance of e.

Contravariant attributes cannot be allowed in gbeta (or BETA), because the semantics of specialization of behavior requires that all mixins be executed, see Sect. 3.8 for details. If an argument to $[y, x]$ were only required to be qualified by a superpattern of what is required with the superpattern $[x]$, the code in x which uses that argument could not safely be executed. Moreover, no real-life examples seem to motivate the need for contravariant arguments.

One technical detail which follows from the semantics of object creation is that a virtual pattern is complete at the first usage. This is different from the situation in C++ where a virtual method during object construction will be invoked according to different classes of the enclosing object, as if the enclosing object started out as object (even though C++ does not have object) and then changed class each time a new constructor was invoked.

Finally, the view on a virtual pattern as a method may be reasonable in a concrete case, but it will always be a partial view, disregarding many capabilities of the virtual pattern. Whenever more class-like usages of the virtual become relevant, the entire class functionality of the virtual is available for use. For example, a virtual method invocation can be treated as an object and, e.g., stored in a list for later execution. This remark will of course apply everywhere.

4.4 Comparison with Type Parameters

Recently, virtual classes, virtual types, and similar concepts with more or less well-defined semantics have been compared with type parameterization mechanisms in several papers [13, 110, 54]. A choice must be made as to what kinds of entities should be supported by the mechanism. It does make a difference whether it is parameterized types or parameterized classes, but we will delay this discussion to the end of this section. These two mechanisms are considered as alternative designs which achieve similar goals, and it has been demonstrated that they are good at different things [13]. They will be exemplified and explained below.

However, it is a common misconception that virtual classes are an inherently type-unsafe mechanism, apparently because it is assumed that no mechanism can exist which removes the covariance. Since object attributes and final-bindings have been used for this purpose in Beta for many years, it can only be hoped that the discussion will become more well-informed in the future.

A type parameterization mechanism may arise in different ways. The most simplistic version is exemplified by C++ templates, which are essentially textual macros, as mentioned near the end of Sect. 3.2. Hence, the entire template must be inspected by the programmer who wants to use it, in order to determine whether a given set of arguments are appropriate. For an ordinary function, in contrast, only the signature (in a 'header' file) is needed, and both the programmer and the compiler may ignore the implementation of the function when using it. In effect, the template instantiation mechanism itself is *typeless*, it just inserts the arguments in the specified positions in the definition and hands over the result to ordinary analysis and code generation. Consequently, as we also mentioned, names which are used in a template definition may be bound differently for each instantiation, and this is a semantic anomaly compared to the static name-binding which is used in all other parts of the language.

Parameterized classes in Eiffel are more well-defined, and may be analyzed once and for all. However, as in other parts of Eiffel, undeclared covariance is allowed as a general principle, and global analysis is necessary [26] in order to check the type correctness of programs[79, p. 633]. The approach used in the global analysis is to keep track of which methods are called on objects whose class is only known by upper bound, and which methods have been overridden by methods with covariantly specialized argument types. The intersection (the 'polymorphic CAT-calls') should be empty. A single assignment statement (which contains an implicit 'upcast') or a single actual argument given to a method invocation (again with an implicit 'upcast') is sufficient to add a class to the set of classes which are accessed polymorphically, so this means that the type safety of a method invocation anywhere in a program may be destroyed by the addition of an extra statement in another part of the program with no explicit connections to the first location. In particular, this means that reusable libraries cannot be type checked once and for all, they must be re-checked for every program in which they are used.

Type parameters in Eiffel are actually quite similar to virtual attributes in BETA with respect to the static analysis, but the decision to make *all* type parameters and *all* method argument types covariant causes severe problems for the static analysis. One way to characterize the situation is that the freedom for programmers to *modify* entities by inheritance has been given absolute priority, and consequently the freedom for programmers to *use* those entities safely has suffered.

Parametrically polymorphic types [103, 95, 96, 92] in languages like Standard ML and Haskell can be described as *type schemes*, i.e., type expressions ◇ containing type variables where a choice of a type for each variable yields an ordinary, so-called ground, type. Such a choice of types for type variables is called an *instantiation* of the type scheme. Universal quantifiers with constraints, e.g. ◇ bounds, may limit the possible choices for some type variables. Note that the type inference and type analysis with parametrically polymorphic types does not need to work on ground types everywhere, it may as well use the *relation* between different type variables to determine whether or not a given expression is type correct, thus proving type safety with all the possible instantiations.

Existential quantifiers [82, 1] have also been considered but do not play as important a role as universal quantifiers in current languages. It might be a good basis for a formalization of virtual types, and it is used in [1, p. 173–184] to formalize self-types (which are also inherently 'covariant'). In any case, the type scheme then stands for the set of ground types which can be obtained by instantiation.

A variant of this concept of polymorphic types which was pioneered by System F [49, 92] is based on letting types be explicit in the language as values, hence allowing type schemes to be represented as functions from types to types. A problem with this approach, often referred to as the *Type:Type* prob- ◇ lem [18, 78], is that it blurs the demarkation line between types and values. That is a problem for type checking, since (interesting) languages in general are Turing complete and termination hence not guaranteed, so if static type checking should be possible then the values which are types should be kept separate from the values which are run-time program entities. If these two are not kept separate then it easily becomes an undecidable problem whether or not a given program is type correct. Nevertheless, parameterized types viewed as functions from types to types are the foundation for the following.

The work done in the area of functional languages has been used as a starting point for type parameterization in object-oriented languages, for instance in Pizza and GJ, which are extensions of Java, but also in Cecil. A simple bound on a type parameter, which constrains the legal arguments in instantiations to be subtypes of a given, known type, enable type checking of a parameterized class once and for all. Firstly, the implementation of the parameterized class can be checked under the assumption that the actual arguments will be subtypes of the bounds, and that generally allows the usage of all the attributes of the bounds. Secondly, each instantiation must be such that the arguments satisfy the bounds. Here is a GJ example:

```
interface Printable { abstract void print(Stream on); };
class List<T implements Printable> implements Printable {
  T storage[10];
  void print(Stream on) {
    for (int i=0; i<10; i++) storage[i].print(on);
  };
  ...
}
class Point implements Printable { ... }
```

Ex. 4-5

The type check can accept the statement `storage[i].print(on)` because the entries in `storage` can be assumed to be instances of some class which implements `Printable`, and an instantiation like `List<Point>` is accepted because `Point` actually implements `Printable`.

A similar example in gbeta would look like this:

```
Printable: (# print: (# on: ^Stream enter on[] #)#);
List: Printable
  (# T:< Printable;
     storage: [10] ^T;
     print::
       (# do (for i:10 repeat on[]->storage[i].print for)#);
  #);
Point: Printable(# ...  #);
```

Ex. 4-6

The main difference between the type parameter based approach and the virtual pattern based approach for this example is that `List` in the first case is a purely compile-time entity, so it does not have any representation at run-time and it cannot be used as such in programs. Only when it is given type arguments (when it is instantiated) does it become a class, which can then be used just like other classes. That makes a difference because we can refer to a `List` polymorphically in gbeta and, e.g., print it without having any compile-time information as to what type parameter the `List` we are actually operating on "has received":

```
(# intList: @List(# T::integer #)
   anyList: ^List;
do intList[]->anyList[];
   screen[]->anyList.print;
#)
```

Ex. 4-7

In the type-parameter approach it is simply not possible to declare a variable like anyList whose qualification is the generic, argumentless `List`. On the other hand, a variable object like `anyList` allows for any value of `T` since there is no lower bound on its qualification, so such a variable cannot be used for, e.g., type safe insertion of new elements. That just means that `anyList` is useful for purposes where the virtuals used as type parameters are not in a contravariant position. For insertion etc. an access path like `intList`, whose pattern is known exactly, is more appropriate. Since `T` is actually final-bound in `intList`, a variable object with `intList` as qualification would also allow all usages.

The approaches are both statically type checked and they are both safe. The difference is that the approach based on type parameters disallows polymorphic access entirely whereas the approach based on virtual attributes allows polymorphic access but only for a *computed, reduced interface* which excludes all the elements where a covariant type is used in a contravariant position. In both approaches it is of course possible to either stop with an error message or give a warning and generate a dynamic type check whenever an unsafe construct is detected.

The type parameter based approach is more convenient than the virtual attribute based approach in one respect, namely in that it makes more classes type equivalent. In gbeta, any two syntactically distinct occurrences of the syntax `List(#T::integer#)` will denote different patterns, even though such collection data structures could often be considered equivalent. This is because a list-of-`X`, for some `X`, does not have any conceptual significance itself, the elements have their own quality but any two lists of them have the same quality (see Sect. 3.4 for details about qualities). However, even collections of other objects may at times have their own qualities, such as when a collection of 'boys' is actually a 'gang'.

For a more sophisticated usage of type parameters we need to introduce the concept of *F-bounded polymorphism* [17]. When considering a parameterized type as a function from types to types, and when operating in a type space which is a partial order, it becomes possible to select types according to their algebraic properties in this type space. In particular it is useful to focus on the pre-fixed points of the parameterized type, i.e., those types τ for which $\Phi(\tau) \leq \tau$, where $\Phi(\cdot)$ is the function which is also the parameterized type. This kind of bound selects all those types which have a particular recursive structure, as in this example: ◇

```
interface Ordered<T>
  boolean lessequal(T other);

interface SortedList< T implements Ordered<T> > {
  boolean insert(T elem);
}
class Integer implements Ordered<Integer> { ... }
class String implements Ordered<String> { ... }
```
Ex. 4-8

This example is inspired by a similar example in [65], and it will be revisited in Sect. 9.2. `Ordered` is a parameterized class which is not intended to receive arbitrary type arguments, it is intended to be used like in the classes `Integer` and `string`. That kind of usage establishes the relation which is required to make `Integers` and `Strings` acceptable type arguments to `SortedList`. The recursive structure which is ensured by a relation like

```
X implements Ordered<X>
```

can be described in terms of an unfolding operation: The insertion of X in place of the formal type argument in the definition of Ordered must create the definition of an interface which is actually implemented by X, i.e., all the methods must be available and each method must have the right signature. But if X is itself considered as a fixpoint of a function which maps an expression Y to the definition of X with Y inserted in place of each occurrence of X, then the criterion above just means that an unfolding of X is less-equal than an unfolding of Ordered, i.e., that the function whose fixpoint is X must be pointwise less-equal than the function Ordered.

By going from the domain of classes to the domain of those ("unfolding") functions whose fixpoints are the classes, the somewhat cryptic formulation X implements Ordered<X> gets reduced to a simple less-equal criterion.

Now, the fixpoint operations produce distinct, unrelated classes, so for example Integer and String above are not related by subtyping in any way, they do not even have a common supertype. Within the F-bounded polymorphism community this is generally considered a feature and not a bug, since it would be against the intentions to, e.g., put an Integer and a String into the same list and then have to support comparisons of them. Integers may be compared, and Strings may be compared, but an Integer and a String cannot. With virtual attributes this is handled in a different way.

A special kind of string such as for instance StyledString could be created such that StyledString implements Ordered<StyledString>, but that does not imply that StyledString would be a subtype of String. In other words, Integer and String are kept separate, but those classes which *should* be related are also kept separate. In Cecil there is support for declaring subtype relations explicitly such that StyledString can be made a subtype of String [65].

Turning to virtuals, it is not necessary to make the classes/patterns entirely unrelated in order to obtain security against the confusion of Integers and Strings:

```
Ordered:
  (# knownType:< Ordered;
     lessEqual:<
       (# other: ^knownType;
          value: @boolean
       enter other[]
       do INNER
       exit value
       #)
  #);
SortedList:
  (# T:< Ordered;
     insert: (# elem: ^T enter elem[] ... #)
  #);
Integer: Ordered(# ...  #);
String: Ordered(# ...  #);
SortedIntegerList: SortedList(# T::Integer #);
```
Ex. 4-9

With these definitions, we cannot insert elements into a sorted list only known by SortedList as an upper bound, because the argument to insert is covariant.

But we can use objects qualified with `SortedIntegerList`, whether or not they are accessed polymorphically.

To summarize, many tasks can be solved similarly well with type parameters and with virtual class attributes. Eiffel demonstrates that the difference between virtual attributes and type parameters may sometimes be reduced almost to a syntactic difference, although the lack of general block structure in Eiffel makes it unsuitable for the expression of mutually dependent families of classes, and the insistence on allowing all parameters to be covariant impedes static type checking. On the other hand, parameterized classes offer more structural equivalence which is attractive in particular with collection data structures. Virtual attributes do not seem to allow for a similar structural equivalence in context of general block structure—at least, it would be the entire universe of enclosing objects which would have to be structurally equivalent, not just the two lists (or whatever it is we are comparing). Finally, the virtual attribute based approach supports a deeper kind of run-time polymorphism because safety is achieved by using computed, reduced interfaces (by "outlawing" methods with covariant arguments, etc.) whereas the type parameter based approaches achieve safety by not having a subtype relation between different instantiations or between instantiations and the parameterized class itself, since it is not even a class.

There is one more difference between the type parameter approach and the virtual attribute approach which we have not yet discussed. As the name says, type parameters are types and not classes, and virtual attributes are normally classes, or they are patterns which are even more capable than classes. It would certainly be possible to reduce virtual attributes to be types, even though this would require introduction of syntax for types in gbeta. However, we would consider this a serious loss of expressive power; for example, it would not be possible to call a virtual method or to create an instance of a virtual class in gbeta if this were realized. The problem is that a type describes some constraints but does not solve them itself, so there has to be a class (or method) available before objects can be created (or methods invoked), and a virtual type would not directly support any way to get access to such a class (method).

Chapter 5

Context Dependency and Block Structure

Consider the two sentences “There was a humongous dog in the book, and the little girl chuckled every time she saw it” and “There was a humongous dog in the room, and the reddish brown stains on the floor made me think about those horrible sounds I had heard the night before”. The introduction and description of the ‘dog’ itself is identical in the two sentences, but the meaning of this description—a picture or a large, dangerous animal—is heavily influenced by the context. Note that in this example, and in this chapter generally, ‘context’ means semantic context and not syntactic context; the ‘little girl’ and ‘chuckled’ are actually syntactically close to each other, but the important issue is that ‘chuckled’ is understood by using the mental model of the little girl as a frame of reference.

Context dependency arises at several different levels. As an example of a quite urgent context dependency at a fine-grained level, consider verbs and verb phrases. Phrases like ‘chuckled’, ‘saw’, ‘made ... think’, and ‘had heard’ are not self-contained, they only obtain sufficient meaning in the mind of a listener when the immediate context within the sentence is taken into account, such that ‘the little girl’ and ‘I’ can be identified as the primary agents of those actions.

The context dependency is not different in nature for verbs and for other words, but verbs characterize such transient phenomena as actions and developments, so they only provide an annoyingly incomplete amount of information unless the entities which perform those actions or undergo those developments are specified by other words. On their own, these transient phenomena are often not particularly significant, but viewed as indicators of developments involving less transient phenomena—such as people, dogs, houses, or even computerized objects—they may become very significant. Similarly, verbs in isolation usually carry only a quite limited and generic amount of information, so the knowledge about the immediate context of the meaning of a verb greatly enhances the depth and detail in the understanding of it; to return to the usual example, ‘the little girl chuckled' lets 'chuckled' provide some extra information in our minds about the 'little girl', and in return it lets the 'little girl' in our minds provide the understanding of 'chuckled' with extra depth.

In summary, contextuality allows us to reach a much richer interpretation of a word-in-context than what we can achieve with a word-in-isolation (where the "dictionary meaning" is all we have), and in return the context dependent meaning may enrich and/or modify our mental model of that context. This chapter is about the exploitation of contextuality in the design of programming languages, in order to make programs more comprehensible for human beings.

◇ *Block structure*, or *nesting* is a programming language concept which corresponds to the notion of context dependency at the conceptual level. This chapter first presents context dependency as it is used in human perception and thinking, in Sect. 5.1. This conceptual basis is used to motivate the introduction of block structure in programming languages in Sect. 5.2, and the concrete mechanism used in gbeta is described in Sect. 5.3. With block structure in place it is possible to give the rules for name lookup in the general case, and Sect. 5.3 also covers this, thereby completing the treatment of lookup which was given in Sect. 3.10. Finally, the similarities and differences between block structure and modules are briefly described in Sect. 5.4.

5.1 Contextuality for People

As we have seen, natural language incorporates several powerful mechanisms which are used in the construction and maintenance of useful mental models of the world. However, even though the model incorporates a choice of a suitable perspective and hence a selection of a few aspects as relevant and a rejection of the rest as irrelevant, there is no way we could maintain a model which would be complete enough to be useful in itself. Our immediate consciousness simply does not have the capacity.

Hence, somehow we are able to live in the world and handle all sorts of upcoming situations while just thinking about a tiny little bit of it at any given point in time. Moreover, the small bits of the world that we can handle in our consciousness are not directly available for us as they exist, they can only be detected by our senses in a very incomplete and sparse manner.

Section 5.1.1 describes this risky guesswork which provides us with the illusion that we simply experience the world as it is, and concludes that we, at this very basic, ontological level, rely on contextuality to make sense of sparse and incomplete information. We choose to focus on perception processes because this demonstrates how deep our dependency on contextuality is; it is really a built-in mechanism that we all use all the time.

After that, Sect. 5.1.2 argues that natural language enhances our ability to rapidly switch between a multitude of contexts. This raises the question of how such contexts may be organized such that it is possible to navigate between them. Section 5.1.3 presents a very basic kind of organization which may be considered as the real-world counterpart of the mechanism which is used in programming languages such as BETA and gbeta.

5.1.1 The Almost Static World

We use our senses in a way that depends on a very fundamental assumption, namely the assumption that the real world is largely static, and changes are in some sense continuous. Note that we are not talking about assumptions that people make as part of their daily mental activities, we are talking about deeply entrenched structural and functional properties of the central nervous system of living creatures, developed by natural selection over billions of years. However, had there been an architect for all this, then that architect would have needed to make such an assumption in order to justify the design.

Without this assumption, i.e., under the alternative premise that any in itself plausible situation[1] migth be immediately followed by any other in itself plausible situation, our senses would not give us any useful information. This is because our senses provide the central nervous system with a highly incomplete stream of measurements of the state of the world. We do not perceive the world as it is, we perceive a few glimpses of it now and then.

This is in particular evident with the eyes, where the very detailed vision is concentrated in a few percent of the area covered with light sensitive cells, such that, for example, we cannot read if we look just a few degrees in a wrong direction. The illusion of being able to see everything clearly arises because we move our eyes frequently, and because the world generally changes so little that we can maintain a sufficiently correct model of the world without actually seeing all of it all the time. Other indications of this wonderful guesswork are simple facts like the following: we only see, hear, feel, smell, or taste a small portion of the world, most phenomena are too far away or for other reasons not discernable for our senses; we only perceive a few different aspects of the world, not, e.g., infrared light or magnetic fields; and finally, our five senses (and all the other senses we haven't mentioned) do not detect the same phenomena, so our perception of the world must continually be reconstructed from a very heterogeneous set of sensory inputs, interpreted as witnesses of more or less overlapping sets of phenomena. The conclusion is that we experience the world by sparse samples of a few measures, not simply "the world as it is".

It is amazing that we can arrive at a useful interpretation of this world, based on only a few, unreliable hints now and then as to what it is really like. This is possible because we are able to hold some kind of a *continuous model* of the ◇ real world inside our mind at all times. Sensory input is then as far as possible interpreted in a way which is meaningful *within* the model, and then used to *modify* the model. This continuous process of model-adjustment equips us with a continuously useful model. This means that every new piece of information injected into the model gets interpreted *in context* of the model. Moreover, when the result of the basic model-adjustment process is unsatisfactory, we have the

[1] What would required for a situation to be 'plausible' under this alternative premise is of course a separate, hard problem

capability to switch to another model. Hence, we must have a rich source of potential models, somehow close to the consciousness, readily available if needed.

◇ This is what we mean by *contextuality*: the phenomenon that a stimulus is interpreted not by looking up an absolute meaning (as in a dictionary), but ◇ relative to an existing body of knowledge, the *context*.

Actually, it is not necessary that the world be static in order to make the model based approach to world-understanding useful; it only has to be *predictable*. Predicting that nothing will change is probably the most basic strategy, and it is almost always right. However, when a rabbit tries to run away from a hungry fox and then barely manages to escape by quickly changing direction, it actually relies on a prediction that the fox will continue in approximately the same direction as it had just a moment ago. Hence, the prediction of no change, and the prediction of approximately constant speed and direction of physical entities "by default", and the prediction that it is a bad idea to let the fox get you in the first place, and countless other predictions, all these are used all the time by all kinds of living creatures.

There are many different mechanisms behind such predictions, starting from a simple nerve loop, a reflex, like the one that make us pull away a hand that gets burned; continuing to largely unconditional patterns of behavior in response to given stimuli, instincts, such as the construction of an anthill; and on to conscious and scenario based planning, like when a cat runs around the house after having found the front door closed, to see if the back door is open.[2] Note that 'prediction' in this context refers to the actual usefulness of the effect on behavior with regard to future developments, not the presence of a conscious, human-like thinking activity which enables an envisionment of that future. E.g., the reflex predicts that pulling back the hand will avoid damage by means of low level nerve system structure, not by means of thoughts; and the anthill is built and maintained because of the fractal-like complex community wide dynamics that arise as a combination of all the rather stereotype individual ant behaviors—the prediction is that each individual ant will do well by exhibiting those stereotype behaviors, in context of the more general prediction that the future world will be a place where it is a good idea for ants to live in anthills. On top of all these ancient techniques we have piled natural language, and that allows us to share and systematize our experiences and thereby enhance our predictional power considerably.

To summarize, we can survive in a world of which we only have very sparse direct knowledge because we maintain a continuous model of it. The model maintenance process is inherently contextual, every new stimulus is understood in context of the model. A variety of prediction techniques have evolved to support appropriate reactions to and maintenance of that model, the most basic and most widely applicable one being the prediction that nothing changes. To manage the possibility of change we need to associate specific phenomena with specific prediction strategies, because reasonable prediction strategies are very

[2]Biologists traditionally avoid such anthropomorphic explanations, but "it doesn't think, only humans do that" just does not seem to make sense.

different for different phenomena. Natural language, in particular concepts, supports such a classification of phenomena, but it does this on the basis of other mechanisms that we share with other living organisms. This means that we cannot arbitrarily choose to use alternative mechanisms, nor can we expect to know consciously how we arrive at everyday conclusions.

In other words, human beings produce understanding by means of contextual interpretation in models containing classified entities. If we are to understand programs then it seems promising to seek to support and exploit this approach.

5.1.2 Natural Language Brings Us up to Warp Speed

Natural language supports a further development of the continuous model adjustment process by allowing an effective maintenance and inter-personal sharing of models, *independently* of the current sensory environment. Because of the continuous sensory input, the model of the immediate environment will always be very close to the consciousness, but we can also consider non-existent or non-present things, and it is language that allows us to *share* such adventures. A language based stimulus is received by the mind, interpreted within the current model in order to provide it with "meaning", and then treated as a request (from another person) to adjust the model—and the model may or may not be concerned with the immediate physical environment. Actually, because of the independence from the immediate physical environment, language based stimuli provide human beings who communicate with each other with a constant opportunity and requirement to switch between more or less unrelated points of view in their thinking. In this respect, verbal communication could be compared to moving around at warp speed in the physical world.

Consider again the two sentences about a dog which introduced this chapter. We must realize that the model context in which any given language stimulus is interpreted is generally tremendously complex. The 'dog' obtains meaning from the contexts which are established by the rest of the two sentences, aided by a large, reader supplied body of knowledge about fictional literary style, and that meaning itself is then being targeted as the topic of the next few sentences, in order to influence the model of production of meaning in the mind of the reader, and that treatment of the treatment is finally made the topic of this sentence—which happens to topicalize the fact that it talks about itself!

The organization of all those possible points of view, or potential current models, is of course highly individual, dynamic, and sophisticated. This is undoubtedly a rich source of inspiration which ought to be explored in future programming language design efforts, but here we will concentrate on a very basic kind of organization which is closely related to the organization of the physical world. After all, the most basic modes of being are likely to be the most robust workhorses, even though we may have a hard time appreciating them because they are so ubiquitous.

5.1.3 Physical Nesting

There is a basic mode of model organization which is derived from the organization of the physical world as we perceive it, namely by starting from one particular point in space, the *location*, and including successively larger portions of the physical world in a series of contexts, each one containing all the previous ones like the layers of an onion. Of course, an actual choice of sharply delineated geometric shapes for this would be meaningless, it is more like a series of fuzzy clouds with functionally defined boundaries like “me”, “the things I can reach out and touch”, “this room”, “the things I can see and could easily walk over to”, “this city”, and so on and so on. Let us call this a *physical nesting* organization of the universe of potential models. Note that this organization automatically orders the models in a way which corresponds well with two different measures: The innermost models have the most immediate relevance and possibly urgency, and models further out gradually become irrelevant for most short-term purposes; at the same time, models further out gradually cover larger and larger portions of the physical space, so they become less and less detailed and thus more suitable for the tracking of changes which are substantial enough to make a difference in spite of the distance.

A very nice property of a physical nesting model organization is that it supports a kind of stability which is somewhat related to the stability of the world which was discussed in the beginning of this section. A largely static world offers the stability property that most of what is true now will still be true in the near future; similarly, with a physical nesting organization of models, a change of model (or viewpoint) to another one with a nearby location will have the stability property that most of the enclosing models remain unchanged. In other words, we can move around and thereby constantly invalidate some of the innermost, nearest models, but the fact that almost nothing has changed when viewed on a larger scale allows us to perceive the world as a comprehensible place, even if the shapes, colors, and sounds that we are immersed in keep changing relentlessly.

The stability of enclosing contexts as a way to make sense of the more transient immediate contexts is at the heart of object orientation, as we shall see in the next sections.

5.2 Contextuality in Programming Languages

It is obviously worth considering if and how a programming language can be designed in such a way that it allows programmers to use their general comprehension competences when reading and writing programs. The previous sections motivate certain directions for the design, given that we set out from the premise that human beings who are dealing with computer programs are first of all optimized for living in the physical world and then, as an afterthought, equipped with natural language and civilization, and finally, basically as a big surprise, immersed in a modern technological world.

The design directions include the following: Human beings are well-trained in exploiting contextuality, that is, in establishing mental models where external stimuli are not just understood by their immanent (stand-alone/dictionary) meaning, but are instead highly enriched by being interpreted in context of a model and then used to update that model. Similar techniques should be applicable when reasoning about computer programs.

Moreover, physical nesting seems to be a very fundamental and useful organization of a universe of potential models, and it has the earlier mentioned nice property of making close models more relevant and urgent, and remote models more coarse-grained and stable. It is also desirable that programmers have support in organizing their understanding of program executions into such a structure.

Context dependencies in a formal system like a programming language must of course be based on a much more rigid and simple-minded mechanism than anything in a natural language. General block structure is one such mechanism which will allow words to derive their meaning from all the enclosing contexts, but a number of restricted versions of block structure are also being used in various programming languages. We briefly describe them first, for comparison.

The simplest approach is to use one large, global, unstructured universe containing all of the available entities, like in traditional FORTRAN, early versions of BASIC, or symbolic machine code ('assembler'). This—degenerate—case of context dependency support works nicely for small, simple projects, because there is no real need for complexity management, and because the lack of this kind of functionality allows for a simpler language.

However, as soon as the complexity rises above the trivial level the unstructured universe becomes hard to manage, because every consideration about a given program basically has to set out from a total and simultaneous awareness of the entire program in full detail. The experience with programs of this kind or just programs of which some aspect was of this kind has given rise to the widely accepted rule that global variables should be avoided. The argumentation given here at least motivates the rule that there should only be *few* global variables.

The next step up is to support localized groupings of state, such as records, or behavior, such as procedures. When a given group of variables are accessed as fields of a record then the programmer is supported in thinking about this group of variables as an entity in its own right, and this alone reduces the complexity of the system by reducing the number of entities. However, it is only really useful if each usage of a variable from this group can be meaningfully explained in terms of the group as a whole, so in particular the group as a whole must be meaningful.

Similarly, a number of statements enclosed as the body of a procedure provide for a simplification, compared to having the same statements on their own at the global level of the program. Again the composite effect of executing those statements should be meaningful as a whole, and each statement should make sense as a contribution to that composite effect. Apart from the fact that the statements can be understood as a group, it is also important that the number of possible execution paths involving such a group of statements becomes much

◇ smaller, especially if the ideals of *structured programming* [116] are taken into account. The core ideal here is that there should only be few and readily recognizable possible execution paths, and this is achieved by having such rules as: each procedure should have only one entry point and one exit point, and the `goto` command should be shunned [28] in favor of a few more specialized control structures (like `if`, `while` and `for` statements).

Even more important, the notion of a procedure invocation is reified: The invocation of a procedure implies not only jumping to and from a group of statements, but also the establishment of a local environment which is specific for the given invocation.[3] This makes it possible to receive arguments, to use local variables, and to return results; it also allows for recursion.

This notion of a local environment for each invocation of a procedure is an example (and an important one) of support for contextuality—the body of the procedure is executed in context of this environment, and since the environment is invocation specific, it is capable of supporting the modeling of a specific action or development which is an example, or instance, of the concept which is associated with the name of the procedure (assuming that the procedure has a name that corresponds to the effect of executing it, which of course it *should* have).

It is no surprise that procedures are the first entities to have such a good support for contextuality, since they are so closely related to verbs, and verbs are so urgently dependent on their context. However, a verb phrase in a sentence is normally associated with a noun phrase which names a primary context for the interpretation of the verb, and then optionally an object which is being manipulated. Procedures support the notion of objects being manipulated, but not the notion of being contextually dependent on a less transient enclosing entity.

This level of support for contextuality corresponds to traditional imperative programming languages like C and Ada83. At this level it is actually possible to create and maintain quite large and complex systems, such as operating system kernels and, particularly using Ada, systems for real-time control of airplanes and weapons. Clearly, procedures and records are so useful that they must be taken seriously.

However, in recent years there has been a strong trend towards using object-orientation in many different application domains, and it is often given as a reason that object-orientation is somehow more "natural" for expressing human thinking than other programming paradigms, e.g. [24]. We basically agree with this line of thinking, but on the very specific grounds that it represents yet another step forward in the support for contextuality.

This step forward, in the mainstream of object-orientation, simply consists in allowing procedures to be defined as contextually located inside records. This is a technical change; the associated change in mindset and terminology is pro-

[3]It is often called an 'activation record' and considered an implementation detail, but that is probably because it isn't a first class entity in most languages; even the most implementation-independent formal semantics would have to specify it as a semantic entity, similar to other environments such as ordinary records

found and differs quite a bit between the user communities of different languages, but the (heretic) use of the old terminology allows us to see more directly what new possibilities it provides us with.

From the point of view of the Scandinavian tradition of object-orientation, this enables us to model a thing (a phenomenon associated with a concept covered by a noun) along with actions and developments that take place in context of that thing. Technically this is unimportant because the 'thing' could easily have been provided to the procedures as, say, the first argument. In languages like CLOS, Dylan, and Cecil, the typical notation for access to an entity inside an object (the 'dot notation', as in `myPoint.move`) is actually explained as a mere syntactic convenience which in reality means that the object is the first argument to the procedure. So objects are not particularly important in object-orientation after all ...

We believe that this entirely misses the point! The improvement in the understandability of programs which can be obtained by supporting the execution of procedures in context of records (which is so profound that we rename it to things like 'the execution of methods in context of objects' or 'sending messages to objects') stems from the fact that contextuality is a complexity management mechanism which has been an inherent aspect of our approach to the world since long before the emergence of the human species. In other words, the important difference is in the support for our basic modes of thinking, and the fact that it technically may be "explained away" as syntactic sugar is an absolutely unimportant curiosity.

To clearly expose the reduction in complexity, compare the following scenarios: Assume that we have a payroll system which models each employee as an object which has various methods for registering performed work and for receiving payments. In daily use the system is used to update the state of each employee-object to reflect the amount of work which was performed by the corresponding employee. Once in a while some money can be transferred to a bank account, and the employee-object is updated to reflect that the payment has been made. A programmer working on such a program can work on basis of a mental image of an employee, and for each method on employees he or she can view it as an action which may depend on and/or change the employee in context of which it happens. The potentially large number of transient phenomona modeled by the methods are organized *in context of* the more long-lived phenomenon modeled by the employee-object.

In contrast, the connection between the transient phenomena and their long-lived context (the employee) has no direct support in procedural languages, so they give much less support to programmers for the context based complexity reduction. In other words, they emphasize each action or development in isolation, and that means that the collection of all actions and developments in a program execution becomes a *much* more complex whirlwind of isolated phenomena than it would be if it were organized into a number of object "life stories".

Most object-oriented languages stop here, but in the Scandinavian tradition of object-orientation a more general approach is taken, as a consequence of not abandoning but instead generalizing the support for block structure in Algol. In particular, Beta supports a very homogeneous and general form of block structure, a generalization of the block structure in Simula, which is again a generalization of the language Algol. Note that the decision to abandon block structure in other communities seems to have been made consciously [23, 51], and this probably illustrates that general block structure used in an unstructured and meaningless fashion can inflict great damage to the comprehensibility of a system.

The technical details of Beta block structure are the same as in gbeta, and they are explained in Sect. 5.3. The implications of this general block structure support at the conceptual level are unsurprising given the discussion so far, but we will briefly summarize them as they represent the logical end point of the development: As a generalization of the previously discussed forms of block structure, general block structure naturally supports the modeling of actions and developments by means of procedure invocations carrying their own, local environment, as well as the modeling of actions and developments in context of noun related phenomena (things, persons, ...). A usage of general block structure which goes beyond mainstream object-orientation, but which was already present in Algol in the sixties, is the notion of nested procedures. They allow the expression of contextually dependent subactions which are used to construct the more complex enclosing action.

Moreover, it is generally possible to define contextually dependent entities explicitly in context. For example, a student role can be modeled in context of the university to which it applies; an airplane ticket can be handled by a computer system using a model of an airplane seat in context of a particular flight; and an object inside a gbeta compiler which is capable of generating code for a specific piece of syntax may be defined in context of the object which represents that piece of syntax, which may again be defined in context of the grammar for the language, in context of the compiler, etc. Whether an object is understood as a model of a real-world phenomenon or it is understood entirely as a computer based phenomenon in its own right, the important message is that programmers are capable of using their real-world comprehension capabilities to understand the semantics of programs, in particular by exploiting the contextual complexity reduction.

However, the rigidity of concrete programming language mechanisms, in particular when static type checkability is an important goal, sometimes makes it necessary to carefully weigh the gains in simplicity and consistency against the need for flexibility for the handling of atypical cases.

For example, a project may be modeled in context of a company, and as a result the project may depend on the company (e.g. on the employees as modeled by their online calenders). The understanding of the life-stories of all projects can then be organized in context of the companies. Because of the regularities (e.g. that meetings for a given project are held in rooms belonging to the company of that project), the contextuality will ensure a basic and intuitive level of consistency that would not be supported if companies, employees, rooms, projects, and meetings were modeled in a flat universe, as global and unrelated classes.

Of course, a usable system must also be able to handle multi-company projects with meetings in third-party rooms. This can for instance be supported by defining abstract, global, unrelated classes for meetings and so on, and then defining concrete subclasses in context of a company for the simple default case (meetings in the company's own rooms), and other concrete, global subclasses for the hard-to-handle general case (meetings anywhere, with participants from anywhere). This gives rise to more classes in the program, but it might very well be worthwhile to exploit the simplicity of the simple case also for users of the program, so that local meetings can be scheduled quickly in an simple dialog box, whereas scheduling of "general" meetings includes an inherently more complex selection of participants from multiple companies, as well as negotiation procedures between the various scheduling systems used in the involved companies.

There will always be a trade-off between "hardwiring" design aspects for simplicity and making them into parameters for generality. As an example of why parameterization should not simply be equated with 'good' and hardwiring should not simply be equated with 'evil', consider a language where a procedure can receive its body as an argument, e.g., as a list of statements. Now we can implement a given program with much fewer procedures, all we need to do is to provide the body (possibly as a result of a clever computation) as an argument to each procedure invocation. Wonderful flexibility! However, the problem is that the *manifest* program entity, the procedure, has lost inherent meaning for the programmer who is trying to understand the program. It appears as an empty shell whose real meaning can only be understood when the infamous body parameter is known, and the choice of actual parameters is such a transient matter that it is hard to predict before run-time. As a result, we have seriously damaged the comprehensability of the program, even if we may have been able to reduce it to a smaller size.

Useful abstraction and parameterization mechanisms in programming languages are those that allow a programmer to construct entities which are *both* meaningful and widely applicable. We believe that the ability to define contextually dependent entities is an example of a mechanism which allows programmers to *reduce* the flexibility of the dependent entities in return for making the group of a context and its dependent entities more comprehensible as a whole. For example, a pattern P in BETA and in gbeta is specific for its enclosing object O, different from the pattern of the same name in context of another object O' even when O and O' are instances of the exact same pattern; similarly, the notion of a virtual or late-bound method in various programming languages implies that the meaning of the name of the method depends on the object in context of which the method is invoked. In both cases, the contextually dependent entity is in an essential way tied to the context, as opposed to the procedural "equivalent" where it is the *same* procedure that receives different objects as the first argument in different invocations. Multiplication mechanisms such as instantiation of objects and incremental specification mechanisms such as inheritance are then more important than ever, since they allow us to manage a larger selection

Global lookup of a name N in a view P:
Perform a local lookup in P; if that succeeds then the global lookup also succeeds, with the same result as the local lookup; otherwise find the view of the enclosing object of the frontmost mixin in P, and perform a global lookup of N from there; if there are no more enclosing objects then the global lookup fails

Figure 5.1: The rule for global lookup

of specialized variants than would otherwise be practical.

5.3 General Block Structure in gbeta

This section describes the syntax and informal semantics of the general block structure in gbeta. Syntactically, the general block structure amounts to the support for declaration of MainParts within MainParts in various ways, as it can be seen in the full grammar in App. A. Already the simplified grammar in Fig. 2.1 on page 23 shows the most important case, namely the possibility to write a MainPart nested inside a MainPart as a part of an attribute declaration, an AttributeDecl.

The semantic support for general block structure is also quite straightforward to describe, because all the descriptions of semantic (or run-time) entities given sofar have been designed to fit into the greater, contextualized framework.

Patterns are by definition context dependent since each mixin, as described in Sect. 3.3, includes the identity of the enclosing part-object. Similarly, each part of an object receives a part-object as its context from the mixin of which it is an instance.

This contextuality support in each part object is used in the general case of name lookup. In Sect. 3.10 the rule for local lookup was given. The global lookup rule builds on the local lookup rule, as specified in Fig. 5.1. Basically, it says that the local lookup rule is applied to each object in the context, starting with the immediate context and one by one searching more and more global ones until the requested name has been found, or the entire context has been exhausted and an 'undefined name' error must be reported.

Note that the search of the next enclosing object in each step always uses that view whose most specific mixin is associated with the next syntactically enclosing MainPart. In other words, lookup starts from the MainParts which are there to look at, directly surrounding the name which is being looked up. This means that the programmer can mentally annotate each MainPart with the environment that it provides, and then any name lookup will simply be

```
1 (#
    company:
     2 (# employee: 3(# ... #);
          project:
           4 (# client: ^customer;
                manager: ^employee
            #);
          realProject: project
           5 (# meeting:
                  6 (# from,to: @dateTime;
                        ... from ...
                        ... manager ...
                        ... employee ...
                     #)
            #)
      #);
    customer: 7(# ... #);
    dateTime: 8(# ... #)
#)
```

Figure 5.2: Global lookup example

a succession of searches in those environments. As a result of the ubiquity of the outermost environments (they are the context of very large portions of the program) and the syntactic and semantic immediacy of the innermost environments, name searches performed mentally by a programmer will often be rather easy, either because the name is nearby or because it is generally well-known.

The organization of objects corresponds to the physical nesting model organization, and the search order corresponds to the relevance and urgency ordering of the models by searching the most relevant objects first. Note that it is perfectly possible to construct program entity nesting structures that do not correspond to a physical nesting of phenomena, but that may just be an example of using our metaphorical capabilities to understand non-spatial issues in terms of a spatial-like organization, such as for example considering a novel in context of the life of the author. Hence, there is no “policy” that prescribes that the physical nesting model organization should only be used to model actual physical nesting. The overall goal is to make programs understandable, and the use of metaphors to go beyond actual physical nesting is an important human capability that should of course be leveraged.

A few examples of global lookup are given in Fig. 5.2. The block structure of executions of a program using the patterns in Fig. 5.2 is illustrated in Fig. 5.3 on page 114. The figure only shows the nesting and inheritance relations between the MainParts, not the actual configuration of part objects and mixins at run-

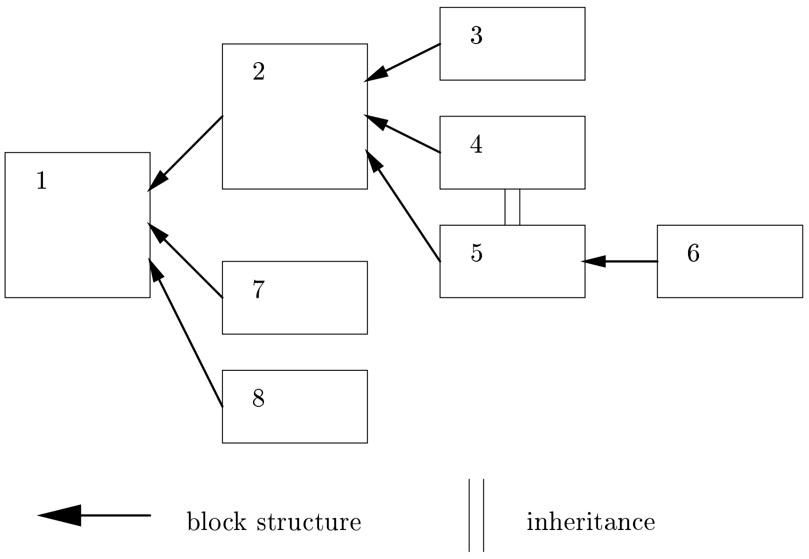

Figure 5.3: The block structure in Fig. 5.2

time. That is because there are so many possible configurations, so we have to give a slightly abstract representation of it.

The relation to potential run-time entities is as follows: Each box in Fig. 5.3 represents the MainPart with the same number in Fig. 5.2, so any `company` pattern will consist of one mixin, and that mixin will be associated with the MainPart (and the box) numbered 2; an instance of such a `company` pattern will consist of one part object which is an instance of the mixin. Similarly for other patterns and objects associated with the source code in Fig. 5.3.

Whenever the code inside a MainPart M is being executed it happens in ◇ context of a *current part object*, and the current object is always an instance of a pattern which includes mixins associated with M and all the MainParts of the statically known superpattern; the current object may also contain additional mixins which are not known statically. For any given part object there will be part objects for all boxes which are reachable by either block structure links or inheritance links (going left along an arrow or up along a double-line in Fig. 5.3). For example, given a `meeting` object (associated with box 6), there will be a uniquely determined object, associated with $[5, 4]$ (an instance of a `realProject` pattern), which is the enclosing object for the `meeting` object. Similarly, there will be an enclosing `company`, [2], around the `realProject`, etc.

Now, when looking up the names which are mentioned inside MainPart 6, we get the results as follows: `dateTime` is looked up by searching 6, 5, 4, 2, 1, in that order; `from` is looked up locally, just 6 has to be searched; 6, 5, and 4 are searched to find `manager`; and finally, 6, 5, 4, and 2 are searched in order to look up `employee`.

The lookup process may be described graphically: starting from a box M that contains the name to search for, we first search upwards, among inherited attributes; if that fails then we take one step outwards from M and repeat. When thinking of the boxes representing all the syntactically enclosing MainParts as a "spine" we may describe the global search as a series of local searches, each one starting from the next more global MainPart in the spine. In the special case where there is no inheritance the global search is reduced to a simple, lexical scoping lookup mechanism.

All the mixins are contextually located inside their specific enclosing objects, so for instance the `employee` pattern which is looked up from our `meeting` context will be *that* `employee` pattern which is contextually located inside *the* `company` where the meeting also (indirectly) belongs. So the `meeting` and the `employee` are a natural pair, as opposed to a combination of a meeting in one `company` and an `employee` from another company.

It is exactly this kind of automatic, statically checked, multi-level object relation consistency support which is the core functionality of general block structure—it allows us to safely and conveniently work with groups of objects and patterns that naturally belong together, and the grouping mechanism offers both great flexibility by being nestable to any desired depth, and comprehensibility because of the deep and life-long experience that human beings have in exploiting contextuality.

As mentioned in the previous section, this kind of consistency support is also a restriction which may be too rigid in some cases—and, as mentioned, there is always the option of using explicit associations, e.g., by manually maintaining an explicit `myProject` reference in each `meeting` object. Such trade-offs between convenient safety and more verbose flexibility must be made all the time in the construction of programs; it is basically the same kind of trade-off as between a `while` statement which allows only a very regular set of control flows, and a `goto` statement, which allows you to jump to any location in a program. History seems to support the assumption that the choice of a rigid but analyzable construct instead of a more "powerful" and flexible construct often makes sense, perhaps because understandability is the more precious resource in the development of complex systems. Again, similar to the case with `goto` vs. `while`, since a flat set of global classes is just a special case of general block structure, it is trivial to see that no expressive power is lost by having support for general block structure in a language.

5.4 The Relation to Modules

General block structure shares a few features with modularization, so it is useful to describe the differences between them. If one of them turned out to be the more powerful mechanism, capable of solving all the tasks assigned to the other, then we had better use that one and get rid of the other mechanism altogether. However, we think that general block structure and modularization are largely orthogonal mechanisms, hence it is actually useful to have both.

Similarities: First a few reasons why they seem to overlap: If a given entity E (such as a class) is used as part of the solution S to a complex problem, but E does not play any rôle in the intended use of S (possibly available in a manifest form as a specification of S or as an interface to S), then we can do two different things to reduce the overall system complexity: We may provide the functionality of S as a module and E may be defined as private in that module, such that nobody needs to worry about what E is or how it is used except for those who implement or maintain that module. Alternatively, we may define E in context of a class that implements a solution to S; we assume that this class is global, to make the alternatives as comparable as possible.

In both cases, E is removed from the global name space, so programmers will not be bothered with E unless they take a look inside S. Similarly, when somebody does need to look at E, it will be clear already from the location of E in the source code that E is supposed to be understood in context of S. Finally, with multiple entities similar to S, $S_1 \ldots S_n$, and many entities similar to E, it will be convenient to compose large, complex systems using a subset of $S_1 \ldots S_n$, because each S_i along with the E-like entities needed by S_i can be provided as a named package (module or class) which can be manipulated as a whole; that amounts to better support for reuse of the S_i's. One thing we have left out of this picture is the need for shared resources—modules will need an import mechanism in order to be useful, and with the block structure approach there would generally be dependencies between the S_i classes such that the use of one of them would imply the use of a number of others. However, that can be taken into account without changing the conclusions above.

It seems that both mechanisms support complexity reduction and reusability in large systems by grouping and containment of entities that do not have to be available for most of the system, thus allowing the remaining, generally useful entities to stand out all the more clearly. However, that description is deceptive for several reasons.

Dissimilarities: Firstly, the nesting of an entity inside another does *not* make it inaccessible for outside entities. As an example of why it would be a bad idea to introduce restrictions that would make nested entities inaccessible from the outside, consider the `company` example from the previous section. Nested entities like `project` and `meeting` are perfectly valid concepts in connection with companies, they are not just implementation details that should be hidden from public view. In some languages, e.g. Smalltalk, there is a rule saying that methods are public, but instance variables are private. This ensures that access from the outside will always be mediated by a computation, as opposed to a simple variable access, and that again ensures that the implementation can be changed more freely without affecting code that uses the class. Now, this argument only makes sense if the access to a method and to an instance variable from the outside look *different*, otherwise the instance variable could simply be changed to one or two methods if needed, and all usage points would remain unchanged. Self, CLOS, Dylan, Cecil, and other languages transform access to variables (data slots) into method invocations by means of accessor methods, and BETA and gbeta support a slightly different kind of transparency by means of coercions. In any case, the distinction between public computation and private state seems to be an artifact of a too meager transparency support. Other languages, including Simula, Java, C++, and Eiffel, use a separate mechanism for access control management, namely explicit declarations of attributes as being 'private', 'public', or accessible from specific entities (e.g. export declarations in Eiffel and 'friend' classes or functions in C++). Since this in all cases amounts to an orthogonal mechanism, independent of the block structure, it actually supports the claim that block structure and privacy management are separate issues.

