Reflection in a Statically Typed and Object Oriented Language

A Meta-Level Interface for BETA

Søren Brandt
Computer Science Department, Aarhus University
Ny Munkegade, DK-8000 Aarhus C, Denmark
Email: sbrandt@daimi.aau.dk

July 14, 1995

reflection: 6: a thought, idea, or opinion formed or a remark made as a result of meditation.

Webster's dictionary

Abstract

This technical report is concerned with issues of type safe and type orthogonal computational reflection in a statically typed, object oriented, and block structured programming language. The main new contributions is the design of a comprehensive meta-level interface for a statically typed language, the introduction of more flexible typing constraints for BETA than currently possible, and the description of a novel language construct, the *attribute reference*. The BETA type system is made more flexible through a generalisation of the qualification concept, adding considerably to the expressive power of BETA, but *without* adding new syntax. Existing BETA already includes types as first-class values, through the concept of dynamic pattern references. With the advent of attribute references, attributes are turned into first-class language citizens as well, allowing type-safe computed attribute access. On top of the generalised qualification concept and the concept of attribute references, a comprehensive meta-level interface to the BETA language is described, along with a number of example uses. Finally, a meta-level architecture enabling the practical implementation of the MLI for a statically typed language is outlined.

Contents

List of Figures

Chapter 1

Reflection in a Statically Typed Language

A core element of the meaning of reflection is the term "self-reference". As might therefore be expected, the following definition of a reflective system from the preface of [Maes & Nardi 88], clearly identifies a reflective system as a self-referential system:

> A computational system is said to be reflective when it is itself part of its own domain. More precisely, this implies that (i) the system has an internal representation of itself, and (ii) the system sometimes shifts its "normal" computation about the external domain to "reflective" computation about itself.

The concept of computational reflection was originally introduced in [Smith 84], along with the 3-Lisp reflective programming language. Since then, most work on computational reflection has been carried out in the context of *dynamically*[1] typed languages such as, e.g., CLOS [Kiczales et al. 91], ABCL [Masuhara et al. 92] and Smalltalk [Goldberg & Robson 83]. This report considers how to turn BETA, a *statically*[2] typed OO programming language, into a reflective programming language.

In a statically typed programming language, the compiler generates code assuming that certain kinds of runtime type-errors cannot occur. Procedure calls and attribute access can therefore be compiled into very efficient code. These assumptions are not valid in dynamically typed languages, and each procedure call therefore gives rise to an implicit runtime type check.

To retain efficiency and safety of a statically typed language, a meta-level language interface cannot be allowed to invalidate the static type-correctness assumptions. However, meta-level code is typically of a very generic and thus dynamic nature, and is therefore often impossible to type-check statically[3]. Compromising static type-safety is therefore inevitable, but compromises should not invalidate the type-correctness assumptions of base-level, i.e., non-meta-level, code.

The meta-level interface described in this report is not statically checkable in general. However, it allows meta-level code to ensure that it does not break static typing assumptions, and enforces these assumptions through runtime type-checks in case of erroneous meta-level code. Furthermore, it allows the programmer

[1]For now, dynamically typed means that most type checks are done at runtime, whereas statically typed means that most checks are done at compile time.

[2]A few runtime type checks on assignments are needed in addition to compile-time checking.

[3]Based on a closed-world assumption, static type-checking of meta-level code may be possible in principle. However, a closed-world assumption does not apply to runtime-extensible systems.

to express static typing constraints even when doing meta-level programming, thereby possibly leading to more readable and efficient meta-level code.

The additions made to the programming language proper are more flexible typing constraints than currently allowed, and the concept of *attribute references*, allowing BETA programs to explicitly speak about