Secondly, the tasks of name space partitioning and visibility management which are associated with module systems are static in nature. They are concerned with proporties of source code, not with properties of run-time entities. In contrast, general block structure is inseparable from run-time entities, it is concerned with the ability of each nested run-time entity to depend on all its enclosing entities. By enabling this it also supports the grouping of run-time entities nested at some level under a common enclosing entity, for instance the grouping of `meeting`s and `employee`s together if and only if they "belong" to the same `company`. From the point of view of a module system there would just be `meeting`s and `employee`s, and an individual relation between a specific `meeting` and a specific `employee` can not be expressed. Conversely, general block structure does not support the separation of individually reusable entities, because even "global" entities are nested inside some outermost "universe" entity. The ability to compose a system from several smaller units is essentially a module-related capability—and the above claim that block structure could be used to support separately reusable packages silently assumed that such separate packages could even be expressed and composed; that would require some form of module system, thereby invalidating the argument that general block structure could support the reuse on its own.

Finally, modules serve well as a means for physical organization of code, for example to separate interface and implementation, or to allow for the combination of a given interface with any of several possible implementations, such as one for each of a number of different hardware and/or software platforms. This again enables separate compilation, and it allows for fine-grained source code control [74, Ch. 17]. General block structure does not support the grouping of source code entities according to such concerns as separation of interface and implementation. For example, an entity in an enclosing scope may be an implementation detail that a given nested entity does not need to depend on—e.g. if the `company` had some attribute which was used in the implementation of `company` itself but not needed by `project` or `meeting`. As another example, an entity may not need or use the context, but it may still be an implementation detail for a nested entity—e.g. if a specialized data structure were used in the implementation of `meeting`, but the data structure did not depend on `company` or `project`. Sometimes it is impossible to use the block structure to hide things appropriately, sometimes it is just wrong, because it introduces useless contextual dependencies. Since contextuality allows us to make entities that may be *considered as a group* more comprehensible, it is confusing and damaging for the usability of an entity if it is nested inside another entity and the enclosing and nested entities make no sense as a group. Hence, block structure should not be used for physical organization of code.

So, to summarize, modules are used to control visibility and/or accessibility for static entities, i.e., for pieces of source code; and for packaging related pieces of source code into conveniently reusable units; and for separating different kinds of pieces of source code independently of the semantic properties such as nesting location, e.g. for separating interface from implementation. General block structure is used to support contextual dependencies between run-time entities. Neither mechanism is able to handle the tasks assigned to the other.

Other points of view: Earlier treatments of related topics do not consider contextuality, which is the main point in our argument for keeping classes and modules separate, but otherwise the argumentation is similar.

In the classic paper [91], modularization of programs is for the first time introduced as a concept and a concern in its own right, and the main criterion given for modularization is that each module should encapsulate a design decision by providing its services to other modules in a form which is useful for the solution of the problem at hand, yet does not have to change if and when another choice is taken with respect to that design decision. Today the phrase 'representation' or 'implementation' seems to cover the term 'design decision' as it is used in [91]. The claimed results are that the system as a whole tolerates many changes inside modules without forcing changes to other modules, the system can be developed in parallel as soon as the interfaces have been chosen, and the system as a whole becomes easier to understand. The results in this paper are so well established today that they seem obvious.

In the Eiffel community, the position is that classes and modules should be unified, such that there is only one structuring construct in the language [79]. This unification is made into a principle, required for 'pure' object-orientation, and the criterion is the same as the one we gave above: If one mechanism can handle all the tasks of another mechanism, then the first one should be used and the second one abandoned. However, the module and class mechanisms can only be unified in Eiffel because there is no support for contextuality except for the nesting of method invocations inside objects. In particular this means that classes can be entirely static entities, and they are all naturally located in one, global name space. We just need to require that each module must consist of exactly one class definition, and then classes and modules are unified!

This does have some confusing consequences, though. For instance, there are standard Eiffel classes such as MATH and BASIC_IO containing sets of procedures and functions for doing trigonometric computations and for receiving keyboard input and writing text to a console. It is necessary to inherit from MATH in order to compute the cosine of an angle, and it is necessary to inherit from BASIC_IO in order to receive keyboard input; this inheritance relation does not make sense as a specialization, and the MATH and BASIC_IO classes themselves do not make sense as generators of contexts for the procedures inside them. On the other hand, it makes good sense to consider MATH and BASIC_IO as importable modules containing global procedures and functions.

Since we are generally in favor of unification of concepts, it is worth considering whether a both-module-and-class concept is a good idea. There are some serious problems, however. Firstly, it is only possible when classes are static entities, so it cannot be applied to languages with general block structure. Secondly, there is no support for physical separation of the interface and the implementation of a class, so even strictly implementation related changes to a class will cause recompilations, new versions of files, etc. It may be possible to compensate somewhat for these problems by using a "smart" compiler and linker and version control system etc., but it seems unnecessary to introduce those problems in the first place. Finally, the functional granularity of the system may not be at the class level—if a group of classes is only meaningful taken together (say, NODE and EDGE which can be used together to create graphs), then it seems counter-productive to require that this group must be handled as a multitude of separate modules.

In [106], the need for modules as a separate construct in addition to classes is treated in detail, and the main reasons given in favor of having both classes and modules in a language are as follows: the import and the inheritance relation should not be confused; groups of classes may need to collaborate in order to maintain invariants; selective export (as in Eiffel or as `friend` in C++) cause hard-to-understand networks of visibility; and modules allow both separate compilation per module and gives good opportunities for optimizations inside a module, since many optimizations are concerned with interactions between tightly cooperating classes. Apart from the fact that this does not cover contextuality, we support this argumentation, and again the conclusion is that it is appropriate to have both classes and modules.

Chapter 6

Propagation of Specialization

This chapter shares material with our paper Propagating Class and Method Combination*, which was accepted for publication and presentation at the ECOOP'99 conference.*

In recent years the management of concerns involving multiple classes and the combination of structure and behavior from separate entities has been a very active area of research. Subject orientation [52], aspect oriented programming [57], and object collaborations [81] are all examples of such efforts. The support for general pattern merging and the semantics of virtuals in gbeta provides a *language integrated* approach to the achievement of these goals. A seamless integration into a statically typed general purpose programming language such as gbeta opens the possibility for type checking at the level of the multi-class constructs, separate type-checking and compilation, and avoidance of the "impedance" problems associated with the use of several different mind-sets, languages, and tools.

The general block structure enables a natural expression of groups of mutually dependent patterns. The very flexible inheritance and pattern combination mechanism interacts with the block structure to support propagation of pattern combinations. The reason why we use the word 'propagation' to describe this phenomenon is that it allows programmers to initiate a complex but regular process by specifying a syntactically simple pattern merging operation, for example by an expression like `a & b`, and as a result of the semantics of virtual attributes (see Chap. 4 for details), this combination of `a` and `b` can propagate to cause the combination of some virtual attributes in `a` and `b`, and possibly also propagate further into virtual attributes nested inside those virtuals etc., and finally it propagates the enrichment of all those virtuals into all the patterns that inherit from them. In other words, one syntactically explicit combination operation may cause many other combination and specialization operations on dependent patterns, where the dependency relations are either 'is-a-virtual-attribute-of' or 'inherits-from'.

This description of propagation as something that moves along the edges of a graph of dependency links (some caused by simple syntactic nesting, some established in static analysis) illuminates how similar it is to a constraint solving process. Constraints are introduced by declarations of virtual attributes—where the constraint is on the form `a` $\leq$ `b`—and by inheritance—where the constraint is often on the form `a` = `b&[(# ... #)]`. Other forms of constraints are also available, for example lower bounds on virtuals, which are presented in Sect. 9.2.

Note that this constraint solving process may happen at run-time or at compile-time. There is full support for performing the process at run-time, as described in Chap. 7, but a warning will be issued for each location in a program where this constraint solving process cannot be analyzed fully at compile-time; that is the case, for instance, when two variable patterns are merged.

This chapter gives a survey of significant usages of the propagating combination mechanism, thus illustrating the semantics and motivating its usefulness.

6.1 Combination of Classes, then Methods

The first example illustrates the use of propagation in only one level; this special case works similarly to CLOS method combination using `before` and `after` methods, thus illustrating the gbeta pattern combination mechanism by showing how it achieves a known goal. Explained in terms of propagation, this is about combination of two classes and—by propagation—combination of the methods inside those classes.

Consider an abstract pattern `Stack` which specifies a stack data structure, along with a specialization `StackImpl` which contributes an implementation of the stack using a list (whose type constraint on contained objects (`element`) is specified to be the same as the constraint given for the enclosing `StackImpl`):

```
Stack:
 1(# element:< object;
     init:< 2(# do INNER #);
     push:< 3(# elm:^element enter elm[] do INNER #);
     pop:< 4(# elm:^element do INNER exit elm[] #)
  #);

StackImpl: Stack
 5(# init::< 6(# do storage.init #);
     push::< 7(# do elm[]->storage.insert #);
     pop::< 8(# exit storage.deleteFirst #);
     storage: @list 9(# element::this(StackImpl).element #)
  #)
```

Ex. 6-1

The `StackImpl` pattern can be used directly, but it does not protect itself from shared access in a multi-threaded context. To add concurrency control we write another specialization of `Stack`:

```
 3 (# elm: ^element enter elm[] do INNER #) (* from Stack *)

12 (# do mutex.P; INNER; mutex.V #)      (* from StackConc *)

 7 (# do elm[]->storage.insert #)       (* from StackImpl *)

   (# elm: ^element                    (* combined result *)
   enter elm[]
   do mutex.P;
      elm[]->storage.insert;
      mutex.V
   #)
```

Figure 6.1: The contributions to push in aThreadSafeStack

```
StackConc: Stack
 10(# init::< 11(# do mutex.init; INNER #);
      push::< protect;
      pop::< protect;
      protect: 12(# do mutex.P; INNER; mutex.V #);
      mutex: @semaphore
   #)
```
Ex. 6-2

In StackConc, the methods push and pop are further-bound to include the pattern named protect.

Combination of the two aspects of the stack, the concurrency control and the implementation, implies a combination of the shared virtuals including the methods push and pop, so all we need to do to obtain a thread safe stack is this:

```
   aThreadSafeStack: @ StackConc & StackImpl;
```
Ex. 6-3

In Fig. 6.1, the contributing mixins in an invocation of push on aThreadSafeStack are listed. The effective, combined push in aThreadSafeStack is given as the 'combined result' in Fig. 6.1. This result is obtained by recursively replacing INNER with the next more special contribution, and that produces a do-part which behaves similarly to push. As desired, it protects the implementation part in a critical region by inserting it between the acquisition and relinquishment of the semaphore. Consequently, only one thread at a time can execute push (or pop) on this particular stack.

To detail how that particular sequence of contributing implementations of push was computed we must consider the combinations of the enclosing patterns. For this chapter it may be assumed that a mixin is just a MainPart, i.e. the syntactic construct (# ... #); a more precise description of mixins is given in Sect. 3.3.

Stack is a pattern which only includes one mixin, which is numbered 1. Both StackImpl and StackConc include two mixins—[5,1] and [10,1], respectively.

The combination of two patterns is a linearization applied to the corresponding two sequences of mixins. This linearization is specified formally in Sect. 3.7.1 so here we just give the result:

$$[10,1]\&[5,1] \;=\; [5,10,1]$$

The mixin sequence for `push` in $[5,10,1]$ is the linearization of the contributions in mixin 1, then 10, then 5:

$$[3]\&[12]\&[7] \;=\; [7,12,3]$$

This makes the contribution to `push` in `StackImpl` the most specific (i.e., the frontmost element in the sequence of mixins), etc., and as we can see, the resulting pattern will execute the same imperatives as illustrated with the 'combined result' in Fig. 6.1.

Note that we could have combined the concurrency control with any implementation, and the implementation could have been combined with one or more auxiliary aspects such as an implementation of concurrency control. The same kind of method combination could have been obtained in CLOS (not typesafe, though) by putting statements before `INNER` into a `before` method and statements after `INNER` into an `after` method, and inheriting from both `StackConc` and `StackImpl` (in that order), yielding a class `ThreadSafeStack`; `aThreadSafeStack` would then be an instance of this class. The reason is that the special case of propagation in one level from a pattern used as a class and into nested virtuals used as methods amounts to a mechanism which is similar to standard method combination in CLOS.

6.2 Combination of Aspects

In this section we consider an example where the merging operation ('`&`') is applied to families of patterns. Each family is realized by having a pattern that represents the family and a number of nested virtuals used as classes, one for each member of the family. The family as a whole is used as a method in this case. Merging of such families creates a combined family, and by propagation merges the contributions to each family member from all contributing families.

For the concrete example we need a few auxiliary patterns, supporting basic financial transactions and transfer of possession:

```
Person: $^{1}$(# name: @string #);
Payer: Person$^{2}$(# pay: $^{3}$(# amnt: @integer .. exit amnt #)#);
Paid: Person$^{4}$(# accept: $^{5}$(# amnt: @integer enter amnt .. #)#);
Receiver: Person$^{6}$(# receive: $^{7}$(# t: ^Thing enter t[] .. #)#);
Deliverer: Person$^{8}$(# deliver: $^{9}$(# t: ^Thing .. exit t[] #)#)
```
Ex. 6-4

A `Person` has a `name`; a `Payer` can `pay` an amount of money, and a `Paid` person can `accept` payments. Moreover, a `Receiver` can `receive` a `Thing` and a `Deliverer` can deliver a `Thing`.

For these patterns it is evident that collaborations may arise. An example could be the activity "to pay":

```
collaboration:
  (# First:< Person;
     Second:< Person;
     fst: ^First;
     snd: ^Second;
  enter (fst[],snd[])
  do INNER
  #);
pay: collaboration
  (# First::< Payer;
     Second::< Paid;
     price:< (# value: @integer do INNER exit value #)
  do price->fst.pay->snd.accept;
     INNER
  #)
```

Ex. 6-5

The pattern `collaboration` introduces two roles, played by `fst` and `snd`, and specified by `First` and `Second`. It is quite common that families of patterns can be described in terms of roles, especially when the pattern family members are used to specify one (possibly variable) object attribute each. Note that this kind of method to some extent support the notion of *activities* which is presented in [61]. ◇

The `pay` method specializes the `collaboration` method by further-binding the role patterns `First` and `Second`, and by adding one statement to the behavior in which the computed `price` is paid by `fst` to `snd`.

We can create a similar activity for a transfer of possession of some item, where the `snd` role player delivers a `Thing` which is then `received` by the `fst` role player:

```
deliver: collaboration
  (# First::< Receiver;
     Second::< Deliverer;
  do snd.deliver->fst.receive;
     INNER
  #)
```

Ex. 6-6

With these activities in place we can create a combination which supports the combination of the activities: both transferring an amount of money and transferring an entity in exchange for the money:

```
(# doTrade: pay & deliver;
   Diamond: @Thing;
   Walrus: @ Paid & Receiver & Deliverer;
   Lucy: @ Payer & Receiver
do
   Diamond[]->Walrus.receive;
   (Lucy[],Walrus[])->doTrade
#)
```

Ex. 6-7

In this piece of code we create the combined method `doTrade`, thus by propagation merging the nested virtual patterns `First` and `Second` and the behavior such that both transfers will occur. Moreover, we declare an object `Diamond`

```
ObserverDesignPattern:
 (# Subject:<
      (# attach: (# enter observers.insert #);
         detach: (# enter observers.delete #);
         notify: observers.scan
           (# do this(Subject)[]->current.update #);
         observers: @set(# element::< Observer #)
      #);
    Observer:<
      (# update:< (# S: ^Subject enter S[] do INNER #)#)
 #)
```

Figure 6.2: Support for the 'observer' design pattern

that can be transferred, and two role players, `Walrus` and `Lucy`, whose patterns have the necessary mixins. Since the `Walrus` must first `receive` the `Diamond` in order to be able to `deliver` it to `Lucy`, there is both a `Deliverer` and a `Receiver` aspect of `Walrus`. `Lucy` could have been a `Deliverer`, too, but she probably won't.

Note that this use of the propagating combination mechanism depends on the tight integration with the type system: We are creating a method whose arguments have types that we obtain by combining the types of the arguments of the method aspects that we combined. Such a type merging capability is not supported by combination mechanisms like those of AspectJ [57] or subject oriented programming [52, 90]. These approaches are otherwise able to combine parts of methods and classes from separately specified aspects/subjects in very flexible ways, but they have no notion of "white-box" combinations, such as combinations of types or interfaces or signatures of entities, only of "black-box" combinations, such as combinations of implementations of methods.

6.3 Mutual Recursion

The last example seems to be almost compulsory in conference articles about advanced languages and type systems recently [13, 65, 110], but in this case we emphasize that it is possible to distribute the implementation over several levels of specialization, in order to deal with various concerns as "soon" as possible—that is, at the most general level where the necessary information is available.

In Fig. 6.2 there is a specification of a pattern `ObserverDesignPattern` which can be used to support the observer design pattern. It contains two nested, mutually recursive patterns `Subject` and `Observer`. An instance of `Observer` may `attach` to an instance of `Subject`. Once inserted into the `set` of `observers` for that `Subject` it will be a target for notifications: each (significant) change in the state of the `Subject` should be followed by an invocation of `notify` (it is a programmer responsibility to remember to invoke `notify` at

```
WindowAndTextODP: ObserverDesignPattern
  (# Subject::< TextBuffer
       (# (* ensure that 'notify' is called after changes *)
          setFileName::< (# do INNER; notify #)
       #);
     Observer::< Window
       (# update::< (# do S[]->getState; refresh #);
          getState:< (# S: ^Subject enter S[] do INNER #)
       #)
  #)
```

Figure 6.3: A specialization, letting `Windows` observe `TextBuffers`

the right places). The `notify` method is a specialization of the `scan` method on the `observers`, and the effect is to visit each of the attached observers and invoke `update` on it. The `Observer` may then update its own state according to the changes in the `Subject`.

To use this we need a couple of "application domain" patterns, for instance a `TextBuffer` to be observed by a `Window`, which could be a `ColorIcon`:

```
TextBuffer:
  (# name: @string;
     setFileName:< (# n: @string enter n .. #);
     getFileName:< (# n: @string .. exit n #)
  #);
Window: (# refresh: (# .. #)#);
ColorIcon: Window(# setIconTitle: (# s: @string enter s .. #)#)
```
Ex. 6-8

Now we can create a specialization of the `ObserverDesignPattern` which lets `Windows` observe a `TextBuffer`, as shown in Fig. 6.3. Note that we have the potential for propagation here: There could be several different specializations of `ObserverDesignPattern` which would contribute a separate aspect each; for example, we could have expressed the `WindowAndTextODP` pattern as a combination of a pattern `TextSubjectODP` (which would only further-bind `Subject`), and a pattern `WindowObserverODP` (which would only further-bind `Observer`).

With a pattern like `ObserverDesignPattern` the propagation would proceed in two levels, from the outermost family of class patterns, over the intermediate nested virtuals which serve as class family members, and finally to the virtuals which are nested inside those family members. However, the mechanics are the same as in the previous examples, so we will not present the details of such a two-level combination operation.

Instead, we will concentrate on the potential for performing some tasks at this intermediate level of specialization, such that all subpatterns will be relieved of these tasks. When an `Observer` learns that the `Subject` has changed (i.e., when `notify` invokes `current.update` with that `Observer` as the argument) then we can get the state and `refresh` the `Window`. We do not yet know *how* to get the state, but that's a virtual method so we can put it in later.

Finally we can create an instance of the design pattern, `myODP`, and populate it with a subject `myBuffer` and an observer `myIcon`:

```
myODP: @WindowAndTextODP
  (# myBuffer: @Subject;
     myIcon: @ (& ColorIcon & Observer &)
       (# getState::(# do S.getFileName->setIconTitle #)#)
  #)
```
Ex. 6-9

The pattern of `myIcon` has two super-patterns, `ColorIcon` and `Observer`. The first would be a standard GUI support pattern, and the second contributes the design pattern related aspect. The newly added mixin provides the implementation of `getState`—now that we have the information about how to implement it. This implementation uses the type knowledge that

- `S` is less-equal than a `TextBuffer`, because `myODP` is a `WindowAndTextODP` which declares `Subject::< TextBuffer...` Hence, it must have a method `getFileName` which takes no arguments and delivers a `string` value
- `myIcon` is a `ColorIcon`, so it has a `setIconTitle` method which accepts a `string` value as argument

This could not be type checked if `S` in the body of `getState` had only had the type declared in the original `ObserverDesignPattern`. However, in both BETA and gbeta, the virtual pattern attributes *are* recognized by the type system as denoting a more specialized pattern when looked up in context of a more specialized enclosing object or enclosing method invocation.

Let us consider how this problem can be handled in other languages. When dealing with a *single* class, simple bounded polymorphism can handle this kind of changing types: The entity whose type should change can be a type parameter, and different instantiations will see the entity with different types. However, bounded polymorphism cannot handle the case where more than one class form a group of mutually recursive classes that should be specialized as a group. In approaches based on F-bounded polymorphism [64, 11] it is possible to establish recursive relations between the members of a type family, so it is possible to create a construct which is somewhat similar to `ObserverDesignPattern`, see for instance [64]. Note that all the relations between the classes in the family must be redeclared in every specialization of the family; [64] suggests some syntactic sugar which can be used to avoid most of these repetitions.

However, the possibility of implementing some of the functionality of the class family, including the possibility to have an attribute such as `observers`, depends on the fact that the different specialization levels of the class family (`ObserverDesignPattern`, `WindowAndTextODP`, and the anonymous pattern of `myODP`) are full-fledged patterns, not just types. In the approaches based on F-bounded polymorphism, this is not possible.

The problem is that the different instantiations of the type family consists of types that are *not* related by subtyping; this corresponds to having a type for

`Subject` in `ObserverDesignPattern` and having another type for `Subject` in `WindowAndTextODP`, but no subtyping relation between them. The reason why there is no subtyping relation between them is that such a relation would make the type systems unsound. That is again because those systems do not have existential types. Consequently, the incremental specification of the implementation of classes with those types cannot be expressed as an inheritance hierarchy in parallel with the subtyping hierarchy—there *is* no subtyping hierarchy between the different versions of `Subject` to follow in parallel.

Chapter 7

Dynamic Features

This chapter shares material with our paper Dynamic Inheritance in a Statically Typed Language*, which was accepted for publication in the Nordic Journal of Computing.*

Actually, the topic of this chapter is in some sense a non-issue. A static feature of a programming language that plays a role in the actual behavior of programs is just a way to perform certain tasks in the execution of programs at an early point in time, compile-time, and those tasks may of course also be performed when their outcome is needed, at run-time. For example, a C compiler may use symbol tables and knowledge about the size and alignment properties of the fields in a `struct` to compile the access to such a field down to the addition of a fixed offset to the address of the `struct` as a whole, and if the information that was used to compute that offset is not thrown away, then the computation may just as well happen when the field is being accessed at run-time.

However, making the computer behave in a particular way is not always the only purpose of a program. We may also want to apply various kinds of theorem provers to the program, such as type checkers, in order to improve the likelihood that the program actually specifies a behavior that is similar to what we intended. Of course, this 'intention' is informal by nature. Moreover, non-trivial questions about the behavior of programs written in non-trivial programming languages tend to be undecidable.

So it may seem like an impossible task to prove that any given program has any specific dynamic properties, even though we know type systems do just that. The underlying notion which has been very successful in attacking this problem is that of formalizable invariants. Invariants allow us to scale up from local to global considerations—a statement X will hold at all times in all executions of a program if every part of the program complies with X. Statements which are not invariants are not so easy to scale up, so the reliance on invariants seems to be crucial if we want to analyze programs. 'Dynamic' and 'invariant' are incompatible concepts, so there may be an issue after all.

In Sect. 7.1, the notions of invariants and promises are used to unfold the meaning of our concept of dynamic features, and to describe how they interact with the static analysis. Section 7.2 presents the concrete mechanism of dynamic pattern merging and presents some ways to use it. Finally, Sect. 7.3 introduces the notion of dynamic specialization of objects, explains why we need it, and gives usage examples.

7.1 Invariants and Dynamic Features

◇ An *invariant* is a statement that is true in all cases within a certain universe of ◇ discourse. In this context we are especially interested in *entity invariants*, which allow us to think of program executions in terms of more complex and useful ◇ semantic entities than individual memory cells; and *safety invariants*, which allow us to trust that certain operations at run-time will never fail.

For example, it may be stated as an invariant that the memory cells 125432–125435 for the duration of an execution of a given C program will only be accessed as an `int` variable. The invariant is the statement "if the current machine code operation accesses any memory cell in the area 125432–125435 then it reads/writes all four cells as a unit". That is all we need to ensure that this particular area of memory can be interpreted as holding an integer value and is being used according to an integer protocol. For a `float` there might also be a few exceptional bit-patterns that must be avoided because they are not representations of floating point values. C and many other languages provide loopholes (such as type casts and unions) that allow programmers to explicitly override invariants, but they are generally treated as an anomaly that must be used with great care. The goal of dealing with semantic entities at a higher level than raw memory cells whenever possible is generally accepted.

The entity invariants mentioned sofar are local in the sense that they can be described in terms of memory cells that are reserved for the entity. The entity invariant for a pointer is global, so we have to mention the "universe" in order to specify it: Given a store which is organized into entities (and possibly some unused space), the entity invariant for a pointer states that it holds the address of an entity.

On top of these primitive entity invariants we can recursively build composite ones: For a `struct`, the entity invariant is the conjunction of the invariants of the fields.

With a traditional run-time system for C, as implied by the above description, there are many different kinds of entities, and each must be treated in a specific way which cannot be inferred from the contents of the raw memory that is reserved for the given entity. This means that the entity invariants can only be maintained at run-time by exercising very strict static control over the execution; basically, every entity usage in the program must be determined as a usage of one particular kind of entity. This is handled by the type system, and the type system propagates precise type knowledge along all potential dynamic usage connections (assuming ANSI C, in one file, and without casts). The invariants enforced by the type system include some that are directly necessary for the maintenance of entity invariants, but most of them are needed because *future* operations might violate some entity invariants if they were not there. Such preparations for the future is what we call safety invariants.

In contrast, a traditional bytecode interpreter for Smalltalk establishes a run-time system with more complex entity invariants, but with a much more homogeneous set of entity protocols [50, Chap. 21]. Basically, an entity is either an object which is an array of slots, or it is a slot which is a pointer to an object. Some objects are classes; each object has a class which is referred by a known slot; classes store methods which are needed for behavior; and a few classes like `smallInteger` receive special treatment. But the important issue is that even though there may be almost as many kinds of entities as in C, the treatment of entities in a Smalltalk program can be almost homogeneous. A message send to an object is a standardized operation, independent of the object. An instance variable lookup is different for each instance variable (they are stored in different locations in the object), but since that can only occur inside a method of the class whose instances have that variable, it is a problem that can be solved just by looking at that class and its superclasses. As a consequence, the entity invariants can be maintained for a Smalltalk program with static knowledge only about each class-with-supers in isolation.

Whereas the C environment forces the notion of entity invariants and safety invariants to be considered together, Smalltalk maintains entity invariants automatically and thereby makes it possible to discard the safety invariants entirely. This is the traditional trade-off, between the safe, fast, highly interconnected systems with rigid static analysis, and the much more flexible and radically modularized systems with more expensive run-time behavior. The notion of dynamic features is commonly associated with various consequences of this flexibility and independence, for instance the fact that sending the same message to two different objects may cause two different methods to be invoked, or the fact that an instance variable may refer to objects with different internal structure at different times.

However, even though Smalltalk objects are all alike as entities, programmers need to consider them different because they are supposed to handle different tasks, are therefore implemented differently, and will behave differently. In particular, since a message send may cause a failed method lookup and thence invoke the method `doesNotUnderstand:`, the need for safety invariants is not entirely removed. Breaking entity invariants, i.e., misinterpreting memory cells, is much worse than invoking `doesNotUnderstand:` which may actually be handled, but generally an invocation of `doesNotUnderstand:` indicates that the program has a defect. The next section takes a look at the connection between different invariant architechtures and the kind of support programmers can have to help them reduce the number of defects.

7.1.1 Invariant Architechtures

Different programming languages have different kinds of invariants, but they all have *some* kinds of invariants, both very statically predictable languages like traditional FORTRAN and Pascal, and very dynamically flexible languages like Self and Smalltalk. Self has a simple invariant architechture—basically the only invariants are that the result of an expression evaluation is an object, and that objects can receive messages that will be looked up using a certain algorithm.

The invariant architecture of FORTRAN is simple, too, but it is different in that it builds on semantic entities that are less capable (because they are close to the actual, physical entities such as memory cells that common computers support directly). On the other hand, the invariant architectures of languages like Beta, Ada, or C++ are very rich and they allow for user-defined extensions such that complicated systems of provably invariant properties of programs can be built and automatically verified.

Imagine that a given task needs to be solved by a computer, and imagine that a particular strategy can be applied to obtain a solution which can be expressed as a traditional FORTRAN program, a Self program, and a Beta program. The programs must in some sense "do the same thing" (we might require that the externally observable behavior for the three programs be indistinguishable, even though we most likely cannot verify that), and they must be "natural" programs for their implementation language, whatever that means. On basis of these assumptions, we expect the following outcome:

With FORTRAN all invariants about the program will be low-level, in the sense that they specify properties that make sense when viewed as statements about the discipline under which the computer hardware is used, and in the sense that these properties are either meaningless or at a very fine-grained level if they are interpreted as statements about the solution of the original problem. With Self the invariants will also be oriented towards the computer and not the problem. They are all entity invariants, so they will specify guarantees for "objectness" of all semantic entities which are manipulated by the program, and such entities may be arbitrarily complex and hence may be designed to be understood in context of arbitrary problem-specific considerations. We might say that invariants in Self are low-level *in context of* a computer which has much more powerful built-in entities than individual memory cells; that computer may then be simulated by a piece of software, running on a more modest piece of hardware. With Beta it is possible to build arbitrarily complex, user-defined systems of invariants, and that may be used to ensure user-defined properties that make sense when viewed in context of the problem being solved. The next section outlines the consequences of these differences for human beings.

7.1.2 Promises

In addition to the invariants, which are formal properties that programming languages can enforce automatically (insofar as the implementation is correct), there is also a belief system, which is established by each individual person who is working with a program, in cooperation with other people who are around and have some relevant knowledge. The belief system is used to build understanding, or models, of the dynamic behavior of a program, and must be complete in the sense that it has the total behavior of the program as its topic, whereas the system of invariants will only cover the special cases that happen to be expressible within the given programming language. The belief system is also very powerful and quite unreliable, because it is an aspect of human thinking.

The belief system contains aspects which are similar to invariants, let us call them *promises*, and the invariants are actually mainly useful because they ◇ can be perceived by human beings and converted to clear and simple promises. Now, with the three imaginary programs in FORTRAN, Self, and Beta, the belief systems have the following working conditions:

For the FORTRAN program, the available invariants establish a very complex and low-level set of promises, and in order to reason about the program in terms of the problem being solved it is necessary to build models of the program behavior which add a large amount of structure to make sense of all those details. It is crucial that the problem-level considerations do not have manifest representations in the program, so the complex structure must be maintained in the mind with little external support.

For the Self program, the available invariants are also far removed from the considerations which are relevant to the problem to be solved, but in this case the basic building blocks, the objects, can solve complex problems in an encapsulated way, so understanding can be established *incrementally*: To understand a `list` (sufficiently well for a given purpose), it is enough to internalize the regularity in its behavior—by looking at a formal specification, by reading the source code, by using a list several times, or by listening to somebody who knows about `lists`. Then it can be used meaningfully, and it can be implemented or debugged by understanding the pieces of its implementation in context of the understanding of the `list` as a whole. The incrementality lies in the fact that only *local* understanding needs to be established—when viewed from the outside, the `list` can be considered as a black box whose behavior is bounded by promises; and when viewed from the inside, the `list` can be understood as a construct that is built by composing a small number of other black boxes with similarly bounded behavior. In this manner, it is possible to understand a complex program by only considering a very small portion of it at any one instant. We may ignore all the *reasons* why said promises can be made, and only remember the promises themselves. Note, however, that only experience from usage and the choice of names is available for building those promises; there is no automatic check which in any way confirms that the entity behind such a name will have the properties signalled by the given name. Without automatically verifiable support the promises may be few and local, but they may still be complex, and for a complex entity there may be many increments of encapsulated complexity stacked on top of each other. That creates a long chain whose weakest link may not always be sufficiently strong.

The only extra facility which is added in context of a language like Beta is that it is possible to give the promises an automatically verifiable support by means of user-defined invariants. This puts special emphasis on those properties that happen to be expressible in the given language. Moreover, it is unavoidable that many programs must be rejected even though they do maintain all the stated invariants, because exact static analysis is undecidable and therefore a safe approximation is generally used. On the other hand, it provides guarantees for certain well-defined properties of the dynamics of programs, and it also documents for a programmer who is new to the code that other parts of the source code can and may rely on those properties, and therefore they should be respected and maintained.

Human reasoning is definitely more powerful than mechanical reasoning when the goal is to quickly obtain approximately correct judgments. Mechanical reasoning is good at handling tedious detail with absolute rigor, and "approximately correct" may be a serious bug when it comes to automatized processes such as computer program executions. This trade-off between the more flexible but less safe dynamic approach and the safer but less flexible static approach does not have an optimal solution, or at least not one that is valid for everybody, or for all purposes.

The gbeta design rests on the assumption that safety invariants ensured via static analysis *is* valuable—the more complex systems and the more people collaborating on it, the more valuable it is—and the dynamic features which are described in this chapter do not in any way open loopholes that make it possible to violate the invariants that gbeta otherwise maintains. There is no pattern or object which is under less strict scrutiny in a program which uses these features than in other programs; there is no way a gbeta program will let an attribute denote or refer to an entity which does not conform to the qualification; and there is no way a name lookup operation can fail at run-time (the `MessageNotUnderstood` error).

So why do we claim that gbeta indeed *has* any dynamic features worth mentioning, compared to other statically type-checked languages? The fact is, creation of new classes and methods at run-time and changing the class and structure of existing objects *are* actually dynamic features which are unusual in statically checkable languages, as well as in programming languages in general.

As the next section argues, the novelty of these dynamic capabilities may not only be a consequence of the fact that it is non-trivial to perform static analysis of programs that may use these features, it may also be connected with the history of technological development.

7.1.3 Performance and Tradition

The fact is that a computer program is a connecting link between human beings and computers. This link should be optimized for *two* radically different purposes, namely enabling human beings to understand and express useful designs, and instructing computers to exhibit certain rigorously defined behaviors. The priority has generally been given to the first of these two tasks in this thesis. That is because the historic development has progressed from a situation with expensive computers and cheaper human labor to the opposite situation, and this trend will probably continue. It is also all the more urgently required to support human beings as well as possible, with systems that are getting so complex that they cannot be managed at all without better support for the human side of the equation.

This does not mean that computer resources can be wasted without limit, just that the optimization of *both* should be based on the right trade-offs!

As a result of the historic conditions, the traditions of computer science and computer practice contain some deeply entrenched trade-offs that give priority to the economy in the use of computerized resources, thereby possibly losing some opportunities for serving human beings better, even if the trade-off is no longer reasonable. A factor which helps maintaining this unfortunate state of affairs is that the alternative trade-off may imply greater implementation complexity, for example in compilers. People are not aware of what they *could* have, since they did not personally make those trade-offs, so they just accept the well-known solutions which are actually good enough for many purposes.

The invariants play an important role in this context. Invariants can in a precise sense reduce the potential range of run-time behavior associated with pieces of source code. In other words, they can make it simpler to execute the program, and thereby enable an implementation which uses the given computer resources more economically. As an example, consider the C `struct` field access mentioned on page 131 near the beginning of this chapter. It consumes far fewer computer resources (time and space) to add a small integer constant to an address than a symbol table lookup followed by a computation of the offset. Since the offset can be used many times, it is not just a question of spending those computer resources *early*, it is also a question of spending them *once* instead of many times. In a way, we might say that it is not only the human being who is relieved of a lot of thinking because of the invariants, it is also the computer.

The entrenched trade-off which is buried in this approach is that entities which are equipped with a description at the source code level will, as far as possible, be "compiled down" to a level where all the consequences of the description have been spelled out into low-level operations, and the description itself is discarded before run-time. This has given rise to some reverse reasoning, where the ability to discard the description has been taken as a criterion, and the semantics of the language adjusted accordingly. An example is the design of the `dynamic_cast` facility in C++, which is only supported with classes that already must have a run-time representation of the class for other reasons, e.g. because they contain virtual member functions.

The opposite philosophy has been adopted in Self. Here, such a basic operation as name lookup is very costly in the general case, but dynamic compilation and maintenance of several versions of the compiled code for each method (with optimizations which are valid under different sets of assumptions) makes it possible to obtain impressive performance without reducing the generality of the language as such [22].

The approach taken in `gbeta` is similar at the outset, but the support for the sophisticated compilation techniques is as yet non-existent, so the performance of the current implementation is poor.

Chapter 11 gives more detailed information about the performance of gbeta as such, but we need to mention two "expensive decisions" here which were taken in the design of gbeta. One is to have a full-featured representation of patterns at run-time, and to support all the operations on patterns that can be specified by means of inheritance and pattern merging also at run-time. The other is to support a mechanism which dynamically modifies the structure of an already existing object, such that it becomes an instance of a more specialized pattern than the pattern it was an instance of before the operation, without affecting the identity of the object (it is the *same* object with a more elaborate structure). These two decisions provide the technical support needed for the dynamic features which are the topic of this chapter.

There are two reasons why these operations do not introduce loopholes in the static analysis. Firstly, the static analysis of gbeta must already be able to handle entities of which just as little is known as the patterns and objects which are the outcome of these operations, so when such an entity has been processed, there is no new issue with it at all. This is the good news.

Secondly, and this is the bad news, the operations which *produce* these new patterns and structure-modified objects at run-time may fail, because the completion of the operation would have violated some safety invariant. The examples where such operations fail are generally contrived, and there are programming conventions which can be used to ensure that the operations do not fail. This is similar to the fact that there can be created many ad-hoc rules and accompanying proofs which will ensure that a given division expression will never cause a 'Divide by zero' error at run-time, the simplest one being an `if`-imperative which only executes the division if the denominator is not zero. The transformation of a run-time error into the execution of an `else`-part does not really solve the problem, but it does allow for a more flexible response than a run-time error (which in gbeta will kill the thread that caused the problem, not the whole program).

Programmers have been able to handle run-time errors like 'Divide by zero' using ad-hoc methods, so this level of safety might actually be acceptable in practice. Nevertheless, it would be a significant step forward if programming conventions that ensure the success of the two dynamic operations in gbeta were formalized and made part of the static analysis.

7.2 Dynamic Patterns

Pattern merging can be performed at run-time, just as well as it can be performed at compile-time. The static analysis of dynamically created patterns happens on exactly the same conditions as the static analysis of any other denotation of a pattern for which only an upper bound is known, as is the case with virtual patterns in many contexts. The same considerations apply, such as covariance of variable objects with a qualification known only by upper bound.

Merging operations can only build a new list of mixins by combining lists of existing mixins. There is no way to create a new MainPart at run-time, short of running the static analyzer and the code generator on a new piece of source code (that is what happens when the implementation of gbeta is run in interactive mode and a 'do' command is executed). Note that this cannot be made safe using simple conventions: there are countless ways for compilation of new source code to fail, and the newly compiled entity can only be integrated into the running program in a type safe manner via some superpattern which is already known in the program [75].

Other than that, the available set of mixins is limited by the constraint that the MainPart M (and therefore all the declarations) of a mixin m must be present in the program, and m must be contextually located within a part object which is associated with the MainPart which lexically encloses M. Of course, we do not create an arbitrary mixin and then check whether it satisfies these criteria, but all patterns, in particular all patterns which are used for pattern computations, *will* only contain mixins that satisfy these criteria.

Since all accesses to attributes are resolved during static analysis and expressed using run-time paths, the attributes of an instance of a given dynamically created pattern can only be accessed using statically available knowledge. This knowledge is guaranteed to describe a subset of the object, i.e., that it is an instance of some superpattern of the actual pattern of the object, and that this is only an upper bound for the real pattern of the object. Hence, it is not possible to access all the attributes of an instance of a dynamically created pattern by means of *one* complete view, but there will generally be a set of (superpattern) views which, taken together, support the access to the whole object. On the other hand, if the computation enhances an abstract pattern with some implementation, it may not be desirable for client code to have access to any more than the abstract pattern interface anyway, so there is no reason to bother with multiple views.

Even though there are no new issues with type safety, there *is* a new potential for a run-time error—the merging operation may fail, and when merging patterns which are only known by upper bound (including all dynamically created patterns), this failure can not be ruled out during static analysis. There should be better support for checking the safety of a merging operation at run-time. Sect. 7.3.2 gives more detailed information on how and why pattern merging may fail, as part of the treatment of dynamic object specialization errors. The following sections contain examples of dynamic pattern computations used in practice.

7.2.1 Dynamic Merging

All the merging operations that can be performed statically can also be performed dynamically. An obvious application of this is to dynamically perform the composition of an interface (an abstract pattern P) and various implementation aspects (patterns which inherit from P and add an aspect to some virtuals and/or introduce some new attributes). The `Stack` example from Sect. 6.1 fits directly into this case.

The main benefit of this approach is probably that the number of patterns does not explode because of combinations. Approximately $n!$ patterns would have to be explicitly and tediously defined in order to provide all the possible combinations of a pattern P and n different aspects, but with dynamic pattern combination it would only be necessary to define the $n+1$ different combinable patterns. Note that such a disciplined use of dynamic patterns could trivially be proved safe by an ad-hoc method: Just create a test program that iterates through all the combinations once; if that does not cause a run-time error, then there will never be a run-time error in the creation of those dynamic patterns. The $n!$ patterns would probably be reduced by the necessity of following conventions such as "take zero or more aspects and merge them, then add one implementation", but even then there might be many combinations.

Another idiom which might be applicable in many places would be to dynamically add one mixin to a given method at the most general position. Such a mixin could "conditionalize" the given method:

```
intFunc: (# i,j: @integer enter i do INNER exit j #)
...
map3to4: intFunc(# do (if i=3 then 4->j else INNER if)#);
modifyAnIntFunc:
  (# arg: ##intFunc
  enter arg##
  exit map3to4 & arg ##
  #)
```

Ex. 7-1

The pattern `modifyAnIntFunc` is invoked with an argument `arg` which is a pattern that is less-equal than `intFunc`, i.e., a function from integers to integers. It returns another function which behaves just like `arg` except that it maps 3 to 4 instead of whatever 3 was mapped to by `arg`. A similar effect can be achieved in BETA using the following approach:

```
modifyAnIntFuncBETA:
  (# arg: ##intFunc;
  enter arg##
  exit intFunc(# do (if i=3 then 4->j else i->arg->j if)#)##
  #)
```

Ex. 7-2

The difference is that the returned pattern is no longer guaranteed to be less-equal than `arg`, and that means that `modifyAnIntFuncBETA` cannot be used to modify patterns which must be usable in some context as some given specialization of `intFunc`. Whatever useful properties `arg` might have beyond those of `intFunc` are lost, because the returned pattern is *not* a modification of `arg`, it is an entirely different pattern which happens to *use* the contextually available value of `arg` in its do-part.

Another usage of this put-something-on-top-of-it idiom is the dynamic addition of concurrency control. One well-known way to avoid deadlock in concurrent systems with guaranteed serialization of the usage of a group of resources $R_1 \ldots R_n$ is to enforce a certain order on the acquisition of access to each of the resources: if all clients acquire R_1 before they acquire R_2, etc., then there can

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

Figure 7.1: A user-defined control structure

never be a loop in which every client is waiting for the next client to release a resource, because such a loop would have to contain a link where a client with access to R_i is waiting for another client to release R_j for some $i > j$. That client would violate the ordering rule. Clients may release their access to resources in order when they are not needed any more.

Since the choice of ordering may greatly affect the overall performance of the system, it may be beneficial to change the ordering dynamically.[1] There may also be other disciplines than the ordering rule which may be more appropriate at times, for instance an optimistic approach when the load is not too high. All it takes is that all methods which need access to some of the resources $R_1 \ldots R_n$ must have added the concurrency control mixin du jour on top of it. The modified method would then be invoked with all the usual arguments and would return results as usual.

With this approach, a method, procedure, or function is not just something that may be invoked, it is instead something which may be passed around and modified, and it may then be executed zero or more times. That is probably a change in programming style for most programmers, but we believe that it is a path worth exploring.

7.2.2 Dynamic Control Structures

There is a special case of usage of dynamic patterns that deserves separate treatment. Technically it is simply the case where a single mixin is added at the *bottom* of a given pattern, i.e., as the most specific mixin. That is just like ordinary single inheritance, except that the superpattern is not a compile-time constant. This section is about that case.

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 any language, where the bodies are the `then`-part and the `else`-part; in this case the name space is empty (no 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.

[1]The transitions must be handled carefully, of course.

```
1  (# myFile: @file; (* an interface to a disk file *)
2
3     (* the iterator interface *)
4     iterator: (# theLine: ^text do INNER #);
5
6     (* concrete iterators *)
7     inputIterator: iterator
8       (# (* read lines from std. input until empty line *)
9       do (while (getline->theLine[]).length>0 do
10            INNER
11         while)
12      #);
13    fileIterator: iterator
14      (# (* iterate through the file, and make the
15          * current line available in 'theLine' *)
16      do (while (not myFile.eof) do
17            myFile.getline->theLine[];
18            INNER
19         while)
20      #);
21    filter: iterator
22      (# do (if theLine.length>0 then INNER if)#);
23
24    (* a method which takes an iterator as argument *)
25    LinePrinter:
26      (# anIter: ##iterator (* a pattern variable *)
27      enter anIter##
28      do (* iterate and print each text *)
29         anIter(# do theLine[]->putline #)
30      #)
31 do
32    inputIterator## -> LinePrinter;
33    'somename'->myFile.name; myFile.openread;
34    fileIterator&filter## -> LinePrinter
35 #)
```

Figure 7.2: `LinePrinter` is parameterized with a dynamic control structure, `anIter`

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. 7.1, showing the control structure `twice` which simply executes its body two times.

In gbeta it is possible to have dynamic control structures, using inheritance from a variable pattern. This makes it possible to parameterize a method with a control structure, delaying the decision about what control structure to use until run-time. Figure 7.2 is an example of this: the method `LinePrinter` (line 25) accepts the argument `anIter` which is some subpattern—not known before run-time—of `iterator` (line 4); `anIter` is then used as a superpattern in line 29.

In line 32, `LinePrinter` is executed with `inputIterator` as argument. This will simply echo what the user types, line by line, until an empty line is entered. Line 33 initializes `myFile`, and in line 34 `LinePrinter` is executed with another control structure as argument, namely the *merge* of `fileIterator` and `filter`. The `filter` control structure (line 21–22) executes its body iff `theLine` is non-empty. As a result, this invocation of `LinePrinter` will print the contents of `myFile`, line by line, but skipping all empty lines. We could also, e.g., compose with an iterator which visits each character of `theLine`; the composition would then visit each character of each line of the file.

Consider a similar example (with just one iterator, for brevity) in C++, as shown in Fig. 7.3. In this version, `anIter` (line 30) is an object, instance of a descendant of the class `iterator` (line 9); `anIter` cannot itself be a class since classes are not first class entities in C++, but this loss of generality does not affect this particular example.

The function call operator (line 11) is used to apply the iterator `anIter` to the callback `body` (line 32). The `fileIterator` implementation of the function call operator then provides itself as an argument to the given callback `cb` (line 20), such that the implementation of `cb` may use `theLine`. If we were to give `theLine` as an argument to `cb` directly, then this and all other iterators would have to be changed if a new member, e.g. `anotherLine`, were added to `iterator`. Hence, this indirect approach is needed to obtain a similar stability towards changes of the control structure name space as the `gbeta` version provides.

However, this is still an incomplete solution since iterators cannot be composed, like `fileIterator` and `filter` are composed in line 34 of Fig. 7.2. There does not seem to be a straightforward way to support iterator composition in C++. Note that composition of control structures is also not supported in the standard approach to iteration in C++ (especially in the Standard Template Library [88]). In this approach, an *iterator* object—specific for the collection being iterated over—is obtained and used in a `for`-statement, e.g.:

```
vector<int>::iterator i;
for (i = myVector.begin(); i != myVector.end(); i++)
  cout << *i << endl;
```
Ex. 7-3

This effectively standardizes a large group of control structures to be `for`-statements, but it does not support all control structures. E.g., a control structure which forks a separate thread to handle each step of the iteration could not be written in this style, because there is no way to write an iterator which changes the semantics of the actual control structure in use, namely the `for`-statement.

The Sather [101, 87] concept of *iters* has the same limitation. A Sather iter is a co-routine which may only be executed lexically nested within a `loop` statement, and it may `yield`, in which case the loop execution continues, or it may `quit`, which works similarly to `break` in C, terminating the `loop`. However, even though iters enable an elegant expression of many control structures, they cannot change the fact that the basic control structure is the built-in `loop`. Again, they do not support control structure composition.

```
#include <iostream.h>
#include <fstream.h>

ifstream myFile("somename");

class iterator;
typedef void (*callback)(iterator&);

class iterator {
public:
  virtual void operator() (callback cb) = 0;
  char theLine[1000];
};

class fileIterator: public iterator {
public:
  void operator () (callback cb) {
    while (myFile) {
      myFile.getline(theLine,999);
      (*cb)(*this);
    }
  }
};

void body(iterator &iter)
{
  cout << iter.theLine << endl;
}

void LinePrinter(iterator* anIter)
{
  (*anIter)(&body);
}

int main(int, char *[])
{
  LinePrinter(new fileIterator());
  return 0;
}
```

Figure 7.3: Using function pointers to simulate a dynamic control structure

Yet another approach is used in Smalltalk, Self, Cecil, Dylan, and Tycoon-II, where a control structure like an `if`-statement is not built-in but instead uses late binding of methods in the objects `true` and `false` to obtain the choice between the `then`-case and the `else`-case, and uses blocks/closures to defer the execution of the two cases. This approach does not build on one fixed control structure, so it transparently allows the same kind of expression to invoke essentially different control structures. However, the name spaces that the bodies are immersed in are provided via arguments to the closures, so they have to be typed in for every usage. Moreover, changes in the number or types of names in these environments will generally require changes to all usage locations. In

Self, blocks will silently discard extraneous arguments, so an argument may be added without having to change all usage locations—but if yet another argument is added and a usage point needs access to that third argument, then the ignored second argument will suddenly become visible anyway. All these name space considerations illustrate differences between explicitly parameterized entities and entities which gain access to the would-be parameters using an environment (in this case inheritance, but contextuality works similarly).

Since the actual control structure is chosen by means of the method selector (the "name of the message" being sent) and the receiver, there is no way one can have a control structure variable without having a message selector variable, and that again requires the use of such devices as, e.g., Dylan's `perform` primitive (which accepts a symbol and a list as arguments and invokes the function whose name is that symbol with those arguments). This is possible, but it will of course not be statically checkable. It would actually be possible to write a method M on a new object O that is just used to hold the two receivers; the method should take a block B as an argument and `perform` the first control structure on a block B_2 that `performs` the second control structure on the argument block B, using the two receivers that O holds. The only missing link is that the name space must be carried over by the intermediate block B_2, so B_2 must explicitly list that name space. Apart from the fact that every distinct name space must have its own version of M, and the fact that static type checking has been evaded (if it were available in the first place), this would actually be a way to establish support for the composition of two control structures.

7.3 Dynamic Specialization of Objects

Dynamic specialization of objects is an extraordinarily natural thing!

When human beings build mental models, for example listening to a story being told, the properties of the modeled phenomena are not all available for the listener at the introduction. After hearing "Once upon a time there was a King ... " we certainly expect to learn more about that king as we hear the rest of the story. So if we want to be able to support human beings in the understanding of programs by adapting to approaches that seem to be reasonable descriptions of ways humans think, then we had better consider this aspect. Let us call it *discovery*. Now, we might think that we can handle discovery quite nicely already with the facilities presented in earlier chapters. For example, if we learn that the king is called 'Bob', then there would be an obvious parallel as follows:

```
person: 1(# name: @string #);
king: person 2(# kingdom: ^country #);
...
hearStory:
 3(# theKing: ^king
  do &king[]->theKing;
     'Bob'->theKing.name;
     ...
  #)
```

Ex. 7-4

Since attributes are not auto-declared we have to add a declaration of `theKing`, but otherwise the mapping seems to work. The problem arises when the story continues "The King was an avid player of golf, and he'd always have problems with the Queen about the broken windows." The problem is that `theKing` does not have the required `demonstrateElegantSwing` method that avid golf players are expected to support. However, that is exactly the kind of problem that dynamic specialization of objects can solve, and it would look like this:

```
avidGolfer: person$^4$(# demonstrateElegantSwing: ... #);
...
hearStory:
 $^3$(# theKing: ^king
  do ...
     avidGolfer## -> theKing##;
  #)
```
Ex. 7-5

The effect of this assignment is that the object referred by `theKing` is modified such that it becomes an instance of `king&avidGolfer`, i.e., an instance of the merge of the previous pattern of the object and the new pattern `avidGolfer`. As a result, `theKing` will be enhanced with new part objects for whatever is missing in order to make him an instance of some subpattern of `avidGolfer`; the change in terms of concrete mixin lists would be from $[2,1]$ to $[4,2,1]$. Note that the addition of new part objects may also lead to a change in the value of virtual attributes; the object will be the same, but it will in every way be enhanced such that it is really an instance of the new, more special pattern.

We believe that it is both a very useful and a quite understandable facility to be able to create objects with a certain structure and then gradually specialize that structure by adding new aspects as they are discovered.

7.3.1 Change of Class in Various Languages

One invariant that most object-oriented languages maintain for objects is that they come into existence with a given structure (interface and implementation) and then do not change that structure ever after. For class based languages it may be phrased as the invariant that each object is an instance of a class, the class is immutable, the class determines the structure of its instances, and an object does not change its class. In gbeta, the last of these restrictions is lifted.

Some languages come in implementations that are complete in the sense that they include the tools needed for creating and modifying programs, for example Smalltalk and Self. This means that it is indeed possible to programmatically change a class in Smalltalk resp. modify the structure of an object in Self. However, this is considered "programming", and it is not commonly used as a technique in ordinary programs. For example, it is stated that the protocol which supports the programmatic editing of classes 'is not generally used by programmers, but it may be of interest to system developers' [50, page 283]. Moreover, after the editing of the class, there is still the question about what to do with existing instances—should they all change? should they remain as instances of the old version of the class? etc. The programming environments handle these issues, but the languages are not oriented towards the widespread use of programmatic object structure manipulations.

However, making changes directly to the structure of an object is not the only way to make it act differently. The rest of this section surveys a number of related mechanisms.

The language Self [114] supports a very general form of object behavior modification: In Self, an object may inherit from another object by having a so-called parent slot which refers to that other object. Parent slots may be assigned dynamically, and that works similarly to dynamically changing the inheritance graph, thereby potentially redefining, adding, or removing inherited methods and state. There is support for type inference in Self [3], 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. Moreover, these types are intended to enable optimizations by discovering some invariants that programs actually support. They are not intended to prove that declared invariants will be supported, e.g., there is no syntax by which such invariants can even be declared.

Smalltalk [50] is class-based, and every object is an instance of one particular class. However, the `become:` primitive swaps 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 [80], 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 possible 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.

Note that a complete object in the metaCombiner approach consists of a group of (traditional Smalltalk) objects, whereas an object in gbeta consists of a group of part objects. In both cases the object is not monolithic, and this enables greater flexibility. Furthermore, it would not be unreasonable to think of a Self object together with all objects reachable directly or indirectly through parent links as an "object", and that again would often be a multi-part entity.

The language Sina embeds the concept of *composition filters* [113, 5] 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 on dynamically evaluated conditions, and it is possible to simulate a standard inheritance mechanism, which may then select varying "parents" dynamically. However, Sina does not have a static type system.

An example of a very general kind of support for actually changing the structure of an existing object is the `change-class` function in CLOS [56]. When the class of an object is changed, a system defined generic function is called which uses various heuristics (such as the spelling of slot names) to determine whether to transfer the value of a slot from the old state of the object to the new one, or to initialize a slot as in a new object. This mechanism provides the programmer with the ultimate flexibility, but an unrestricted mechanism like this will of course potentially break any safety invariant that the run-time system might try to maintain, and is thus incompatible with any static analysis that attempts to guarantee that `MessageNotUnderstood` errors cannot occur.

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 [56] convention of using some classes for "mixin" inheritance has been developed [10, 25, 100] into a separate concept of mixin-based inheritance. For more information about mixins, please see Sect. 3.2. In [100, 68, 67] it is described how Agora 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 is not acceptable in practical software engineering.

In Cecil [19, 20, 21, 43], 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 annotates the code, a type check can be made. Predicate dispatching allows for an automated check of the disjointness and completeness properties, but only insofar as the boolean expressions can be considered as black boxes (so it cannot detect that, e.g., `x>0` and `x<=0` are disjoint and complete). However, programs are 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. In `gbeta` there is 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.

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 pattern, not generalized to an instance of a superpattern or to an unrelated pattern. 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 gbeta, because of the static name binding. Compared to Agora, gbeta does not require that each object foresees its entire potential for structure development statically. Compared to predicate based inheritance, gbeta does not support changing the structure of an object automatically, an explicit statement must be executed. But unlike Cecil, the structure enhancements in gbeta may certainly change an object in such a way that it supports interfaces that it did not support before the modification, and this is handled in a type safe manner.

The next section details the mechanism behind dynamic specialization of objects in gbeta.

7.3.2 The Dynamic Specialization Mechanism

The dynamic object specialization mechanism in gbeta supports the modification of an object at run-time such that it becomes an instance of a pattern that is less-equal than the pattern it was an instance of before the operation. This modification does not affect the identity of the object, and in particular all existing references to the object will now "see" the more specialized object. This may for instance become evident because the behavior exhibited by the object when executing some methods has changed; or it may enable an alternative in a `when` imperative (a typecase) that was not enabled before, such that it becomes possible to access the object through a richer view, for example in order to call methods that the object did not have earlier. More information on the `when` imperative is given in Sect. 9.1.

Consider an object O which is an instance of a pattern P. After being dynamically specialized with the pattern P', O will be an instance of the pattern $P\&P'$, and this may affect O in various ways:

A. If $P \leq P'$ then O is left unchanged: it was already an instance of P'; this will of course always succeed.

B. If P and P' share no mixins then all the part objects corresponding to the mixins of P' are added to O (in more specific positions than all the existing P part objects, because that is the way '`&`' works); there will be no interaction between the old and the new part objects except for a possible change in overall behavior due to added `do`-parts; the operation will always succeed.

C. If $P' < P$ then O will be enhanced with part objects corresponding to the mixins in P' which are missing in P; this operation will succeed, unless it causes error 3, which is explained below. Note that error 3 only occurs when there is a variable attribute which is not `NONE`, so it does not occur with new objects.

D. If P and P' share mixins but the shared mixins occur in the same order (this is the case if there is a pattern Q such that $P \leq Q$, $P' \leq Q$, and Q contains all the shared mixins) then part objects will be added to O for the missing mixins of P'; this operation will succeed, unless it causes error 3 or error 2, explained below.

E. In the general case, part objects will be added to O for the missing mixins of P', in the positions specified by the semantics of '&'; this may fail with error 3, 2, or 1.

As we can see, dynamic object specialization is dangerous in the general case, but we believe that the less dangerous cases form recognizable and useful classes of usage of this mechanism; for the remaining cases, the potential errors will have to be handled, and alternative actions must be taken when it is detected that a given object specialization has failed. The failures come in three varieties, listed in order of their occurrence during the execution of the dynamic specialization operation:

1. The merge $P \& P'$ may fail; if this happens then the requested object structure is inconsistent, i.e., there is a pair of mixins, i and j, that both P and P' contain, and P requires i to be more specific than j whereas P' requires the opposite. Since the requested object structure cannot be built, the operation raises a run-time error immediately. Note that this error can only occur if both P and P' have at least two mixins each; a single-mixin pattern (just one MainPart) is immune. Moreover, it can never happen for mixins i and j which are inherited from the same superpattern, directly or indirectly.

2. If the merge $P \& P'$ succeeds then the new object structure can be built. However, propagation may cause additional dynamic merge operations. If P and P' further-bind the same virtual attribute in two different ways, then this will cause a dynamic merge that may fail. Further propagation is also possible, for instance to virtuals nested inside a shared virtual in P and P'. Note that if P and P' do not have shared virtuals, or if at most one of them further-binds each shared virtual, then there will be no propagation and this error cannot occur.

3. If the merge $P \& P'$ and subsequent propagated merges succeed then it is possible that the qualification of a variable object or pattern attribute has been made more special, because a virtual pattern was used for the qualification and that virtual pattern has been further-bound during the operation; if the variable attribute already refers to an object or a pattern then the strengthened qualification may be too strict for the object, and in that case a run-time error is raised. Note that a virtual which is used as a method (not as a qualification) can never cause this error.

There is actually yet another run-time error which may occur when an object is about to be specialized dynamically, but it is orthogonal to the others and does not depend on the two involved patterns. It is instead associated with exact qualifications. There is an inherent conflict between exact qualifications and dynamic specialization. It is not possible to maintain the safety invariant of an exact reference and to dynamically specialize an object referred by that reference. As a result, a run-time error is raised if there is an attempt to specialize an object which is or has been referred by an exact reference. It would be possible to maintain a reference count and thus be able to remove the "is-exact" mark when the last exact reference to an object disappears, but the current implementation just marks objects and never removes the marks again. This probably means that it is necessary to divide the universe of patterns into two sub-universes: the patterns whose instances may be specialized dynamically, and the patterns whose instances may be referred by exact references. As long as these subuniverses are kept apart, there will be no conflict. It might even make sense to elevate this to a property of the pattern itself and thereby make it statically enforcable, but this has not been designed in detail nor implemented in gbeta.

Here is an illustration of how the other errors can arise. Since the first two errors are associated with the merging operation, we present them in context of a dynamic pattern merging operation. The same merging operation could of course have been provoked implicitly during an object specialization, but this seems to expose the cause more directly. The third error can only happen in context of dynamic object specialization. An error of type 1 arises if we merge two patterns dynamically which both contain two given mixins (here it is mixin 1 and 2), but in different order:

```
(# a: 1(# ... #); b: 2(# ... #); vp1,vp2: ##object
do a&b## -> vp1##;
   b&a## -> vp2##;
   vp1 & vp2 (* Error 1! *)
#)
```
Ex. 7-6

We use the variable patterns `vp1` and `vp2` to "hide" the actual patterns being merged, such that the compiler cannot detect the merging failure statically—it makes no attempt to do flow analysis, so `vp1` and `vp2` are only known as "some pattern which is less-equal than object" when the merging of them is analyzed. Since this is not known to be safe, a compile-time warning is issued, and the program actually fails with a run-time error during that merging operation. The next case, errors of type 2, is concerned with propagation. To provoke this error, we create two patterns `q1` and `q2` which both inherit the virtual `v` from the superpattern `p`, so they have a shared virtual and therefore propagation of the combination operation will occur:

```
(# a: 1(# ... #); b: 2(# ... #);
   p: 3(# v:< object #);
   q1: p 4(# v::< a&b #);
   q2: p 5(# v::< b&a #);
   vp1,vp2: ##object
do q1## -> vp1##;
   q2## -> vp2##;
   vp1 & vp2 (* Error 2! *)
#)
```
Ex. 7-7

When `vp1` and `vp2` are merged, a run-time error is raised. This is because of the same merging conflict as in the previous example, only this time it occurs in the merging of the contributions to the shared virtual `v`. Again we use the generic variable patterns `vp1` and `vp2` to make this a run-time error and not a compile-time error. Finally, an error of type 3 can be provoked as follows:

```
(# p: ¹(# v:< object; aV: ^v #);
   aP: ^p
do
   &aP[];
   &integer[]->aP.aV[];
   p ²(# v::< string #)## -> aP## (* Error 3! *)
#)
```
Ex. 7-8

We create an instance of `p`, make `aP` refer to it, and make its `aV` variable object attribute refer to a fresh `integer` object. The dynamic object specialization operation causes a warning, because it *may* hide the kind of problem that we are planning to create. Then we specialize `aP` such that the qualification of `aV` changes from `object` to `string`. While the `integer` object was perfectly acceptable for `aV` before the specialization (because `integer` $\leq$ `object`), it is a violation of a safety invariant after the specialization (because `integer` $\not\leq$ `string`). Hence, a run-time error is raised.

After having visited this tarpit of trouble, we should remember that there are systematic ways to maneuver around the problems. Let us summarize some rules which are sufficient to avoid run-time errors in dynamic object specializations:

- It is always safe to add unrelated mixins to an object (this is case B).
- It is always safe to specialize a *new* object with a subpattern of its current pattern (this is case C, and it is known that all variable attributes are `NONE`, so error 3 cannot occur). One way to use this is to gradually build any instance from a *single* inheritance hierarchy by creating it as `object` and then adding up mixins along some path down through the inheritance tree (first making it a `Point`, then proceeding to a `ColorPoint`, then a `SingingAndDancingColorPoint`, etc.).
- It is always safe to specialize an object with a pattern that does not further-bind any existing virtuals in the object (new virtuals and other attributes can be added).
- It is always safe to specialize a new object with a pattern that only further-binds virtuals in the object that it "owns". E.g., the virtual attributes of an abstract pattern may be divided into groups, and then an object may be built gradually by specializing it dynamically several times, each time with a pattern that takes care of exactly one group of shared virtuals.

So there are many ways in which dynamic object specialization can fail in the general case, but there are also many different conventions which can be employed to ensure that they will actually never fail. It would be much better if the static analysis could detect exactly when the specialization operation would be guaranteed to succeed, but that question is of course undecidable, and the safe, algorithmic approximations that we have been able to come up with are too restricting to enforce generally.

7.3.3 Incremental Object Creation

This section presents the notion of incremental object creation, which is one example of a disciplined way to use dynamic object specialization. It motivates this technique from a software engineering point of view. We claim that it enables an improved modularization of large and complex systems, thereby removing ripple effects from the dynamics of system development which otherwise make it "exponentially" harder to manage large systems than small ones. The argumentation is given in the frame of a single, running example, namely that of modeling cars from multiple perspectives.

In large projects it is important to apply "divide & conquer" strategies whenever possible, e.g. by dividing the system into many small clearly defined modules, and by keeping module dependencies at a minimum. Information which is used or even available globally challenges this strategy by creating many dependencies between modules. Dependency management as a core concern in software engineering has been advocated and substantiated in particular by Robert C. Martin [77].

One important measure according to this perspective is how much work it requires to bring a complex system back to a usable state after a given change. For example, if many modules (say, in a C++ system) use a certain function, and this function is changed to take one more argument, then many changes will have to be made before the system is again usable. Modules and module elements may thus be characterized as more or less "heavy" to change, and the ideal is to minimize unnecessary "weight" everywhere in the system, as well as to ensure that the "heavy" elements do not *need* to change so often. The act of separating a module into two (modules or parts) where one is the interface (what the other modules need to know) and the other is the implementation (whatever is left over when the interface has been extracted) is one typical technique which takes us toward that goal, but the choice of elements (e.g. the set of classes in a system), the partitioning of elements into modules, and the import network between modules are also important factors.

The ability to create objects incrementally gives a new opportunity to remove global knowledge about the total structure of highly visible objects, and thereby makes it easier to change this total structure of an object without affecting the system globally. This is especially beneficial in those cases where no single module needs to use this total structure anyway, because individual modules only are concerned with one aspect of it.

Related problems have been studied in the area of Subject-Orientation [52]. In subject oriented programming, each subject is a separate, static "universe" consisting of fragments of classes in the system. This makes it possible for one (complete) class to participate in several different subjects with different inter-

```
class Car
{ public: int registration_number; };
class Property
{ public: int price; };
class Schedulable
{ public: void reserve(time from, time to); };

class TotalCar: public Car,
                public Property,
                public Schedulable
{};
```

Figure 7.4: A direct, naive design of the `Car` domain

faces in each subject, hence allowing designers to concentrate on one perspective at a time and later combine the subjects to complete systems. In contrast, incremental object creation in `gbeta` is a dynamic mechanism, and the different views on a given object are not separated—any given piece of code might obtain any view it wishes on a given object (which supports it). Apart from these differences it is certainly possible to view dynamic object specialization as one possible mechanism giving language support for approaching the goals stated by the subject orientation community.

Now let us turn to the concrete example. Consider a situation where many different departments of a large organization need to interact with the same computerized model of an entity, e.g., a car. The full, combined specification of the car depends on many subsystems, each providing one aspect. The car 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, so we do not want to model one car with many separate computer objects. Cars are probably used in many places, too. As a result, the system contains many cross-module interdependencies, and it becomes hard to maintain. Incremental object creation avoids that problem since the combination class does not need to be declared explicitly anywhere.

Figure 7.4 shows a first, naive approximation to such a design, using C++. We have no intention of making these classes useful in a real project, but even with toy-requirements this is a naive design. It is considered bad style to have publically accessible data members, but that is only a problem because of the lack of uniform access, so that is not our problem, either. We must think of these classes as defined in separate modules, one for each class, because each of the aspects of the car is a part of a much larger system. For example, the accounting department handles everything that has to do with buying property and maintaining the tax related issues as the property gets older and looses value; so `Property` lives in that world. Similarly, `Schedulable` is maintained in the logistics department.

The real problem lies in the module interdependencies. Let us call the central

```
// Interface layer
class Car
{ public: virtual int registration_number() = 0; };
class Property
{ public: virtual int price() = 0; };
class Schedulable
{ public: virtual reserve(time from, time to) = 0; };

class TotalCar: public Car,
                public Property,
                public Schedulable
{};

TotalCar *newTotalCar()
{ return new ConcreteTotalCar(); }

// Implementation layer
class ConcreteCar: public Car { ... };
class ConcreteProperty: public Property { ... };
class ConcreteSchedulable: public Schedulable { ... };
class ConcreteTotalCar: TotalCar { ... };
```

Figure 7.5: A standard improvement over Fig. 7.4

module the *hub*; this is the module which contains the class `TotalCar`. Since all departments need to use instances of `TotalCar`, and since `TotalCar` depends on all the aspects, the system ends up having dependency links back *and* forth between the hub and all the department modules. Any little change will cause a total recompilation, which is already a serious problem in a large system. Moreover, when the compiler has to consider the entire system it is likely that some human beings should also take a look at the whole system, to see if the changes break any of the many conventions that compilers cannot handle. This system will be very unstable, in the sense that machines and/or human beings will have to reconsider all of it all the time during development.

Now, there are well-known ways to handle this; we just wanted to show the naive approach first because standard solutions might be two steps forward and one step backward compared to the naive solution, and if alternatives allow us to avoid going backward then we should keep it in mind. Figure 7.5 shows such a standard solution.

With the design in Fig. 7.5 there could be two layers of modules: Each department would have an interface module layer and an implementation module layer; for instance, the accounting department would keep `Property` at the interface level, and hide `ConcreteProperty` away in the implementation layer. The hub interface would contain `TotalCar`, and it would depend on the department interfaces. The departments would depend on the hub interface, too. The

```
(# Car: (# registration_number: @integer #);
   Property: (# int: @price #);
   Schedulable: (# reserve: (# from,to: @time enter(from,to)..#)#);
   myCar: ^Car
do
   (* create a new car and make 'myCar' refer to it *)
   &myCar[];
   (* build the structure of 'myCar' dynamically *)
   Property## -> myCar##;
   Schedulable## -> myCar##;
#)
```

Figure 7.6: Incremental Object Creation

hub implementation would contain `ConcreteTotalCar`, and it would depend on the department implementation modules. This *is* an improvement, because the loop has been broken: if the implementation in one department is changed, then the hub implementation is affected, but the buck stops here because the hub interface is unaffected. The interfaces still have the looping dependencies, but they are not expected to change so often.

Languages like Ada95, Turbo Pascal, and BETA allow for a separation of interface and implementation without requiring an extra set of classes, but since the extra set of classes allow such things as dynamically choosing one of several available implementation classes, it might be relevant to have the `Concrete` ... classes even in those languages.

In gbeta we could have created `ConcreteTotalCar` by merging `ConcreteCar`, `ConcreteProperty`, and `ConcreteSchedulable`. This is not possible in languages like C++ or Eiffel, because the result would not be a subclass of `TotalCar`, because these languages do not support the mixin based coarse-grained structural equivalence that gbeta has. However, this could for example be handled by having a `ConcreteCar`, a `ConcreteProperty`, and a `ConcreteSchedulable` as data members and manually delegating to them. To be able to get a real, implemented car via the hub interface we may use a factory method like `newTotalCar`. This is necessary because the departments should *not* need to depend on the hub implementation.

This adds up to a solution which has reduced the dependency problems in the original, naive approach. However, we have paid for this in terms of a significantly more complex design. Moreover, it would still affect all parts of the system if a new aspect were to be added, for example if the legal department wanted to be able to handle cars in their registration of work related injuries.

The approach which builds on incremental object creation removes the global representation of the total structure of car objects entirely, thereby making it possible to use a naive design without creating the dependency loop. Figure 7.6 shows some patterns for the different aspects, corresponding to Fig. 7.4. In the `do`-part, a plain `Car` is created and then enhanced with the two aspects

`Property` and `Schedulable`. This provides the car with new attributes such as `price` and `reserve`, but they cannot immediately be accessed because the type of `myCar` is `Car`. To use the enhanced interface we must obtain a reference to `myCar` with a more specialized type; this can be done with a `when` imperative, see Sect. 9.1 for details.

This dynamic approach allows us to work with objects that support the same, rich interface as `TotalCar` does in the previous approaches, but the system does not need to contain a class for this explicitly, and hence there is no need to have a hub interface that all modules depend on, or to have a dependency loop.

The dynamic construction of the total car may happen gradually, by a separate step taken in each of the departments. There would then be a *dynamic hub*, i.e., a central location in the system which would have access to a list of entities, one from each department. In this case we would need to have access to one 'car enhancer method' from every department, where a car enhancer method is some pattern which is less-equal than this one (note that it does not even have to depend on `Car`, even though it seems reasonable to qualify `target` by `Car`):

```
carEnhancer: (# theCar: ^Car enter theCar[] do INNER #)
```
Ex. 7-9

Each department would not need to depend on the patterns provided and used by the other departments, they would only need to know this simple method signature pattern, `carEnhancer`, to be able to provide a car enhancer method to the dynamic hub. Conversely, the hub would not need to depend on any department specifics, such as the pattern `Property` etc. Hence, the dependencies between the hub and the departments have been almost completely *removed*—it is not just a loop which has been broken. Since the simple `carEnhancer` pattern is unlikely to need to change, the whole system will be very stable even if some departments keep changing *their* stuff.

As mentioned, the dynamic hub would keep a collection of car enhancer methods; they would be used whenever a new car were to be created, like this:

```
(# carEnhancers: [n] ##carEnhancer;
   theNewCar: ^Car
do &theNewCar[];
   (for i:carEnhancers.range repeat
       theNewCar[]->carEnhancer[i]
   for)
#)
```
Ex. 7-10

This would iterate through all the available car enhancer methods and run them with `theNewCar` as the argument, and that would give each department an opportunity to enhance the car in whatever way it wants; for example, the accounting department would provide this pattern:

```
AccountingCarEnhancer: carEnhancer
  (# do Property##->theCar## #)
```
Ex. 7-11

Note that these dynamic specialization operations would be guaranteed to succeed, for example, if the different aspects were unrelated, or if they shared only a common superpattern and did not further-bind the same virtuals in this shared superpattern. Such a convention might be reasonably easy to specify and acceptable to conform to.

To avoid having global knowledge about department specific patterns like `Property` and at the same time avoid dynamic typecasing to obtain a reference to any given `Car` qualified by a department specific pattern, it might be beneficial if every department kept a database of references to all cars, qualified by their own department specific views.

Alternatively, it might be acceptable, and it certainly would save some space and administration, if cars were passed around globally as `Car`, and whenever a department needed to work with a car it would use a **`when`** imperative to obtain a more specific view on it, such as `Property`. If that view were not available in a given car, then somebody violated the car enhancer convention, which would be a bug. Nevertheless, it would actually be possible to add missing aspects later, so the situation could just be repaired when detected. This would be dangerous in the general case, but it would be safe if the added aspect shared no mixins with other aspects, or if it only shared mixins whose virtuals it did not further-bind.

The car would thus support the global set of interfaces, but no part of the system would depend on this global set as a whole. Furthermore, a new aspect could be added without recompiling any of the existing departmental subsystems.

There are still more advantages with the dynamic, incremental object creation which are not parallelled in the static approach. Firstly, if we do not need the `TotalCar` but would rather prefer to use any of the many different selections from a set of n aspects, then the static approach leads to a tedious definition of the many 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., a system may contain many cars, but only few of them need to carry legal department information about the consequences of a traffic accident. If we need this aspect for a given `Car` then we can test to see if it is already there, and we can add it dynamically if it is not, subject to the same safety considerations as above.

In any case, the improved independence between all the modules of this multi-aspectual car system is obtained by taking away type information from the global universe, thereby making the system as a whole much more resilient against changes in that type information. This will inevitably cause either the need to reestablish the type information locally (by means of typecasing), or the need to apply extra resources in order to avoid losing that information in the first place (such as the local databases of cars with department specific views). It should probably not be expected that the improved independence can be obtained as an entirely free lunch ...

Chapter 8

Miscellaneous Enhancements

This chapter presents a number of language features in gbeta that have not been covered elsewhere. They are optional enhancements, in the sense that they could be removed without affecting the rest of the language. None of these enhancements required changes to the basic strategies in the static analysis or to the run-time entities, nor did they cause any major implementation efforts. However, they may certainly make a significant difference for users of the language.

The enhancements can be divided into two major groups. The first group, treated in Sect. 8.1, contains elements of gbeta that are not backward compatible with BETA. The language constructs in the second group, Sect. 8.2, support a more concise and convenient expression of semantics that could already be expressed using such things as a few extra imperatives or an extra attribute declaration.

8.1 Incompatible Changes

In almost all respects, correct BETA programs are also correct gbeta programs, and they will behave identically. However, a couple of incompatible changes were introduced in gbeta, because we considered the BETA semantics improvable in those few cases. These changes are genuine language design issues; there are also some implementation specific issues which cause programs written for the Mjolner BETA implementation to be rejected by gbeta; more about this can be found in Sect. 11.2.

8.1.1 Repetition Evaluations

Repetitions have not been presented in earlier chapters, because we wanted to concentrate on the core of gbeta. However, we have to present them now in order to be able to specify how gbeta repetitions differ from BETA repetitions.

Repetitions are not entities. They are collections of attributes whose names ◇

are computed by combining a shared name for all the attributes in the collec-
◇ tion with a number, the *index* of the chosen attribute. The index can be any
◇ member of the set $\{1 \dots n\}$ for some natural number n, the *range* of the repetition. The range is the number of attributes in the repetition; it is obtained by evaluation of an integer valued expression when the part object containing the repetition is created (and all its attributes are created with it), and it may be changed dynamically. Repetitions can be recognized syntactically by the use of expressions enclosed in square brackets, both in declarations and in the use of computed names. For example:

```
(# R: [3] @integer; (* 3 object attrs: R[1], R[2], R[3] *)
do (* R.range = 3 *)
   (* access the second attribute in 'R' *)
   5->R[2];
   (* use of 'R[4]' would be an error here *)
   (* add two attributes 'R[4]' and 'R[5]' to 'R' *)
   2->R.extend; (* R.range = 5 *)
   (* new attributes have the usual initial values *)
   R[4]+1->R[1]; (* R[1] = 1 *)
   (* throw out all 5 attributes, get 4 new ones *)
   4->R.new; (* R.range = 4, R[1] = 0 *)
#)
```
Ex. 8-1

Note that the notion of computed names introduces a special version of the MessageNotUnderstood error which cannot be ruled out statically. The problem is that both the range of a given repetition and the indices used in computed names may be the result of arbitrary computations, so it is generally an undecidable problem to determine statically whether or not a given attribute in a repetition exists. However, it is almost safe to test whether a given repetition attribute access will succeed by means of an if imperative:

```
(if R.range>=3 then 0->R[3] if)
```
Ex. 8-2

This only fails if another thread modifies R between the Evaluation and the body of the if imperative. This is not normally considered as a loophole in the type analysis but rather as an entirely separate (and more politically correct) kind of run-time error, 'Index out of range'. We have not added a construct in gbeta which makes it possible to implement this example in a safe way, but a variant of the when imperative could be used; see Sect. 9.1.3 for more information on when.

So far, this describes repetitions in BETA and in gbeta equally well. However, there is a special syntactic shorthand for multiple assignments associated with repetitions. BETA and gbeta have different semantics for such repeated assignments in some cases. The syntax for a repeated assignment is just like that of an ordinary AssignmentEvaluation, but the source and destination entities are repetitions. The semantics is a repeated assignment operation which evaluates the first attribute of the left hand side and assigns the resulting value to the first attribute of the right hand side, then repeats for the two attributes with index two, and so on. The receiving repetition is in all cases adjusted such that both repetitions have the same range. For example, the assignment R1->R2 has the same effect as the whole do-part in context of the following example, both in BETA and in gbeta:

```
(# R1: [3] @char;
   R2: [0] @char
do (if true
    // R1.range>R2.range then R1.range-R2.range->R2.extend
    // R1.range<R2.range then R2.range-R1.range->R2.delete
   if);
   (for i:R1.range repeat R1[i]->R2[i] for)
#)
```
Ex. 8-3

BETA and gbeta do not include the primitive repetition operation delete, but such an operation is needed in the implementation in order to get the right semantics for these repeated assignments; the delete operation would remove as many attributes as the given integer argument specifies, starting from the highest indices and going downwards. The whole operation happens atomically, such that the range of the two repetitions will be equal until the end of the for imperative.

In Mjolner BETA, repetitions of object attributes have different assignment semantics, depending on whether they denote instances of basic patterns, like integer and char, or instances of composite patterns. For composite patterns, the repetition assignment works like a repeated reference assignment. In gbeta there is no such distinction between basic patterns and composite patterns, but the repeated reference assignment can be specified with another syntax in gbeta, namely a syntax which is similar to ordinary, single reference assignment. For example, the imperative R3[]->R4[] in gbeta and R3->R4 in BETA work like the whole do-part in the following context:

```
(# Point: (# x,y: @integer #);
   R3: [2] @Point;
   R4: [7] ^Point
do (if true
    // R3.range>R4.range then R3.range-R4.range->R4.extend
    // R3.range<R4.range then R4.range-R3.range->R4.delete
   if);
   (for i:R3.range repeat R3[i][]->R4[i][] for)
#)
```
Ex. 8-4

In BETA it would be possible to declare R4 as a repetition of object attributes, e.g. R4: [7] @Point, and carry out the same repetition assignment. That is rejected in gbeta because the implied execution of R3[i][]->R4[i][] would be rejected—reference assignment can only be applied to a variable object attribute, because it means that the attribute must *vary*.

In general, the coercion markers applied to repetition assignments in gbeta are carried over to the body of the implied for imperative, so for example R5[]->R6 means the same as the do-part in this example:

```
(# printer: (# s: ^string enter s[] do INNER #);
   R5: [2] @string;
   R6: [7] ^printer
do (if true
    // R5.range>R6.range then R5.range-R6.range->R6.extend
    // R5.range<R6.range then R6.range-R5.range->R6.delete
   if);
   (for i:R5.range repeat R5[i][]->R6[i] for)
#)
```

Ex. 8-5

One way to explain the effect of R5[]->R6 is that it gives some string objects as arguments, by reference, to some stored procedure invocations, which are less-equal than printer, and then calls them.

The benefits obtained by redefining the semantics of repetition assignment in gbeta compared to Beta are the following:

- There is no distinction between basic patterns and composite patterns; such a distinction is nowhere else assumed and it would be a semantic anomaly to introduce it here.
- There is no support for 'reference assignment to an object attribute', such as with R4 in the Beta imperative R3->R4 above. Again, that would be a semantic anomaly. We should note that this is the actually implemented semantics in the Mjolner compiler. The explanation of repetition assignment in [74] could very well be interpreted to mean something much closer to the semantics in gbeta.
- The simple, mechanical construction of the implied for imperative from any given repetition assignment ensures that the connection between syntax and semantics for repetition assignment is consistent with the rest of the language.
- The various combinations of coercion markers on repetition assignments mean different things, and they may all be useful.

However, there are certainly also some problems left:

- It would probably not be meaningful to require that arbitrarily complex assignment operations should be executed repeatedly as an *atomic* language operation. What if such a complex repeated assignment would attempt to change the range of one of the repetitions during the operation? The repetitions should not support that, that's what the atomicity is all about. What would it *mean* to execute such a repeated assignment "atomically" if each iteration might execute arbitrary code? Should all other threads stop, or should the run-time system be required to maintain locks on repetitions that are being used in assignments?
- The alternative, being that such an operation would not have to be atomic, does not seem to be attractive either. It would be a very unpleasant perspective if all repetition assignment operations could potentially cause 'Index out of range' errors.

- It seems inconsistent that a repetition of object attributes supports the `new` operation. This allows us to, e.g., access *two different* objects via the computed name `R1[2]` in the first example above. Normally, an object attribute will denote the same object in the entire lifetime of the part object of which it is an attribute. However, removing this possibility would seriously damage the backward compatibility of `gbeta` and create a need for alternative ways to handle such common facilities as the manipulation of mutable strings (i.e. the `text` pattern in BETA).

It might be beneficial to exploit and widen the static knowledge about the length of certain repetitions and thereby be able to declare certain operations safe. This could be done by making some (explicitly marked) repetitions fixed-length, and then statically prohibit the operations `new` and `extend` on these repetitions. If the range were a compile-time constant then even some accesses to attributes in the repetition could be checked statically. Fixed-length repetitions might also allow for a more efficient representation in some cases. Nevertheless, there would be many cases where the variable and mutable range would be very useful, and arbitrary computations to select attributes from a repetition are also useful indeed.

The conclusion is probably that repetition assignments are too complex to allow for a definition which is both consistent and safe. Either we must single out the basic patterns and disallow repeated value assignments for the general case (reference assignments are simpler), or we must accept that repetition assignment cannot be executed atomically, even though it is a primitive operation in the language. Actually, there would be problems with requiring atomicity for assignment of very long repetitions, also in the case of repetitions of instances of basic patterns. The basic problem with a non-atomic failable primitive operation is that a highly safety oriented programming style must abandon it entirely; the `extend` and `new` operations on the two repetitions are made into run-time errors for arbitrary periods of time, and there is no test which will determine whether or not they can be executed safely at any given point in time. This is no better than having the potential for a `MessageNotUnderstood` error at all times.

One solution could be to define the semantics of repeated assignments to carry out as many iterations as possible after the initial length synchronization, i.e., until the end of the shortest repetition is reached. That would make the operation safe, it would not require atomicity and would therefore allow the consistent, fully general value assignment semantics. It would then be possible for programmers to check if the expected number of assignments were performed, and handle the problem gracefully if not.

Another solution would be to remove the support for repetition assignments altogether and require that programmers write the corresponding `for` loops explicitly. Since repetitions should arguably not be used “publicly” anyway—they are simply too low-level and too inflexible to be part of widely used interfaces—such a forced extra verbosity might actually make it even more clear that they are meant to be used locally in the implementation of “real” abstractions. It might be necessary to add a primitive operation which would adjust the range of a given repetition efficiently. It should not be impossible for a production quality compiler to generate code with the same performance for the explicit `for` imperative version as it could do for the repetition assignment shorthand.

8.1.2 The Type of `this`

In BETA, the name application which appears in the syntactic construct ThisObject, e.g., `Point` in `this(Point)`, is used for two purposes. First, it is used to select one of the enclosing objects—the nearest enclosing object whose pattern is statically known to be less-equal than the pattern of the attribute found by looking up that name. Second, the object which is accessed through such a ThisObject construct is accessed with the view of that name application—e.g. `this(Point)` will be treated like a `Point` even if it is a `ColorPoint`. To see what that means in practice, consider the following example:

```
1 (# (* This is in BETA, not gbeta *)
2    p: $^1$(# x: @integer #);
3    q: p$^2$(# x: $^3$(# exit 2 #)#);
4    r: @q$^4$(# do x->this(p).x #);
5    aP: ^p
6 do
7    r[]->aP[];
8    r.x->aP.x
9 #)
```
Ex. 8-6

The topic of this example is how to access multiple attributes with the same name, in the same object. As mentioned in Sect. 3.6, this is deprecated in gbeta, but it might be inevitable, e.g., when combining two large, independently developed bodies of code. Whenever a single object has more than one attribute with a given name, the specificity of the mixins in the view determines which one of these attributes will be denoted by a given name application (statically as well as dynamically). In the example, the object `r`, with part objects $[4, 2, 1]$, contains two attributes named `x`, and the usage of `x` in its do-part will access the pattern in its part object 2 (line 3).

If there is a need to access another attribute than the chosen one, then we need to obtain another view on the object in which the desired attribute is the most specific one. For example, the construct `this(p).x` in line 4 is used to access the `x` object attribute in part object 1 of `r`, line 2.

However, that approach does not work from outside of `r`, as in the main do-part of the above example. Here, `r.x` in line 8 is used to access the pattern `x`, but a separate variable object attribute such as `aP` must be declared, line 5, in order to access the `integer` object `x` with `aP.x`, line 8. The two do-parts have the same effect, but they are too different. Moreover, it is confusing to have to perform a small data flow analysis to discover that `r` and `aP` is actually the same object, only seen in two different ways.

For this reason, the view manipulation effect of ThisObject was removed in gbeta, and a construct similar to the **qua** expression in Simula was reintroduced (see Sect. 8.2.2) to manage views, and to do that only. As a result, a ThisObject construct in gbeta will be useful in all the same ways as in Beta when used to get access to one of the enclosing objects, but that enclosing object will be accessed under the view which represents the full available knowledge about that particular object. With this change, and the introduction of a **qua**-like construct, two concerns have been separated, namely access to enclosing objects and access to any object via a specified view. We think that this is an improvement for the readability of source code.

This is not the only benefit. Another improvement is that the ThisObject construct supports access to the entire known structure of enclosing objects, even if their patterns do not have a name, for example:

```
1  list:
2    (# element:< object;
3       scan: (# current: ^element do INNER #)
4       scanReverse: (# current: ^element do INNER #)
5    do scan
6       (# previous: ^element
7       do scanReverse
8          (# previous: ^element
9          do this(scan).current[]->previous[];
10            this(scan).previous[]->current[]
11         #)
12      #)
13   #)
```
Ex. 8-7

This piece of code illustrates a problem with access to identically named attributes in enclosing objects. We can get access to `this(scan).current` (line 9) in both Beta and gbeta because `current` is inherited from `scan` (line 3), but only the gbeta analysis allows us to get access to `this(scan).previous` (line 10). In this case the problem has been created artificially in order to keep the example short, but this problem *does* arise now and then, and the ability to use the selected enclosing object in full via ThisObject seems consistent and natural.

However, this change is actually insufficient. Another generalization of ThisObject should be introduced, but this has not yet been implemented in gbeta: The name application is often not sufficiently flexible for cases like the above example, for instance if the two nested control structure patterns are specializations of `oneList.scan` and `anotherList.scan`—which would be quite realistic. We would only need to allow a general AttributeDenotation instead of the NameApl which is currently allowed, such that we could write, e.g., `this(anotherList.scan).current[]`. It would also be convenient to allow `this` alone, meaning `this(object)`. The implementation would be easy, so this will probably happen soon.

Another case where the gbeta analysis of ThisObject is useful is when methods return a reference to the enclosing object:

```
link:
  (# T:< object;
     value: ^T;
     next: ^=this(link);
     append:<
       (# other: ^=this(link)
       enter other[]
       do other[]->next[]; INNER
       #)
  #);
doubleLink: link
  (# prev: ^=this(doubleLink);
     append::< (# do this(doubleLink)[]->next.prev[] #)
  #)
```

Figure 8.1: Linked lists

```
Counter:
  (# x: @integer
     inc: (# do x+1->x exit this(Counter)[] #);
     dec: (# do x-1->x exit this(Counter)[] #)
  #)
```

Ex. 8-8

With this style (where methods return the enclosing object if they do not have to return anything else), it becomes natural to cascade method invocations like this:

```
(# ct: @Counter
do
   ct.inc.dec.inc
#)
```

Ex. 8-9

This example also uses the enhancement described in Sect. 8.2.5 to avoid some parentheses, but the point we want to make here is that this breaks down with the approach taken in BETA when we create a subpattern of `Counter`. Assume that we want to use cascading method calls on an instance of such a subpattern; since the inherited methods have `this(Counter)[]` in their `exit` lists, the result will always be considered an instance of `Counter` with the BETA analysis, and this means that the cascade is broken—after the first invocation of an inherited method, only the inherited methods can be used.

As a final example illustrating how useful it is to be able to express a genuine `SelfType` (a qualification which is recognized by the static analysis as being equal to the actual, run-time pattern of a given enclosing object), consider the `link` and `doubleLink` patterns in Fig. 8.1. A linked list can be built from `link` elements, and a doubly linked list from `doubleLink` elements, with each link (double or not) carrying one value by reference via the `value` attribute.

Such a list will be heterogeneous in the sense that the `value` attribute of each `link` may refer to an instance of any subpattern of `T`. The virtual `T` can be further bound, hence restricting `value` to be an instance of a more specific pattern than `object`; we may for example have a `link` which requires that its `value` is an `integer`, whereas another `link` might require that *its* `value` is a `boolean`. The crucial point here is that this will never be the case for two `link`s in the *same* list, and that is again because the `next` attribute in `link` (and also the `prev` attribute in `doubleLink`) have the *exact* qualification `this(link)`. This exactness also ensures that there will never be a list which contains both some single `link`s and some `doubleLink`s.

This means that every list which is built from instances of `link` or a subpattern of `link` is homogeneous—in the sense that the "spine" of the list, the `link`s themselves, consists of instances of the exact same pattern. As a result, it is known to be safe to reference assign a new `value` to an arbitrary element in such a list as soon as it is known to be safe to do it with just one `link` in the list. Consequently, the whole list is adequately constrained for static analysis by each of its elements, and that means that such lists can be passed around just by passing abound a reference to one of the elements.

This homogeneity also makes `append` safe: It would not be safe to assign both `other[]->next[]` in `append` in `link` and `this(doubleLink)[]->next.prev[]` in `append` in `doubleLink` with inexact qualifications. Moreover, if we changed the qualification of `next` and `prev` to be an ordinary virtual then it would be possible to final-bind it in all kinds of `link`s which are used to create instances, and this could make usage of `link`s from the outside safe, but it still would not help for the implementation of methods inside `link`. Hence, `link` and `doubleLink` could not be implemented safely using virtuals.

To conclude, the combination of a true `SelfType` and exact qualifications make it possible to maintain a strict homogeneity in the patterns of objects in dynamic configurations of unbounded size, and this makes designs like `link` and `doubleLink` useful. Note that the typical design in BETA for situations like this is to have a wrapper pattern which contains a virtual pattern attribute which plays a similar role as `link` does here. A virtual pattern has the property that a usage of the name of the virtual inside its own definition will actually be a genuine `SelfType` (unless inheritance from virtuals is supported). However, with a `list` which is built on this principle (such as the standard Mjolner BETA `list` pattern where the nested "`doubleLink`" virtual is called `theCellType`) it will not be possible to transfer links between different `list`s—they have a wrong origin for all other `list`s than their enclosing one. This would be particularly nasty if a large list were to be reorganized, perhaps by splitting it into two smaller lists with roughly the same number of elements. That would require creation of new link objects for half of the elements in the large list.

8.2 Convenience Constructs

This section describes some constructs in gbeta which could be removed from the language without restricting the expressive power much; the effect of these constructs could be achieved using local rewriting techniques which might introduce a few extra imperatives and auxiliary attributes. Some of them are old, well-known ideas that just never were given sufficiently high priority to actually be implemented in BETA.

8.2.1 Computed Object Evaluation

◇ Computed object evaluation, marked by '`!`', has been present in the BETA grammar for many years, but has not yet been implemented. We decided to invent and implement a precise semantics for this; it seems to be useful, though more experience with it would still be beneficial. It enables the use of an object which has no direct denotation, but which is available via the evaluation of a Reference. It may be described as a dereferencing operation, somewhat like the '`*`' operator in C. The difference is that it implies the evaluation of the Reference. For example:

```
(# intFunc: (# i,j: @integer enter i do INNER exit j #);
   getAnIntFunc: (# f: ^intFunc ...  exit f[] #);
   x: @integer
do
   3 -> getAnIntFunc! -> x
#)
```
Ex. 8-10

The effect of the main `do`-part in this example is to compute the value of an `intFunc` at 3. The `intFunc` itself is obtained as the result of an invocation of the method `getAnIntFunc`. If there is a need to give arguments to `getAnIntFunc` or otherwise perform a more complex complex computation, it can be expressed by enclosing it in a MainPart:

```
3 -> (# exit (a+b)->getAnIntFunc->modifyAnIntFunc #)! -> x
```
Ex. 8-11

8.2.2 Explicit Choice of View

As described in Sect. 8.1.2, the syntactic construct ThisObject cannot be used in gbeta to obtain a specific view on an object, like it can in BETA. Instead, a construct similar to the **qua** construct in Simula has been reintroduced in gbeta. This construct is similar to a type cast operation in C, but it is safe: it will not "cast" an object to any pattern unless it is statically known to be greater-equal than the actual pattern of that object. The Simula **qua** construct can also "downcast" an object to a more specific class, but we have reserved another construct for that, see Sect. 9.1. For example:

```
(#
   p: ^1(# x: @integer #);
   q: p^2(# x: ^3(# exit 2 #)#);
   r: @q^4(# do x->this(:p:).x #)
do
   r.x->r(:p:).x
#)
```
Ex. 8-12

It is illustrative to compare this example with the example at the beginning of Sect. 8.1.2; note that we have used the not yet implemented shorthand `this`, which means `this(object)`. The syntax of this construct is based on the '`(: ... :)`' brackets. The use of parentheses with markers placed just inside them is standard BETA syntactic style. The choice of colon as the marker is supposed to signal the relation to declarations (which are the primary tools for specifying and looking up views), and the placement *after* the entity to be viewed differently is both more convenient syntactically—these constructs can be applied more than once in the same AttributeDenotation—and it places the view in view for the person who is reading the source code.

It is sometimes claimed, e.g. [98, p. 82] where `super`-sends are considered, that the correctness of a program is threatened if it uses a construct which lets programmers explicitly select the class in which the search for a given attribute should start. This is indeed the case in languages like Smalltalk where such a choice would cause an "inverted override" phenomenon: If a subclass redefines a method then the redefined version of the method should be the correct choice for an instance of that subclass in all cases, and subtle bugs could be introduced by "freezing" the choice to a method implementation in any fixed class for individual invocations of that method. This might be seen as a reason to avoid such constructs entirely.

The situation is different in gbeta, because the choice of a particular attribute is determined by the view and is therefore *always* made at compile-time, so a construct that allows explicit view selection is not semantically inconsistent with the rest of the language. There is no such thing as "choosing the wrong implementation of a late-bound method" in gbeta, because virtual attributes are never overridden, they are *unified*, and the pattern which is the result of the unification can be obtained by looking up any of the declarations for that virtual—they all deliver the same pattern because they are all the same attribute.

8.2.3 Control Structure Evaluations

An old idea which has not yet been implemented in BETA is that of "value" versions of the built-in control structures. This is in some ways similar to the `if` statement in Standard ML and other functional languages. For example:

```
boolean2string:
  (# b: @boolean
  enter b
  exit (if b then 'true' else 'false' if)
  #);
x,y: @integer;
setSome: @
  (# b: @boolean
  enter (if b then x else y if)
  #);
```

Ex. 8-13

This facility has not nearly been implemented in the full generality; for example, it is not supported to evaluate a GeneralIfImp, only a SimpleIfImp can be evaluated. It seems to be quite useful in practice, but the usefulness would probably not rise very steeply if it were implemented fully, so it is likely to stay at approximately this level of support for some time.

Note that the static analysis of these constructs might be handled by introducing a general notion of the least upper bound of any given finite set of types, such that, e.g., an expression which might evaluate to an integer or to a char value might be typed as being any of those two. Since this is not needed anywhere else in the language and it would complicate the rest of the language considerably, for instance by introducing the need for declarations of attributes of all those least upper bound types, we have chosen the simplistic rule that these constructs must evaluate to the exact same type of value for every branch.

8.2.4 Renewal of Variable Objects

◇ In gbeta it is possible to *renew* a variable object by means of the '&' operator. This mechanism corresponds to the !!SomeAttribute construct in Eiffel. The effect is to create a new instance of the qualification of the variable object attribute and then make the attribute refer to the new instance. For example:

```
(# Point: (# x,y: @integer #);
   Pair2Point: (# pt: ^Point enter &pt exit pt[] #);
   myPoint: ^Point
do
   (2,3) -> Pair2Point -> myPoint[];
#)
```

Ex. 8-14

The invocation of Pair2Point causes the renewal of the attribute pt, and the new instance of Point is then value-assigned the pair of integer values which are given as arguments. The identity of pt is then delivered as the return value, and that is reference assigned to myPoint. Note that this construct violates the transparency of the variable object attribute *less* than the explicit construct that it is a shorthand for (which would include something like &Point[]->pt[] in the example), because the reference assignment to an attribute unambiguously reveals that it is a variable object attribute. With '&' it could be a pattern, a variable pattern, and a variable object.

The use of '&' in this renewal construct is historically related to the use of '&' in BETA in front of an ObjectSpecification which denotes a pattern, in order to create a new instance of that pattern. That use of '`&`' is actually less relevant in gbeta, where the implicit coercion—see Sect. 2.3.4 for details—ensures that new instances of patterns are created also when '`&`' is not present. Historically, the version without '`&`' in BETA was intended to mean that the compiler could use the *same* object several times when a pattern was used as an imperative and in assignments (that was called an 'inserted item', which is still the name of a non-terminal in the BETA grammar). This has never been implemented, and it is hardly considered desirable by anyone in the BETA community any more. So, in practice, there is little difference also in BETA between using such expressions as `myPoint.move` and `&myPoint.move`.

As a result, the '`&`' operator can be understood as an explicit constraint that a *new* object must be created at that point, which might be beneficial for readers of the source code to know. That must then be weighed against a (slight) loss in transparency, because an attribute which can be used with '`&`' cannot be a (non-variable) object attribute.

The usage of '`&`' for creation of new objects is historically entrenched, and the usage of '`&`' for pattern merging seems to be the natural choice. It is unfortunate, though, that the same symbol is used for two different purposes. One way to keep them apart is to remember that '`&`' for object creation appears at the *beginning* of a syntactic construct, whereas '`&`' for pattern merging appears *between* two constructs.

8.2.5 Improved Transparency

Greatly inspired by David Ungar, a mechanism has been added to gbeta which improves on the transparency of computed vs. stored values. This mechanism does not introduce new syntax. It is very simple, and it is backward compatible with BETA, because it gives meaning to constructs that would otherwise have caused static semantic errors. The error that used to be reported was that there was an attempt to look up an attribute in a pattern (the Mjolner compiler would say 'An object is expected here').

The new meaning is obtained by evaluating the pattern; if that delivers the identity of an object, then that object is used. For example:

```
(# door: (# open: (# ... #)#);
   house1: @(# frontDoor: @door #);
   house2: @
     (# frontDoor:
          (# theDoor: ^door
          do (* let me see, I think it is this one *)
             ... ->theDoor[]
          exit theDoor[]
          #)
     #)
do
   house1.frontDoor.open;
   house2.frontDoor.open
#)
```

Ex.
8-15

The point is that the `frontDoor` attribute in `house1` is stored and the `frontDoor` attribute in `house2` is computed, i.e., the former *is* an object and the latter *produces* an object when needed. Without this mechanism, it would have been necessary to change the usage of the `frontDoor` of `house2` (by adding parentheses to make it `(house2.frontDoor).open`); with this mechanism the two usage points can be identical. As always, the fact that the usages are independent of the chosen implementation of a given attribute improves on the possibilities for program reorganization and maintenance.

This transparency mechanism is specifically oriented towards attributes that are used to access other attributes. If an attribute is itself the last element in the chain (e.g., a stand-alone name, or the last name in a Remote construct), then the coercion which is available also without this mechanism already provides good support for changing the kind of an attribute without affecting usage points. Of course, it is still possible to become dependent on whether `frontDoor` is a pattern or an object by means of explicit coercion markers; for example `house1.frontDoor##` is less-equal than `door##`, but `house2.frontDoor##` is not. This underscores the fact that BETA and gbeta have very good transparency support as long as attributes are used without explicit coercion markers, but the transparency gets more and more compromised the more coercion markers there are, so they should preferably be kept in rather private areas of the source code, such as with parameters that are only used locally within a method.

Chapter 9

Improving the Static Analysis

This chapter describes some mechanisms in `gbeta` which enable a more flexible and detailed management of the information which is used in the static analysis, thus allowing for a statically safe expression of some designs which could not be expressed safely if these mechanisms were removed from the language. Since we highly value the safety guarantees provided by static analysis, these mechanisms are important additions to `gbeta`.

However, they have been designed with the rather idiosyncratic and formalistic requirements of the static analysis in mind, so they do not support the comprehensibility of the source code very well. The decisions about where and how to use them will usually be concerned with technicalities rather than understanding the overall conceptual universe of the program. It would probably be a good habit to maintain awareness about what aspects of a program express the essential design and what aspects are there to help the static analysis. In return, the static analysis might actually help improving the understanding of the program because false impressions of regularities may be unveiled when the static analysis is unable to verify them.

Section 9.1 describes the `when` imperative which supports a dynamic case selection based on patterns (type casing). Section 9.2 presents a mechanism which allows the specification of a lower bound on a virtual, thereby making certain usages of that pattern in contravariant positions safe. The next section, 9.3, describes virtual objects which allow for a more statically restricted way to provide arguments than the normal approach based on evaluations and `enter`-lists, and Sect. 9.4 describes the concept of disownment of a virtual, which makes it possible to further-bind a given virtual in certain ways, because certain other kinds of further-binding have been declared to be illegal.

Finally, Sect. 9.5 deals with a more profound change in the development of `gbeta` from BETA: There are two kinds of object-like entities in BETA, namely items and components. The difference is concerned with sequential vs. alternating or concurrent execution. These two kinds of entities have been unified into the single concept of objects in `gbeta`, and the support for non-sequential execution has been transferred to the domain of patterns by introducing a new, basic pattern. One of the main benefits obtained by this change is a clearer and more manageable static analysis treatment of non-sequential entities, and that is the reason why it is located in this chapter. Section 8.1.2 would actually fit very well into this chapter, but we decided to put it in Sect. 8.1 because it is not just an improvement for static analysis but also an incompatible change in relation to BETA.

9.1 Type Casing—the `when` Imperative

◇ A *type case* is a control structure which allows the selection of one of a given set of bodies according to the actual (run-time) descriptive value associated with a given run-time entity, for example the class of an object.

Many people in the object-orientation community consider type casing an inherently inappropriate activity, as explained in Sect. 9.1.1. However, we believe that some mechanism for dynamic rediscovery of the applicability of static information is needed, as argued in Sect. 9.1.2—not just because real projects are imperfect, but because there is a genuine trade-off between the overall complexity of a system and the completeness of statically checkable information about run-time entities such as objects. Next, Sect. 9.1.3, describes how BETA can already do type casing, why that is not completely safe, how the `when` imperative in `gbeta` solves the safety problem, and how it works in more detail. Finally, Sect. 9.1.4 explains why the `when` imperative is a more powerful construct than traditional type casing based on purely static denotations of the descriptive entities.

9.1.1 Why Type Casing is Bad

It is a widely accepted tenet within the object-orientation community that type casing is a bad thing. For example, it is stated in [79, p. 1116] that the **inspect** construct in Simula 'runs into conflict with the Open-Closed principle'. The problem is that a type case, such as an **inspect** statement, contains an explicit list of cases and then handles the problem at hand in various different ways, one specific way for each case, and this creates unmanageable dependencies. The conflict with the Open-Closed principle is that every new class which may occur in context of the problem which is handled in such a type case will create the need for addition of a new case to the type case statement. Hence, the type case is not open for extension because the set of classes associated with that problem cannot be extended without affecting this particular piece of code. The preferred alternative is to declare a method in a suitable type, and implement it in each of the implementations of that type, basically by using the corresponding case from the type case as the method body. With this approach it becomes possible to add a new class without changing existing code—the new class just has to implement that method in its own way.

During the development and standardization of C++ [105] it was for a long time not considered appropriate to add constructs that would enable type casing, for example by means of dynamic class tests like `dynamic_cast<···>`, for similar reasons as the ones given above (space economy considerations were also taken into account, because this would require run-time type information even for classes that would not otherwise need that).

As a third and final example, at a Borland product presentation for more than a hundred programmers in 1991 in Copenhagen, the basic ideas of object-orientation were presented in such a way that this topic was made *the* most important benefit of using object-oriented languages. The central example was one where different graphical shapes were drawn, first using `switch` statements, and then using virtual methods. They added a new kind of shape and demonstrated how this required changes in several places with the switch based implementation but not with the virtual method based one.

In other words, this argument is at the heart of the folklore of object-orientation.

However, with functional programming languages such as Standard ML and Haskell it is a standard technique to write each function as a number of cases where each case deals with arguments of a certain kind, for example one case dealing with the empty list and another case dealing with non-empty lists. Since algebraic datatypes allow for the definition of one type containing a specified set of variants which may each contain composite values, this style of programming may resemble object-oriented programming in some ways. In this community the explicit casing on the structure of entities is not considered problematic. The benefit of such a program structure is that it is possible to write a new function which handles this type of values, each variant in a specific way, without changing the algebraic datatype or all the other functions for use with that type. With the object-oriented approach where the casing is made implicit and the cases stored in different places, that would correspond to the addition of a new (late-bound) method to the class hierarchy, along with the introduction of method implementations in each of the subclasses. This point is for instance made very clearly in Sect. 4 of [89].

So it seems that these two ways to distribute the implementation of a set of behaviors associated with a set of related kinds of entities each have their own benefits, and everyone should feel free to choose one or the other, depending on whether adding classes or adding methods seems the more likely. However, we do not think the choice is that symmetric.

Firstly, we believe that the ability to use context based reasoning makes it more profitable to keep the method implementations associated with the classes, in stead of keeping the cases for all classes together in the same procedure. In the latter case it is necessary to inspect one case in every one of the associated functions in order to see how the instances of a given class are actually implemented, and the implementations of all the other classes will be right there, disturbing the impression of the class being considered. What we need to know in order to be able to implement or maintain a method is what it should *do* and in what *context* it should do that, not how all the other classes implement it.

Secondly, there is no need to choose between the two kinds of extensibility, as we shall see in Chap. 10, we can have them both at the same time.

This leads to the conclusion that it is indeed not appropriate in the typical case to organize the implementations of a range of behaviors for a range of different kinds of entities by type casing on the entities in separate, global implementations of each behavior.

9.1.2 Why Type Casing is Good

As argued in the previous section, the organization of the implementations of a range of behaviors for a range of different kinds of entities by type casing on the entities is not recommended in the typical case; but there are also other uses for type case constructs, and they motivate the introduction of such a construct in gbeta.

To see why these other uses of type casing are significant we have to consider some possible trade-offs in the management of static information, assuming that type casing is *not* available. Note that a reference assignment in BETA and gbeta which causes a warning during static analysis and generation of a dynamic qualification check is in fact a special case of type casing, where the two cases are: (1) perform a reference assignment, or (2) raise a run-time error. Consequently, we must assume that such unsafe reference assignments are also not available in the following discussion, except for Smalltalk where the maintained static information is incomplete and we might say that such implicit type cases are present in every message send.

The maintenance of static knowledge about program execution entities is not a free lunch. As an example of a radical trade-off in this respect, the lack of explicit typing of instance variables in Smalltalk make it possible to change and recompile a class C in a system without affecting other classes than C and its subclasses, and without considering any other classes than the superclasses of C (and that is only to find the slot numbers of inherited instance variables).[1] This approach basically reduces the static knowledge about run-time entities to a minimum, thereby gaining flexibility and discarding safety.

At the other end of the spectrum we have a language like Pascal, where the strict typing of all entities ensures that the information about the structure of a given entity is propagated at least as far out into the source code as that kind of entity itself. This approach insists on complete static information about all accessed entities, thereby guaranteeing safety, but paying for it with a vastly increased network of program part interdependencies.

The complexity of global interdependencies tends to grow faster than proportionally when systems get larger, so the cost of maintaining complete static information gets gargantuan with complex systems. Moreover, each global concern has to be considered *again* in every new context where it is injected, so a module which carries a global concern with it may be a very heavy burden if it is to be integrated into a large system.

[1] Self has an even more radical separation in principle, but the very sophisticated compilation techniques reintroduce many globally scoped interdependencies; similar phenomena probably occur with highly optimizing modern implementations of Smalltalk.

Statically type checked object-oriented languages support the notions of subtyping and polymorphism, i.e., an attribute may have the safety invariant that it always refers to an entity of a given type, and a type may be defined to include objects which are instances of a given class or any of its subclasses, so a given name may give access to various different but related kinds of entities. This allows for intermediate trade-offs, where the static information about a given entity is not completely generic like in Smalltalk, and not rigidly complete like in Pascal.

An important reason why it is sufficient in many cases to have incomplete information about the structure of an object is that different method implementations may perform a range of actions with the same "meaning" using late binding. So, polymorphism and late binding play together to make unknown parts of an object structure useful—or to allow useful parts of an object structure to be unknown.

However, that approach can only be used as long as there is a way to express the desired interaction with a given object in terms of some sequence of invocations of methods which are known to be supported. The situation is different if we do not know exactly what object to use, such that we cannot invoke the desired method before we have found a suitable object. Similarly, we may need to obtain additional information because an object which is already at hand *might* support certain methods that we need to use. Note that this discussion is not about subversion of information hiding and ADTs (abstract data types), it is about retrieving information which is "public" and relevant, but currently unavailable in the given context.

In particular, for a method M to be invokable at all on a given object O, there must exist a reference to O somewhere in the program execution whose statically known type includes knowledge about that method M (note that this might also be a `this`/`self` reference, possibly implicit). Whether or not there exists at least one reference to a given object with a given type is a global consideration—just like garbage collection, which is concerned with a similar question. It is well-known that the global strategies which are needed in order to manage memory reclamation when garbage collection is not available—for instance conventions about "ownership" of objects—are the cause of a severe increase in overall system complexity and interconnectedness.

A typical example where it is hard to maintain the complete static knowledge about an object is when it is stored together with similar objects:

```
(# vehicle: (# ... #);
   car: vehicle(# ... #);
   bus: vehicle(# ... #);
   theCompany: @
     (# theVehicles: @list(# element::vehicle #);
        findBus: (# theBus: ^bus do ...  exit theBus[] #)
     #)
#)
```
Ex. 9-1

The challenge in this example is to write the implementation of the `findBus` method, which is supposed to deliver the identity of some `bus` which is associated with `theCompany`. The `vehicle`s which are associated with `theCompany` are conceivably acquired over a long period of time and under various different circumstances, and it is consequently hard to keep track of where each `vehicle` came from and what kind of vehicle it is. This is an example where it is tempting to throw away static knowledge about each individual `vehicle`, and that is exactly what we are doing if those `vehicle`s are only stored in `theVehicles`, as in the example above. In that case, we cannot implement `findBus` because there is no readily available reference to such a `bus` which has `bus` as its statically known qualification, and indeed there may not exist such a reference at all. We consider several solutions below, some of which will use more than one attribute in place of the single `list` called `theVehicles` above.

It is possible to introduce a virtual method `returnThisIfBus` in `vehicle` and implement it such that it delivers the identity of the object qualified as a `bus` if it is a `bus`, and delivers `NONE` otherwise. That is a bad solution for two reasons. One reason is that this style in general requires changes to `vehicle` whenever we add a new subpattern X of `vehicle` that we might want to call such a `returnThisIf`X method on. This is a global dependency loop because it makes `vehicle` depend on all its subpatterns in the general case. Another reason is that this amounts exactly to a manual reconstruction of the type casing mechanism that we are considering—if we really need `returnThisIf`X methods then it would be better to add support for type casing, which does not cause additional global dependencies.

An alternative strategy could be to introduce separate `list`s of `bus`es and of other `vehicle`s, but that would only work until we need a `getCar` method, then we would need to maintain a `list` of `bus`es, a `list` of `car`s, and a `list` of other `vehicle`s. We could put each kind of vehicle into a separate `list`, but then we would need to change `theCompany` every time a new subpattern of `vehicle` was added—exactly one of those global dependencies that we want to avoid.

To conclude, it may be a noble goal in the name of safety and correctness to preserve full static knowledge about all run-time entities somewhere in the system, and even maintain enough knowledge to know where to go to find a reference which offers sufficient static information for a given purpose. But we do not think that this is realistic in large and complex systems, or it will incur severe disadvantages in terms of increased complexity. The complexity cost can even be disproportionately large—if a module is to be reused in a complex system then any global concerns that it carries with it (such as keeping track of the number of references of a certain kind) will tend to be as complex as the total system to maintain, not just as complex as the module.

Finally, the tendency to lose static information about objects over time is of course aggravated even more if the scope in time or space is widened, for example for objects which are stored in a database and retrieved in the execution of another program, or objects which are exchanged or just accessed across a network.

Hence, it *is* sometimes justified to recover information about the type or structure of run-time entities, i.e., to do type casing.

9.1.3 How to Make Type Casing Safe

BETA supports a very general notion of type casing, in addition to the unsafe reference assignments, because patterns are first class values which can be obtained from many kinds of attributes and compared:

```
 1 findBus:
 2   (# theBus: ^bus
 3   do theVehicles.scan
 4      (#
 5      do (if current## <= bus## then
 6             current[]->theBus[];
 7             leave findBus
 8         if)
 9      #)
10   exit theBus[]
11   #)
```

Ex. 9-2

This pattern is a possible implementation of the `findBus` pattern from the previous section, and it should be understood in that context. It uses a few constructs which are characteristic of BETA standard libraries. We will explain the overall effects of executing this method, but not go into details about how the BETA standard libraries are implemented. We hope that this will be sufficient to make the example comprehensible also for those who have no experience with the BETA libraries.

All the `vehicles` in `theCompany` are stored in the `list` called `theVehicles`, so we must iterate over all the elements in that `list` in order to find a `bus`. That is the effect of executing `theVehicles.scan(#..#)`, line 3–9; `scan` is a control structure pattern supporting iteration that is defined for all `lists`. The most specific `do`-part of the control structure, line 5–8, is executed once for each element in the `list`, and in each iteration the variable object attribute `current` refers to the currently visited element. Hence, the `if` imperative will be executed once for each element of the `list`.

With each `current` element it is tested, line 5, whether or not it is an instance of a pattern which is less-equal than `bus`, and if that is the case then the identity of that object is reference assigned to `theBus`, line 6, the iteration is terminated, line 7, and the result is returned via `theBus`, line 10.

The only problem with this approach is that it is not safe. The test for the pattern relation and the reference assignment are separate actions, so there might be another thread which modifies `current` between the test and the assignment, and then the assignment might cause a qualification error even though that is exactly what the test is there to prevent. It would indeed be a contrived program where that would actually happen, but type checkers cannot assume that programs are non-contrived. They would not have any idea about contrivedness either, of course.

Consequently, it is necessary to generate a dynamic qualification check in the reference assignment, and emit a warning that it might fail at run-time.

There are a significant number of tests of this kind in the BETA program which implements `gbeta` (more about this program in Chap. 11), and it is a recurrent source of irritation that the “real” qualification checks (which are considered bugs that should be removed) sometimes slip by unnoticed because all these “safe” constructs cause so many warnings.

This creates a strong motivation for adding a construct which is actually safe, and which is recognized by the static analysis such that the semi-spurious warnings can be avoided. Luckily, this is no problem.

◇ The *`when` imperative* was added to gbeta in order to solve this problem. It is used in the following way, in a modification of the previous example which does the same thing, only safely:

```
1  findBus:
2    (# theBus: ^bus
3    do theVehicles.scan
4       (#
5       do (when it: current
6            // bus then it[]->theBus[]; leave findBus
7          when)
8       #)
9    exit theBus[]
10   #)
```

Ex. 9-3

The `if` imperative is now replaced by a `when` imperative, line 5–7. This syntactic construct, WhenImp, resembles the construct GeneralIfImp, as it can be seen in the full grammar for gbeta in App. A. Both have a structure similar to `case` or `switch` statements in other languages.

Just after the keyword `when` in line 5 there is a name declaration, named `it` in the above example. This declares the name which will be accessible under the given qualification of each case (marked by `//`) inside the `when` imperative. It is an object attribute, and not a variable one, so it cannot be made to denote any other object during the execution of the `when` imperative. Let us call the
◇ object that this name denotes the *target* of the `when` imperative, and let us call
◇ each AttributeDenotation which occurs after `//` the *guard* of that case.

The execution proceeds as follows: First an object is obtained by looking up the AttributeDenotation which is just after the colon (`current`, line 5, in the example); then the target name (`it`, also line 5) is bound to that object—it becomes the target; then the cases, each starting with `//`, are tested one by one in source code order, until the first time where the test succeeds; a test succeeds iff the pattern of the target is less-equal than the pattern of the entity denoted by the guard (`bus`, in the example); when a test has succeeded the Imperatives just after the `then` are executed, and the remaining parts of the `when` imperative are skipped. If all tests fail and there is an `ElsePart` then that is executed; if all tests fail and there is no `ElsePart` then no cases are executed, and no error is raised.

The Imperatives in each case are analyzed statically in an environment where the target is assumed to have a pattern which is less-equal than the AttributeDenotation that it was compared against, and that is of course safe because that piece of code is only executed when that is true. Note that it is essential for the soundness of the analysis that the target can *not* be reference assigned, and that it will therefore denote the same object for the entire duration of the `when` imperative.

There is also a case variant, marked by `//=`, where the pattern match must be exact. The run-time semantics and static analysis are modified accordingly for such a case. Note that the usage of `//=` on an object will mark it such that it cannot thereafter be dynamically specialized.

9.1.4 Non-static Type Casing

It might seem that type casing by nature is concerned with the test for applicability of static information, and nothing beyond that.

This is actually not true with the `gbeta` `when` imperative, because the semantics is to compare the actual (run-time) pattern of the target with the actual pattern of each guard. This makes a difference in that it is possible to obtain knowledge about the *relation* between two patterns without necessarily knowing any of them exactly during static analysis. For example:

```
1  transferAppropriate:
2    (# src,vardst: ^list
3    enter (src[],vardst[])
4    do (# dst: @vardst
5       do src.scan
6          (#
7          do (when elm: current
8                // dst.element then elm[]->dst.append
9              when)
10         #)
11      #)
12   #)
```

Ex. 9-4

The pattern `transferAppropriate` is a procedure which takes two `list`s as arguments, `src` and `vardst`, and inserts some elements from `src` into `vardst`, namely exactly those elements which can be inserted into `vardst` without violating the qualification constraint. Like a few times earlier, this example is easiest to understand for those that already know the BETA standard libraries, but it works as follows: The virtual pattern `element` is used as the qualification of all attributes in `list` which are supposed to refer to an element in the `list`. In particular, it is used for the arguments to methods such as `append`, which will add its argument as a new element at the end of the list. The pattern `src.scan(# ... #)`, line 5–10, will iterate through all elements of `src` as usual, and the `when` target `elm`, line 7, is the `current`ly visited element, which is appended to the second argument `list` iff it has an appropriate pattern.

As a technicality, we need to obtain a non-variable attribute which denotes the object referred by the second argument, `vardst`. This is obtained by using an extra object, described by the outermost MainPart, line 4–11, in the `do`-part of `transferAppropriate`, and declaring an object attribute `dst`, line 4, such that it denotes the object referred by `vardst` just after `transferAppropriate` is invoked. The `when` imperative then uses `dst`; if it had used `vardst` directly then the invocation of `append` would have been unsafe, and a warning would have been emitted. Note that a virtual object attribute (described in Sect. 9.3) could also have been used to obtain a non-variable denotation of the second argument, but that would cause a syntactic overhead at each call site and provide some extra constraints that we do not need here, so we preferred the more encapsulated approach in this case.

The method `transferAppropriate` is type safe, and it is polymorphic across all combinations of specializations of `list`. This is an example where the type casing construct does not reduce the flexibility or reusability of the source code. In contrast, it provides a type safe polymorphic functionality which cannot be matched by any of the languages where the support for dynamic type tests has been excluded from the run-time system—such as Cecil or GJ—and even if support for dynamic type tests were added to those languages then this particular kind of polymorphism could not be supported without the addition of existential types or another mechanism which would make it possible to even have a polymorphic reference to `list`s of all kinds. With the approach based on parametric polymorphism (and F-bounds, or not, that makes no difference), no such polymorphic reference can exist, because there is no subtyping relation between `list[`P`]` and `list[`Q`]` when $P < Q$.

9.2 Lower Bounds on Virtuals

A lower bound on a virtual attribute allows for more control over that fine balance between two incompatible dimensions of freedom which is characteristic of virtuals. A virtual attribute provides the programmer with a very valuable freedom, namely the freedom to elaborate on a pattern attribute, for instance in order to define a method in an abstract pattern and then implement that method in various concrete subpatterns; or in order to parameterize a pattern with a class, for instance the `element` attribute in `list` which is the qualification of elements contained in the `list`. However, a virtual attribute also *reduces* the freedom of programmers (who want to write type-safe code), because a virtual pattern generally varies along with the pattern of the enclosing object, i.e., it is covariant, and this means that it cannot be used in a contravariant position. For example, if an object is only statically known to be an instance of `list` or a subpattern of `list` then we cannot insert new elements into the `list`, because the actual qualification could be any subpattern of `object`, i.e., any pattern whatsoever.

Virtual attributes are generally known either by upper bound or exactly. A virtual attribute is introduced with a declaration like `v:<p`, may be further-bound a number of times with declarations like `v::<q`, and it may be final-bound with a declaration like `v::r`. Virtual introductions and further-bindings provide upper bounds, only. A final binding provides both an upper bound and a lower bound and thereby freezes the virtual attribute to be one particular pattern (at least if all the declarations for that virtual gave contributions which are compile-time constant patterns, but not, e.g., if the virtual is final-bound to an open virtual). The static knowledge is in all cases either an upper bound

(so it may be *any* pattern less-equal than the bound), or it is a fixed pattern.

An explicit lower bound on a virtual can help establishing a more flexible limit on the covariance than the entirely frozen limit of a final-binding, and it is at the same time more strict than the upper bound alone. This makes it possible to ensure type safety in cases which would be hard to handle without this tool. To give a concrete example we will need a few patterns; the central pattern for this example is shown in Fig. 9.1. It is the pattern `ordered`, which is used to represent values which are organized into some total order relation. This example is a variation of a Cecil example given in [64]. This example has already been discussed in Sect. 4.4, but there are additional aspects now.

One of the main topics of this example is the notion of *binary methods*, ◇ namely methods which take an argument of the same type as the receiver of the message. This is hard to handle if it is combined with subtype polymorphism, because it is then typically only known that the two involved objects have types T_1 and T_2 which are less-equal than a given upper bound T, and that says nothing about whether or not $T_1 = T_2$. Assuming that all combinations of T_1 and T_2 are acceptable we may just choose the most specific type T_0 such that $T_1 \leq T_0$ and $T_2 \leq T_0$ and then execute the method which expects two arguments of type T_0. That is exactly what multi-methods like in Cecil will do, so the use of a multi-method is one possible solution to the problem of handling binary methods safely. This approach will of course not work if it is required that $T_1 = T_2$. Other approaches discard the subtyping polymorphism, but retain the potential for implementation inheritance by separating subtyping and inheritance (which is in this case called matching [16]).

Binary methods have been treated in several contexts, for example [16, 14, 15]. Usually they are concerned with the case where two arguments to a procedure must be of the same type, or the receiver and an argument of a message must be of the same type. This can for example be made type safe by ensuring that the exact type of both of the involved objects is known; with matching this is achieved by removing the support for subtype polymorphism. In this context we widen the concept such that the acceptable pairs of arguments must be in the same *family* of patterns, i.e., some combinations of different patterns are acceptable whereas others are not.

The pattern `ordered` uses the virtual attribute `cmpType` to define the patterns which are expected to be known for comparisons. Initialization of instances of subpatterns goes into further-bindings of `init`. Comparisons are supported by the method `lessEqual`, and it is possible to select of the greater of two `ordered` objects using the method `max`. Finally, the method `asString` makes it possible to obtain a `string` which describes the given `ordered` object.

The virtual pattern `cmpType` deserves closer consideration. Inside `ordered`, this attribute is not known exactly, but it is intended to be known exactly from the outside, when actual instances are to be compared. There will be a final bound on it in these cases. However, since `cmpType` is an open virtual as seen from inside `ordered`, we would not be able to implement the `max` method safely without the lower bound on `cmpType`. Assume that this lower bound were removed; in that case the reference assignment in line 12,

```
ordered:
  (# cmpType:< ordered :> this(ordered);
     init:< (# do INNER exit this(ordered)[] #);
     lessEqual:<
       (# other: ^cmpType; b: @boolean
       enter other[] do INNER exit b
       #);
     max:
       (# other,maxi: ^cmpType
       enter other[]
       do (if other[]->lessEqual
           then this(ordered)[]->maxi[]
           else other[]->maxi[]
          if)
       exit maxi[]
       #);
     asString:< (# s: @string do INNER exit s #)
  exit this(ordered)[]
  #);
```

Figure 9.1: The pattern `ordered` relies on a virtual lower bound

`this(ordered)[]->maxi[]`, would not be type safe, because the qualification of `maxi` (which is `cmpType`) could be any pattern less-equal than `ordered`.

Now consider the situation where `cmpType` has `this(ordered)` as a lower bound, such as the example is actually written in Fig. 9.1. The lower bound has no run-time semantics, it is only a restriction which makes some programs illegal (which is unusual for a gbeta construct). The effect of the lower bound on the analysis is two-sided. Firstly, the analysis is allowed to assume that the lower bound is respected whenever the pattern `cmpType` is used—so it is safe to reference assign `this(ordered)` to `maxi` because the qualification of `maxi` by the lower bound is guaranteed to be greater-equal than the pattern of `this(ordered)`. Secondly, in every pattern that inherits `cmpType` it is checked that any new *upper* bounds, which are applied to `cmpType` by further- or final-binding declarations, are actually compatible with the given lower bound. In other words, no matter what happens to `cmpType` it must remain a superpattern of the pattern of its enclosing object. Of course, that is exactly what is needed in order to ensure soundness of the abovementioned assumption which is made wherever `cmpType` is used.

The intention with `cmpType` is that it should present a well-defined *partial view* of instances of subpatterns of `ordered`. This is exactly what the lower bound says, and we expect that this may be a quite useful feature in practical programming.

In the concrete example we use it to make comparisons between certain ordered objects type safe and to detect and reject all other comparisons. To do this we divide the subpatterns of ordered into families. The members of these families should be comparable freely and safely amongst each other, but members of different families should not be compared, and there should be a compile-time complaint if anybody tries to do such a thing. The families we will consider are the one-member family text, and the family of number and its two subpatterns int and float. The source code for these will be shown below, but let us first consider a seemingly natural approach and explain why it can *not* be done in that way.

As explained in [64], a simple approach would be to let the qualifications of the arguments to lessEqual and max be ordered in stead of cmpType, thus removing the need for cmpType entirely:

```
ordered:
  (# init:< <<as in Fig. 9.1>>
     lessEqual:<
       (# other: ^ordered; b: @boolean
       enter other[] do INNER exit b
       #);
     max:
       (# other,maxi: ^ordered
       enter other[] do <<as in Fig. 9.1>> exit maxi[]
       #);
     asString:< <<as in Fig. 9.1>>
  exit this(ordered)[]
  #)
```
Ex. 9-5

With this design we can actually implement max safely, but lessEqual could then be called with arbitrary ordered objects, so this would require that we either choose a way to compare a number and a text, or that lessEqual implement certain comparisons, for instance of two numbers, and raise an exception in the remaining cases, such as comparing a number and a text. In the first case we would have to implement meaningless or contrived comparisons, and that was exactly one of the things we wanted to avoid. In the second case we would have to expect potential run-time errors at every comparison, and that is of course not desirable. In other words, this simple approach is not appropriate. Hence, in the following we will assume the definition as it appears in Fig. 9.1.

The definition of the text pattern which forms the first family all by itself is as follows:

```
text: ordered
  (# cmpType::text;
     init::(# enter value #);
     lessEqual::(# do (other.value<=value) -> b #);
     asString::(# do value->s #);
     value: @string
  #);
```
Ex. 9-6

This definition of text final-binds cmpType to text, thereby making it possible to safely compare instances of text or a subpattern of text with each other.

Moreover, the virtual methods are implemented such that `text` can be used as a concrete class. The other family is rooted in `number`:

```
number: ordered
  (# cmpType::number;
     lessEqual::(# do (other.asReal<=asReal) -> b #);
     asReal:< (# r: @real do INNER exit r #)
  #);
```

Ex. 9-7

Again, `cmpType` is final-bound to `number` in order to make all `number`s comparable. We introduce the method `asReal` which is needed in the implementation of `lessEqual`; this implementation assumes that conversion to and comparison as real values is appropriate for all kinds of `number`s. Other choices might be better, but this is simple and not unreasonable. The other members of the family are `int` and `float`:

```
int: number
  (# init::(# enter value #);
     asString::(# do '<anInt>'->s #);
     asReal::(# do value->r #);
     value: @integer
  #);
float: number
  (# init::(# enter value #);
     asString::(# do '<aFloat>'->s #);
     asReal::(# do value->r #);
     value: @real
  #)
```

Ex. 9-8

These patterns just add implementations of various methods. With all these patterns in place we can begin to use some `ordered` objects. The following MainPart is assumed to be in a context where `ordered` and its subpatterns are available:

```
1  (# t1,t2: ^text;
2     n1,n2: ^number;
3     r: @real
4  do
5     'Hello, '->(&t1).init;
6     'world!'->(&t2).init;
7     (if t1->t2.lessEqual then t1.asString->puttext if);
8     (t1->t2.max).asString->putline;
9     3.14159->float.init->n1[];
10    4->int.init->n2[];
11    (n1->n2.max).asReal->r
12 #)
```

Ex. 9-9

In this example, a couple of `text`s and `number`s and an auxiliary attribute are declared in line 1–3. In line 5 and 6 the two variable `text`s are renewed and initialized with the two literal strings. The two `text`s are then compared in line 7 and the string value of `t1` is printed if `t1` is less-equal than `t2` (and it is). Next, the greater of the two `text`s is selected by `max`, and the string value of that `text` (which is `'world!'`) is printed; that was line 8.

In line 9, a new `float` object is created and initialized with a π-like value, and `n1` is made to refer to that `float`; similarly, line 10 creates a new `int` object and initializes it with the `integer` value four. Finally the greater of the two numbers is selected using `max` in line 11, and the `real` value of that number is assigned to the auxiliary `r`.

In [64] it is explained how F-bounded polymorphism can handle this example. With F-bounded polymorphism it is possible to let `ordered` be a parameterized class whose type argument is used in F-bounds, like in this Pizza example:

```
interface Ordered<T> {
  bool lessEqual(T other);
  T max(T other);
};
class Text implements Ordered<Text> {
  bool lessEqual(Text other) { ... };
}
class Number implements Ordered<Number> {
  bool lessEqual(Number other) { ... };
}
```
Ex. 9-10

This does enable instances of `Text` to be compared to each other in a type safe manner. It also allows subclasses of `Number` to be defined and instances thereof compared, and it does make it a static error to compare a `Text` and a `Number`, as desired. The implementation of the `max` method could not be given in `Ordered` in Pizza because `Ordered` is an interface, but it can be given at the abstract level in Cecil:

```
forall T where T isa Ordered[T]:
  method max(x:T, y:T): T { if x>=y then x else y }
```
Ex. 9-11

In Cecil, methods can be defined as infix operators like '>=', so the expression `x>=y` corresponds to an invocation of `greaterEqual` in the other languages. This implementation is type safe because the compiler will only accept invocations of `max` with such arguments where there is a type `T` to which the actual arguments are known to conform and such that `T` is less-equal than `Ordered[T]`.

To summarize, both `gbeta` and Cecil have the ability to safely support inter-family comparisons for different families of subpatterns of `ordered`, respectively instantiations of `Ordered[T]`, and at the same time statically detect attempts to compare across families. Moreover, this happens in a way that allows a type safe implementation of such a method as `max` which is shared between *all* `ordered` objects—without that constraint there would not be much of a challenge because each family could then be implemented from scratch as unrelated class hierarchies, with a syntactically identical implementation of `max` in the root of every family.

The most important differences are that Cecil infers better return types for `max`, and `gbeta` allows for a more complete kind of dynamic polymorphism. For the first issue, Cecil will infer that the maximum of two `float` objects is again a `float`,[2] whereas `gbeta` will only have a return type of `number`. This

[2] If `Ordered[T]` is declared to be *contravariant* in `T`, as explained in [64, p. 401].

difference is caused by the fact that Cecil uses unification of actual arguments with formals at every call site in order to infer type arguments, whereas gbeta uses virtual attributes for type parameterization. This is discussed in more detail in Sect. 9.3. On the other hand, gbeta supports polymorphic access to `ordered` objects:

```
(# o: ^ordered
do (if ...
     then 3.14159->float.init->o[]
     else 'world!'->text.init->o[]
    if);
    o.asString->putline
#)
```
Ex. 9-12

This would not be possible in Cecil (or any other approach based on F-bounded polymorphism or simple type parameters), since it is not possible to have an attribute which is capable of referring to *all* instances of `Ordered`—the problem is that `Ordered` is not a type but a function from types to types, so it cannot be the type of a reference.

9.3 Virtual Objects

Virtual objects were originally invented by Mads Torgersen, and he developed the idea in his progress report [112]. We have adopted this concept in its original form and given it a precise semantics that fits into gbeta. Virtual objects turned out to solve a longstanding problem in the type systems of both BETA and gbeta: Without virtual objects, neither BETA nor gbeta can handle methods with arguments or returned results whose types are polymorphic and depend on each other. Virtual objects support these cases in a general way, even though the syntax is more verbose than what one might expect or desire for a parameter passing mechanism. There may also be other beneficial uses of virtual objects, but we have not had the time to experiment with them so this treatment will concentrate on the parameter passing viewpoint.

Section 9.3.1 explains why languages like BETA and gbeta are not optimized for implementation reuse as thoroughly as some other languages; this is the core of the motivation for introducing virtual objects. Section 9.3.2 presents one specific problem where the need for improvement in the support for implementation reuse is particularly evident, namely the problem of handling method signatures with interrelated type variables. It is explained why this is inherently difficult to do in languages like BETA and gbeta. Finally, Sect. 9.3.3 shows how parameter passing by means of virtual objects handles the problem.

9.3.1 Functional Languages and Implementation Reuse

The Hindley-Milner style types [92] which are used in functional languages such as Haskell are particularly convenient for giving examples of signatures where different argument and result types may be chosen rather freely but only in ways which exhibit certain patterns of relations between the types. For example, consider the standard function `map` which takes a function f and a list l as arguments and returns another list containing the images by f of the elements in l. This function has the following type:

$$\forall\alpha,\beta\,.\,(\alpha \to \beta) \to [\alpha] \to [\beta]$$

This type specifies that `map` can be applied to any function f and list l for which there exists a choice of values for the type variables α and β such that f is a function from α to β and l is a list of values of type α; the result is then known to be a list of values of type β. Wherever `map` is used, the type analysis will combine the above type of `map` with the types of nearby expressions by means of a unification process which determines how (and if) it is possible to consistently choose values for all the type variables [92].

This approach enables a type safe reuse of a given function implementation for all those types of arguments where a choice of values for type variables is possible, i.e., for all those types of arguments which are known to have a certain minimal top-down structure. The implementation of `map` will work correctly no matter what structure the elements in the list have, it just has to be a list of *something*, and the function argument can be called on arbitrary values as far as `map` is concerned. There are some internal relations between the arguments—that the function argument type α must be the same as the type of the elements in the list, and that the result type $[\beta]$ can be computed from the function type $\alpha \to \beta$. These internal relations are irrelevant for the implementation of `map`, but the consistent use of such internal relations in composite types allow for very well-studied and solid type correctness properties for programs as a whole.

In this tradition, the highest priority is given to reusing function implementations as widely as possible. For example, there should only be one `map` function which would then be reused for all the possible choices of values for type variables. The `map` function will actually work just fine in all those cases, and having several structurally identical implementations of the `map` functionality—e.g. one for each choice of type variables—would be confusing and redundant. The ideal is that programming language entities such as functions should be pure structure, purely transparent, carrying no other properties than the minimum which is derived from the language semantics. So for example, the "meaning" of `map` is just to apply a function to all elements in a list, no less and in particular no more.

The natural consequence is that structural type equivalence is considered "correct" in the functional language community, and anything else is seen as a compromise which must be rooted in performance considerations or sheer bad judgment; types should be computable (if we have a function type and a list type then we can take them apart and recombine the elements to make another list type); and the structure of composite values should not be taken to mean anything but that structure, and we can of course take composite values apart and recombine them any way we like. We might use the phrase 'no nonsense' to describe this mindset, noting that the nonsense is everything beyond the operational semantics of the language. Functional languages are really very well optimized along these lines, and the topic of virtual objects is all about obtaining some of the benefits of this approach.

However, gbeta has grown out of an entirely different mindset. In the gbeta approach the highest priority is given to how well the programming language will support programmers in the construction of complex systems. We are less concerned with the fact that a given implementation may be redundant in a purely operational, semantic sense—the implementation should not play such a dominant role. In contrast, we find it important to support programmers in the construction and use of program entities based on an understanding of the role and purpose of those program entities which goes far beyond the narrow semantic properties and into the minds of people. Hence, a composite entity is *more* than the sum of its parts. The programming language should support programmers in thinking about composite entities as meaningful in and of themselves and containing parts that make sense in their context, rather than thinking about composite entities as arbitrary conglomerates whose parts may be repackaged freely into other conglomerates.

Consequently, languages like BETA and gbeta are actually quite poorly optimized for a consistent and ubiquitous exploitation of opportunities for implementation reuse based on purely structural and operational similarities. A typical example where this seems to create problems is with container data structures such as `list`s and `set`s. The problem is that these languages insist that a `list`-of-`integer` pattern must have its own unique identity, different from any other `list`-of-`integer` pattern which happens to be defined by some other possibly syntactically identical declaration. As far as both are simply considered as lists of `integer` objects without any further meaning, this is a clear example of harmful and confusing redundancy, and it is inconvenient in large projects to have to standardize on using one particular declaration in order to avoid having several type-incompatible `list`-of-`integer` patterns.

However, in practice this does not seem to be a problem, because the generic `list` functionality can be implemented as methods on the generic `list` pattern and the concrete lists tend to be used only very locally in the implementation of other patterns which do actually carry a meaning of their own. It seems that "meaningless" concepts such as `list`-of-`integer` should preferably be kept hidden inside the implementation of meaningful patterns, because the requirement that an entity be meaningful becomes ever more urgent the larger the universe is where it is intended to be known and used. This means that it is not a problem in practice that two `list`-of-`integer` patterns are distinct, because there will not be any source code which uses both of them. Hence, we do not intend to change or complicate the design of gbeta in any profound way in order to provide structural equivalence between (some) patterns. Besides, container data structures seem to be an exception where structural equivalence is "obviously" desirable, other examples come up rather seldom and seem less compelling.

There is another case where the need to improve on the support for implementation reuse in gbeta is more acute. This case is in some sense inverse to the problem of excessive uniqueness of container data structures—we might say that a structural equivalence on container data structures would allow us to use many different entities in the same context, but this improvement is about allowing one entity to be used in many different contexts. The problem is that the static analysis *and* the run-time semantics of Beta and gbeta do not support the capture of regularities such as the multiple occurrences of a type variable in a Hindley-Milner style polymorphic type, and hence it is hard to write routines which are parametrically polymorphic in the same sense as the map function in a functional language. The next section will detail why this is a hard problem in Beta and gbeta and outline some partial solutions.

9.3.2 Interrelated Types In a Method Signature

The map function mentioned in the previous section is actually a good example of interrelated types in a signature, but the creation of a list of elements of type β (which was the result type for the function argument) requires structural type equivalence in order to be useful; otherwise the returned list could not be accessed as a list of elements of type β, only as a list of bare objects, because the actual pattern of the result would be known only inside the implementation of map. Hence, we will consider an example which does not "repackage" composite entities into other composite entities. As explained, this is something that gbeta is not optimized for.

Let us assume that we want a generally applicable procedure which can extract an element from a list. This problem is simple and still requires the handling of polymorphically interrelated types which is hard in Beta and gbeta. We can easily create a partial solution to this problem with a procedure which uses subpattern polymorphism to handle all kinds of lists:

```
getElement:
  (# theList: ^list;
     theElement: ^object
  enter theList[]
  do theList.first.elm[]->theElement[]
  exit theElement[]
  #)
```
Ex. 9-13

This procedure receives theList as an argument and then uses the method first to extract the first element from the list. In the standard Beta list we have to access the elm attribute of the result returned by first to get hold of the element itself, so we do that; the element is then reference assigned to theElement so that we can deliver it as the result of getElement.

This design is type safe and it will actually deliver an element from the list as desired (ignoring possible errors such as 'List is empty'). The problem with this design is that it needlessly throws away static information about the properties of the element: If we use getElement to extract an element from a list of integer objects then the result will certainly be an integer, but the static analysis will claim that we only know that it is an instance of object or a subpattern. Since statically provable information about run-time entities is a valuable resource it is important to try to avoid such a loss.

One way to do this is to have a separate copy of getElement for each specialization of list. Note that this is exactly what we do—even though there is only one copy of the syntax—if we change getElement from a (stand-alone) procedure to a method of list:

```
list:
  (# element:< object;
     ...
     getElement:
       (# theElement: ^element
       do first.elm[]->theElement[]
       exit theElement[]
       #)
  #)
```
Ex. 9-14

Since a method of a list object depends on that list it is possible for the static analysis to retain the information about the qualification of elements in that particular list, so getElement on a list whose element attribute is known to be integer would be known to deliver an integer, not just an object.

This technique uses contextuality to remove the need for the equivalent of call-site instantiations of parametrically polymorphic functional types like $\forall\alpha\,.\,[\alpha] \to \alpha$, and it may be applicable in many cases. It is also specifically object oriented because it builds on context dependency as opposed to parameterization. However, it does not provide us with genuine parametric polymorphism because all those getElement methods, one in each list, would be distinct and non-interchangeable both in the static analysis and in the dynamic semantics. In a functional language we could actually have *one* single function with the type $\forall\alpha\,.\,[\alpha] \to \alpha$, and that function could be used for all choices of α. For simple calls it makes little difference whether we have many distinct getElement methods or just one getElement procedure, but if we wanted to get an element from many lists and getElement were a method then we could not obtain one single getElement run-time entity and reuse it with all of those lists.

There is a profound reason why such a polymorphic entity could not be created in an imperative language such as Beta or gbeta. The reason is that functional languages use immutable bindings of names to values whereas Beta and gbeta use destructive assignments to variable entities. Because of this, it is actually possible to view the type analysis and type inference of functional languages as a ubiquitous data flow analysis which delivers the results in the form of types of functions. It does not exactly determine the flow of data, but it does establish some connections between the types in type variable expressions which can only be proved because the flow of data is so restricted.

In the functional equivalent to our getElement procedure the type system knows that the returned element is of type α for such an α that the argument is of type $[\alpha]$, which almost amounts to knowing that the element actually came from that list; in the Beta and gbeta approach it is only known that every operation respects all safety invariants (in particular that a variable object conforms to its qualification). Hence, as far as the type checker is concerned there is absolutely no way we could use knowledge about the object referred by theList to conclude anything about the object referred by `theElement`. This is indeed a necessary restraint because the language allows multiple destructive assignments to both `theList` and `theElement`, and it would simply be unsound to assume anything beyond the respect for safety invariants. Hence, if we want the type system to prove that a call to `getElement` returns an `integer` then we must ensure that the qualification of `theElement` is `integer`—there is no other way. This again means that it cannot be the same pattern as one which provably returns, say, a `string`.

In the next section we will describe how virtual object attributes provide a solution to the original problem of handling call-site specific parameter/result type interdependencies, in spite of the fact that there is no way we could use the exact same pattern at all those call sites.

9.3.3 Virtual Object Attributes

It is clear that we would not—without fundamental design changes in the language—be able to create a single pattern which would exhibit the parametric polymorphism that allows it to provably return an `integer` when used in one context and a `string` when used in another context. In other words, patterns cannot exhibit parametric polymorphism in the sense that a function in, e.g., Standard ML does it. We should mention that the *combination* of mutability and parametric polymorphism is not supported in the functional languages either; one example is that methods in Objective CAML must be monomorphic, so for example the polymorphic function `map` could not be a method without being restricted to one particular choice of values for the type variables.

Hence, we must be prepared to create a distinct pattern at each call site. Moreover, we need to have a mechanism that allows us to declare that a given object (`theList`) will be available, such that it can be used in the implementation (of `getElement`), and we need to ensure that this object attribute is non-variable. An ordinary object attribute would have exactly these properties, but since it is immutable it does not work as a parameter—there is no way to do parameter passing at the call site. The notion of a *virtual object* attribute solves ◇ exactly this problem by providing an object attribute which can be introduced and used in the implementation and final-bound to an actual object in a subpattern, i.e., at each call site. For example, we can define a version of `getElement` which receives its `theList` argument using a virtual object attribute:

```
getElement:
  (# theList:< @list;
     theElement: ^theList.element
  do theList.first.elm[]->theElement[]
  exit theElement[]
  #)
```
Ex. 9-15

A virtual object is declared with the same syntax as a virtual pattern, except that an '`@`' sign is added to make the new attribute kind different from all others and yet make it similar to an ordinary object attribute declaration. There is no further-binding declaration, but there is an introduction declaration marked by '`<@`' (e.g., '`theList:<@list`'), and a final-binding declaration marked by '`:@`' (e.g., '`theList::@myIntList`'). When a virtual object parameter is to be passed at a call site it must be done by means of a final-binding of that virtual object:

```
(# myIntList: @list(# element::integer #);
   i: ^integer
do getElement(# theList::@myIntList #) -> i[]
#)
```
Ex. 9-16

There are some disadvantages with this parameter passing mechanism. It is significantly different from—and more verbose than—the usual AssignmentEvaluation based parameter passing style. The parameters are identified by name and not by position, so the type analysis will not warn you if you ought to give three arguments and forgot one of them. It is a non-trivial design question whether it should be considered an error to create an instance of a pattern that has one or more virtual objects *without* a final bound. To make it an error would seem to improve the support for automated bug-detection. However, it would also be a serious fault in context of the general style of BETA and gbeta where there are no "abstract" entities, so for example it is possible to create an instance of *any* pattern whatsoever. If it were made an error then it would make the language unsafe, because there are so many cases where an entity access is subpattern polymorphic that it would make practically every statement a potential run-time error if *some* patterns could not be instantiated. Consequently, we decided to use the same rules as for ordinary object attributes, i.e., if a virtual object is not final-bound to an object then the bounding pattern is used to obtain a fresh object. It would probably be good to supplement this with a compile-time warning in the cases where it is statically known that one or more virtual objects have not yet been final-bound.

Finally, it breaks the transparency in the sense that `getElement` is known to be used as a pattern because it is the superpattern in an ObjectDescriptor, and the ObjectDescriptor itself will always denote a pattern, not an object. In fact, there is no way we could have avoided to break the transparency because we must use a pattern which is different from `getElement`, and there must be an explicit construct somewhere which creates this pattern.

Nevertheless, virtual objects provide a mechanism which is consistent with the rest of the language, both semantically and syntactically. It does support the generic specification of type relations between arguments and results, and at each call-site it is statically checked that the specific binding to arguments actually exhibits the required type relations.

Now let us reconsider the `map` function which we deemed impractical in gbeta because it needed to create a list from a pattern which would not be known outside `map`. There are actually other ways to express a similar functionality as that of `map` which are much better adapted to the imperative context:

```
map:
  (# src:< @list; dst:< @list;
     f:< (# s: ^src.element; d: ^dst.element
         enter s[] do INNER exit d[]
         #)
  do dst.clear;
     src.scan(# do current[]->f->dst.append #)
  exit dst[]
  #)
```
Ex. 9-17

In this pattern, the two lists are both provided from the outside, such that they may have useful types, and the function `f` is made a virtual pattern whose argument and return types match up with the two `element` virtuals in the `lists`. It can be used like this:

```
(# l1: @list(# element::int #);
   l2: @list(# element::text #)
do
   map
   (# src::@l1; dst::@l2;
      f::(# do s.asString->text.init->d[] #)
   #)
#)
```
Ex. 9-18

It is necessary to either learn the names `src`, `dst`, and `f` and the role they play in the use of `map`, or to have programming environment support for the creation of skeleton expressions which just need to be filled in. However, our experience with such constructs as `scan` on collection data structures is that it is easy to remember how to write it and to understand it on a later reading. Thus, we cannot precisely imitate the functional approach for such a function as `map`, but we can create a native variant which seems to be quite acceptable.

To illustrate yet another aspect of the virtual object mechanism, consider again the example with a `company` pattern that contains two nested patterns called `employee` and `project`, see Fig. 5.2 on page 113. Using these patterns we may for instance write a procedure which brings together two instances of `company` matched up with a `project` and a representative `employee` for each:

```
synchronizeProjects:
  (# company1:< @company;
     company2:< @company;
     employee1: ^company1.employee;
     employee2: ^company2.employee;
     project1: ^company1.project;
     project2: ^company2.project
  enter (employee1[],project1[],employee2[],project2[])
  do ...
  #)
```
Ex. 9-19

This procedure can only be given parameters whose types match in the following way: The virtual object parameters `company1` and `company2` can be arbitrary instances of `company` or a subpattern, but once these two have been final bound they determine the qualifications of the remaining attributes. Hence, employee1 and project1 must be contextually located in company1, and similarly for employee2, project2, and company2. This means that the arguments are organized into two groups, each group having the statically guaranteed consistency property that the members of the group "belong together". This illustrates that even though virtual objects are in some ways less elegant than the call-site unification of type expressions in functional languages, they do allow for some kinds of statically checkable consistency properties which are unparalleled in those functional languages, and indeed in *all* languages without support for general contextuality.

9.4 Disownment of a Virtual

Disowning a virtual attribute means giving the promise that *you* do not intend to further- or final-bind it. As a result, there will not be a conflict when *somebody else* final-binds it, and hence there is no need for the compiler to complain about those final-bounds.

To enrich this explanation with a little more technical content, let us consider the situation where virtuals cannot be disowned and see what problems this causes. Assume that we have a pattern which contains two nested virtuals and a subpattern of the outermost virtual:

```
a: (# v:< (# w:< object #);
      b: v(# w:: integer #)
   #)
```
Ex. 9-20

This is not a legal gbeta pattern. The reason is that we can create a conflict by means of a seemingly innocent subpattern:

```
c: a(# v::< (# w::< string #)#)
```
Ex. 9-21

The conflict is that the pattern b inside c will contain a w virtual with contradictory requirements. First, the introduction w:<object tells users of w that it is a pattern which is less-equal than object; no problems with that. Second, the final bound w::integer tells users of w in context of an instance of b that it is exactly the pattern integer. This means that it is safe to reference assign an integer object to a variable object attribute with w as qualification. Now, since c further-binds v and w, and since b inherits w from v, the w in a b in a c must live up to the following constraints:

- it must be exactly integer, as promised with the final-binding
- it must be less-equal than string, because of the further-binding

There is no way we could satisfy both of these constraints, so the situation as a whole must be reconsidered and something in it deemed illegal.

Since c is derived from a we would consider it unnatural to declare a illegal *because of* the existence of c. It would be very confusing if some patterns from a third party library would be used and work perfectly well for some time and then suddenly would cause a compile-time error because a new—innocently looking—subpattern of a library pattern was added to the system. Generally, in any language, it would be incompatible with separate compilation if the creation of a subclass C' of a class C could make C invalid, because C would already be compiled when C' is added and the compiler should not be required to recheck C when subclasses of C are defined—the source code for C may not even be available.

On the other hand, it would be unacceptable to declare `c` as illegal, because the conflict is between the further-binding of `w` in `v` and the final-binding of `w` in a *subpattern* of `v`. Since the fragment system (see Chap. 10) makes it possible for such a subpattern to be located in a different file than both `a` and `c`, it would not be possible to detect the problem before link-time, because the compiler would never see the conflicting patterns, here `b` and `c`, together. Consequently, we could not make `c` illegal *because of* the existence of `b`.

However, we can make `a` illegal without even considering `c`, and this leads to a rule which is simple and safe. The solution we decided to use was that of making the final-binding of `w` illegal:

It is not allowed to final-bind a virtual
which is inherited from an open virtual.

Unless it is *disowned*, that is. With this rule in place it is convenient to be able to shift the burden from one set of shoulders to another, and that is exactly what a disownment mark, a '`-`', is designed to do: ◇

```
a: (# v:< (# w:< - object #);
      b: v(# w:: integer #)
   #)
```
Ex. 9-22

The '`-`' on the virtual introduction `w:<-object` means that any further- or final-binding of `w` in `v` is declared illegal: If a pattern like `c` above is created then it will be rejected at compile-time because it violates the disownment. This allows programmers to explicitly choose one of two paths. The default path is where a virtual w inherited from an open virtual v cannot be final-bound such that v can further-bind w without danger of conflicts. On the alternative path the virtual is explicitly "given away" (disowned) to subpatterns so they can final-bind it, because the enclosing virtual will not be allowed to further- or final-bind it.

One case where disownment is necessary is in the example given as Fig. 9.2. This example is a solution to the so-called *expression problem*, named by Philip Wadler but known from various contexts over several years. This problem has been discussed extensively on the Java genericity mailing list where such people as Philip Wadler, Kim Bruce, Didier Remy, Jacques Garrigue, Gilad Bracha, Matthias Felleisen, Shriram Krishnamurthi, Kresten Krab Thorup, Mads Torgersen, and I have participated in the discussion. The original presentation of the expression problem is shown in App. C. ◇

The tentative solution given in App. C actually turned out to be impossible to type check, as Philip Wadler explained on the mailing list in a later message. The adjustment of the type checking algorithm for GJ which is mentioned in section 2 of this presentation turned out to be non-trivial, because it would require a type judgment which is contrary to what is actually known. The problem is that the type argument `This` which is used to play the role as the type of the current object (a `Lang` or a `Lang2`) is restricted by an F-bound, e.g. `This extends Lang2<This>`. As a consequence `This` is known to be a *subtype* of the type of the current object (`This`$\leq$`SelfType`), but the reverse subtyping relation (`SelfType`$\leq$`This`) is not known and would not be sound to assume (the instantiation `Lang<This>` in the superclass clause for `Lang2` actually violates it), so `This` cannot be considered a true `SelfType`. Since GJ does not have a primitive notion of `SelfType`, and since F-bounded polymorphism is not capable of expressing it (we want a fix-point but can only select for pre-fixed points), the example must be rejected by the type analysis. The concrete location where it fails is in `Lang2.forPlus` where it is attempted to reference assign an object of type `Eval` to a reference (an argument of `e1.visit`) with type `This.Eval`, and since all inner classes are potentially covariant and only `This`$\leq$`SelfType` is known, this is not type safe.

Even though a solution in GJ seems to require additional investigations, there are statically type checked solutions in gbeta and Objective Label. Objective Label builds on Objective CAML and adds support for labeled arguments, polymorphic variants, and first-class parametric polymorphism [48]. We will return to Objective Label after considering the gbeta solution in Fig. 9.2. This solution defines two patterns, `Lang` and `Lang2`, implementing two tiny "languages" of expressions. Expressions in `Lang` can only be numbers, `Num`, and expressions in `Lang2` can be numbers (`Num` is inherited from `Lang`) and `Plus`, which is a sum with two operands which are other expressions in the language. One of the challenges of the expression problem is to be able to create new languages which add more kinds of expressions, and that is just what `Lang2` does with `Plus`.

For these expressions there are a set of possible actions, each action being implemented by a subpattern of `Visitor`. Actions are carried out by executing `visit` on an expression with a `Visitor` as argument (see `Exp` in `Lang`). Note that `visit` needs to use a virtual object for its argument, `theVisitor`, in order to be able to return an object whose statically known qualification depends on the actual argument to which `theVisitor` is final-bound. Also note that there is a disownment mark on `theVisitor`, such that methods which inherit from `visit` will be allowed to final-bind `theVisitor`. It is no problem that `visit` thereby also promises *not* to final-bind `theVisitor`—`visit` is not designed to be called, but to be used as a superpattern for invocations, since the parameter transfer is based on writing an ObjectDescriptor at the call-site.

Another challenge in the expression problem is to be able to add new actions, i.e., new subpatterns of `Visitor`, to languages. The only action in both `Lang` and `Lang2` is `Eval`, but the modularized version of the expression problem in Chap. 10 shows how a new action can be added to both `Lang` and `Lang2` in a

```
Lang:
  (# Visitor:<
       (# R:< - object;
          for1: (# result: ^R do INNER exit result[] #);
          forNum:< for1(# val: ^integer enter val[] do INNER #)
       #);
     Eval:< Visitor
       (# R:: integer;
          forNum::< (# do val[]->result[] #)
       #);
     Exp:
       (# init:< (# do INNER exit this(Exp)[] #);
          visit:<
            (# theVisitor:< - @visitor;
               result: ^theVisitor.R
            do INNER
            exit result[]
            #)
       exit this(Exp)[]
       #);
     Num: Exp
       (# val: ^integer;
          init:: (# enter &val #);
          visit:: (# do val[]->theVisitor.forNum->result[] #)
       #)
  #);
Lang2: Lang
  (# Visitor::<
       (# forPlus:< for1
            (# e1,e2: ^Exp enter (e1[],e2[]) do INNER #)
       #);
     Eval::<
       (# forPlus::<
            (# fst,snd: ^integer;
            do e1.visit(# theVisitor::@this(Eval) #)->fst[];
               e2.visit(# theVisitor::@this(Eval) #)->snd[];
               fst+snd->&result
            #)
       #);
     Plus: Exp
       (# e1: ^Exp; e2: ^Exp;
          init:: (# enter (e1[],e2[]) #);
          visit::(# do (e1[],e2[])->theVisitor.forPlus->result[] #)
       #)
  #)
```

Figure 9.2: The Expression Problem in gbeta

separate file. We could of course have added this new Visitor to Lang2, like in App. C, but the separate file is just an even more flexible approach.

The implementations of visit in each kind of expression calls the method for$X$ on theVisitor, thereby enabling it to perform an action which is specific for the kind X; for instance, visit in Num executes theVisitor.forNum such that theVisitor can visit that Num expression in a Num specific manner. This is a well-known technique which is called double dispatch, or it is called the 'Visitor' design pattern [46]. The forNum method in Eval in Lang is simple, it just returns the integer object in that Num; the forPlus method in Eval in Lang2 visits both operands to evaluate them, and then adds up the returned values and returns an integer whose value is that sum. In these invocations of visit the returned value is the identity of an object with qualification this(Eval).R, which is a virtual that is final-bound to integer in Eval in Lang. Hence, it is known to be type safe to reference assign that result to the local attributes fst and snd.

We may now use the languages, assuming that Lang and Lang2 are available contextually (the commented out imperative uses the Show visitor which is added in Chap. 10):

```
(# L: @Lang;
   L2: @Lang2;
   e: ^L.Exp;
   e2: ^L2.Exp;
   i: @integer; s: @string
do
   42->L.Num.init->e[];
   e.visit(# theVisitor::@L.Eval #)!->i;
   (37->L2.Num.init,5->L2.Num.init)->L2.Plus.init->e2[];
   e2.visit(# theVisitor::@L2.Eval #)!->i;
   (* e2.visit(# theVisitor::@L2.Show #)!->s *)
#)
```
Ex. 9-23

Note that *being* a virtual and *inheriting* from another virtual allows Eval in Lang2 to obtain properties in several ways, thereby illustrating that many kinds of information need only be given once because it may propagate to many places:

- Since Eval in Lang inherits from Visitor it can be used as a Visitor which is, e.g., known to support the forNum method.

- Since Visitor in Lang2 is given a forPlus method, Eval in Lang2 is also known to support a forPlus method even when accessed as a Visitor. If Eval had instead inherited from a simple (non-virtual) pattern then Eval could not have been used as a Visitor because it would not automatically get the for$X$ methods that Visitor is enhanced with for every subpattern of Lang.

- Since Eval in Lang2 is further-bound, the forPlus method is implemented in a way which is specific for evaluation (this could not have been inherited from the generic `Visitor` because `forPlus` must do something else in other subpatterns of `visitor`).

Also note that virtuals like `Visitor` and `Eval` and expressions like `Num` and `Plus` are mutually linked to each other by the contextual placement inside an instance of `Lang` or `Lang2`. This allows them to be mutually dependent on each other and still be reused in a more specialized context, without any danger of mixing, e.g., a `Num` from an instance of `Lang` with a `Plus` from an instance of `Lang2`. This again makes it possible and safe to use a language polymorphically, for instance such that the statically known pattern of the language is `Lang` but the actual pattern is `Lang2`; an expression from such a polymorphically accessed language could be evaluated without depending on what kinds of expressions that language actually contained:

```
useSomeLanguage:
  (# varL: ^Lang
  enter varL[]
  do (# L: @varL;
        e: ^L.Exp;
        i: ^integer
     do
        <<obtain an expression>> -> e[];
        e.visit(# theVisitor::@L.Eval #) -> i[]
     #)
  #)
```

Ex. 9-24

In this example, the use of an ordinary `enter`-list based argument `varL` and then a nested object with a non-variable object attribute `L` is preferred over the use of a virtual object based parameterization, because this makes `useSomeLanguage` truly polymorphic in the argument; with a virtual object as the parameter we would have obtained access to the statically known pattern of the language at each call site, and the point here is exactly that this is not necessary. We can handle *any* language in exactly the same way.

Let us take a look at the solution in Objective Label which was presented on the Java genericity mailing list on February 17, 1999, by Didier Remy and Jacques Garrigue. This solution is shown in Fig. 9.3. As is often the case with functional languages, the solution is impressively compact. It relies on the use of *polymorphic variants* [47]. These are similar to the tagged variants that are used to define algebraic datatypes, such as `Empty` and `Node` in the following Standard ML definition: ◇

```
datatype 'a Tree = Empty | Node of ('a Tree * 'a * 'a Tree);
```

Ex. 9-25

The difference is that the organization into groups is fixed and explicit with ordinary algebraic data types (`Empty` and `Node` can be used in a `Tree` and nowhere else), but polymorphic variants can simply be used, without needing any centralized declaration à la algebraic data types, and they will be grouped into arbitrary finite sets according to their use. For example, `Empty` would not be restricted to be one of the variants of the data type `Tree`, it could be used

```
let rec call self = self (fun x -> call self x);;

module Lang = struct
  let num x = `Num x
  let eval self (`Num x) = x
end;;

module Lang2 = struct
  let num = Lang.num
  let plus x y = `Plus(x,y)
  let eval self = function
      `Plus(x,y) -> self x + self y
    | `Num _ as x -> Lang.eval self x
  let show self = function
      `Num x -> string_of_int x
    | `Plus(x,y) -> "(" ^ self x ^ " + " ^ self y ^ ")"
end;;

(* the test *)
open Lang2;;
let e = num 42;;
let e2 = plus (num 5) (num 3);;
call eval e2;;
call show e2;;
```

Figure 9.3: The Expression Problem in Objective Label

in many different groups of variants, whenever "emptiness" was needed. As an example of a polymorphic variant type consider the function `eval` in module `Lang2` in Fig. 9.3, which has the following type:

$$(\alpha \to \texttt{string}) \to [< \texttt{Num(int)}\ \texttt{Plus}(\alpha * \alpha)] \to \texttt{string}$$

The expression enclosed in square brackets in the middle is a polymorphic variant type, namely the type which contains a `Num(int)` variant or a `Plus`$(\alpha * \alpha)$ variant. The '<' marker indicates that this set of variants is an upper bound, so the function may be called with an argument at this argument position whose type is known to include *at most* these variants.

The type analysis of polymorphic variants is related to the type analysis in [55], in that it represents types as finite sets of (variant resp. class) identifiers. It differs in that the polymorphic variants have type specifiers, and polymorphic variant types are not just absolute sets, they can also be bounded by lower or upper bounds, or both. A common property of the two approaches is that it is not possible to add a variant (class) and recompile without rechecking all the expressions where a value of that variant (an instance of that class) could occur. This means that it is impossible to add a new variant without rechecking and recompiling all the usage points. In contrast, the standard object-oriented analysis which does *not* attempt to maintain explicit and complete lists of variants/classes allows for such additions. For example, it is no problem to reuse some code with a `ColorPoint` even though that code was compiled to be used with a simple `Point`, but with the polymorphic variants we would have to go in and add `ColorPoint` to the list of variants everywhere it could occur. Moreover, the distribution of the implementation, with all cases collected into global functions such as `Lang2.eval`, will of course make it necessary to change those functions every time a new variant is added.

The reason why Objective Label is better than other functional languages is that it is actually possible to write a new module, like `Lang2`, which supports the extended language without creating conflicts between the different sets of variants, e.g., {`Num`} vs. {`Num,Plus`}. However, as opposed to `gbeta`, there is no inheritance relation in Objective Label between `Lang` and `Lang2`, so the implementation has to be written twice, although it *is* possible to reuse a method by means of explicit delegation (e.g., the use of `Lang.eval self x` in the implementation of `eval` in `Lang2`).

Finally, the languages are not first-class entities. They are modules which can be `open`ed such that the contained declarations become available, or they may be used by "dotting" into their name spaces (`Lang.eval` is an example), but there is no way to obtain a dynamically polymorphic reference to a module such that a piece of code would only depend on the type of `eval` in `Lang` but would actually at run-time work on `eval` in `Lang2`. The reason why this would not be possible is that the types are *not* mutually recursively dependent, as is demonstrated by the fact that a `Lang.Num` can actually be used in context of `Lang2` (see the declaration `let num = Lang.num`). This means that there is no way Objective Label could avoid mixing up expressions in `Lang` and expressions in `Lang2`, and that again necessitates keeping waterproof barriers between `Lang` and `Lang2` in the static analysis—that kind of dynamic polymorphism would simply not be type safe.

9.5 Concurrency as a Type Issue

Concurrency has always been an integrated part of BETA, possibly because of the simulation background in the Simula community. There are two kinds of non-sequential execution, alternating and concurrent. Alternating execution is performed by coroutines, and concurrent execution is performed by so-called systems. Both coroutines and systems are based on object-like entities, called *components*. Where `gbeta` has objects and patterns, BETA has objects, components, and patterns.[3] The unification in `gbeta` of components and objects into just objects removes some typing and conversion complications. The change is backward compatible because the special syntax which is used in BETA wherever ◇

[3]Actually the BETA terminology is items, components, and patterns.

a component is used is reinterpreted in gbeta to be a simple merging operation. The syntax associated with components in BETA always uses the '|' character, for example:

```
aComponent: @|
  (#
  do 'This is executed '->puttext;
     SUSPEND;
     'by a coroutine.'->putline;
  #)
```
Ex. 9-26

If a component in BETA is executed by the same syntax as used with an object, here simply the imperative `aComponent`, then it will execute as a coroutine, i.e., it will establish its own stack and then execute until it is suspended or until it terminates. A suspended component which is executed again will continue from the point where it was suspended, it will not start again from the beginning. Note that the run-time stack of a component may contain any number of ordinary objects, so it is a computation involving the component and zero or more objects which is being suspended and later resumed, not just the execution of a single `do`-part as in the above example. In that example, executing `aComponent` twice will print "`This is executed by a coroutine`".

If the predefined `fork` command is executed on a component in BETA then a new thread is created, and it will execute separately while the thread which invoked the `fork` continues without waiting.

The dynamic semantics are the same in gbeta, except that a component can be executed more than once, just like all other objects. The difference is that the support for having a separate stack, and with it the coroutine and thread functionality, has been moved out of the domain of the special object-like entities called components in BETA and into the domain of ordinary objects and ◇ patterns. To do this a new, basic pattern called *component* was introduced. A component in gbeta is simply an object which is an instance of a pattern which is less-equal than `component`. To ensure backward compatibility, the '|' marker is reinterpreted to mean merging with the new, basic pattern `component`:

```
a_Beta_or_gbeta_Component: @|aPattern;
same_in_gbeta: @ component & aPattern;
```
Ex. 9-27

This change has several consequences. For example, it is possible in gbeta to use the `when` imperative to determine whether or not a given object (whose statically known qualification does not include `component`) is actually a `component`. In BETA it is necessary to have special rules for the reference assignment to dynamic references to components from an object-or-component, and for reference assignment to dynamic references to objects from attributes which are known to denote components. All these special cases are effortlessly normalized into the well-known rules for reference assignments with the approach in gbeta. Similarly, it is possible to dynamically specialize a given object which was not created as a `component`, such that it becomes a `component` and can, e.g., be `fork`ed.

```
symmetricCoroutineSystem:
  (# symmetricCoroutine: component
       (# resume:<
            (# (* Here is the assignment which becomes safe *)
            do this(symmetricCoroutine)[]->next[]; SUSPEND
            #)
       do INNER
       #);
     next: ^symmetricCoroutine;
     run:
       (# active: ^symmetricCoroutine
       enter next[]
       do (if (next[]->active[])=NONE then leave Run if);
          NONE->next[];
          active;
          restart run
       #)
  do INNER
  #)
```

Figure 9.4: Improving analysis with a pattern which is a `component`

Finally, it improves the static analysis that it is possible to include the `component` aspect directly into a pattern. In BETA it is always a property of an object-like attribute, never of a pattern, so the use of a ThisObject construct in a BETA pattern is always analyzed as if the enclosing entity were an object, not a component. An example where the gbeta approach leads to a better preservation of the static knowledge about run-time entities is the classic example in Fig. 9.4, adapted from a similar example in [74]. The example demonstrates how a symmetric coroutine system can easily be built from BETA's built-in asymmetric coroutine support.

In the original BETA version of the `symmetricCoroutineSystem` in Fig. 9.4, the variable object attributes `next` and `active` were declared with the '`|`' marker which specifies that it must refer to a component, not an object. In this version the `component`ness has been moved by removing those '`|`' markers and adding `component` as a superpattern of the pattern `symmetricCoroutine`. The difference is that instances of this pattern in *all* parts of the program are known to be `components`, and that again makes the reference assignment to `next` inside `resume` safe. In the BETA version this reference assignment is not statically known to be safe, because the static analysis cannot verify that the enclosing `symmetricCoroutine` is actually guaranteed to be a `component`, it might as well be an ordinary object.

As a final example of the enhanced expressive power of `components` in gbeta—even though it might at first look like a curiosity which ought to be avoided—consider adding the `component` mixin in the *middle* of a pattern in-

```
(# tokens: @list(# element::string #);
do
   tokens.scan & component
   (#
   do <<setup-header-color>>;
      (while <<in-header>> do current->puttext; SUSPEND while);
      <<setup-body-color>>;
      (while <<in-body>> do current->puttext; SUSPEND while);
      (while true do
          'Unexpected token: '->puttext; current->puttext; SUSPEND
      while)
   #)
#)
```

Figure 9.5: One way to use a `component` in the middle of an object

stead of as the most general mixin. The effect of this is that the object will obtain a behavior which starts out as the ordinary, sequential behavior of most objects, but then at some level the execution of the `INNER` imperative will call the special, pre-defined `component` behavior, and that changes the rest of the behavior into a coroutine which will be executed, suspended and resumed just like any other coroutine. For example, assume that we have a `list` of `string`s which holds the contents of an HTML document, one token per element. The piece of code (with pseudo-code elements) in Fig. 9.5 then illustrates how we could print the HTML document with a different background color for the header and the body parts.

Chapter 10

The Fragment Language

Modularization of gbeta programs is facilitated by the fragment language which is a separate language that coexists with the gbeta source code in modules. The design of the fragment language is entirely as in Beta, so any prior experience with modularization of Beta programs can be applied directly to gbeta programs. However, the implementation in gbeta lifts a number of restrictions that the Mjolner Beta system imposes, and the increased freedom to use hitherto unavailable parts of the fragment language does indeed make a difference at the systems design and the software engineering level. In other words, even though readers who already know how to modularize Beta programs may want to skip the basic presentation in Sect. 10.1, the description of some practical consequences of the enhancements in Sect. 10.2 goes beyond the well-known and might serve to illustrate useful techniques which can only be applied when some of the restrictions have been removed.

10.1 Fragment Language Basics

This section presents the basic elements of the fragment language. This language is a separate language which coexists with gbeta source code in modules. It facilitates the combination of modules into larger systems, thereby making it meaningful to write the modules in the first place and providing the well-known benefits of modularization in Beta [74, Chap. 17]:

- Support for code reuse by use of the same module in several programs.
- Parallel and independent development by separate teams of programmers for those parts of large systems which do not need to depend on each other.
- Reduction of the complexity and interconnectedness of large systems by separation of a given functionality into a widely used interface module and an implementation module which is normally not used by any other modules.

- Separate static analysis and compilation of each module, reusing but not affecting the results of the static analysis of other modules that it depends on.

- Provision of separate variants of an implementation, e.g. for a number of different hardware architectures, with separate compilation and with peaceful coexistence of the variants, instead of a plethora of `#ifdef`s.

- Separation of different elements in an interface by "topic", such that a module which needs only one group of methods in a given class does not have to depend on all the other methods.

- Various "acceleration" effects where one improvement leads to another which may again improve on the first. For example, dependency chains may be broken: Assume that module A uses module B, and module B uses module C. If module B is separated into an interface B_1 and an implementation B_2 then it may be sufficient that B_2 uses C, and then A may use only B_1 and be relieved of its dependency on C.

As with most good ideas, the basic idea behind the fragment language is simple. The idea is that a syntactic construct can be expressed as a sentential form in the grammar of the language, i.e., as a partial grammatical derivation, i.e., as an ordinary piece of source code in the language except that certain parts of the source code have been left unspecified, marked only by placeholders saying "here should be something, but I won't tell you what it is". For example:

```
getElement:
  (# theList:< @list;
     theElement: ^theList.element
  <<SLOT GetElement:dopart>>
  exit theElement[]
  #)
```

Ex. 10-1

◇ In this pattern, a piece of code is missing and a placeholder, a *SLOT application*, is found in its place. It is the piece of text between the '`<<...>>`' brackets. Since the placeholder for the missing piece of code is marked as a '`dopart`', and the syntactic category DoPart is the kind of expression that is expected at this position according to the grammar of gbeta, the pattern as a whole is accepted by the parser, and it is known that there should be a piece of code somewhere in any complete system which is a DoPart and which has been given the name `GetElement`. The compiler does not need to look at that piece of code when compiling the pattern `getElement`, but if there is no such piece of code when a complete program using `getElement` is linked, then there will be a linker error. In other words, the interface of this method may be compiled separately from its implementation. Such a language which enables the combination af source code units into a system is called a module interconnection language [93].

The piece of code carrying the name `GetElement` must be syntactically a DoPart and it can be declared like this:

```
-- GetElement:dopart --
do theList.first.elm[]->theElement[]
```
Ex. 10-2

This is a *SLOT declaration*, saying that in the name space for pieces of code, ◇ the piece with the name `GetElement` is the DoPart which follows right after the '`--...--`' line. Note that the name space for pieces of code is constructed in and used by the fragment language only, so it is possible (and not uncommon) to use the same name for a piece of code as the name of some nearby attribute in the gbeta source code. Hence, there is no confusion between the `getElement` pattern and the piece of code named `GetElement`. Like in gbeta, names in the fragment language are case insensitive.

Modules are (currently) identified with files, so each source code file is a module. Since the traditional name for a SLOT declaration is *fragment form*, ◇ and since each module may contain any number of these, modules are traditionally called *fragment groups*. Both fragment groups and fragment forms are ◇ traditionally called *fragments* when it is clear from the context whether it refers ◇ to one or the other.

A fragment form, or SLOT declaration, *defines* a named entity (a piece of source code), and a SLOT application *uses* such an entity, so there is a need for lookup rules in connection with SLOT names. These lookup rules are entirely separate from the lookup rules in gbeta, and much simpler. Each fragment group is considered as a flat name space containing a set of named pieces of code, each piece of code typed as one particular syntactic construct, e.g., a DoPart. The granularity for the lookup process is the fragment group, so lookup happens either within one fragment group, or by going to other fragment groups which are reachable through certain links. Each module may include a number of links to other modules, in one of the following varieties:[1]

ORIGIN: This kind of link specifies the direction in which a fragment form may be *used*; one way to think of it is that each fragment form can "travel" from fragment group to fragment group, but only via ORIGIN links, in order to "go home" to the spot where it is used.

INCLUDE: This kind of link provides visibility. A declaration can only be used in a fragment group if it is located in a fragment group which is reachable through a (possibly empty) path of ORIGIN and/or INCLUDE links.

BODY: This is a link which has no effect before link-time;[2] one way to think of it is that it is a "blind" link, because the source code which is reachable through BODY links *will* be included in the final program, but the contents of such a fragment group is entirely invisible at compile-time—it's

[1]These are not the only kinds of 'properties' that may occur at the beginning of a fragment group, but they suffice for the presentation given here.

[2]It may cause some files to be considered and possibly compiled because of the global dependency analysis, but that is essentially also 'link-time' even if it is not the linker which performs this analysis.

like saying "I need this, but I don't want to know what it is!" which is exactly the suitable attitude to have towards an implementation module.

It is important to realize that it may not be enough to INCLUDE a module in order to use something which is declared in it; the INCLUDE link ensures that the contents of the module will be visible *in its context*, but since BETA and `gbeta` have general block structure there are many other possible contextual placements than the outermost, global name space. For example, if the fragment system is used to add a `getElement` method to the `list` pattern then an INCLUDE link to the fragment which contains the declaration of `getElement` will make it possible to invoke that method in context of an instance of `list` or a subpattern, but it will not add any `getElement` pattern to any other context. E.g., it will not be possible to invoke `getElement` as a standalone procedure outside of any `list`, unless of course a pattern named `getElement` has been added to that context by some other means. This may seem obvious, but experience shows that it often leads to confusion when, e.g., it is not possible to use such a pattern as `Window` at the global level of a BETA program even though the fragment group where it is defined has been included, when in fact `Window` is defined as an attribute of the pattern `guienv` which represents and supports the use of the graphical user interface framework on a number of platforms in the Mjolner BETA system. The solution is typically to wrap everything written by the programmer inside an instance of `guienv`, such that the programmer may think of all the graphical user interface patterns as being "global", though it might also be relevant to use several instances of `guienv` to interface to several screens for a distributed program (no, the current implementation of `guienv` does not allow this, but it might).

The easiest way to understand why the fragment system has these properties
◇ is to think of it in terms of two concepts [74, Chap. 17], namely the *domain* of
◇ a fragment form, and the *extent* of a fragment form; in both cases we call
◇ that fragment form the *root* of the domain or extent. Both of these concepts denote a syntactic construct, a piece of `gbeta` source code, and both of them are created by selecting a certain set of fragment groups and performing a search-and-replace process. For the domain, the selected fragment groups are all the groups reachable through either ORIGIN or INCLUDE links, and for the extent it is all groups reachable through either ORIGIN, INCLUDE, or BODY links—in both cases directly or indirectly.

The search-and-replace process works as follows: For each SLOT application the corresponding SLOT declaration is looked up, and the SLOT application is replaced with the piece of source code in the declaration, such that each placeholder for a "missing piece" in the source code is replaced with the actual piece of source code that is defined with the same name. The lookup process may only select a SLOT declaration in a fragment group from which there is a (possibly empty) path of ORIGIN links to the fragment group containing the SLOT application—we might say that the scope for a SLOT name is the transitive closure of the inverse ORIGIN links. In this replacement process there must be exactly one SLOT declaration with the given name in scope for each syntactic construct which is not a list, and there may be zero or more SLOT declarations for such syntactic categories as Attributes, where the grammar allows for an unordered list of AttributeDecls of varying length, such that all SLOT declarations of that name can simply be collected to make a longer list. For lists where the order is semantically significant it is generally only allowed to have one SLOT declaration, but Alternatives form a notable exception, as explained in the next section.

In other words, the domain contains all the source code which is available for *lookup* from the root fragment, and the extent contains all the source code which is available at *run-time*, namely the domain plus something which can usually be described as the implementation.

For example, consider the situation where we have a fragment group in the file `betaenv.gb` containing just one *universe fragment form* named `betaenv` ◇ which is a `descriptor` (an alias for ObjectDescriptor):

```
-- betaenv:descriptor --
(# list:
     (# <<SLOT listlib:attributes>>;
        element:< object;
        append: ...
        first: ...
        ...
     #);
   program: @<<SLOT program:merge>>
do
   program
#)
```
Ex. 10-3

The `betaenv` fragment form is by convention *not* looked up so there should not be a SLOT application for the name `betaenv`, and this fragment form will therefore be the outermost construct into which all other pieces of code will be placed, directly or indirectly. Consequently, it is the universe within which everything will ultimately be located. Furthermore, we have a fragment group in `getElement.gb` which adds a method to the `list` pattern:

```
ORIGIN 'betaenv';
BODY 'getElementbody'
-- listlib:attributes --
getElement:
  (# theElement: ^element
  <<SLOT GetElement:dopart>>
  exit theElement[]
  #)
```
Ex. 10-4

We need another fragment group in order to provide the implementation; that kind of fragment group is conventionally given names ending in `..body`, here `getElementbody.gb`:

```
ORIGIN 'getElement'
-- GetElement:dopart --
do first.elm[]->theElement[]
```
Ex. 10-5

Finally, we have a file which can tie all the pieces together to a complete program, `myProgram.gb`:

```
ORIGIN 'betaenv';
INCLUDE 'getElement'

-- program:merge --
(#
   myList: @list(# element::integer #);
   i: ^integer
do
   integer[]->myList.append;
   myList.getElement->i[];
#)
```

Ex. 10-6

Taking `program` as the root fragment form, we obtain the following domain:

```
(#
   list:
     (# getElement:
          (# theElement: ^element
            <<SLOT GetElement:dopart>>
            exit theElement[]
            #);
        element:< object;
        append: ...
        first: ...
        ...
     #);

   program: @
     (#
        myList: @list(# element::integer #);
        i: ^integer
     do
        integer[]->myList.append;
        myList.getElement->i[];
     #)
do
   program
#)
```

Ex. 10-7

Notice that the implementation of `getElement` is still missing in the domain. The extent is the following piece of code, and this time the implementation is included:

```
(#
   list:
     (# getElement:
          (# theElement: ^element
          do first.elm[]->theElement[]
          exit theElement[]
          #);
        element:< object;
        append: ...
        first: ...
        ...
     #);

   program: @
     (#
        myList: @list(# element::integer #);
        i: ^integer
     do
        integer[]->myList.append;
        myList.getElement->i[];
     #)
do
   program
#)
```
Ex. 10-8

Of course, this search-and-replace process is incompatible with separate compilation, but an implementation which provides separate compilation—such as the Mjolner BETA system—must behave in such a way that the name lookup and the behavior of the program at run-time is as if the search-and-replace process had actually taken place.

In fact, the gbeta implementation, about which more information can be found in Sect. 11, performs the search-and-replace process as described for the extent, and then checks the fragment graph to see for each name lookup whether or not a given declaration with the name N is visible from the fragment where the application of N is located. This means that the implementation of the fragment language in gbeta takes the easy route and thus is far more complete than the implementation in Mjolner BETA. Here, a SLOT can only be one of a few, carefully selected syntactic categories (ObjectDescriptor, Attributes, DoPart, and soon also MainPart). Moreover, an Attributes SLOT can only contain simple pattern attributes, not objects, variable objects, virtual patterns, or variable patterns.

However, there are some problems which have to be solved before the enhanced generality can be obtained in a system with separate compilation—gbeta currently generates code with access to the entire program at the same time. For separate *static analysis* there are only few new problems; the Mjolner BETA system stores the results of the static analysis in files along with the source code, and that is basically what we need in order to analyze gbeta code separately; it is certainly enough to perform name binding and to ensure that there will not be any `MessageNotUnderstood` errors. The analysis of conflicts—for example when there are several attributes with the same name in the same context—requires a link-time check, and that check may be complicated by the need to investigate the legality of merging of virtuals in gbeta, but the need to have a link-time check in the first place is not new.[3]

On the other hand, for *code generation* it is necessary to handle the problem that the size of objects is generally not known at compile-time with the generalized fragment system, and therefore it is not possible to allocate fixed offsets for attributes (in objects for BETA and in part objects for gbeta), and that is a significant change, compared to the currently used assumptions for code generation in the Mjolner BETA system. For instance, if an object attribute is placed in an Attributes SLOT then the size of the enclosing object is not known during compile-time of the SLOT application, but it is not even known during compile-time of the SLOT declaration either, because there may be additional Attributes SLOT declarations with the same name in other fragment groups. In other words, the compilation of the pattern cannot determine how much memory to allocate for a new instance, and it cannot even use a symbolic name and let the linker pick up the value from code generated for the SLOT declaration. A simplistic solution which looks up the attributes dynamically for every access would be prohibitively slow, but there are many systems where similar problems have been attacked with great success, so it should not be entirely impossible. More about this in Sect. 11.

10.2 Enhancements in gbeta

Actually, there are no enhancements in the fragment system of gbeta compared to the fragment system of BETA; but most people who know BETA do this via the Mjolner BETA system, and this means that the restrictions in the implementation of the fragment system which were mentioned near the end of the previous section may often be considered part of the fragment system as such rather than temporary restrictions caused by the finiteness of resources in the development of that BETA system.

Hence, we find it beneficial to give some hints at what kind of consequences the restrictions have for practical systems development, by means of a number of examples which are supported in the gbeta version of the fragment system, but which violate some of the restrictions in the Mjolner version.

Firstly, it is a very serious restriction that the Mjolner fragment system only allows one kind of attributes in an Attributes SLOT, namely simple patterns. It is possible to have private methods (*so* private that the class does not have to be recompiled when they are added), but that only works for non-virtual methods—a virtual method, both declaration and all the further- and final-bindings, must be physically located in context of the MainPart of which it is an attribute. Especially further-bindings seem to be purely a matter of implementation in many cases. Moreover, it is not possible to separate a group of virtual methods out into a library fragment, again because virtuals have to be located physically

[3]The Mjolner BETA system actually omits the link-time check and allows the existence of more than one attribute in the same context with the same name as long as they come from different files, but we are not convinced that this is a good approach.

inside the source code for their enclosing object. It is possible and very useful to move a group of simple patterns out into a library fragment; it is used in order to provide these patterns optionally and in meaningful clusters, such that only the clients who actually need them have to become dependent on them.

As an example where we separate out a virtual attribute in gbeta, consider again the expression problem described in Sect. 9.4. The patterns in Fig. 9.2 on page 199 may be modularized in the following manner:

```
(* FILE Lang.gb *)

ORIGIN 'betaenv'

-- lib:attributes --

Lang:
  (# <<SLOT LangLib:attributes>>;

     <<as in Fig.~9.2>>

  #)
```

Ex. 10-9

The first fragment, `Lang.gb`, places `Lang` in a context with the SLOT application `lib` of the syntactic category Attributes, and this is traditionally a SLOT at top level in the universe fragment `betaenv`, so this means that we are making `Lang` a globally available class for all those fragments that INCLUDE the fragment `Lang`, directly or indirectly. The only difference in the pattern itself compared to Fig. 9.2 is that we have added a SLOT application named `LangLib`. Similarly for `Lang2`:

```
(* FILE Lang2.gb *)

ORIGIN 'betaenv';
INCLUDE 'Lang'

-- lib:attributes --

Lang2: Lang
  (# <<SLOT Lang2Lib:attributes>>;

     <<as in Fig.~9.2>>

  #)
```

Ex. 10-10

With `Lang2` we also have to INCLUDE the fragment form `Lang`, because the pattern `Lang` from that fragment is being used as a superpattern. Now we add an extra `Visitor` to both of these patterns:

```
(* FILE LangShow.gb *)

ORIGIN 'Lang';
ORIGIN 'Lang2'

-- LangLib:attributes --

Show:< Visitor
  (# R:: string;
     forNum::< (# do '(some number)'->&result #)
  #);

-- Lang2Lib:attributes --

Show::<
  (# forPlus::<
       (# fst,snd: ^string
       do e1.visit(# theVisitor::@this(Show) #)->fst[];
          e2.visit(# theVisitor::@this(Show) #)->snd[];
          fst+' + '+snd->&result
       #)
  #)
```

Ex. 10-11

Note that the two SLOTs LangLib and Lang2Lib are used to put something into two *different* contexts from the same file. The enhanced generality of the gbeta fragment system comes into play in two ways here: Firstly, the fact that attribute SLOTs may contain other kinds of attributes than simple patterns is used to add a virtual to Lang and also to further-bind it in Lang2. Secondly, since these two SLOT declarations are used in two different fragment groups we have given two ORIGIN links; in the Mjolner system only one ORIGIN is allowed.

We could of course have introduced and implemented Show in Lang2 alone, and included it already in Fig. 9.2, just like it is done in the GJ version in App. C. This version demonstrates that it is also possible to enhance an already existing class with a nested virtual class, or method, or type parameter ... depending on what kind of use the virtual attribute is intended for.

With these fragments it is possible for a client program to use the original patterns Lang and Lang2 as they were, and it is possible to use the enhanced version which also supports Show by including the fragment group LangShow. Note that it is possible to include LangShow in a program and still not let all parts of the program see Show—seeing something means depending on it whether or not it is actually *used*, so that may be significant. The fact that these attributes are simply not present in a program that does not in any way refer to LangShow (it is the *client*, not Lang that brings LangShow into the program) opens some interesting possibilities in the area of saving space and thereby also time by having more lightweight objects than would otherwise be practical—programs that do not use a given aspect of a general pattern would leave that aspect out entirely by not including certain files.

Now consider the two forPlus methods in Show (above) and in Eval (defined in Fig. 9.2). They could actually share a large part of the implementation from a purely textual point of view, namely the two evaluations of `visit`. If we separate out those two imperatives and put in an Imperatives SLOT instead then we get the following:

```
forPlus::<
  (# fst,snd: ^string
  do <<SLOT ForPlus:Imperatives>>;
     fst+' + '+snd->&result
  #)
```
Ex. 10-12

With this definition of `forPlus` in `Show`, and a similar definition of `forPlus` in `Eval`, these two methods may both use the same piece of syntax for the missing piece of the implementation:

```
-- ForPlus:Imperatives --
e1.visit(# theVisitor::@this(Visitor) #)->fst[];
e2.visit(# theVisitor::@this(Visitor) #)->snd[];
```
Ex. 10-13

We have to use `this(Visitor)` to select the right enclosing object because both `Show` and `Eval` are less-equal than `Visitor`, but otherwise the source code works without changes in both contexts. However, the analysis and the code generation are quite different in the two cases—in `Eval` the `fst` and `snd` attributes are `integers`, but in `Show` they are `strings`. The difference would have been even bigger if they had been, e.g., patterns in one case and objects in the other.

This mechanism can be used to exploit the similarities between different contexts where two pieces of code are very similar because they do the same thing in some sense, but the type system is too rigid to see the similarities. It remains to be seen whether this is a useful tool in software development or just a way to create some terrible problems for maintenance and readability.

Finally, there is one case in which it may be justifiable to allow for more than one SLOT declaration for an *ordered* list even though these SLOT declarations will be collected in an arbitrary order, namely with the Alternatives of a GenerallfImp. The GenerallfImp construct is the `if` imperative which allows for testing against several different guards, similarly to a `switch` or a `case` statement in other languages. The semantics of a GenerallfImp is to evaluate the guards in the order they appear in the source code. This rule was chosen because it is hard to ensure that (non-trivial) expression evaluation in a language like gbeta will not have side-effects, so a rule which says that the guards will be evaluated in an unspecified order would simply be too much of a source of subtle bugs. Nevertheless, allowing multiple SLOT declarations to contribute to the same Alternatives SLOT application is so interesting that we decided to live with the fact that these SLOT declarations will be collected in an arbitrary order.

The motivation for allowing multiple Alternative SLOT declarations is that it provides an entirely modularization based approach to support for *modes* in ◇ objects [107]. This concept is associated with the observation that many objects may be described in a simpler way by first dividing their life history into a sequence of phases, where the behavior of the object is relatively homogeneous within each phase. For example, a mobile phone offers different operations and behaves differently when a connection has been made than it does when there is no connection; similarly, a bank account behaves differently when it is empty and when it is full ...

There has been some work recently in the direction of supporting modes in BETA as a new language construct (not just a new library), but it is not yet clear how exactly this will work. On the other hand, we would like to stress the possibility of using the fragment system to divide an ordinary BETA program into pieces in an unusual way (we do not need any special gbeta features for this), and thereby allowing for that kind of separation of mode specific concerns that mode support is all about.

Consider the mobile phone we mentioned before; assume that it has a 'Talk' button which is used to go off-hook and dial the current number, and an 'End' button which is used to take the connection down again and go on-hook:

```
(* FILE phone.gb *)
mobilePhone:
  (# offHook: @boolean; (* this is the current mode *)
     onTalkPressed:
       (# do (if offHook <<SLOT Talk:Alternatives>> if)#);
     onEndPressed:
       (# do (if offHook <<SLOT End:Alternatives>> if)#);
     ...
  #);
```

Ex. 10-14

The value of the boolean object offHook is the mode, so the mode space only has two elements. The behavior of the two methods when the phone is on-hook, i.e., offHook is false, is as follows:

```
(* FILE phoneOnHook.gb *)
ORIGIN 'phone'
-- Talk:Alternatives --
// false then <<dial>>; true->offHook

-- End:Alternatives --
// false then <<beep>>
```

Ex. 10-15

The Alternatives are similar in form, and this should make it easy to ensure consistency. Since all the source code in this file is concerned with the phone when it is in the on-hook mode, it should also support programmers in thinking about just one mode at a time, along with the mode changes out of the on-hook mode. In the other mode the situation is similar:

```
(* FILE phoneOffHook.gb *)
ORIGIN 'phone'
-- Talk:Alternatives --
// true then <<beep>>

-- End:Alternatives --
// true then <<shut-down-connection>>; false->offHook
```

Ex. 10-16

There are some concerns that this approach does not address. For example, there are often thousands of modes in real-world systems, perhaps even in mobile phones, and this approach does not immediately offer meaningful ways to handle this. We believe that the best approach to handling this problem is to cut down on the mode space by separating some concerns, such that a system contains, e.g., 10 independent elements with 2 modes each rather than one element with 1024 modes. The problem of switching mode in the middle of a method invocation (should we jump to the middle of another case?!) is not addressed, but it seems that there are no good solutions to this problem anyway, so that will probably be handled by programmers from case to case. At least the semantics of this approach is simple since it only uses very well-known constructs and just optimizes the physical organization of the source code for a mode centric working style. All in all, we feel that this approach is simpler than the special-purpose constructs which have been proposed, but it still does the job relatively well.

Chapter 11

Implementation

The language gbeta could not have been purely a thought experiment.

A programming language is in some sense similar to a fractal set like the Mandelbrot set [76], where the rapidity of divergence of certain sequences of complex numbers in the vicinity of the set can be used to draw complex, but somehow regular, beautiful and surprising pictures. The similarity lies in the fact that the language itself (the syntax and accompanying semantics) is such a relatively simple core entity from which a large wilderness of different programs may be grown using human ingenuity and perseverance. Another analogy would be a chaotic system like the weather, where developments over periods of weeks or months are so acutely sensitive to the conditions at the beginning of the development that even very small changes initially lead to entirely different scenarios later on [66]. With the Mandelbrot set, the complex outcome is a result of a rigid application of a single rule; a small seed grows up and becomes a complex phenomenon. With the weather, the complex outcome can not readily be traced back to such a simple core cause, but there are still connections from small causes to large consequences.

We believe that a programming language should be considered as an intermediate form between fractals and the weather, even if different from both in many ways. The set of potential programs is about as rigidly defined as the Mandelbrot pictures, at least if the language in question has a formal semantics or an implementation. But the subset of these programs that human beings will actually come up with and consider well-designed is more like the weather: There is a complex basis of all the *possible* programs and then on top of that a much more complex process of human beings trying to navigate the large universe of possibilities in order to produce programs at the center of a fuzzy cloud of *appropriate* programs for a given usage context. Like the computation rule for the Mandelbrot pictures, the programming language serves as a small seed from which a large set of possible programs arise, and even small changes to the language may have profound consequences for this set of possible programs.

In order to do good programming language design one must try to understand this duality of the dynamics of unfolding, from actual language design decisions, over the formal consequences in terms of possible programs, and, last but certainly not least, to the consequences for human beings who are trying to create useful programs.

From this point of view it was evident that an implementation of gbeta would be a necessary tool in the development of the language, not just as a "proof of concept" that could be added at a late stage in the process. It is crucial that the language development can be accompanied with excursions into the universe of possible programs. It will be extremely valuable to gather experiences from many people writing larger systems, when the implementation becomes sufficiently mature for that, and if a multitude of people can be lured into doing it.

As a matter of fact, the development of gbeta started out as a plan to implement a BETA interpreter, mainly in order to establish some working knowledge about the precise semantics of BETA, such that a formal semantics for BETA could be finished. However, it soon became apparent that such an interpreter would be a wonderful tool for trying out experiments with the language design, and that became the main topic of our research. As it turned out, it is not only important to be able to write programs in the new language, it is also important to use the process of implementing the language to improve the precision and verify the technical feasibility of the intended semantics. An implementation represents a precise choice of semantics, and the fact that the implementation cannot be fuzzy helps greatly in the process of ensuring that the language semantics is well-defined, and that such a choice of semantics is realistic in the sense that it *can* actually be implemented.

In summary, the implementation of gbeta has served as an essential vehicle for the exploration of the consequences of language design decisions, both in terms of the experience from writing small example programs as a method to sample the qualities of new kinds of possible programs, but also in terms of the lessons learned from implementing the language, ensuring precision and technical feasibility of chosen semantics.

Note that the considerations in other chapters of this thesis about human beings and their understanding of programs serve to establish guidelines for the design of programming languages, such that the design process—an exploration of potential universes of possible programs—does not happen blindfolded. The fact is that we cannot directly investigate the universes of possible programs because they are so huge, so we need to search for principles to guide us, and they must be concerned with human beings.

11.1 A Chronological View

The idea of creating gbeta was conceived in the autumn of 1995. The gbeta implementation has been sufficiently complete to perform static analysis since October 1996. In particular, it has been able to analyze the special '`tst.bet`' program. This program contains about 2500 lines of BETA code which represents a collection of test cases that is used as a baseline test suite for new versions of the Mjolner BETA compiler. The `tst.bet` program had to be changed slightly, as described below in Sect. 11.2.

The ability to handle `tst.bet` correctly is a demonstration of a certain level of backward compatibility and a certain level of correctness, since `tst.bet` contains a number of constructs that originally were used to demonstrate bugs or test "hard cases" in the Mjolner compiler. However, in 1996 gbeta could not execute even the simplest program, it could only analyze it.

In spring and summer 1997, the core dynamic semantics (the run-time system) in gbeta was implemented. This gave rise to several non-trivial changes in the static analysis. These changes did not affect BETA programs, but they affected the analysis of gbeta programs that went beyond the boundaries of BETA. In particular, the dynamic semantics of the merging of virtual attributes was implemented differently than it had been planned, because the original semantics were hard to analyze *correctly* statically—virtuals could not in general be determined to be type safe without a global analysis, i.e., they were incompatible with an 'open world assumption' and hence incompatible with separate compilation, and with reusable libraries. That was not considered acceptable, and actually the semantics which was chosen to overcome this problem is better from so many points of view that there has been no consideration of trying to handle the original semantics since then. This was a case where even the implementation of static semantic analysis did not clearly enough reveal that the (slightly vague) intended semantics was ill defined.

In December 1997 a version of gbeta was complete and stable enough to allow people with interests in programming languages to experiment with the language as such. This version was announced as being available by anonymous FTP, with an accompanying web-site at the address `http://www.daimi.au.dk/~eernst/gbeta/index.html`. Version 0.8 of gbeta is expected to be released here in July 1999. The source code of the implementation is available, under the GPL (open source) copyright license.

In 1998 and 1999, the language has evolved along four axes. Firstly, the static analysis has stabilized, such that there have been no changes at the design level since spring 1998, but there have been many bug-fixes. In other words, it seems clear how to analyze gbeta correctly, but the implementation of the analysis had to be shaked down to a reasonably bug-free state. Secondly, the implementation of the dynamic semantics has been improved—some constructs were not supported, or they were only supported in "most" cases (for example, the dynamic specialization of objects containing repetitions was implemented in March 1999, until then an attempt to specialize such an object would cause a 'Sorry, not yet implemented!' error message). Thirdly, the basic architecture of the execution of programs has been changed from a closure based model to a model based on generation (once) and later (possibly many times) execution of bytecode. This is treated in greater detail below. Finally, several new constructs have been added to the language, for example virtual objects and the `when` imperative.

There is still a long way to go before the implementation will have sufficiently good performance (in time, space, and stability) for larger, real-world projects, but getting people to use it is certainly an important goal.

11.2 Compatibility Issues

◇ The language gbeta is essentially *backward compatible* with BETA—a BETA program is also a gbeta program, and it behaves the same under both languages. However, there are differences. There are some syntactic issues, because gbeta has a few additional reserved words, and there are some genuine semantic issues, which are the topic of Sect. 8.1. Finally, there are some more spurious semantic differences which are described in this section in context of the modifications to `tst.bet` that were needed in order to make it a gbeta program.

The modified version of `tst.bet` is available as part of the release of gbeta which is mentioned above. Incompatibilities between the semantics of BETA and gbeta are described in Sect. 8.1. Some other incompatibilities that make gbeta reject Mjolner BETA programs are caused by implementation artifacts in the Mjolner BETA compiler, or facilities that simply have not yet been implemented in gbeta. All in all, this causes three kinds of modifications to `tst.bet`.

Firstly, the Mjolner compiler expects the basic patterns like `integer` to be defined by "magic" declarations such as `integer:(##)` in the outermost ('`betaenv`') object; these declarations must be removed before gbeta runs the program, because they actually declare that `integer` is a pattern that has no `enter` and `exit` lists and no attributes. The "magic" with the Mjolner compiler is that these declarations are treated differently than all other declarations, and it was not considered appropriate for gbeta to do this; in gbeta the predefined patterns are really predefined and do not have any syntactic representation. The second kind of changes was the deletion of the few lines in `tst.bet` which were used to test the invocation of external routines (such as C functions); no support for external routines has been implemented in gbeta as yet. Primitives (such as directly reading or writing a specific memory address) are not yet supported either. Finally, since gbeta deliberately has a different semantics than BETA in repetition assignments, a couple of repetition assignments had to be changed.

Since external routines are not yet supported, a new primitive entity `stdio` was added to gbeta, and this makes it possible to print to standard output and to read the standard input. This does not represent careful language design, it was just an easy way to make it possible to write programs which can at least support old-fashioned, console based interaction.

In summary, gbeta will not run any Mjolner BETA programs unmodified, because they all contain calls to external routines (e.g. to be able to access standard input/output), and because of the "magic" patterns in `betaenv.bet`. However, BETA programming experience can generally be applied directly when writing gbeta programs, just noting the "real" semantic changes presented in Sect. 8.1.

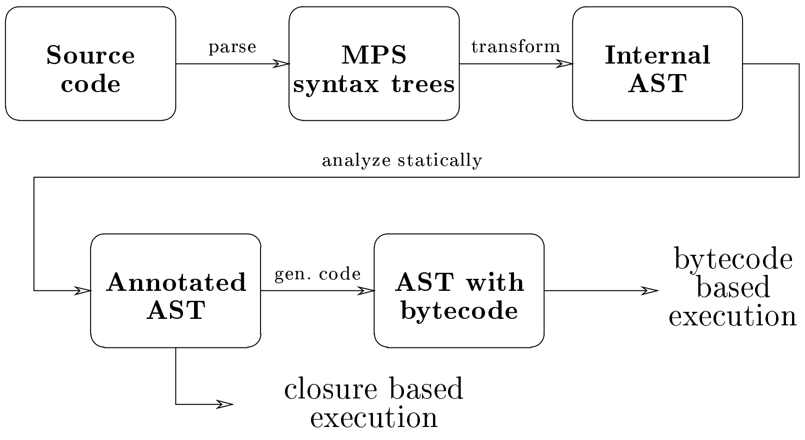

Figure 11.1: The architecture of the gbeta implementation

11.3 Architecture

The implementation of gbeta is written in Beta. It includes approximately 110 source code files, 70 KLOC, written specifically for the implementation of gbeta. Additionally, it uses some standard libraries provided by Mjolner, such as the meta-programming system, *MPS* [84], which is used to obtain abstract ◇ syntax tree representations of source code. The GNU '`readline`' library is used to provide line editing facilities for interactive use of gbeta.

The architecture of the implementation of gbeta, i.e., the most coarse-grained view of the design, is illustrated in Fig. 11.1. Each rounded box in the figure represents some representation of a given, complete program which is being analyzed and executed, and the arrows show how representations can be transformed into other representations. This process is a simple, linear progression which finally produces two directly executable representations of the program.

At the beginning, the program is provided as a set of textual source code files or files containing abstract syntax trees. If the programs are constructed using such a tool as Sif [83] then the textual files need not exist because the abstract syntax trees are generated directly, otherwise the textual representation is parsed to get the abstract syntax trees. The parsing is based on the grammar in App. A, and the parser has been generated using MPS.

A simple transformation process is applied to the MPS syntax trees to produce an internal representation which may also be described as abstract syntax trees, or *AST*s. This extra transformation degrades the performance somewhat, ◇ but it also yields several benefits. First, the internal ASTs may be generated from several different kinds of MPS ASTs, so there may be several different grammars. This is used to ensure that Beta source code can be parsed according to the standard Beta grammar, while gbeta programs can be parsed with a grammar that differs from the Beta grammar in many ways. It may be beneficial to provide other grammars for gbeta than these two, e.g., a "mainstream" grammar that tries to mimic the grammar of a language like Java might ease the communication of ideas to people who are not familiar with the BETA style of syntax. The internal representation is the same for all grammars, though some nodes (such as `when`) will never be generated from a BETA MPS AST.

Second, the fact that the MPS ASTs and the internal ASTs are separate makes it easy to make small adjustments to the gbeta grammar, regenerate the patterns that are used to access the MPS ASTs, and then update the transformation. The internal AST nodes carry most of the implementation of gbeta, and it would be very inconvenient to re-insert all those virtual methods etc. after each little grammar adjustment.

Third, the internal representation is more abstract than the MPS ASTs, in particular because the MPS grammar needs to allow for unambiguous parsing. This allows the use of simpler ASTs, especially the ASTs for expressions become smaller.

Fourth, each child node under a node in an internal AST is accessed through a reference with a qualification which is optimally precise according to the grammar (e.g., the two nodes hanging under a binary expression, `l2BinaryExp`, are qualified by `l2Expression`), whereas each child node in an MPS AST is only known by the type system as an '`ast`'. Consequently, a program which traverses abstract syntax trees repeatedly will avoid a large number of dynamic type checks by using internal ASTs.

Fifth, the MPS ASTs allow for annotation of syntax trees with semantic information, but it is not easy to allocate space for information with variable size, which is needed in order to support gbeta static analysis.

The main disadvantage is that the internal ASTs do not have integrated support for persistence. The MPS ASTs are stored efficiently on disk files together with their static semantic annotations, but as it is now, gbeta recomputes the static information every time a file is loaded, because the internal ASTs from previous runs are lost. This means that the potential for separate, reusable analysis and code generation is not leveraged. That should be corrected as soon as possible.

◇ The next step in the processing of a gbeta program is the *static analysis*. This process has two aspects. The first aspect is essential for the run-time semantics of programs, and that is the annotation of each name application, NameApl, with a run-time path that specifies how to find the entity which is denoted by that NameApl in a given execution context. A necessary support facility for this annotation is the capability to compute the type of accessible run-time entities.

◇ The second aspect of the static analysis is the *type checking*, where it is ensured that the statically known properties of entities guarantee that the execution of the program will not exhibit type errors. This part of the static analysis is not needed in order to run the program, and gbeta can be instructed to skip it (by using the '`-l`' option, for 'lazy' analysis, where static information is computed on demand and only as needed for the execution of the program).

A program which has been transformed into an internal AST and annotated with run-time paths can be executed directly by means of a closure based execution. This technique is described in Sect. 11.5.

However, the closure based execution technique is inappropriate for several reasons, and that motivates yet another annotation phase, namely a code generation phase where each imperative is decorated with a list of bytecodes. When a program is available as an internal AST with bytecodes it can be executed in a much more appropriate way, using a stack based virtual machine to interpret the bytecodes.

11.4 Source Code Naming Conventions

To support the readability of the source code we mention briefly some naming conventions that may help to place individual pieces of code in the right context, in addition to the division into phases that Fig. 11.1 on page 225 describes. The processing of gbeta programs is divided into three levels, namely the abstract syntax (the program); the run-time entities (objects and patterns and so on, which may be accessed using syntax); and the transient entities, implementing values (always implicit—even literals just *produce* values when evaluated). The transient entities are called 'level zero', the run-time entities are called 'level one', and the syntactic entities are called 'level two'.

This affects the naming of many parts of the implementation. For example, the syntactic construct ObjectDescriptor is represented in internal ASTs by an instance of the pattern `l2ObjectDescriptor`, where `l2` means level two. Similarly, a (gbeta) pattern is represented at run-time by an instance of a (BETA, implementation level) pattern `l1PatternEntity`, where `l1` means level one. And, finally, an integer value which arises as a result of a computation (such as an evaluation of the value of an `integer` object) is represented by an instance of the pattern `l0TransientInteger`. Names in the source code are often prefixed with `l0`, `l1`, or `l2`, to reflect whether they refer to gbeta-level values, gbeta-level objects/patterns, or gbeta-level syntax. For example, `l1obj` is an object, `l1pat` is a pattern, and `l2ndcl` is an l2NameDcl abstract syntax node, a name declaration.

Note that level zero entities are only used in the closure based execution, and the ability to avoid explicit representation of transient entities as implementation level BETA objects is one of the many reasons why the bytecode based execution is so much more efficient. In bytecode based execution the transient entities at the gbeta level are mapped to transient entities at the implementation level.

11.5 Closure Based Execution

The implementation of the run-time semantics of gbeta comes in two versions, two generations. The first generation of the implementation was created at a stage where it was not at all clear that it would eventually be possible to implement the language as it was intended. Getting the right semantics (and discovering what the detailed semantics should be) was a sufficiently lofty goal, and performance considerations did not enter the equation.

It was actually possible. This implementation is based on closures, i.e., on objects which serve as contextually dependent deferred computations that can be collected from run-time entities and then executed using a well-defined protocol. For example, consider the interpretation of an assignment evaluation such as `(3,4)->myPoint.move` in context of the following program:

```
(# Point:
     (# x,y: @integer;
        move: (# dx,dy: @integer enter (dx,dy) ... #);
     #);
   myPoint: @Point
do
   (3,4)->myPoint.move
#)
```
Ex. 11-1

The execution would proceed as follows: First the evaluation list '`(3,4)`' would be asked to provide an '`ExitIterator`' which would be able to deliver the values 3 and 4 as level zero entities, in that order and one per request. The `gbeta` implementation defines many kinds of `ExitIterator`s, one for each syntactic construct that can be evaluated. Then `myPoint.move` would be asked to deliver an '`EnterIterator`' (again, there are many kinds) which would be able to accept two level zero entities containing integer values (or some other values that can be coerced into integer values), and then the two iterators would be brought together by asking the `ExitIterator` for the next level zero entity and giving that entity to the the `EnterIterator` as often as one had something to give and the other would accept more values. Static information is necessary for the creation of such iterators, because the `enter`- and `exit`-lists which are statically known at the location of the assignment evaluation, and *only* those `enter`- and `exit`-lists, must participate in this process. A simplified version of this algorithm (which is the closure-based implementation of the `execute` method of an `l2AssignmentEvaluation`) looks as follows:

```
leftHandSide.getExitIterator -> leftIter[];
rightHandSide.getEnterIterator -> rightIter[];

(while leftIter.hasMoreToGive and rightIter.willAcceptMore do
    leftIter.getNextTransient ->
    rightIter.acceptNextTransient
while);

(if leftIter.hasMoreToGive or rightIter.willAcceptMore then
    (* the iters expected a different number of transfers *)
    'OOPS! Static analysis bug detected, please report!' ->
    internalError
if)
```
Ex. 11-2

Essentially the same process must be carried out during static analysis to ensure that the two iterators will never have a differing number of delivered resp. accepted level zero entities, and to ensure that each pair of delivery and acceptance of a level zero entity will have compatible types. If indeed they had incompatible types in the above example, there would be an `internalError` during the execution of `acceptNextTransient`.

The fully typed representation of all run-time entities and the frequent use of consistency tests (and invocation of `internalError` if a test fails) degrade the performance of gbeta considerably, but it also ensures that the soundness of the static analysis is monitored very closely.

Since one and the same `enter`- or `exit`-list may be associated with different type information when used in different contexts, there is no easy way to store the results of the static analysis that determines the correct behavior of iterators. Hence, the closure based design leads to a repetition of the static analysis of an assignment evaluation (and many other syntactic constructs) for *each execution* of it. Apart from the fact that this is ridiculously slow, it also gives rise to a natural suspicion that there is no static analysis at all. If the "static information" must be computed at run-time again and again then it does not seem to have an appropriate name. Consequently, it would not be particularly convincing to claim that the language supports static type checking.

As a result, it was a high priority task to redesign the execution model in the gbeta implementation. But it was not an easy task, so it remained a high priority task for a long time.

11.6 Code Generation

The improved implementation based on bytecode came into existence in the autumn of 1998, based on a design where a stack based virtual machine executes static bytecodes. With this model, the program is analyzed statically and a list of bytecodes is generated for each imperative in the program. After that, the imperative will be executed by letting the virtual machine read and execute the bytecodes; the syntax is not needed any more, and the execution of bytecodes does not need to obtain information about the involved entities by means of static analysis methods. In other words, the static analysis is actually static.

A virtual machine and its bytecodes may be defined together as a software based simulator for a CPU and its instruction set. Each bytecode may be executed, and the effect of doing that is a simple action which is characteristic for the bytecode but possibly modified by (compile-time fixed) parameters. The main benefit of using a virtual machine based design in the implementation of gbeta is that it allows for the definition of an explicit representation of the semantics of a given program which is at a considerably lower level than the original source code, but at the same time at a level which is high enough to make it reasonably convenient to implement. As a consequence, it becomes possible to inspect such aspects of the semantics as the time and space complexity of individual imperatives in a program. This will be treated in Sect. 11.7.

The virtual machine which is used is a special purpose virtual machine whose instruction set is optimized for the execution of gbeta. It would be tremendously useful, for instance, to be able to compile gbeta down to the kind of bytecode

```
virtualMachine:
  (#
     execute:
       (# program: ^byteCodeList;
          dynamicContext: ^partObject
       enter (program[],dynamicContext[])
       do ...
       #);

     saveFrame: (# ... #);
     restoreFrame: (# ... #);
     resetFrame: (# ... #);

     tmpObjs: @stack(# element::l1ObjectEntity #);
     booleans: @stack(# element::boolean #);
     chars: @stack(# element::char #);
     integers: @stack(# element::integer #);
     reals: @stack(# element::real #);
     strings: @stack(# element::string #);
     objRefs: @stack(# element::l1ObjectEntity #);
     patterns: @stack(# element::l1PatternEntity #)
  #)
```

Figure 11.2: Pseude-code for important parts of the gbeta virtual machine

that Java virtual machines can run. But that is not easy, because each gbeta concept (object, pattern, etc.) does not at all map directly to similar concepts in Java, and the JVM is optimized for handling those concepts [62].

Figure 11.2 presents an overview of the gbeta virtual machine. Each thread has its own virtual machine, so the concurrency support is not visible inside each individual virtual machine. It contains an `execute` method which receives two arguments, a program (a list of bytecodes) and a dynamic context in which to execute that program. It also contains methods to maintain stack frames—marks on the stack of temporary objects that are used as base levels for access to temporaries. For example, if three temporaries are needed for the execution of an AssignmentEvaluation then those temporaries may be pushed on the stack of temporaries in the preparation phase of the execution, and they may then be addressed relative to the base level on the stack of temporaries. They cannot be addressed relative to the *top* of the stack of temporaries because there may be other computations going on, such as the execution of methods which are called as part of the execution of the AssignmentEvaluation.

The virtual machine contains many stacks. There could have been just the stack of temporary objects (`tmpObjs`) and one expression evaluation stack, but the expression evaluation stack has been split into 7 separate stacks, one for each kind of transient value. This makes it possible to maintain the type information about each transient entity, and that again enhances the constant paranoia that the run-time system exhibits towards the soundness of the static analysis. Faults in the type correctness of the dynamic semantics are caught very quickly. An implementation optimized for space and speed would of course just trust the static analysis and store all expression evaluation intermediate values (i.e. transients) on the same stack.

The bytecodes represent the kind of small tasks that the execution of a gbeta program can be divided into. Some bytecodes simply push a compile-time constant value on a stack. Others pop a value from a stack and store it in a given entity (e.g. an object reference may be stored in a variable object attribute, specified by the run-time path which leads to it from the current part object). Yet others may pop two values, add them, and push the result again. The complete list of bytecodes is given in App. D.

There are three levels of bytecodes in the instruction set. First, there are the *simple bytecode* instructions like the abovementioned ones. Then there are byte codes for built-in control structures (such as an `if`-imperative); a bytecode for a control structure is executed at a high level, referring to syntax and in terms of execution of other, syntactically nested imperatives. These *high-level bytecodes* make it possible to use an instruction set that does not contain jumps or labels, and that again allows for a simpler implementation, without compromising the improvements that were obtained by going from closures to bytecodes. Finally there are a group of intermediate bytecodes which are concerned with the semantics of repetitions. In these bytecodes there is sometimes a need to compute the value of an Evaluation (e.g., to access `R[i+1]` it is necessary to compute the value of `i+1`), and that is again expressed in terms of syntax (the syntax '`i+1`'). However, both for repetition related instructions and for control structures, the "high-level" aspect could be further compiled down to simple bytecodes by means of labels and jumps, and the time complexity of executing one or the other would be the same.

In any case, the interesting issues with "the time complexity of a language", to the extent that such a concept is even meaningful, is the time complexity of executing the small basic blocks of the program that use the kind of language defined functionality (such as looking up a name in the current execution environment) that gets invoked everywhere. The gbeta bytecode instruction set supports such investigations just as well with high-level bytecodes as it would have done if they had been compiled away.

Here is an example of a listing of bytecodes which was printed during an interactive execution of a program like the example in the previous section:

```
PUSH-ptn {"myPoint","move"}
NEW, ptn->tmp
PUSHI  3
POP-integer {tmp(1),"dx"}
PUSHI  4
POP-integer {tmp(1),"dy"}
CALL  {tmp(1)}
```

Ex. 11-3

This list was obtained using the command ‘bytecode’ which prints the bytecodes for the current imperative, which was (3,4)->myPoint.move. Note that a list enclosed in braces ({..}) is the gbeta printed representation of a *run-time path*, and each element in a run-time path is a *run-time step*.

A run-time step can be a *lookup step* (such as ‘"myPoint"’), which will cause a *part object local* lookup (and make the most specific part object of the result the current part object, if the result is an object). Note that it is not a full local or global lookup, but just the fetching of an attribute at a statically known offset in the currently selected part object. A run-time step can also be an *out step* (such as ‘<-1’), which simply selects the part object which encloses the currently selected part object (thereby normally also selecting another *object*). A step like ‘<-3’ works like {<-1,<-1,<-1}. A run-time step can also be an *up step* (such as ‘^`258’), which searches through the less specific part objects of the *same* object as the one that is currently selected until it finds the requested one, as specified by the string after the backquote. Backquotes are used to mark parts of the gbeta output as references to pieces of source code (in this case a MainPart), such that it can be double-clicked on when gbeta runs inside Emacs, to jump to that position in the source code. Finally, a run-time step can be a *temporary step* (such as ‘tmp(1)’), which selects the most specific part object of the object in the specified position on the stack of temporaries.

Note that it is statically known exactly what kind of entity is found at each step of a run-time path traversal, and it is in all cases a part object[1] for every step, except for the last step where it may be any kind of run-time entity, e.g. a pattern.

When executed, the bytecodes shown above give rise to the following actions: In the current object, the attribute myPoint is looked up, and in the object which is obtained from that, the attribute move is looked up, yielding a pattern. That pattern is pushed on an expression stack. The second bytecode pops the pattern again, creates a new instance of it, and pushes the new object on the stack of temporary objects. The third bytecode pushes the immediate integer value three unto an expression stack, and the fourth bytecode pops it off again, storing it in the dx attribute of the object which is in the first slot of the stack of temporaries. Similarly, the next two bytecodes store the value four into the dy attribute of the first temporary. Finally the last bytecode calls the do-part of the first temporary, i.e., it calls that instance of the move method.

The bytecodes in the implementation are actually full-fledged BETA objects, with an execute method. They are also capable of printing themselves, and they even store a text string which specifies what source code file and line originally gave rise to the generation of the given bytecode. This means that they are convenient to implement and debug, and that the behavior of the virtual machine is nicely modularized into all the execute methods. It also means that they take up a large amount of space, but a more efficient implementation may benefit from the level of debugging that was facilitated by this implementation,

[1]More precisely, it is a *context* (in the implementation in it an instance of the pattern substanceSlice or a subpattern of that), which may be a part object or the entity associated with an execution of a for imperative, or a few other things.

without inheriting the space efficiency problem.

11.7 Performance Implications

The implementation of gbeta does not allow for meaningful measurements of performance; the execution is so heavily unoptimized that the results would just confirm the fact that any programming language can be implemented in such a way that the performance is too slow for real-world use.

However, the bytecode generation which was briefly introduced in the last section does establish a baseline of information which can be used to argue that the performance need not be that bad. The main observation we need to make is that the generated bytecodes can be used to inspect the micro-level time and space complexity of execution of gbeta programs. Since this *has* been implemented, it proves that an implementation of gbeta with this performance profile (or better) *can* be implemented.

The benefit of having bytecodes to look at, compared to an execution based on closures, is that they allow for a very immediate complexity inspection process. Each simple bytecode has a near-constant execution time. The only element which is not a constant time operation is the traversal of an up step in a run-time path. The up step is currently implemented as a linear search in the list of part objects that constitute the current object, so it is linear in the number of part objects. It is not expected that objects will have more than a few part objects each in the typical case, but even two or three extra pointer indirections for an access to an attribute are a significant overhead when comparing to, e.g., the execution of C code—where we might say that every run-time path has exactly one step, and that is a lookup step.

In general the task of performing an up step is similar to a situation in C++, where the lookup of a data member in presence of virtual inheritance also requires extra work: at first the part object which represents the given superclass must be looked up, and then the member can be looked up at a statically known offset inside that part object. It might be possible to leverage experiences from the handling of this problem to obtain an efficient solution in gbeta.

There are some “expensive” operations in the gbeta semantics. For example, the process of dynamically specializing an object may cause the object to become larger, and that may require that it gets moved in memory. To handle this it is possible to scan the entire pool of objects and update pointers (which makes dynamic object specialization *very* expensive), or it is possible to access every object through an extra level of indirection (which makes *every* object access a bit more expensive, which is bad), or it may be possible to force a garbage collection during which the object being specialized would be relocated anyway, or some objects could be allocated with some “waste” space at the end to be prepared for dynamic specialization, etc. The main point is that new functionality should preferably not cause degradation of the performance of code that does not use it, but if it is performing a task which is both complex and useful then it may actually be allowed to take some time when it *is* used.

Moreover, there may be reasonable trade-offs where it does cost a little bit extra even for code that does not use it. Different compilers may make different trade-offs.

The implementation of gbeta right now just uses the approach which was easiest to implement, degrading the performance in a variety of ways. However, it should be noted that such facilities as dynamic specialization of objects may be included or excluded with any given implementation without affecting the rest of the language. There could be a compiler for the full language gbeta which would not be able to optimize a number of cases. There could be another compiler for gbeta-except-dynamic-specialization, and that might provide better performance for systems that do not need dynamic specialization. There might also be a highly optimizing compiler for gbeta-without-dynamic-allocation (and whatever implies dynamic allocation), and such a language might be convenient for systems with hard real-time constraints. It would allow for FORTRAN-like optimization strategies, because all objects could be allocated at absolute addresses which would be known at compile-time; there would be opportunities for aggressive parallellizing optimizations if not even variable object attributes were allowed, such that no entity could be aliased.

These restrictions would interact gracefully with abstraction mechanisms—for example, it would still be meaningful to be able to express designs incrementally using (possibly propagating) specialization, even if polymorphism were made unavailable because variable objects were excluded.

With such scenarios it would be possible for programmers to reuse their experience with a given programming language in widely different application areas, and if the constraints on a given project were to change then the rewriting to the next-more-constrained or next-more-expressive language might well be more manageable than, say, switching from Smalltalk to Ada.

11.8 Separate Compilation

The fragment system provides gbeta programmers with powerful support for composing large programs by composition of modular units; see Chap. 10 for more information about the fragment system. This works just like in Mjolner BETA, except that the gbeta implementation of the fragment language is more complete. So we may claim that the current gbeta implementation has very good support for modularization of programs.

However, that is not the full story. The current implementation of gbeta does not support separate compilation, and every time a program is executed gbeta will analyze all the fragments all over again and generate new bytecode. That is a consequence of the fact that the internal ASTs (introduced in Sect. 11.3) cannot be stored on disk together with the associated static information.

For technical reasons, the Mjolner BETA persistence support [85] cannot be used to store objects that are involved in concurrency, and that clashes with the implementation of gbeta which uses BETA concurrency extensively. Moreover, it would probably be better to define some optimized representation of the semantic information which can be stored in the Mjolner AST files, or something that works similarly to that, instead of storing all the BETA objects that actually make up the internal ASTs and their associated static information.

Apart from the fact that separate compilation does not happen now, it is not different from the same task for BETA, where it has been implemented and used for many years. Each fragment group F can be analyzed by loading the fragment groups in its domain, $D_1 \ldots D_n$, and binding the fragment forms in F in the context where the corresponding fragment slot is found (that will be in one of $D_1 \ldots D_n$ which is reachable from F via ORIGIN fragment links). Assuming that all the fragment groups $D_1 \ldots D_n$ have been checked, the existing semantic information in $D_1 \ldots D_n$ will be sufficient for the analysis of F, and the code in F will not be able to affect the correctness or content of the static information in $D_1 \ldots D_n$. That is all well-known from BETA, and gbeta does not in itself add new difficulties.

However, the more general implementation of the fragment language which is supported in gbeta does cause a new difficulty. This difficulty would also be present in context of BETA, it is not caused by the extra generality of gbeta. The difficulty is that the support for adding substance attributes (such as object attributes) in an Attributes SLOT makes it impossible to determine at compile-time where in an object a given attribute will be allocated—in other words, it makes it impossible to compile an attribute access operation down to a simple addition of a statically known offset to the address of the object (in gbeta that is an offset to the address of the *part* object).

However, that is a situation which has been handled gracefully in many languages. Sather uses a very small, automatically generated method, a thunk, to look up an attribute whose offset is not known statically, similarly to the implementation of call-by-name in Algol-60 [70]; CLOS, Dylan, Cecil, and others use accessor methods which may be compiled down to simpler constructs (inlined) when sufficient information is available, and dynamic recompilation and method splitting like in Self [22] can be used to obtain better performance at run-time, when a closed world assumption is natural. Similarly, some Objective C implementations[111, 109] set up tables of methods at program loading time (not link time, but just before running the program), and that could also be used to set up offset tables for attributes.

Note that all the attributes which are declared in the same fragment group as the enclosing pattern may have assigned offsets at compile-time, it is only the attributes which are added in other fragment groups that have to have offsets assigned at link-time or later (or they may be looked up in some other way, not using fixed offsets at all), and it is only the patterns that contain a declaration of an Attributes SLOT that may have such attributes. That may very well be resolved quite quickly for most programs. We believe that the potential improvement in the modularization of programs enabled by the full implementation of the fragment language is probably well worth the cost in performance degradation, if any, and in implementation complexity.

Another small implementation problem with the more general support for fragmentation as it is implemented in gbeta is that a given piece of source code in a fragment form, S, may be inserted into more than one context; see Sect. 10.2 for an example. This means that name applications in S may be resolved to different name declarations in different contexts, and consequently they must be annotated with more than one set of static annotations. Similarly, there will have to be more than one portion of generated code for S, and there must be ways to choose what version of static information and generated code to use in different situations. It may be a bit tedious to handle this correctly, but should not present any deep problems.

Chapter 12

The Core Language

This chapter shares material with our paper Propagating Class and Method Combination*, which was accepted for publication and presentation at the ECOOP'99 conference.*

This chapter presents the core of gbeta in an indirect manner, by describing an untyped functional calculus, **gb**. This core expresses the essence of the semantics of object creation and attribute lookup in gbeta, including the semantics of virtual pattern attributes and the combination mechanism. In **gb** there is syntax for specifying a program; moreover, there is a rule outside **gb** for building a pattern from such a program, a rule for creating an object as an instance of a given pattern, and a rule for looking up a name in a given object. This means that it is possible to specify some structure with a **gb** program and then use the rules to explore that structure. This makes it possible to keep **gb** minimal and still enable arbitrary objects and patterns to be created without inventing expression syntax for it.

The abstract syntax for **gb** programs is given in Fig. 12.1. It includes blocks (corresponding to MainParts in gbeta), descriptors (similar to ObjectDescriptors in gbeta), and specifications (the right hand side of declarations). The symbol l denotes a label, i.e., one of a predefined set of identifiers. The only label with a predefined semantics is '`object`' which is the pattern with no mixins; there is no need for the basic mixins like `integer` and `component` because they only affect evaluation semantics, their role in name lookup and pattern merging is no different from that of other mixins; the predefined names like '`integer`' may

$$\begin{array}{lcll}
b & = & (\# \; l_i : s_i \;^{i \in I} \; \#) & \text{(block)} \\
d & = & l \; b & \text{(descriptor)} \\
s & = & l \mid d & \text{(specification)} \\
l & & & \text{(label)}
\end{array}$$

Figure 12.1: The abstract syntax of **gb**

$$\begin{aligned}
\mathsf{Block} &= (\mathsf{Label} * \mathsf{Spec})\ \mathsf{set} \\
\mathsf{Descriptor} &= \mathsf{Label} * \mathsf{Block} \\
\mathsf{Spec} &= \mathsf{Label} \mid \mathsf{Descriptor} \\
\mathsf{Pattern} &= \mathsf{Mixin}\ \mathsf{list} \\
\mathsf{Mixin} &= \mathsf{Env} * \mathsf{Block}
\end{aligned}
\qquad
\begin{aligned}
\mathsf{Env} &= \mathsf{Object}\ \mathsf{list} \\
\mathsf{Object} &= \mathsf{Attribute}\ \mathsf{set} \\
\mathsf{Attribute} &= \mathsf{Label} * \mathsf{EnvSpec}\ \mathsf{list} \\
\mathsf{EnvSpec} &= \mathsf{Env} * \mathsf{Spec}
\end{aligned}$$

Figure 12.2: Semantic Entities

be considered to be declared in a predefined scope which encloses the outermost syntactically described object, '`betaenv`', and which contains no mutable state, only a few patterns. The syntax includes only one kind of attribute declarations, corresponding to virtual pattern attribute declarations in gbeta. Hence, all attributes are virtuals. This is sufficient for the following reasons:

- Simple pattern attributes are (in this untyped world) like virtuals which happen to have no further-bindings.
- The variability of variable patterns plays no role for name lookup, and pattern merging follows the same rules for all patterns no matter what kind of entity they are obtained from.
- Object attributes are looked up according to the same rules as pattern attributes, and merging always happens in terms of patterns.

There is no statement syntax in **gb**, but the rules for creating instances and looking up names can be applied repeatedly, so objects and patterns from anywhere in the program can be created.

The semantic entities are shown in Fig.12.2. They include the syntax as Block, Descriptor, and Spec. The central concept of mixin is represented by Mixin which is a block in an environment. A pattern is simply a list of mixins. An environment, Env, is not only the enclosing object but the list of *all* enclosing objects, ending in the outermost object which contains everything in the program execution. An Object is a set of attributes, and an Attribute is a pair of a label and its value. The value of an attribute is a list of specifications, each in its own environment. It can be thought of as a list of expressions whose names have not yet been looked up, packaged together with *almost* all the information in which they will be looked up when needed. The missing part in the environments for the value of an attribute A is the object of which A is an attribute, and it will be inserted into the environments when the value is being looked up.

Since the result of looking up a label in **gb** is always a Pattern, it would have been natural to use the definition $\mathsf{Attribute} = \mathsf{Label} * \mathsf{Pattern}$, but that definition conflicts with the dynamic semantics for objects which contain self references. The definition of Attribute in Fig. 12.2 is one way to handle recursive objects, namely by evaluating specifications lazily.

$$\begin{array}{rcl}
\textsc{New}(C : \mathsf{Pattern}) & = & \{\ (l, \textsc{Val}(l, C))\ \mid\ l \in \textsc{Labels}(C)\} \\
\forall j \in I.\ \textsc{Val}(l_j, (\#\ l_i : s_i\ {}^{i \in I}\ \#)) & = & s_j \\
\textsc{Val}(l, (e, b) : \mathsf{Mixin}) & = & \begin{cases} [(e, \textsc{Val}(l, b))] & \text{if } l \in \textsc{Labels}(b) \\ [\,], & \text{otherwise} \end{cases} \\
\textsc{Val}(l, [\,] : \mathsf{Pattern}) & = & [\,] \\
\textsc{Val}(l, (h :: t) : \mathsf{Pattern}) & = & \textsc{Val}(l, h) +\!\!+ \textsc{Val}(l, t) \\
\textsc{Labels}((\#\ l_i : s_i\ {}^{i \in I}\ \#)) & = & \{l_i \mid i \in I\} \\
\textsc{Labels}((e, b) : \mathsf{Mixin}) & = & \textsc{Labels}(b) \\
\textsc{Labels}([\,] : \mathsf{Pattern}) & = & \emptyset \\
\textsc{Labels}((h :: t) : \mathsf{Pattern}) & = & \textsc{Labels}(h) \cup \textsc{Labels}(t)
\end{array}$$

Figure 12.3: Creation of objects ($+\!\!+$ concatenates lists)

12.1 From Program to Pattern to Object.

A **gb** program is a block (just like a gbeta program is the `betaenv` ObjectDescriptor, i.e., normally a MainPart). For a given program b we construct the initial pattern $[([\,], b)]$, which contains one mixin which places b in the empty environment; we could also have placed it in an environment containing the handful of predefined patterns, but this approach is the simplest. This pattern can then be instantiated like any other pattern, and that initiates the **gb** 'execution'—which is a chain of evaluations of $\textsc{New}(\cdot)$ and $\textsc{Lookup}(\cdot, \cdot)$.

Any given pattern can be instantiated using the function $\textsc{New}(\cdot)$ which takes a pattern and yields an object. It is defined in terms of the auxiliary functions $\textsc{Labels}(\cdot)$ and $\textsc{Val}(\cdot, \cdot)$. See Fig. 12.3.

Figure 12.4 presents the semantics of attribute lookup. Given an object O and a label l, $\textsc{Lookup}(O, l)$ delivers the result of looking up l in O. It yields a pattern if l is defined in O, and raises an error otherwise. To lookup l in O we search the labels of O using $\mathrm{L_{obj}}(O, \cdot, l)$. If we find l then we have an EnvSpec list, *ess*, which is then looked up in O using $\mathrm{L_{esps}}(O, \mathit{ess})$. Note that *ess* is the result of collecting all contributions to a given attribute—**gb** has virtual attributes, only.

The next step is crucial. The use of $\mathrm{C3}(\cdot, \cdot)$ in the definition of $\mathrm{L_{esps}}(\cdot, \cdot)$ constructs the virtual by linearizing all the contributions. A similar core language for BETA would not linearize at this point; it would *replace* the definition in the less specific enclosing pattern with the definition in the more specific one. Moreover, the static analysis in BETA ensures that this always replaces the virtual pattern with a descendant. Since $A\&B \leq X$ for $X \in \{A, B\}$ and $A\&B = B\&A = B$ whenever $B \leq A$, the BETA semantics comes out as a special case of the gbeta semantics. Finally, $\mathrm{L_{env}}(\cdot, \cdot)$ is used to look up labels in the given environment e, enhanced with the current object to $O :: e$; this (very late)

$$
\begin{array}{rcl}
\textsc{Lookup}(O : \mathsf{Object}, l : \mathsf{Label}) & = & \mathrm{L}_{\mathrm{obj}}(O, O, l) \\
\mathrm{L}_{\mathrm{obj}}(O, [\,] : \mathsf{Object}, l) & = & \text{raise Undefined} \\
\mathrm{L}_{\mathrm{obj}}(O, ((l', ess) :: t) : \mathsf{Object}, l) & = & \begin{cases} \mathrm{L}_{\mathrm{esps}}(O, ess), & \text{if } l = l' \\ \mathrm{L}_{\mathrm{obj}}(O, t, l), & \text{otherwise} \end{cases} \\
\mathrm{L}_{\mathrm{esps}}(O, [\,] : \mathsf{EnvSpec\ list}) & = & [\,] \\
\mathrm{L}_{\mathrm{esps}}(O, (h :: t) : \mathsf{EnvSpec\ list}) & = & \mathrm{C3}(\mathrm{L}_{\mathrm{esp}}(O, h), \mathrm{L}_{\mathrm{esps}}(O, t)) \\
\mathrm{L}_{\mathrm{esp}}(O, (e : \mathsf{Env}, l : \mathsf{Label})) & = & \mathrm{L}_{\mathrm{env}}(O :: e, l) \\
\mathrm{L}_{\mathrm{esp}}(O, (e : \mathsf{Env}, (l, b) : \mathsf{Descriptor})) & = & (O :: e, b) :: (\mathrm{L}_{\mathrm{env}}(O :: e, l)) \\
\mathrm{L}_{\mathrm{env}}([\,] : \mathsf{Env}, l) & = & \begin{cases} [\,], & \text{if } l = \text{"}\mathtt{object}\text{"} \\ \text{raise Undefined}, & \text{otherwise} \end{cases} \\
\mathrm{L}_{\mathrm{env}}((h :: t) : \mathsf{Env}, l) & = & \begin{cases} \textsc{Lookup}(h, l), & \text{if } l \in \textsc{Labels}(h) \\ \mathrm{L}_{\mathrm{env}}(t, l), & \text{otherwise} \end{cases}
\end{array}
$$

Figure 12.4: Looking up a label in an object

enhancement of the environment to include the current object is actually the essence of the lazy evaluation that makes it possible to handle recursion. This ends the brief presentation of **gb**.

12.2 The Relation to gbeta.

The core language **gb** described in the previous section is of course very different from gbeta. It is purely functional, so the **gb** objects (in environments) are replaced with store locations ("pointers") in gbeta. In **gb**, names are matched according to their spelling. Since gbeta uses static name-binding, the identification of names in **gb** is much more inclusive than in gbeta (**gb** considers two declarations related in many cases where gbeta considers them unrelated). To obtain the effect of static name binding in **gb** we would need to rename identifiers in a given program, but since the static analysis of gbeta determines exactly what names are equivalent, it is certainly a tractable problem to choose new names such that only the gbeta-equivalent names are spelled identically.

In **gb**, the immanent recursion of objects is handled using lazy evaluation of attributes. In gbeta, the exact mixins contributing to a given declaration are determined at compile-time,[1] and cycles (e.g., a pattern which indirectly inherits from itself) are detected using a graph coloring algorithm: Whenever the type of a declaration depends on itself, the program is rejected with a 'cyclic dependency' error message. The run-time context is represented *relative* to a current object in the gbeta static analysis, since the actual objects are of course not available before run-time.

[1]Unless we use dynamic features, as described in Chap. 7

However, **gb** accurately reflects the semantics of looking up names in gbeta, starting with declarations in the currently selected mixin (e.g., the method being executed) and continuing through all enclosing objects until the outermost "universe" object is reached. Similarly, the semantics of virtuals is the same in **gb** and gbeta (apart from the name binding issue which was mentioned above). Each attribute includes the full context (potentially many objects) in **gb**, but this has been reduced to one pointer shared by several attributes in gbeta. The semantics comes out clearer and simpler with the complete environment attached to each attribute, that is the reason why **gb** is handled in that way. Finally, note that **gb** does not need to include the explicit linearization operator '`&`' since the semantics of that operator can be obtained using a couple of auxiliary patterns and virtuals. This is because virtual pattern contributions are linearized just like '`&`' expressions.

Chapter 13

Core of the Static Analysis

This chapter presents the core of the static analysis of gbeta, based a core language **eta**. Note that the core language **gb** which was used in Chap. 12 was concerned with the dynamic semantics, whereas **eta** is concerned with the static analysis. These two core languages are different because they have different purposes.

The **eta** language has no expressions, no assignments, no arguments to methods, and no statements. There is a large body of considerations associated with the static analysis of these constructs. However, **eta** is sufficiently rich to illustrate the basic issues and techniques of type analysis in gbeta.

After a presentation of **eta** and the basic concepts behind the analysis, we give a brief presentation of appendix E which contains a specification of the type analysis. Finally the relation to the full analysis of gbeta is discussed.

The syntax of **eta** is shown in figure 13.1. An **eta** program is a Block. This is just a list of attribute declarations, but because of nesting, programs can be complex nevertheless. The 'Meta-variables' summarize the systematic naming; for example, D is a descriptor.

Meta-variables	Non-terminal	Expansion
B	Block	= `"(#"` AttrDecl* `"#)"`;
A	AttrDecl	= NameDecl Declarator ObjectSpec;
δ	Declarator	= `":"` | `":<"` | `"::<"`;
O	ObjectSpec	= Descriptor | NameAppl | Object;
	Object	= `"object"`;
D	Descriptor	= Prefix Block;
P	Prefix	= Object | NameAppl;
a, b, c	NameAppl	= Identifier;
a_d, b_d, c_d	NameDecl	= Identifier;

Figure 13.1: Syntax and meta-variables for mini-gbeta

◇ The analysis of **eta** (and gbeta) is *rooted*: Whatever type information is obtained about a piece of syntax is only valid relative to the current analysis root. The root is always a Block. The analysis of a program is then the analysis of each of its blocks.

◇ The analysis uses *typing entities*. A typing entity describes an aspect of the statically known environment, as seen from the root of the analysis. Typing entities contain indications of positions in the environment, in the form of paths

◇ from the root. The current position during movements is called the *focus*.

Typing Entities:	Step	=	OUT(n) | UP(B);
	Path	=	Step list;
	Mixin	=	Path × Block;
	Type	=	Mixin list;
	Context	=	Path × Type;
	Universe	=	mutable Context set;

It would be cleaner to keep syntax and typing entities separate, but this allows for a shorter presentation. Hence, there are Blocks in Mixins.

A Step OUT(n) describes the movement of focus from a Context to the enclosing Context, repeated n times. E.g., OUT(1) leads to the enclosing Context. A Step UP(B) describes the movement of focus within the current Context towards more general Contexts until one is reached which contains the Block B. A Path is a sequence of Steps. It is always implied that this sequence starts with the root as focus.

A Mixin is a Block together with a Path which leads from the root to the Context which encloses the Mixin, and a Type is a sequence of Mixins. Hence, a Type describes a list of potential mixins, each corresponding to a Block and each positioned in an environment. A Type could be described as the type of a pattern.

A Context is a Type together with a Path which leads to the Context. A Context could be described as the type of an object. Note that the type of an object specifies how it is possible to access exactly *that* particular object at run-time, relative to a root which does of course not exist before run-time.

Finally, a Universe is a collection of information about objects in the environment which is built during analysis. Each Block has its own, unique Universe. This universe is a mutable entity which is brought along everywhere during the analysis and enhanced with every new Context created.

The static analysis is presented in appendix E as a program in pseudo-code in a slightly enhanced version of Standard ML. In the following we will comment on that program in order to make it easier to read. The algorithm executes in a context where the **eta** program is available in the form of an abstract syntax tree, and this syntax tree can be navigated using the '`syn_..`' functions. They do simple things in terms of the structure of abstract syntax trees, like looking up the right hand side of a declaration.

Most functions are partial, and failure is signaled by returning the special bottom value "$\perp$" which either leads to trying something else (by constructs like if Result<>⊥..) or to a failing termination of the algorithm, if delivered as the result of a type request.

After a few auxiliary functions, the typing functions are given, and they specify the core of the analysis. The "U" argument is always the current universe. In the following, the individual typing functions in App. E are briefly presented.

First comes merge which implements the two-list C3 merging algorithm. getFocus searches the Type T for the index of the Mixin whose Block is B.

Two lookup functions follow: blockLookup searches the declarations in the Block B for the given name and delivers the NameDecl together with a Path which leads to the given mixin (ρ',B). It is assumed that ρ leads to a Context containing the mixin. localLookup searches the Context (ρ,T) starting in the Mixin at index focus for the NameAppl a and delivers its NameDecl and the Path to it. Hence, blockLookup searches one part object, and localLookup searches all part objects of a type from a point, in most-specific-first order. lookup, defined later, uses localLookup to search a Context and then continue as necessary with all enclosing Contexts.

gatherVirtualChain delivers a list of NameDecls from the given Context C. Each NameDecl declares a virtual of the same name as the given NameDecl $\mathtt{a}_d$, and getVirtualDecl is used to check that each refers to the same initial ":<" declaration. I.e., gatherVirtualChain delivers the list of contributing declarations to a given virtual.

Next, the typeOf family of functions compute the type of a given piece of syntax, Typable, in the Context C, starting from the Mixin at index focus. Name applications, name declarations, and descriptors have a type.

rawTypeOf determines the type of the given syntax without considering that this syntax may be part of a virtual.

typeOfNameAppl is given a description of the placement of this NameAppl, namely (C,focus), and it uses staticWalk to transform this into the placement (C',focus') of the associated NameDecl. The type of the NameDecl in this environment is then delivered.

typeOfDescriptor delivers the raw type, unless the descriptor is part of a virtual. In that case, the NameDecl which declares this virtual is used to find the type.

typeOfNameDecl tests whether the given NameDecl declares a simple pattern or not. In the first case, the type is the type of the right hand side of the declaration. In the latter case, the declared entity is a virtual pattern, and gatherVirtualChain and merge are used to construct the type of the virtual.

Next, some universe building functions are given. getContext constructs a Context associated with the Block B and located at the end of the Path ρ. This Context may contain any number of Mixins, but one of them is associated with B, and focus is set up to point at that Mixin. enclosingContext constructs the enclosing Context of the given Context C at the given focus. Note that a Context may have several different enclosing Contexts, depending on focus.

The function declOf is special. It looks up the NameDecl associated with the given NameAppl, and the Path which leads from the Context of the NameAppl to the Context of its NameDecl. Note that this analysis is made in the *local* universe of the name application. This ensures that the binding from a name application to its associated declaration is the same, no matter what universe asks for this information. Consequently, name binding is not only static but also invariant with respect to the viewpoint.

Similarly, the function `getVirtualDecl` uses the local universe because it is also a binding between two names and must also be independent of universes, such that any given further- or final-binding will be associated with one and the same virtual attribute from all points of view. The difference to `declOf` is that `getVirtualDecl` computes a binding from one NameDecl to another NameDecl. The fact that the NameDecl of a further- or final-binding declaration is looked up gives it an intermediate position between a name declaration and a name application; this seems in some sense consistent with the fact that many further- or final-bindings provide implementation and are not needed as declarations.

Hence, the path taken from a given name application (or further- or final-binding) to its associated declaration is globally invariant, but each point of view—each Block—has its own type information about the environment, so the `staticWalk` will be in universes of varying richness. The local analysis of a name application always yields the poorest universe, any other viewpoint knows at least as much. This makes it possible to have different types associated with the same name applications, depending on the point of view, and that is necessary for the handling of virtuals. This concludes our walk-through of the functions in App. E.

This type analysis is sufficient to bind names correctly in **eta**, but it does not accept a sufficiently large class of programs as type-safe. This is caused by the fact that types are generally represented as lists of mixins with the implied invariant that the run-time patterns and objects being described by these lists will have *at least* these mixins; in other words, the patterns are assumed to be known by an upper bound in all cases. It is not registered when the list is known to be exact, and this produces a loss of precision which propagates and thereby makes the type analysis vastly more pessimistic with respect to type safety than need be.

To improve on this, the type analysis in gbeta associates some extra information with types. In particular, it is noted whether any contribution to a type is virtual. If this is not the case then the type is known to be exact, and this is a tremendous help when determining type safety. This knowledge comes in two versions: First, if the type of a qualification of a variable attribute is exact, then both evaluation of and assignment to this variable attribute can be determined to be safe, depending on the other parts of the syntactic context which is being type checked. However, if the type of a qualification is only an upper bound then only evaluations can be determined to be safe, assignments will always be unsafe (unless relative information is available, see below). The second version of exactness information is associated with the type of objects. If the type of an object is exact then the type of virtual attributes in that object will also be exact, and this is one of the most important ways to get rid of covariant types in BETA and gbeta, for example by using an object attribute instead of a variable object attribute.

Other information gathered is the relations between virtuals: if we have, e.g., `v::< w(# .. #)` and `w` is virtual then we know that `v` is a specialization of `w` no matter what the rest of the program declares. This kind of information is essential for determining type safety of assignments and argument transfers among references whose declared types are virtual or depend on virtual types. Lower bounds are similarly registered such that they may be exploited, e.g., in case a variable object is being reference assigned.

The explicit merging operator "`&`" is not included in **eta**, but it does not imply any new issues. The machinery for handling virtuals is there, and explicit merging is a simple special case of that.

An obvious question is, "Does the universe make any difference, isn't it just a cache?" The answer is "No, It is not just a cache!" because an inherited attribute could depend on its enclosing contexts. If these contexts were computed from scratch each time then some enclosing objects would be computed without having the complete environment of the more specialized viewpoint. The root of the universe is the most specialized, "most knowledgeable" point, and Mixins further UP may "know less." The result would be that the enclosing objects of inherited parts of an object would receive a too general type. An example is:

```
p: 1(# v:< object; r: 2(# x: @v; .. #)#);
q: p3(# v::< integer; s: r4(# do 5->x #)#);
```

Ex. 13-1

When computing the type of `x` in mixin 4, the type of the virtual `v` is needed, and this depends on the type of the enclosing instance of `p`. In this case, that enclosing instance is a `q`, and the resulting type of `v` should be `integer`. Without the universe, it would be typed as `object` and the program would be rejected.

Note that the analysis does not assume that *any* pattern is non-virtual. Consequently, it is possible to inherit from virtual patterns, to combine virtual patterns, and to do everything with virtual patterns which can be done with ordinary patterns. In BETA it is only possible to create instances of virtual patterns and to use them as qualifications of references.

When inheriting from dynamically constructed patterns, names are bound according to the statically known types. In other words, we can use what we know about from a dynamic pattern, and the rest lies in "darkness." The dark parts of objects may still affect the behavior, because they may contribute to `do`-parts and virtuals, and they may be discovered via `when` imperatives.

Chapter 14

Conclusion

The programming language gbeta was presented. This language is a deep generalization of the language BETA, almost backward compatible but significantly more general already at a very basic level of the semantics. BETA provides virtual attributes and general block structure in context of strict static type checking, and gbeta integrates this with a class and method combination mechanism which propagates through the block structure and thereby enables complex but orderly processes of combination of classes and methods, both at compile-time and at run-time. By enabling programmers to express separate concerns separately and later combine the parts into complete solutions by means of a recursively applied multiple inheritance like mechanism, this represents a new kind of abstraction mechanism which has other mechanisms as special cases and adds new possibilities, too. One way to describe it is as a tightly language integrated support for aspect-oriented programming.

A simple special case of the propagating combination mechanism works similarly to the method combination mechanism in CLOS with `before` and `after` methods, only type safe. Another example is to combine two methods and thereby create a method whose argument types are obtained by combination. The fact that this mechanism extends to the types of method arguments illustrates the tight integration. An example of a run-time mechanism is the dynamic specialization of objects which allows an object to become an instance of a more specialized class; it is also possible to create new classes and new methods at run-time, by recombining the building blocks which are available in the program. A number of constructs not in BETA are available in gbeta, helping to write programs which are more tractable for the static analysis and hence diminish the need for circumventing the type system.

The module system of gbeta was presented; essentially a more complete implementation of the fragment language which is also used in BETA, it mainly serves as an appetizer which shows that significant new possibilities arise by lifting some of the restrictions in the current BETA module system.

The implementation of gbeta was presented briefly, chronologically and architecturally, and with brief glimpses of the approach to execution of programs either via closures or via execution of bytecode by a gbeta-specific virtual machine. It was argued that the implementation was crucial to the language design process, doing language design without being able to run programs is a futile exercise—at least for some people.

Finally, three more formally precise presentations of core aspects of gbeta were presented. First, the linearization algorithm which is at the core of the gbeta semantics was presented, formalized in a declarative manner, generalized, and proved to have certain properties. Next, a small, functional object calculus with virtual attributes and general block structure was presented, giving the basic dynamic semantics of gbeta. Last, another small language was used to present the most essential parts of the static analysis of gbeta.

All in all, this project generated a large amount of experience with language design and implementation, including the creation of a highly non-trivial static analysis and accompanying run-time system. One of the lessons which stand out clearly is the demonstration of how deep a conflict there is between on one side the dynamic freedom to do and change whatever you want, and on the other side the possibility to statically keep track of what may or may not happen at run-time.

Part II

Appendices

Appendix A

Grammar for gbeta

The context-free grammar for gbeta is given below. Terminals are enclosed in single quotes, `'like this'`, and non-terminals are enclosed in angle brackects, `<like this>`.

```
<ObjectDescriptor>          ::= <PrefixOpt> <MainPart>;
<MainPart>                  ::= '(#' <Attributes> <ActionPart> '#)';

<PrefixOpt>                 ::? <Prefix>;
<Prefix>                    ::| <SimplePrefix> | <CompositePrefix>;
<SimplePrefix>              ::= <AttributeDenotation>;
<CompositePrefix>           ::= '(' '&' <Merge> '&' ')';

<Attributes>                ::+ <AttributeDeclOpt> ';';
<AttributeDeclOpt>          ::? <AttributeDecl>;
<AttributeDecl>             ::| <PatternDecl>
                              | <SimpleDecl>
                              | <RepetitionDecl>
                              | <VirtualDecl>
                              | <BindingDecl>
                              | <FinalDecl>;

<PatternDecl>               ::= <Names> ':' <Merge>;

<SimpleDecl>                ::= <Names> ':' <ReferenceSpecification>;

<RepetitionDecl>            ::= <Names> ':' '[' <Index> ']'
                                <ReferenceSpecification>;

<VirtualDecl>               ::= <Names> ':' '<' <DisownOpt> <Merge>
                                <RestrictionOpt>;
<BindingDecl>               ::= <Names> ':' ':' '<' <DisownOpt> <Merge>
                                <RestrictionOpt>;
<FinalDecl>                 ::= <Names> ':' ':' <Merge>;

<DisownOpt>                 ::? <Disown>;
<Disown>                    ::= '-';

<RestrictionOpt>            ::? <RestrictionPart>;
<RestrictionPart>           ::= ':' '>' <Restrictions>;
<Restrictions>              ::+ <AttributeDenotation> ',';

<VariablePattern>           ::=  '##' <AttributeDenotation>;

<ReferenceSpecification>    ::| <StaticItem>
                              | <VirtualStaticItem>
```

```
                                | <FinalStaticItem>
                                | <DynamicItem>
                                | <StaticComponent>
                                | <DynamicComponent>
                                | <VariablePattern>;

<StaticItem>                  ::= '@' <Merge>;

<VirtualStaticItem>           ::= '<' <DisownOpt> '@' <AttributeDenotation>;
<FinalStaticItem>             ::= ':' '@' <AttributeDenotation>;

<DynamicItem>                 ::= '^' <ExactOpt> <AttributeDenotation>;

<ExactOpt>                    ::? <Exact>;
<Exact>                       ::= '=';

<StaticComponent>             ::= '@' '|' <Merge>;
<DynamicComponent>            ::= '^' '|' <ExactOpt> <AttributeDenotation>;

<ObjectSpecification>         ::| <ObjectDescriptor>
                                | <AttributeDenotation>;

<Merge>                       ::+ <ObjectSpecification> '&';

<Index>                       ::| <SimpleIndex> | <NamedIndex>;

<SimpleIndex>                 ::= <Evaluation>;
<NamedIndex>                  ::= <NameDcl> ':' <Evaluation>;

<ActionPart>                  ::= <EnterPartOpt> <DoPartOpt> <ExitPartOpt>;

<EnterPartOpt>                ::? <EnterPart>;
<DoPartOpt>                   ::? <DoPart>;
<ExitPartOpt>                 ::? <ExitPart>;

<EnterPart>                   ::= 'enter' <Evaluation>;
<DoPart>                      ::= 'do' <Imperatives>;
<ExitPart>                    ::= 'exit' <Evaluation>;

<Imperatives>                 ::+ <ImpOpt> ';' ;
<ImpOpt>                      ::? <Imp>;

<Imp>                         ::| <LabelledImp>
                                | <LeaveImp>
                                | <RestartImp>
                                | <InnerImp>
                                | <SuspendImp>
                                | <Evaluation>
                                | <WhileImp>
                                | <WhenImp>;

<LabelledImp>                 ::= <NameDcl> ':' <Imp>;

<ForImp>                      ::= '(' 'for' <Index> 'repeat'
                                      <Imperatives>
                                  'for' ')';

<WhileImp>                    ::= '(' 'while' <Evaluation> 'repeat'
                                      <Imperatives>
                                  'while' ')';

<GeneralIfImp>                ::= '(' 'if' <Evaluation>
                                      <Alternatives>
                                      <ElsePartOpt>
                                  'if' ')';

<SimpleIfImp>                 ::= '(' 'if' <Evaluation> 'then'
                                      <Imperatives>
```

```
                                   <ElsePartOpt>
                               'if' ')';

<WhenImp>                   ::= '(' 'when' <NameDcl> ':'
                                          <AttributeDenotation>
                                   <WhenAlternatives>
                                   <ElsePartOpt>
                               'when' ')';

<LeaveImp>                  ::= 'leave' <NameApl>;
<RestartImp>                ::= 'restart' <NameApl>;

<InnerImp>                  ::= 'inner' <NameAplOpt>;

<NameAplOpt>                ::? <NameApl>;

<SuspendImp>                ::= 'suspend';

<Alternatives>              ::+ <Alternative>;
<Alternative>               ::= <Selections> 'then' <Imperatives>;

<Selections>                ::+ <Selection>;
<Selection>                 ::| <CaseSelection>;
<CaseSelection>             ::= '//' <Evaluation>;

<WhenAlternatives>          ::+ <WhenAlternative>;
<WhenAlternative>           ::= '//' <ExactOpt> <AttributeDenotation> 'then'
                                   <Imperatives>;

<ElsePartOpt>               ::? <ElsePart>;
<ElsePart>                  ::= 'else' <Imperatives>;

<Evaluations>               ::+ <Evaluation> ',';
<Evaluation>                ::| <Expression> | <AssignmentEvaluation>;
<AssignmentEvaluation>      ::= <Evaluation> '->' <Transaction>;

<Transaction>               ::| <Reference>
                              | <ObjectReference>
                              | <EvalList>
                              | <StructureReference>
                              | <ForImp>
                              | <SimpleIfImp>
                              | <GeneralIfImp>;

<Reference>                 ::| <ObjectDenotation>
                              | <DynamicObjectGeneration>
                              | <ComputedObjectEvaluation>
                              | <RepetitionSlice>;

<DynamicObjectGeneration>   ::| <DynamicItemGeneration>
                              | <DynamicComponentGeneration>;

<ObjectDenotation>          ::= <Merge>;
<ComputedObjectEvaluation>  ::= <Reference> '!';
<ObjectReference>           ::= <Reference> '[]';
<StructureReference>        ::= <Merge> '##';
<EvalList>                  ::= '(' <Evaluations> ')';
<DynamicItemGeneration>     ::= '&' <Merge>;
<DynamicComponentGeneration> ::= '&' '|' <Merge>;

<AttributeDenotation>       ::| <NameApl>
                              | <Remote>
                              | <ComputedRemote>
                              | <Indexed>
                              | <ThisObject>
                              | <QualifiedAttrDen>;

<Remote>                    ::= <AttributeDenotation> '.' <NameApl>;
```

```
<ComputedRemote>            ::= '(' <Evaluations> ')' '.' <NameApl>;
<Indexed>                   ::= <AttributeDenotation> '[' <Evaluation> ']';
<ThisObject>                ::= 'this' '(' <NameApl> ')';
<QualifiedAttrDen>          ::= <AttributeDenotation>
                                '(' ':' <Merge> ':' ')';

<Expression>                ::| <RelationalExp> | <SimpleExp>;

<RelationalExp>             ::| <EqExp> | <LtExp> | <LeExp>
                              | <GtExp> | <GeExp> | <NeExp>;

<SimpleExp>                 ::| <AddExp> | <SignedTerm> | <Term>;

<AddExp>                    ::| <PlusExp> | <MinusExp> | <OrExp> | <XorExp>;

<SignedTerm>                ::| <UnaryPlusExp> | <UnaryMinusexp>;

<Term>                      ::| <MulExp> | <Factor>;

<MulExp>                    ::| <TimesExp> | <RealDivExp> | <IntDivExp>
                              | <ModExp> | <AndExp>;

<EqExp>                     ::= <SimpleExp> '=' <SimpleExp>;
<LtExp>                     ::= <SimpleExp> '<' <SimpleExp>;
<LeExp>                     ::= <SimpleExp> '<=' <SimpleExp>;
<GtExp>                     ::= <SimpleExp> '>' <SimpleExp>;
<GeExp>                     ::= <SimpleExp> '>=' <SimpleExp>;
<NeExp>                     ::= <SimpleExp> '<>' <SimpleExp>;

<PlusExp>                   ::= <SimpleExp> '+' <Term>;
<MinusExp>                  ::= <SimpleExp> '-' <Term>;
<OrExp>                     ::= <SimpleExp> 'or' <Term>;
<XorExp>                    ::= <SimpleExp> 'xor' <Term>;

<UnaryPlusExp>              ::= '+' <Term>;
<UnaryMinusExp>             ::= '-' <Term>;

<TimesExp>                  ::= <Term> '*' <Factor>;
<RealDivExp>                ::= <Term> '/' <Factor>;
<IntDivExp>                 ::= <Term> 'div' <Factor>;
<ModExp>                    ::= <Term> 'mod' <Factor>;
<AndExp>                    ::= <Term> 'and' <Factor>;

<Factor>                    ::| <TextConst>
                              | <IntegerConst>
                              | <RealConst>
                              | <NotExp>
                              | <NoneExp>
                              | <Transaction>;

<RepetitionSlice>           ::= <AttributeDenotation> '[' <Low:Evaluation>
                               ':' <High:Evaluation> ']';
<NotExp>                    ::= 'not' <Factor>;
<NoneExp>                   ::= 'none';

<Names>                     ::+ <NameDcl> ',';
<NameDcl>                   ::= <Name>;
<NameApl>                   ::= <Name>;

<IntegerConst>              ::= <SignOpt> <Natural>;
<SignOpt>                   ::? <Sign>;

<RealConst>                 ::= <IntegerConst> '.' <Natural> <ExpOpt>;
<ExpOpt>                    ::? <Exp>;
<Exp>                       ::= <ExpMark> <IntegerConst>;
```

At the lexical level the specifications are quite simple, except for `<TextConst>` which defines the format of literal strings. A `<TextConst>` is enclosed in single quotes and may contain C-like escape sequences, e.g., `'\n'` is a literal string containing one character, namely a newline. Because of these complications we omit a precise definition of `<TextConst>`. Apart from that, the lexical level can be specified as follows:

```
<Name>                          = "[A-Za-z_][A-Za-z0-9_]*";
<Natural>                       = "0" | "[1-9][0-9]*";
<Sign>                          = "[+-]";
<ExpMark>                       = "[Ee]";
<TextConst>                     = ...;
```

Appendix B

Linearization Proofs

Proof: (**Proposition 1**) Let $R \triangleq (R_1 \cup R_2)^*$ as in the definition, and let $S \triangleq \mathrm{dom}(R_1) \times \mathrm{dom}(R_2) \setminus \overline{R}$. Observe that R and $\overline{R}$ are reflexive because union, transitive closure, and inversion preserve reflexivity. Moreover, since $\mathrm{dom}(S) \subseteq \mathrm{dom}(R_1) \cup \mathrm{dom}(R_2) = \mathrm{dom}(R)$, also $R_1 \lhd R_2$ is reflexive—"S does not touch any new elements compared to R."

For transitivity, note that $R_1 \lhd R_2 = R \cup S$, and $(x, y) \in S \Rightarrow (y, x) \notin R$. Assume $(x, y), (y, z) \in R_1 \lhd R_2$. We show that $(x, z) \in R_1 \lhd R_2$:

- If $(x, y), (y, z) \in R$, then $(x, z) \in R \subseteq R_1 \lhd R_2$ because R is transitive.
- If $(x, y) \in R$ and $(y, z) \in S$ then $(z, y) \notin R$ by definition of S. If $(z, x) \in R$ then by transitivity of R we get $(z, y) \in R$, contradiction, hence $(z, x) \notin R$. Observe that $y \in \mathrm{dom}(R_1)$ and $z \in \mathrm{dom}(R_2)$ because $(y, z) \in S$. If $x \in \mathrm{dom}(R_1)$ then $(x, z) \in \mathrm{dom}(R_1) \times \mathrm{dom}(R_2) \setminus \overline{R} = S \subseteq R_1 \lhd R_2$. Otherwise $x \in \mathrm{dom}(R_2)$, but then $(z, x) \in \mathrm{dom}(R_2)^2$, and by totality of R_2, $(x, z) \in R_2 \vee (z, x) \in R_2$. Since $(z, x) \in R_2$ contradicts $(z, x) \notin R$ we must have $(x, z) \in R_2 \subseteq R_1 \lhd R_2$.
- The case $(x, y) \in S$ and $(y, z) \in R$ is similar.
- If $(x, y), (y, z) \in S$ then $x \in \mathrm{dom}(R_1)$, $y \in \mathrm{dom}(R_1) \cap \mathrm{dom}(R_2)$, and $z \in \mathrm{dom}(R_2)$. Moreover $(y, x), (z, y) \notin R$, by definition of S. Then $(x, y) \in R_1$ by totality of R_1, and $(y, z) \in R_2$ by totality of R_2, hence $(x, y), (y, z) \in R$ and by transitivity of R finally $(x, z) \in R \subseteq R_1 \lhd R_2$.

For totality of $R_1 \lhd R_2$ we choose arbitrary $x, y \in \mathrm{dom}(R_1 \lhd R_2)$, and show that either $(x, y) \in R_1 \lhd R_2$ or $(y, x) \in R_1 \lhd R_2$:

- If $x, y \in \mathrm{dom}(R_1)$ then $(x, y) \in R_1 \vee (y, x) \in R_1$ by totality or R_1.
- If $x \in \mathrm{dom}(R_1)$ and $y \in \mathrm{dom}(R_2)$ then either $(y, x) \in R$ or $(x, y) \in S$.
- The remaining two cases are similar.

This proves that $R_1 \lhd R_2$ is reflexive, transitive, and total, i.e. it is a total preorder. $\square$

Proof: (**Lemma 3**) Given an acyclic relation R. With $R_0 \triangleq R$, $\forall i \in \omega.\ R_{i+1} \triangleq R_i^{+1}$ we have $R^* = \bigcup_{i\in\omega} R_i$. Assume that R^* has a cycle and let $k \in \omega$ be the least number such that R_k has a cycle, say $d_1 \ldots d_n$; then $k > 0$ because R is acyclic. Since

$$\{(d_i, d_{i+1}) | i \in 1 \ldots n-1\} \cup \{(d_n, d_1)\} \subseteq R_k$$

and $R_k = R_{k-1}^{+1}$ we can choose $c_1 \ldots c_n \in \mathrm{dom}(R)$ such that

$$\begin{array}{l} \quad \forall i \in \{1 \ldots n-1\}.\ (\{(d_i, c_i), (c_i, d_{i+1})\} \subseteq R_{k-1}) \vee ((d_i, d_{i+1}) \in R_{k-1})) \\ \wedge \\ \quad (\{(d_n, c_n), (c_n, d_1)\} \subseteq R_{k-1}) \vee ((d_n, d_1) \in R_{k-1}) \end{array}$$

which provides us with a cycle in R_{k-1}, contradicting the minimality of k. $\square$

Proof: (**Proposition 2**) Since R_1 and R_2 are total preorders we get reflexivity, transitivity, and totality directly from proposition 1. Only anti-symmetri remains to be proved. Assume that $(x,y), (y,x) \in R_1 \lhd R_2$; we must then prove that $x{=}y$. Let $R \triangleq (R_1 \cup R_2)^*$ and $S \triangleq \mathrm{dom}(R_1) \times \mathrm{dom}(R_2) \setminus \overline{R}$ such that $R_1 \lhd R_2 = R \cup S$ and $(x,y) \in S \Rightarrow (y,x) \notin R$.

- If $(x,y), (y,x) \in R$ then $x = y$ since R is acyclic, by lemma 3.
- Both $(x,y) \in R \wedge (y,x) \in S$ and $(y,x) \in R \wedge (x,y) \in S$ are impossible by definition of S.
- Similarly, if $(x,y), (y,x) \in S$ then $(y,x), (x,y) \notin R$. This is a contradiction since $x, y \in \mathrm{dom}(R_1) \cap \mathrm{dom}(R_2)$ and in particular by totality of R_1, $(x,y) \in R_1 \vee (y,x) \in R_1 \subseteq R$.

$\square$

Appendix C

The Expression Problem

This appendix contains the original presentation of the expression problem which spurred the discussion on the `java-genericity` mailing list.

The Expression Problem
Philip Wadler, 12 November 1998

The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts). For the concrete example, we take expressions as the data type, begin with one case (constants) and one function (evaluators), then add one more construct (plus) and one more function (conversion to a string).

Whether a language can solve the Expression Problem is a salient indicator of its capacity for expression. One can think of cases as rows and functions as columns in a table. In a functional language, the rows are fixed (cases in a datatype declaration) but it is easy to add new columns (functions). In an object-oriented language, the columns are fixed (methods in a class declaration) but it is easy to add new rows (subclasses). We want to make it easy to add either rows or columns.

The Expresion Problem delineates a central tension in language design. Accordingly, it has been widely discussed, including Reynolds (1975), Cook (1990), and Krishnamurthi, Felleisen and Friedman (1998); the latter includes a more extensive list of references. It has also been discussed on this mailing list by Corky Cartwright and Kim Bruce. Yet I know of no widely-used language that solves The Expression Problem while satisfying the constraints of independent compilation and static typing.

Until now, that is. Here I present a solution to this problem in GJ, as extended by the mechanism I described in my previous note ‘Do parametric types beat virtual types?’. (However, there is a caveat with regard to inner interfaces, see below.)

1. A solution

Figure 1 shows a solution to the Expression Problem in GJ. The two phases of the problem are clumped into two classes, LangF and Lang2F, each of which defines several mutually recursive inner classes and interfaces.

In the first phase, the class LangF define an interface Exp with a subclass Num representing constants, and an interface Visitor with a method forNum specifying functions over constants. The Visitor class is parameterized on the result type of the function. Class Eval implements Visitor<Integer> and specifies evaluation of expressions.

In the second phase, the class Lang2F extends Exp with an additional subclass Sum representing the sum of two expressions, and extends Visitor with an additional method forSum specifying how to act on sums. Class Eval is extended appropriately, and class Show implements Visitor<String> and specifies conversion of an expression to a string. Finally, a test class creates, evaluates, and shows expressions from both phases.

The class Eval in the second phase extends the class Eval from the first phase and implements the interface Visitor from the second phase. So it is essential that Visitor be an interface, not an abstract class.

The LangF class is parameterised on a type parameter This that is itself bounded by LangF<This>, and the Lang2F class is parameterized on a type parameter This that is bounded by Lang2F<This>; further, Lang2F<This> extends LangF<This>. This use of ‘This’ is the standard trick to provide accurate static typing in the prescence of subtypes (sometimes called MyType or ThisType). As usual, we tie the knot with fixpoint classes Lang and Lang2.

The key trick here is the use of This.Exp and This.Visitor, via the mechanism described in ‘Do parametric types beat virtual types?’. Recall that mechanism allows a type variable to be indexed by any inner class defined in the variable’s bound; in order for this to be sound, any type which instantiates a type parameter must define inner classes that extend those in the bound. Here we can refer to This.Exp and This.Visitor because This is bound by LangF<This> which defines Exp and Vistor; soundness is satisfied since Lang2F<This>.Exp extends LangF<This>.Exp, and Lang2F<This>.Visitor extends Lang2F<This>.Visitor.

This solution is remarkably straightforward, once one is familiar with the techniques for simulating ThisType and virtual types. However, I must admit it took me a while to see the solution, even after I went looking for it. (Some of you will have seen an earlier solution, similar in structure but impossible to implement since it had the type variable This.Exp as a supertype of Num and Sum; the current version has no such problem.)

```
-----------------------------------------------------------------------
Figure 1:  A solution to The Expression Problem
-----------------------------------------------------------------------
```

```
class LangF<This extends LangF<This>> {
  interface Visitor<R> {
    public R forNum(int n);
  }
  interface Exp {
    public <R> R visit(This.Visitor<R> v);
  }
  class Num implements Exp {
    protected final int n_;
    public Num(int n) {n_=n;}
    public <R> R visit(This.Visitor<R> v) {
      return v.forNum(n_);
    }
  }
  class Eval implements Visitor<Integer> {
    public Integer forNum(int n) {
      return new Integer(n);
    }
  }
}
final class Lang extends LangF<Lang> {}

class Lang2F<This extends Lang2F<This>> extends LangF<This> {
  interface Visitor<R> extends LangF<This>.Visitor<R> {
    public R forPlus(This.Exp e1, This.Exp e2);
  }
  class Plus implements Exp {
    protected final This.Exp e1_,e2_;
    public Plus(This.Exp e1, This.Exp e2) {e1_=e1; e2_=e2;}
    public <R> R visit(This.Visitor<R> v) {
      return v.forPlus(e1_,e2_);
    }
  }
  class Eval extends LangF<This>.Eval implements Visitor<Integer> {
    public Integer forPlus(This.Exp e1, This.Exp e2) {
      return new Integer(
        e1.visit(this).intValue() + e2.visit(this).intValue()
      );
    }
  }
  class Show implements Visitor<String> {
    public String forNum(int n) {
      return Integer.toString(n);
    }
    public String forPlus(This.Exp e1, This.Exp e2) {
      return "(" + e1.visit(this) + "+" + e2.visit(this) +")";
    }
  }
}
final class Lang2 extends Lang2F<Lang2> {}

final class Main {
  static public void main(String[] args) {
    Lang l = new Lang();
    Lang.Exp e = l.new Num(42);
    System.out.println("eval: " + e.visit(l.new Eval()));
    Lang2 l2 = new Lang2();
    Lang2.Exp e2 = l2.new Plus(l2.new Num(5), l2.new Num(37));
    System.out.println("eval: "+e2.visit(l2.new Eval()));
    System.out.println("show: "+e2.visit(l2.new Show()));
  }
}
```

----

2. A caveat with regard to inner interfaces

In GJ as it is currently implemented, type parameters do not scope over static members, and further, a type parameter may be indexed only by non-static classes or interfaces defined in the bound. And in Java as it is currently defined, all inner interfaces are taken as static, whether declared so are not. This makes the mechanism for indexing type variables by inner classes useless for interfaces, greatly reducing its utility. In particular, it invalidates the solution just presented, which depends on Visitor being an interface.

Fortunately, it looks possible to relax either the GJ or the Java constraint. So far as I can see, the only difference in making an interface non-static is that it can now include non-static inner classes; so the change would not render invalid any existing Java programs. But this point requires further study. Also, I should note that since the changes have not been implemented yet, I have not actually run the proposed solution. (I did translate from GJ to Java by hand, and run that.)

3. Related work

It is instructive to compare this solution with previous solutions circulated by Corky Cartwright and Kim Bruce. Corky's solution requires contravariant extension -- that is, even though Lang2F.Exp extends LangF.Exp, one may use LangF.Exp in place of Lang2F.Exp and not conversely. This partly explains why fixpoints are required here: though LangF is a superclass of Lang2F, the classes Lang and Lang2 are unrelated. Short of complicating the language with contravariance, unrelated classes is the best we could hope for.

```
          LangF
          /   \
      Lang     Lang2F
               /
          Lang2
```

Kim's solution required a type to be parameterized over a type constructor (rather than another type). In terms of our example, it required Exp to be parameterized on Visitor. Here, instead of paramerizing Exp on Visitor, Exp refers to This.Vistor<R>. Although GJ supports parametization over types, it does not support parameterization over higher-order type constructors. However, virtual types (as simulated by GJ) in effect support higher-order type parameters for free. I'm grateful to Mads Torgersen and Kresten Krab Thorup for this insight, which they passed on when we discussed this problem at OOPSLA a few weeks ago. (Ironically, though, it looks like this solution won't work in Beta, which lacks interfaces or any other form of multiple supertyping; there also may be a problem in having a single expression type that allows visitors with different result types, like Integer and String.)

The solution presented here is similar to the Extended Visitor pattern described by Krishnamurthi et al. Their solution differs in that it is not statically typed; they cannot distinguish Lang.Exp from Lang2.Exp, and as a result must depend on dynamic casts at some key points. This isn't due to a lack of cleverness on their part, rather it is due to a lack of expressiveness in Pizza.

I am aware of two solutions to the expression problem, but both depend on special-purpose language extensions designed specifically for that problem. One appears in the Krishnamurthi et al. paper, the other in a master's thesis by a student of Martin Odersky. In contrast, the solution presented here arises from the general purpose mechanisms of type parameters and virtual types.

I'd be grateful for pointers to other solutions to the Expression Problem. How do Beta, Sather, Ocaml, and others fare?

Cheers, -- P

```
References

W. R. Cook (1990). Object-oriented programming versus abstract data
  types.  REX workshop on Foundations of Object-Oriented Languages,
  Springer-Verlag LNCS 489, 1990.

S. Krishamurthis, M. Felleisen, and D. Friedman (1998).  Synthesizing
  object-oriented and functional design to promote re-use.  ECOOP 1998,
  Springer-Verlag LNCS 1445, July 1998.

J. C. Reynolds (1975).  User-defined types and procedural data as
  complementary approaches to data abstraction.  In S. A. Schuman,
  editor, New Directions in Algorithmic Languages, IFIP Working Group
  2.1 on Algol, INRIA, 1975.  Reprinted in D. Gries, editor, Programming
  Methodology, Springer-Verlag, 1978, and in C. A. Gunter and
  J. C. Mitchell, editors, Theoretical Aspects of Object-Oriented
  Programming, MIT Press, 1994.
```

Appendix D

Bytecode Instruction Set

`ADD-mainpart` *`mainpart where`* **:** Pop a pattern P, then add a new, most specific mixin M to P which is associated with the given *`mainpart`* and whose origin can be found by traversing the run-time path *`where`*. Then push the resulting pattern $P\&[M]$.

`ADDOP( + )` *`type`* **:** Pop two values of type *`type`*, add them (yielding a result of type *`type`*), and push the result.

`ADDOP( - )` *`type`* **:** Pop two values of type *`type`*, subtract the first from the second (yielding a result of type *`type`*), and push the result.

`ADDOP( or )` **:** Pop two boolean values and push the boolean value which is the logical disjunction of them.

`ADDOP( xor )` **:** Pop two boolean values and push the boolean value `false` if they are equal and true otherwise.

`CALL` *`where`* **:** Execute the object found by traversing the run-time path *`where`*.

`CALL-rep` *`rdecl where kind`* **:** Lookup the repetition by traversing the run-time path *`where`*. If the *`kind`* is object then it is a repetition of objects; execute each of them in index order (execute `rep[1]`, then `rep[2]`, etc.). If the *`kind`* is variable object then it is a repetition of variable objects; for each of them, in index order, raise a run-time error if it is NONE and otherwise execute it. If *`kind`* is variable pattern, then it is a repetition of patterns; for each of them, in index order, create an instance of the pattern and execute it. The declaration of the repetition, *`rdecl`*, is only stored for documentation and debugging purposes.

`CHK NONE` **:** Peek^p an object reference and raise a run-time error if it is NONE, otherwise do nothing.

`CHK PTN NONE` **:** Peek^p a pattern and raise a run-time error if it is NONE, otherwise do nothing.

`ENSURE-component` : Pop a pattern P, then push the pattern `component` & P. Note that `component` & P is the same pattern as P if P is already less-equal than `component`. Needed for expressions like `&|p` when `p` is a pattern variable whose value may or may not be less-equal than `component`.

`EXTEND-rep` *`rdecl`* *`where`* : Lookup a repetition by traversing the run-time path *`where`*. Pop an integer value N and extend the repetition with N entries. The declaration of the repetition, *`rdecl`*, is only stored for documentation and debugging purposes.

`FORK` *`where`* : Lookup a `component` part object by traversing the run-time path *`where`*. Fork a new thread which runs that `component`. Note that this fails if the `component` is already run by another thread, even if it is suspended.

`GETSIZE-rep` *`rdecl`* *`where`* : Lookup a repetition by traversing the run-time path *`where`*. Push the number of entries in the repetition as an integer value. The declaration of the repetition, *`rdecl`*, is only stored for documentation and debugging purposes.

`generalIf` : Evaluate^j the Evaluation of the GeneralIfImp imperative, yielding a value V. For each Alternative, in the order they appear in the source code, evaluate each Selection, also in the order they appear in the source code, until an evaluation yields the value V. At this point, stop the evaluations and execute the list of imperatives for that Alternative. Otherwise, if no evaluation yields the value V, execute the list of imperatives for the ElsePart.

`KILL` *`where`* : Lookup a `component` part object by traversing the run-time path *`where`*. Kill the thread that runs this `component`. Note that it is an error to kill a `component` that is not running.

`locatedSimpleIf` *`where`* : Lookup^j a context C by traversing the run-time path *`where`*. In context of C, evaluate the Evaluation of the SimpleIfImp imperative which was provided as a parameter to the initialization of this bytecode at compile-time. Obtain the result of the evaluation by popping a boolean value V. If V is true then execute the imperatives in the `then`-part of the SimpleIfImp, else execute the imperatives in the `else`-part.

`MERGE-ptn` : Pop a pattern P_1 and pop a pattern P_2, then push the pattern $P_2 \& P_1$.

`MULOP( * )` *`type`* : Pop two values of type *`type`*, multiply them (yielding a result of type *`type`*), and push the result.

`MULOP( / )` : Pop two real values, divide the second by the first (yielding a real result), and push the result.

`MULOP( and )` : Pop two boolean values and push the boolean value which is the logical conjunction of them.

`MULOP( div )` : Pop two integer values, divide the second by the first, and push the result. This instruction actually also handles real values, because the Mjolner compiler allows things like '`1.2 div 2.4`', but it is not recommended to use this facility.

`MULOP( mod )` : Pop two values of type ***type***, find the value of the second modulo the first, and push the result which is also of type ***type***. Note that the only allowed value type is integer. It may be changed to support real values as well.

`NEG(integer)` : Pop an integer value N and push the integer value $-N$.

`NEG(real)` : Pop a real value R and push the real value $-R$.

`NEW, ptn->obj` : Pop a pattern P, create a new instance O of P, and push a reference to O.

`NEW, ptn->tmp` : Pop a pattern P, create a new instance O of P, and push a reference to O unto the stack of temporary objects.

`NEW-rep` ***rdecl where*** : Lookup a repetition by traversing the run-time path ***where***. Remove all entries from the repetition, pop an integer value N and create N new entries. The declaration of the repetition, ***rdecl***, is only stored for documentation and debugging purposes.

`NOT` : Pop a boolean value B and push the boolean value $\neg B$.

`namedFor` : Create^j a `for` substance entity (to hold the index variable). Evaluate the Evaluation of the NamedIndex of the NamedForImp imperative which was provided as a parameter to the initialization of this bytecode at compile-time. Obtain the result of the evaluation by popping an integer value N. Execute the Imperatives of the NamedForImp N times, with the index variable bound to i during the i'th iteration.

`osSystem/in` : Pop a string value, execute this as an operating system command. The standard output and the standard error streams are sent to the same destinations as the standard output/error of the gbeta run-time. For further information and a disclaimer see `osSystem/inout` .

`osSystem/inout` : Pop a string value, execute this as an operating system command, and push the resulting standard output as a string value. The standard error stream is sent to the same destination as the standard error of the gbeta run-time. This operation is sometimes useful, but it is of course a purely pragmatic, non-portable facility which is not part of the language design. It uses the UNIX system call `exec`.

`PEEK-inx-objref` ***where eval evalWhere*** : Lookup a repetition R of variable objects by traversing the run-time path ***where***. Lookup a dynamic context for the evaluation of the Evaluation ***eval*** by traversing the run-time path ***evalWhere***. Evaluate^j ***eval*** and pop the integer value N that this produces. Check that N is a valid repetition index (greater than zero, less-equal than the number of elements in R). Peek[p] an object reference and store it in R at entry N.

`PEEK-inx-tmpref` *`where eval evalWhere`* **:** Lookup a repetition R of variable objects by traversing the run-time path *`where`*. Lookup a dynamic context for the evaluation of the Evaluation *`eval`* by traversing the run-time path *`evalWhere`*. Evaluate[j] *`eval`* and pop the integer value N that this produces. Check that N is a valid repetition index (greater than zero, less-equal than the number of elements in R). Peek[p] an object reference from the stack of temporary objects and store it in R at entry N.

`PEEK-objref` *`where`* **:** Lookup a variable object attribute A by traversing the run-time path *`where`*. Peek[p] an object reference r. Check that r is `NONE` or that it refers to an object which has a pattern which is less-equal than the qualification of A. Store r in A if the qualification test succeeds, otherwise raise a run-time error.[q]

`PEEK-tmpref` *`where`* **:** Lookup a variable object attribute A by traversing the run-time path *`where`*. Peek[p] an object reference r from the stack of temporary objects. Check that r is `NONE` or that it refers to an object which has a pattern which is less-equal than the qualification of A. Store r in A if the qualification test succeeds, otherwise raise a run-time error.[q]

`POP-boolean` *`where`* **:** Lookup an object O by traversing the run-time path *`where`*. Find the boolean part object o in O. Pop a boolean value B and change the state of o to B.

`POP-boolean-value` *`where`* **:** Lookup a `boolean` part object o by traversing the run-time path *`where`*. Pop a boolean value B and change the state of o to B.

`POP-char` *`where`* **:** Lookup an object O by traversing the run-time path *`where`*. Find the char part object o in O. Pop a char value C and change the state of o to C.

`POP-char, C-->I, PUSH-integer` **:** Pop a char value, coerce[c] it into the corresponding integer value, and push it.

`POP-char, C-->R, PUSH-real` **:** Pop a char value, coerce[cr] it into the corresponding real value, and push it.

`POP-char, C-->S, PUSH-string` **:** Pop a char value, coerce it into the corresponding string value (a string of length 1 containing that character), and push it.

`POP-char-value` *`where`* **:** Lookup a `char` part object o by traversing the run-time path *`where`*. Pop a char value C and change the state of o to C.

`POP-int, PUSH-char-at-inx` *`where`* **:** Lookup a `string` part object o by traversing the run-time path *`where`*. Pop an integer value N. Check that N is greater than zero and less-equal than the length of the string. If the test succeeds then push the char value in the string value of o at index N. If the test fails then raise a run-time error.

`POP-integer` *`where`* **:** Lookup an object O by traversing the run-time path *`where`*. Find the `integer` part object o in O. Pop an integer value N and change the state of o to N.

`POP-integer, I-->C, PUSH-char` **:** Pop an integer value, coerce[c] it into the corresponding char value, and push it.

`POP-integer, I-->R, PUSH-real` **:** Pop an integer value, coerce[c] it into the corresponding real value, and push it.

`POP-integer-value` *`where`* **:** Lookup an `integer` part object o by traversing the run-time path *`where`*. Pop an integer value N and change the state of o to N.

`POP-inx-objref` *`where eval evalWhere`* **:** Lookup a repetition R of variable objects by traversing the run-time path *`where`*. Lookup a dynamic context for the evaluation of the Evaluation *`eval`* by traversing the run-time path *`evalWhere`*. Evaluate[j] *`eval`* and pop the integer value N that this evaluation produces. Check that N is a valid repetition index (greater than zero, less-equal than the number of elements in R). Pop an object reference r. Check that r is `NONE` or that it refers to an object that has a pattern which is less-equal than the qualification of R (a repetition of variable objects has one shared qualification for all its entries). Store r in R at entry N if the qualification test succeeds, otherwise raise a run-time error.[q]

`POP-inx-ptnref` *`where eval evalWhere`* **:** Lookup a repetition R of variable patterns by traversing the run-time path *`where`*. Lookup a dynamic context C for the evaluation of the Evaluation *`eval`* by traversing the run-time path *`evalWhere`*. Evaluate[j] *`eval`* in context of C and pop the integer value N that this produces. Check that N is a valid repetition index (greater than zero, less-equal than the number of elements in R). Pop a pattern (or `NONE`) p. Check that p is `NONE` or a pattern which is less-equal than the qualification of R (a repetition of patterns has one shared qualification for all its entries). Store p in R at entry N if the qualification test succeeds, otherwise raise a run-time error.[q]

`POP-obj, O-->P, PUSH-ptn` **:** Pop a reference to an object O, obtain the pattern of O, and push that pattern.

`POP-obj, PUSH-tmp` **:** Pop an object reference and push it on the stack of temporary objects.

`POP-objref` *`where`* **:** Lookup a variable object attribute A by traversing the run-time path *`where`*. Pop an object reference r. Check that r is `NONE` or that it refers to an object that has a pattern which is less-equal than the qualification of A. Store r in A if the qualification test succeeds, otherwise raise a run-time error.[q]

`POP-ptn, SPECIALIZE-obj` *`where`* **:** Lookup an object O by traversing the run-time path *`where`*. Pop a pattern P. Dynamically specialize O such that it becomes an instance of pattern $P_o\&P$, where P_o is the pattern of which O was an instance before this operation. Note that this may both add more part objects to O and change pattern values, e.g., virtual patterns may become further-bound and qualifications may become more special.

`POP-ptnref` *`where`* **:** Lookup a variable pattern attribute A by traversing the run-time path *`where`*. Pop a pattern (or `NONE`) p. Check that p is `NONE` or a pattern which is less-equal than the qualification of A. Store p in A if the qualification test succeeds, otherwise raise a run-time error.[q]

`POP-real` *`where`* **:** Lookup an object O by traversing the run-time path *`where`*. Find the `real` part object o in O. Pop a real value R and change the state of o to R.

`POP-real, R-->I, PUSH-integer` **:** Pop a real value, coerce[c] it into the corresponding integer value, and push it.

`POP-real-value` *`where`* **:** Lookup a `real` part object o by traversing the run-time path *`where`*. Pop a real value R and change the state of o to R.

`POP-string` *`where`* **:** Lookup an object O traversing the run-time path *`where`*. Find the `string` part object o in O. Pop a string value S and change the state of o to S.

`POP-string --> [char]` *`rdecl where`* **:** Lookup a repetition R of entities with pattern `char` or a subpattern of `char`, by traversing the run-time path *`where`*. Pop a string value V. Adjust the number of items in R to be the same as the length of V. Then assign the char values in V to the entries in R one by one: the first char value in V is assigned to the first entity in R, then the second, etc. The declaration of the repetition, *`rdecl`*, is only stored for documentation and debugging purposes.

At compile-time when this bytecode is generated, it is initialized to handle one of three cases (there could as well have been three bytecodes): where R is a repetition of objects, where R is a repetition of references to objects, and where R is a repetition of patterns. In the first case, each assignment of the iteration obtains the current entry of R, which is an object O_i, and stores the current char value from V in the `char` part object of O_i. In the second case, each assignment obtains the current entry of R, which is an object reference r_i, checks that it is not `NONE`, raises a run-time error if it is, and otherwise stores the current char value from V in the `string` part object of the object referred by r_i. Finally in the third case, each assignment obtains the current entry of R, which is a pattern P_i, creates a new instance O_i of P_i, and stores the current `char` value from V in the `char` part object of O_i.

There is also a variant of this bytecode which is parameterized with two evaluations, i.e., an upper and a lower bound expression, and with this variant the target is not the entire repetition R but only the repetition slice from and including the lower bound and to and including the upper bound. For this, the lower bound must be greater than zero and less-equal than the number of entries in R, but the upper bound must just be greater than zero (a "too large" number means "up to and including the last entry"). If any of these checks fails then a run-time error is raised. The bounded variant of this bytecode is used in assignments that include repetition slices, like for instance `'***'->R[2:3]`; if `R` is a repetition of 4 `char` objects with values `'1234'` before the assignment then it will contain 5 char objects with the values `'1***4'` after the assignment.

`POP-string-value` *`where`* **:** Lookup a `string` part object o by traversing the run-time path *where*. Pop a `string` value S and change the state of o to S.

`POP-string1, S1-->C, PUSH-char` **:** Pop a string value V which is statically known to have length exactly one, extract the single `char` value it contains, and push that `char` value. See `PUSHI` .

`POP-string1, S1-->I, PUSH-integer` **:** Pop a string value V which is statically known to have length exactly one, extract the single `char` value it contains, coerce^c it to the corresponding `integer` value, and push it. See `PUSHI` .

`POP-string1, S1-->R, PUSH-real` **:** Pop a string value V which is statically known to have length exactly one, extract the single `char` value it contains, coerce^{cr} it to the corresponding `real` value, and push it. See `PUSHI` .

`PUSH-boolean` *`where`* **:** Lookup an object O by traversing the run-time path *where*. Find the `boolean` part object o in O and push its value.

`PUSH-boolean-value` *`where`* **:** Lookup a `boolean` part object o by traversing the run-time path *where*. Push the `boolean` value which is its state.

`PUSH-char` *`where`* **:** Lookup an object O by traversing the run-time path *where*. Find the `char` part object o in O and push its value.

`PUSH-char-value` *`where`* **:** Lookup a `char` part object o by traversing the run-time path *where*. Push the `char` value which is its state.

`PUSH-index` *`where`* **:** Lookup a `for` statement substance F by traversing the run-time path *where*. Obtain the `integer` value of the index variable of F and push it.

`PUSH-integer` ***where*** **:** Lookup an object O by traversing the run-time path ***where***. Find the `integer` part object o in O and push its value.

`PUSH-integer-value` ***where*** **:** Lookup an `integer` part object o by traversing the run-time path ***where***. Push the integer value which is its state.

`PUSH-inx-obj` ***where eval evalWhere*** **:** Lookup a repetition R of objects by traversing the run-time path ***where***. Lookup a dynamic context C for the evaluation of the Evaluation ***eval*** by traversing the run-time path ***evalWhere***. Evaluate^j ***eval*** in context of C and pop the integer value N that this produces. Check that N is a valid repetition index (greater than zero, less-equal than the number of elements in R). Push a reference to the object stored in R at entry N.

`PUSH-inx-objref` ***where eval evalWhere*** **:** Lookup a repetition R of variable objects by traversing the run-time path ***where***. Lookup a dynamic context for the evaluation of the Evaluation ***eval*** by traversing the run-time path ***evalWhere***. Evaluate^j ***eval*** and pop the integer value N that this evaluation produces. Check that N is a valid repetition index (greater than zero, less-equal than the number of elements in R). Push the object reference stored in R at entry N.

`PUSH-inx-ptnref` ***where eval evalWhere*** **:** Lookup a repetition R of variable patterns by traversing the run-time path ***where***. Lookup a dynamic context C for the evaluation of the evaluation ***eval*** by traversing the run-time path ***evalWhere***. Evaluate^j ***eval*** in context of C and pop the integer value N that this produces. Check that N is a valid repetition index (greater than zero, less-equal than the number of elements in R). Push the pattern (or `NONE`) stored in R at entry N.

`PUSH-inx-qual` ***where eval evalWhere*** **:** Lookup a repetition R of variable objects by traversing the run-time path ***where***. Push the pattern which is the qualification of R (a repetition of variable objects has one shared qualification for all its entries).

`PUSH-inx-tmpobj` ***where eval evalWhere*** **:** Lookup a repetition R of objects by traversing the run-time path ***where***. Lookup a dynamic context for the evaluation of the Evaluation ***eval*** by traversing the run-time path ***evalWhere***. Evaluate^j ***eval*** and pop the integer value N that this produces. Check that N is a valid repetition index (greater than zero, less-equal than the number of elements in R). Push a reference to the object stored in R at entry N, on the stack of temporary objects.

`PUSH-inx-tmpobjref` ***where eval evalWhere*** **:** Lookup a repetition R of variable objects by traversing the run-time path ***where***. Lookup a dynamic context for the evaluation of the Evaluation ***eval*** by traversing the run-time path ***evalWhere***. Evaluate^j ***eval*** and pop the integer value N that this evaluation produces. Check that N is a valid repetition index (greater than zero, less-equal than the number of elements in R). Check that the object reference r stored in R at entry N is not `NONE`. If the test succeeds then push this object reference on the stack of temporary objects, otherwise raise a run-time error.

`PUSH-obj` *where* **:** Lookup an object O by traversing the run-time path *where*. Push an object reference which refers to O.

`PUSH-objref` *where* **:** Lookup an object reference r by traversing the run-time path *where*. Push this object reference.

`PUSH-ptn` *where* **:** Lookup a pattern by traversing the run-time path *where*. Push it.

`PUSH-ptn "object"` **:** Push the pattern `object`, i.e., the empty list of mixins.

`PUSH-ptnref` *where* **:** Lookup a variable pattern p by traversing the run-time path *where*. Push p (it may be `NONE` or a pattern).

`PUSH-qual` *where* **:** Lookup a variable object attribute A by traversing the run-time path *where*. Push the pattern which is the qualification of A.

`PUSH-real` *where* **:** Lookup an object O by traversing the run-time path *where*. Find the `real` part object o in O and push its value.

`PUSH-real-value` *where* **:** Lookup a `real` part object o by traversing the run-time path *where*. Push the real value which is its state.

`PUSH-str-len` *where* **:** Lookup a `string` part object o by traversing the run-time path *where*. Push the integer value which is the length of the value of o.

`PUSH-string <-- [char]` *rdecl where* **:** Lookup a repetition R of entities with pattern `char` or a subpattern of `char` by traversing the run-time path *where*. Push the string value which consists of the char values obtained from the entries in R, one by one. There are three variants of this bytecode for the cases where R contains objects, references to objects, or patterns. The declaration of the repetition, *rdecl*, is only stored for documentation and debugging purposes. There is also a variant which is parameterized with a lower and an upper bound which is used when R is specified by a repetition slice. For example, if `myString` is a `string` object, and `R` is a is a repetition of 4 `char` objects with values `'1234'` at some point, then `myString` will have the value `'23'` after an execution of the assignment `R[2:3]->myString`. For more details about the variants of this bytecode, see `POP-string --> [char]` .

`PUSH-string` *where* **:** Lookup an object O by traversing the run-time path *where*. Find the `string` part object o in O and push its value.

`PUSH-string-value` ***`where`*** **:** Lookup a `string` part object o by traversing the run-time path ***`where`***. Push the string value which is its state.

`PUSH-tmpobj` ***`where`*** **:** Lookup an object O by traversing the run-time path ***`where`***. Push an object reference which refers to O on the stack of temporary objects.

`PUSH-tmpobjref` ***`where`*** **:** Lookup an object reference r by traversing the run-time path ***`where`***. Check that r is not `NONE`. If the test succeeds then push this object reference on the stack of temporary objects, otherwise raise a run-time error.

`PUSHI NONE(obj)` **:** Push a `NONE`-valued reference to an object.

`PUSHI NONE(ptn)` **:** Push `NONE` as a pattern (meaning there is no pattern here).

`PUSHI` ***`boolean-literal`*** **:** Push the given literal boolean value.

`PUSHI` ***`char-literal`*** **:** Push the given literal char value.

`PUSHI` ***`integer-literal`*** **:** Push the given literal integer value.

`PUSHI` ***`real-literal`*** **:** Push the given literal real value.

`PUSHI` ***`string-literal`*** **:** Push the given literal string value. Note that literal string values with length exactly one are statically recognized, and this information is used to allow coercion from a string value to a char value if and only if that string value is known to have length one. That means that there is no need to have a special syntax for char literal values.

`RELOP( < )` ***`type`*** **:** Pop two values of type ***`type`***, compare them, and push true if the second is less than the first, otherwise push false.

`RELOP( <= )` ***`type`*** **:** Pop two values of type ***`type`***, compare them, and push true if the second is less than or equal to the first, otherwise push false.

`RELOP( <> )` ***`type`*** **:** Pop two values of type ***`type`***, compare them, and push true if they are different, otherwise push false.

`RELOP( = )` ***`type`*** **:** Pop two values of type ***`type`***, compare them, and push true if they are equal, otherwise push false.

`RELOP( > )` ***`type`*** **:** Pop two values of type ***`type`***, compare them, and push true if the second is greater than the first, otherwise push false.

`RELOP( >= )` ***`type`*** **:** Pop two values of type ***`type`***, compare them, and push true if the second is greater than or equal to the first, otherwise push false.

`RESIZE-rep` ***`rdecl where`*** **:** Lookup a repetition by traversing the run-time path ***`where`***. Pop an integer value N and delete or create new entries such that the repetition has exactly N entries after the operation. The declaration of the repetition, ***`rdecl`***, is only stored for documentation and debugging purposes.

`SEM-Count` *`where`* **:** Lookup a `semaphore` part object o by traversing the run-time path *`where`*. Push the integer value which is the number of threads that are blocked in a `P` operation on o, i.e., the number of threads that are "waiting for access to the resource which is guarded by o".

`SEM-P` *`where`* **:** Lookup a `semaphore` part object o by traversing the run-time path *`where`*. Execute the `P` operation on o. This may cause the current thread to be blocked (stopped) by the scheduler, because it maintains the invariant that the number of `P` operations on o is at all times less-equal than the number of `V` operations.

`SEM-TryP` *`where`* **:** Lookup a `semaphore` part object o by traversing the run-time path *`where`*. Execute the `tryP` operation on o. This operation has the same effect as the `P` operation followed by pushing the boolean value true would have had, if the `P` operation *not* would have blocked. If the `P` operation would have blocked then `tryP` just pushes the boolean value `false`. In other words, it tries to do a `P` operation, and then tells whether it succeeded.

`SEM-V` *`where`* **:** Lookup a `semaphore` part object o by traversing the run-time path *`where`*. Execute the `P` operation on o. This may cause a thread to be unblocked (resumed) by the scheduler, because it maintains the invariant that the number of `P` operations on o is at all times less-equal than the number of `V` operations.

`SUSPEND` *`where`* **:** Suspend the current `component`.

`simpleFor` **:** Create^j a `for` substance entity. Evaluate the Evaluation of the SimpleForImp imperative which was provided as a parameter to the initialization of this bytecode at compile-time. Obtain the result of the evaluation by popping an integer value N. Execute the Imperatives of the SimpleForImp N times.

`simpleIf` **:** Evaluate^j the Evaluation of the SimpleIfImp imperative which was provided as a parameter to the initialization of this bytecode at compile-time. Obtain the result of the evaluation by popping a boolean value V. If V is true then execute the Imperatives in the `then`-part of the SimpleIfImp, else execute the ElsePart.

`stdio/in` **:** Read a line of text from the standard input of the gbeta run-time, then push it as a string value. Pragmatic facility which is useful but not part of the gbeta language design.

`stdio/out` **:** Pop a string value and print it on standard output of the gbeta run-time. Pragmatic facility which is useful but not part of the gbeta language design.

`when` **:** Create^j a `when` substance entity (to hold the immutable reference to the object on which we are typecasing). Lookup the AttributeDenotation of the WhenImp imperative which was provided as a parameter to the initialization of this bytecode at compile-time. This step yields an object O (or fails), and it is done by using information such as run-time paths which was added to the AttributeDenotation during static analysis. Since the AttributeDenotation lookup may include the computation of a repetition entry index it may execute arbitrary code, and hence it may cause any kind of run-time error. This may be seen as evidence of the fact that the `when` imperative is not yet being compiled down to as primitive constructs as most other constructs. That will be corrected as soon as possible.

For each WhenAlternative, in the order they appear in the source code, obtain the pattern P of the AttributeDenotation of the WhenAlternative, and test the pattern of O, P_o, against P. If the WhenAlternative has an exact marker (a '`=`' just after the '`//`') then we test whether P_o and P are equal, otherwise we test whether P_o is less-equal than P.

When the first test succeeds execute the Imperatives of the current WhenAlternative and then terminate the execution of the WhenImp without considering the remaining cases or the ElsePart. Otherwise, if no test succeeds, execute the ElsePart.

`while` **:** Evaluate[j] the Evaluation of the WhileImp imperative which was provided as a parameter to the initialization of this bytecode at compile-time. Obtain the result of the evaluation by popping a boolean value V. If V is true then execute the Imperatives and repeat the execution of the WhileImp, otherwise terminate it.

Notes to the bytecode descriptions:

c The coercion used in the implementation is the built-in value coercion which is provided by the Mjolner BETA compiler.

j The control structure imperatives are compiled into one bytecode for each imperative (e.g. `(if ... if)`) as a whole. This bytecode is initialized with a reference to the abstract syntax tree for its imperative during code generation, and its execution is defined at a higher level than with normal bytecodes. The execution consists of evaluation of Evaluations and execution of Imperatives. With this design of the bytecode instruction set there is no need to handle labels (addresses) and jumps in the generation and execution of bytecode, which was nice because the implementation thereby became simpler and less prone to subtle errors in address calculations and the like. When a specific entry of a repetition is accessed, the computation of the index (as in `R[a]` where `a` may have a `do`-part) also invokes arbitrary code.

The fact that some bytecodes cause arbitrary computations and not just a fixed operation with near-constant time complexity may seem to defeat the goal of making it possible to inspect the time/space complexity of executing gbeta programs by looking at the generated code. However, it does not affect the complexity of execution: The high-level operations occur either in those places where there would have been jumps and labels in the bytecode if it had been compiled down to a level that did not contain high-level operations, or it occurs when an Evaluation is being evaluated, in which case the byte code of that Evaluation can be inspected separately.

In fact it can be argued that there is even better support for complexity inspection with the current design, because every list of bytecodes that the gbeta compiler prints will be (a part of) a basic block.

p To peek is to read the value at the top of the stack. Apart from the better performance and the atomicity of the operation it is equivalent to a pop operation yielding a value V, immediately followed by a push operation which pushes the same value V.

q Note that this qualification check is unnecessary in all cases except where a compile-time warning about a possible run-time error was issued. The test is performed every time, even though it is redundant in all the places where no warnings were issued, because this will detect faults in the soundness of the static analysis. I.e., if there is ever a run-time qualification error in a place where no warnings were issued, there is a bug in the static analysis. No bugs related to the soundness of the static analysis have been observed since early 1998.

r Coercion between char and real is considered an error, except that the Mjolner compiler supports this coercion implicitly as part of a real division operation. This is explicitly marked by Mjolner as a feature that may be removed in a future release, so it is not recommended to use it. If you need to coerce between char and real values then use an intermediate `integer` object.

Appendix E

Static Analysis Functions

```
(* SYNTAX *)
datatype NameDecl   = NAMEDECL of string;
datatype NameAppl   = NAMEAPPL of string;
datatype Prefix     = "object" | NameAppl;
datatype Declarator = ":" | ":<" | "::<";
datatype Descriptor = Prefix*Block
and      ObjectSpec = Descriptor | NameAppl | "object"
and      AttrDecl   = (NameDecl*Declarator*ObjectSpec)
and      Block      = AttrDecl list;

(* TYPING ENTITIES *)
datatype Step     = OUT of int | UP of Block;
type     Path     = Step list;
type     Mixin    = Path * Block;
type     Type     = Mixin list;
type     Context  = Path * Type;
type     Universe = Context list ref;
val      Ω = ..: Path; (* non-existent path *)

(* SYNTAX NAVIGATION *)

(* deliver the nearest enclosing block of ‘s’ *)
fun syn_enclosingBlock (s:NameAppl|..) = ..: Block;

(* deliver the descriptor which contains ‘b’ *)
fun syn_enclosingDescriptor (b:Block) = ..: Descriptor;

(* deliver the right hand side of the ‘nd’ decl. *)
fun syn_declaredTo (nd:NameDecl) = ..: ObjectSpec;

(* deliver the attribute declarations of ‘B’ *)
fun syn_attrsOf (B:Block) = ..: AttrDecl list;
```

```
(* deliver 'nd' such that 'D=syn_declaredTo(nd)' *)
fun syn_declOf (D:Descriptor) = ..: NameDecl;

(* true iff 'nd' declares a pattern, (δ=":") *)
fun syn_isPattern (nd:NameDecl) = ..: bool;

(* true iff 'nd' declares a virtual, (δ=":<") *)
fun syn_isVirtual (nd:NameDecl) = ..: bool;

(* deliver the string value of 'id' *)
fun syn_string (id:NameAppl|NameDecl) = ..: string;

(* AUXILIARY FUNCTIONS *)

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

fun findPos pred xs =
  let fun find [] n = ⊥
        | find (x::xs) n = if pred x then n else find xs (n+1)
  in  find xs 0
  end;

fun sameIdentifier name a_d = toLower(name)=toLower(syn_string(a_d));

(* TYPING FUNCTIONS *)

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

fun getFocus (T,B) =
  let fun getf [] n = ⊥
        | getf ((ρ',B')::T') n = if (B'=B) then n else getf T' (n+1)
  in  getf T 0
  end;

fun blockLookup((ρ',B),ρ,name) =
  let fun search [] = ⊥
        | search ((b_d,δ,O)::Attrs) =
          if sameIdentifier name b_d then b_d else search Attrs
      val Result = search (syn_attrsOf B)
  in  if Result=⊥ then ⊥ else (Result,ρ@[UP(B)])
  end;

fun localLookup(U,(ρ,T),focus,a) =
  let val Result = blockLookup (nth(T,focus),ρ,syn_string(a))
  in  if Result<>⊥ then Result
      else if (focus+1)>=length(T) then ⊥
      else localLookup(U,(ρ,T),focus+1,a)
  end;

fun getVirtualDecl(a_d) =
  let val B = syn_enclosingBlock(a_d)
      val U = <<the universe of B>>
      val (T,focus) = getContext(U,B,[])
  in getVirtualDecl1(T,focus,a_d)
  end

and getVirtualDecl1(T,focus,a_d) =
  let val S = nth(T,focus)
      val Result = blockLookup(S,[],syn_string(a_d))
  in  if Result<>⊥ then
        (* S does declare something of name a_d *)
        let val (b_d,ρ) = Result
        in  if syn_isPattern(b_d) then getVirtualDecl1(T,focus+1,a_d)
            else if syn_isVirtual(b_d) then b_d
            else if focus+1>=length(T) then ⊥
            else getVirtualDecl1(T,focus+1,a_d)
        end
      else if focus+1>=length(T) then ⊥
      else getVirtualDecl1(T,focus+1,a_d)
  end;

fun gatherVirtualChain(U,C,a_d) =
  let val (ρ,T) = C
      (* find the canonical (":<") declaration *)
      val v_d = getVirtualDecl(a_d)
      fun isMine b_d = (getVirtualDecl(b_d)=v_d)
      (* create list of contributing declarations *)
      fun gather [] = []
        | gather (S'::T') =
          let val Result = blockLookup (S',[],syn_string(a_d))
              val Head = if Result=⊥ then [] else
                         let val (b_d,ρ) = Result
                         in  if isMine b_d then [b_d] else []
                         end
              val Tail = gather T'
          in  Head@Tail
          end
  in  gather T
  end;

fun typeOf(U,Typable,C,focus) =
  case Typable of
       NameAppl(a) => typeOfNameAppl(U,a,C,focus)
```

```
     | NameDecl(a_d) => typeOfNameDecl(U,a_d,C,focus)
     | Descriptor(D) => typeOfDescriptor(U,D,C,focus)

and rawTypeOf(U,O,C,focus) =
  case O of
       Descriptor(D) =>
         let val (P,B) = D
             val (ρ,T) = C
             val B' = syn_enclosingBlock(B)
             val T' = (* type of superpattern *)
                 case P of
                      NameAppl(a) => typeOfNameAppl(U,a,C,focus)
                    | Object => []
         in  (ρ@[UP(B')],B)::T'
         end
     | NameAppl(a) => typeOfNameAppl(U,a,C,focus)

and typeOfNameAppl(U,a,C,focus) =
  let val (a_d,ρ) = declOf(a)
      val (C',focus') = staticWalk(U,ρ,C,focus)
  in  typeOfNameDecl(U,a_d,C',focus')
  end

and typeOfDescriptor(U,D,C,focus) =
  let val a_d = syn_declOf(D)
  in  if syn_isPattern(a_d) then rawTypeOf(U,D,C,focus)
      else typeOfNameDecl(U,a_d,C,focus)
  end

and typeOfNameDecl(U,a_d,C,focus) =
  if syn_isPattern(a_d) then
  let val O = syn_declaredTo(a_d)
  in  case O of
          Descriptor(D) => typeOfDescriptor(U,D,C,focus)
        | NameAppl(a) => typeOfNameAppl(U,a,C,focus)
        | Object => []
  end
  else (* virtual pattern *)
  let val vchain = gatherVirtualChain(U,C,a_d)
      fun doMerge [] = []
        | doMerge (b_d::Rest) =
          let val B = syn_enclosingBlock(b_d)
              val O = syn_declaredTo(b_d)
              val (ρ,T) = C
              val bfocus = getFocus(T,B)
              val T = rawTypeOf(U,O,C,bfocus)
              val Base = doMerge Rest
          in  merge T Base
          end
  in  doMerge vchain
  end

and getContext(U,B,ρ) =
  if ∃ (ρ',T) ∈ U. ρ' ≅ ρ then (ρ',T)
  else if syn_enclosingBlock(B)=⊥ then
     (* B is the outermost block *)
     let val T  = [(Ω,B)];
         val C  = (ρ,T);
         val focus = 0; (* B alone, must be focus *)
     in  U := U ∪ {C}; (C,0)
     end
  else
  let val B' = syn_enclosingBlock(B);
      val D  = syn_enclosingDescriptor(B);
      val (C',focus') = getContext(U,B',ρ@[OUT(1)]);
      val T  = typeOfDescriptor(U,D,C',focus');
      val C  = (ρ,T);
      val focus = findPos (fn (ρ,B) => B=B') T;
  in  U := U ∪ {C}; (C,focus)
  end

and enclosingContext(U,(ρ,T),focus) =
  let val (ρ',B) = nth(T,focus)
      val B' = syn_enclosingBlock(B)
  in  if ∃ (ρ'',T') ∈ U. ρ'' ≅ ρ'
      then ((ρ'',T'),getFocus(T',B'))
      else getContext(U,B',ρ')
  end

and lookup(U,C,focus,a) =
  let val Result = localLookup(U,C,focus,a)
  in  if Result<>⊥ then Result else
      let val (C',focus') = enclosingContext(U,C,focus)
      in  if C'=⊥ then ⊥ else lookup(U,C',focus',a)
      end
  end

and declOf(a) =
  let val B = syn_enclosingBlock(a)
      val U = <<the universe of B>>
      val (C,focus) = getContext(U,B,[])
  in  lookup(U,C,focus,a)
  end

and staticWalk(U,ρ,C,focus) =
   let fun walkOut n (C',focus') =
         if n=0 then (C',focus') else
         let val (C'',focus'') = enclosingContext(U,C',focus')
         in  walkOut (n-1) (C'',focus'')
         end
```

```
    fun walkUp B' (C',focus') =
      let val (ρ',T') = C'
          val (ρ'',B'') = nth(T',focus')
      in  if (B''=B') then (C',focus')
          else if focus'+1>=length(T') then ⊥
          else walkUp B' (C',focus'+1)
      end
    fun walk [] (C',focus') = (C',focus')
      | walk (σ'::ρ') (C',focus') =
        let fun walkOneStep (UP(B')) = walkUp B'
              | walkOneStep (OUT(n)) = walkOut n
        in  walk ρ' (walkOneStep σ' (C',focus'))
        end
in  walk ρ (C,focus)
end;
```

Bibliography

[1] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, New York, 1996.

[2] 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.

[3] 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.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, USA, 1986.

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

[6] Ken Arnold and James Gosling. *The Java^{TM} Programming Language.* The Java$^{\text{TM}}$ Series. Addison-Wesley, Reading, MA, USA, 1998.

[7] 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 (OOPSLA '96)*, volume 31, 10 of *ACM SIGPLAN Notices*, pages 69–82, New York, October6–10 1996. ACM Press.

[8] Jan Bosch and Stuart Mitchell, editors. *Object-Oriented Technology: ECOOP'97 Workshop Reader.* LNCS 1357. Springer-Verlag, Heidelberg, Germany, 1997.

[9] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* Ph.D. thesis, Dept. of Computer Science, University of Utah, March 1992.

[10] 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.

[11] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices volume 33 number 10, pages 183–200, Vancouver, BC, October 1998.

[12] 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.

[13] K. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. *Lecture Notes in Computer Science*, 1445:523–??, 1998.

[14] K. B. Bruce, L. Petersen, and A. Fiech. Subtyping is not a good "match" for object-oriented languages. *Lecture Notes in Computer Science*, 1241:104–??, 1997.

[15] K. B. Bruce, R. Van Gent, and A. Schuett. PolyTOIL: A type-safe polymorphic object-oriented language. *Lecture Notes in Computer Science*, 952:27–??, 1995.

[16] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[17] Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *Fourth International Conference on Functional Programming and Computer Architecture*. ACM, September 1989. Also technical report STL-89-5, from Software Technology Laboratory, Hewlett-Packard Laboratories.

[18] L. Cardelli. Structural subtyping and the notion of power type. In ACM, editor, *POPL '88. Proceedings of the conference on Principles of programming languages, January 13–15, 1988, San Diego, CA*, pages 70–79, New York, NY, USA, 1988. ACM Press.

[19] 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.

[20] Craig Chambers. Predicate classes. In O. Nierstrasz, editor, *Proceedings ECOOP'93*, LNCS 707, pages 268–296, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[21] Craig Chambers. *The Cecil Language, Specification and Rationale.* Dept. of Comp.Sci. and Eng., Univ. of Washington, Seattle, Washington, 1997.

[22] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. *SIGPLAN Notices*, 25(6), June 1990.

[23] L.A. Clarke, J.C. Wilden, and A.L. Wolf. Nesting in ada programs is for the birds. *ACM SIGPLAN Notices*, 15(6), November 1980. 139–145.

[24] P. Coad and E. Yourdon. *Object-Oriented Analysis, Second Edition.* Yourdon Press/Prentice Hall, 1991.

[25] W. Codenie, K. de Hondt, T. D'Hondt, and P. Steyaert. Agora: message passing as a foundation for exploring OO language concepts. *ACM SIGPLAN Notices*, 29(12):48–57, December 1994.

[26] William Cook. A Proposal for Making Eiffel Type-safe. In S. Cook, editor, *Proceedings of the ECOOP'89 European Conference on Object-oriented Programming*, pages 57–70, Nottingham, July 1989. Cambridge University Press.

[27] Serge Demeyer and Jan Bosch. *Object-Oriented Technology: ECOOP'98 Workshop Reader.* LNCS 1543. Springer-Verlag, Heidelberg, Germany, December 1998.

[28] E. W. Dijkstra. GOTO statements considered harmful. *Comm. ACM*, 11:147–148, 1968.

[29] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *Proceedings OOPSLA'92*, volume 27, no 10, pages 16–24, Vancouver, USA, October 1992. ACM SIGPLAN Notices.

[30] 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.

[31] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, USA, 1990.

[32] Erik Ernst. Constraint based inheritance. In Ernst et al. [42], pages 25–34.

[33] Erik Ernst. *Constraint Based Inheritance.* In Bosch and Mitchell [8], 1997. (Short section in [41]).

[34] Erik Ernst. Dynamic inheritance and static analysis can be reconciled. In Mughal and Opdahl [86].

[35] Erik Ernst. *Programming Language Development.* In *LNCS 1543* [27], December 1998. (Short section in [41]).

[36] Erik Ernst. Dynamic inheritance in a statically typed language. *Nordic Journal of Computing*, 6(1):72–92, Spring 1999.

[37] Erik Ernst. Propagating class and method combination. In *Proceedings ECOOP'99*, 1999. (to appear).

[38] Erik Ernst. Relative types. In Ernst et al. [39], pages 21–36.

[39] Erik Ernst, Frank Gerhardt, and Luigi Benedicenti, editors. *Position Papers from the 8th Workshop for PhD Students in Object-Oriented Systems*, DAIMI-PB 535. Department of Computer Science, University of Århus, Denmark, 1999.

[40] Erik Ernst, Frank Gerhardt, and editors Luigi Benedicenti. *The 8th Workshop for PhD Students in Object-Oriented Systems*, chapter 1, pages 1–43. In *LNCS 1543* [27], December 1998.

[41] Erik Ernst, Lutz Wohlrab, and editors Frank Gerhardt. *The 7th Workshop for PhD Students in Object-Oriented Systems*, chapter 13. In Bosch and Mitchell [8], 1997.

[42] Erik Ernst, Lutz Wohlrab, and Frank Gerhardt, editors. *Proceedings of the 7th Workshop for PhD Students in Object-Oriented Systems*, DAIMI PB-526, Århus, Denmark, 1997. Dept. of Computer Science, University of Århus.

[43] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings ECOOP'98*, LNCS 1445, pages 186–211, Brussels, Belgium, July 1998. Springer-Verlag.

[44] Charles N. Fischer and Richard J. LeBlanc, Jr. *Crafting a Compiler with C*. Benjamin/Cummings Publishing Company, Inc., 1991.

[45] 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.

[46] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[47] Jacques Garrigue. Programming with polymorphic variants. In *Proceedings, The 1998 ACM SIGPLAN Workshop on ML*. ACM, `http://wwwfun.kurims.kyoto-u.ac.jp/~garrigue/papers/index.html`, 1998.

[48] Jacques Garrigue and Jun P. Furuse. The objective label trilogy. `http://\-wwwfun.kurims.\-kyoto-u.ac.jp/\-soft/\-olabl/`, June 1999.

[49] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972. A summary appeared in the Proceedings of the Second Scandinavian Logic Symposium (J.E. Fenstad, editor), North-Holland, 1971 (pp. 63–92).

[50] Adele Goldberg and David Robson. *Smalltalk–80: The Language.* Addison-Wesley, Reading, MA, USA, 1989.

[51] D.R. Hanson. Is block structure necessary? *Software – Practice and Experience*, 11(8):853–866, August 1981.

[52] William Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 411–428, October 1993. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, number 10.

[53] G. Hedin and B. Magnusson. Supporting exploratory programming in simula. In *Proceedings of the 15th Simula Conference*, pages 73–88, September 1987.

[54] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. In *European Conference on Object-Oriented Programming (ECOOP). Also in informal proceedings of the Sixth International Workshop on Foundations of Object-Oriented Languages (FOOL)*, 1999.

[55] Michael I. Schwartzbach Jens Palsberg. *Object-Oriented Type Systems.* John Wiley & Sons, New York City, 1994.

[56] Sonya E. Keene. *Object-Oriented Programming in Common Lisp.* Addison-Wesley, Reading, MA, USA, 1989.

[57] 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, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer.

[58] J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, and B. Magnusson. *Object-Oriented Environments – The Mjølner Approach.* Prentice Hall, Hertfordshire, GB, 1993.

[59] Jørgen Lindskov Knudsen. Name collision in multiple classification hierarchies. In S. Gjessing and K. Nygaard, editors, *Proceedings ECOOP '88*, LNCS 322, pages 93–109, Oslo, August 15-17 1988. Springer-Verlag.

[60] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Classification of actions, or inheritance also for methods. In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP'87*, LNCS 276, pages 98–107, Paris, France, June 15-17 1987. Springer-Verlag.

[61] Bent Bruun Kristensen and Daniel C. M. May. Activities: Abstractions for collective behavior. In P. Cointe, editor, *Proceedings ECOOP'96*, LNCS 1098, pages 472–501, Linz, Austria, July 1996. Springer-Verlag.

[62] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification, Second Edition.* Addison-Wesley, Reading, MA, USA, 1999.

[63] Luigi Liquori. An extended theory of primitive objects: First order system. In *Proceedings ECOOP'97*, LNCS 1241, pages 147–169, Jyväskylä, June 1997. Springer-Verlag.

[64] Vassily Litvinov. Constraint-based polymorphism in cecil: Towards a practical and static type system. In Craig Chambers, editor, *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98), Special Issue of SIGPLAN Notices*, volume 33, 10, Vancouver, October 1998. ACM Press.

[65] Vassily Litvinov and Craig Chambers. Constraint-based polymorphism in Cecil. Technical report, Department of Computer Science and Engineering, University of Washington, 1998. Technical Report UW-CSE-98-01-01.

[66] Edward N. Lorenz. *The essence of chaos.* University of Washington Press, 1993.

[67] 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.

[68] 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).

[69] B. J. MacLennan. Values and objects in programming languages. *ACM SIGPLAN Notices*, 17(12):70–79, December 1982.

[70] Bruce J. MacLennan. *Principles of Programming Languages.* Oxford University Press, New York, N.Y., third edition, 1999.

[71] O. L. Madsen. Block structure and object oriented languages. *ACM SIGPLAN Notices*, 21(10):133–133, October 1986.

[72] Ole Lehrmann Madsen. Block structure and object oriented languages. In Bruce Shriver and Peter Wegner, editors, *Workshop on Object-Oriented Programming*, Cambridge, MA, 1987. MIT Press.

[73] Ole Lehrmann Madsen, Boris Magnusson, and Birger Moller-Pedersen. Strong typing of object-oriented languages revisited. *ACM SIGPLAN Notices*, 25(10):140–150, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[74] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, MA, USA, 1993.

[75] Jawahar Malhortra. *Tailorable Systems: Design, Support, Techniques, and Applications*. PhD thesis, Department of Computer Science, University of Århus, Århus, Denmark, 1994.

[76] Benoit B. Mandelbrot. *The fractal geometry of nature*. W. H. Freeman, New York, 1983.

[77] Robert Cecil Martin. *Designing Object-Oriented C++ Applications Using the Booch Method*. Prentice Hall, Englewood Cliffs, N.J., 1995. ISBN 0132038374.

[78] Albert R. Meyer and Mark B. Reinhold. 'Type' is not a type. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 287–295, St. Petersburg Beach, Florida, January 13–15, 1986. ACM SIGACT-SIGPLAN, ACM Press.

[79] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.

[80] Mira Mezini. Dynamic object evolution without name collisions. In *Proceedings ECOOP'97*, LNCS 1241, pages 190–219, Jyväskylä, June 1997. Springer-Verlag.

[81] Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. *ACM SIGPLAN Notices*, 33(10):97–116, October 1998.

[82] J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 37–51. ACM, ACM, January 1985.

[83] Mjolner Informatics, Århus, Denmark. *MIA 90-11: Sif - Mjolner BETA Source Browser and Editor*, 1991.

[84] Mjolner Informatics, Århus, Denmark. *MIA 91-14: The Metaprogramming System*, 1991.

[85] Mjolner Informatics, Århus, Denmark. *MIA 91-20: Persistence in BETA*, 1991.

[86] Khalid Mughal and Andreas L. Opdahl, editors. *Proceedings of NW-PER'98: The Eighth Nordic Workshop on Programming Environment Research*, Ronneby/Sweden, August 1998. Department of Informatics and Department of Information Science, University of Bergen, Norway.

[87] Stephan Murer, Stephen Omohundro, and Clemens Szyperski. Sather Iters: Object-oriented iteration abstraction. Technical Report TR-93-045, International Computer Science Institute, Berkeley, August 1993.

[88] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, MA, USA, 1996.

[89] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 15–17 January 1997.

[90] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems, Wiley & Sons.*, 2(3), 1996.

[91] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[92] Benjamin Pierce. *Type Systems*. Not yet published, but used in courses at Univ. of Pennsylvania, 1999. See `http://www.cis.upenn.edu/~bcpierce/`.

[93] R. Prieto-Diaz and J.M. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4):307–334, November 1986.

[94] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 40–53, Paris, France, 15–17 January 1997.

[95] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Proc. Colloque sur la Programmation*, LNCS 19. Springer-Verlag, 1974.

[96] J. C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. North-Holland, Amsterdam, 1983.

[97] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass., 1997.

[98] David N. Smith. Smalltalk faq. `http://\-www.dnsmith.com/\-SmallFAQ/\-PDFfiles/\-DavesSmalltalkFAQ.pdf`, June 1999.

[99] 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 of the 7^{th} European Conference on Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 197–219. Springer-Verlag, 1993.

[100] 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.

[101] David Stoutamire and Stephen Omohundro. The sather 1.1 specification. Technical Report TR-96-012, International Computer Science Institute, Berkeley, 1996.

[102] David Stoutamire and Steven Omohundro. *Sather 1.1*. International Computer Science Institute, Berkeley, California, 1996.

[103] C. Strachey. Fundamental concepts in programming languages. Lecture notes for International Summer School in Computer Programming, Copenhagen, August 1967.

[104] Bjarne Stroustrup. *The C++ Programming Language (Second Edition)*. Addison-Wesley, Reading, MA, USA, 1991.

[105] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, Mass., 1994.

[106] Clemens A. Szyperski. Import is not inheritance - why we need both: Modules and classes. In *Proceedings of the Sixth European Conference on OOP (ECOOP'92)*, number LNCS 615 in Lecture Notes in Computer Science. Springer Verlag, June 1992.

[107] Antero Taivalsaari. Object-oriented programming with modes. *Journal of Object-Oriented Programming*, 6(3):25–32, June 1993.

[108] Kristine Stougård Thomsen. *Multiple Inheritance, a Structuring Mechanism for Data, Processes and Procedures*. Datalogisk afdeling, AArhus Universitet, Århus, Denmark, 1986. DAIMI PB-209.

[109] Kresten Krab Thorup. Optimizing method dispatch using sparse arrays. In *Proceedings of FreeSoft'93, Moscow, Russia*, March 1993.

[110] Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings ECOOP'97*, LNCS 1241, pages 444–471, Jyväskylä, June 1997. Springer-Verlag.

[111] Kresten Krab Thorup. Objective c. In Saba Zamir, editor, *Handbook of Object Technology*, chapter 18. CRC Press, 1999.

[112] Mads Torgersen. Unifying abstraction. Progress Report, in partial fulfillment of the requirements for the Ph.D. degree, 1998.

[113] A. Tripathi, E. Berge, and M. Aksit. An implementation of the object-oriented concurrent programming language SINA. *Software Practice and Experience*, 19(3):235–256, March 1989.

[114] 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.

[115] Guido van Rossum. Why are python strings immutable? Section 6.4 of Python FAQ, jun 1999.

[116] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Yourdon Press, New York, 2 edition, 1978.

Index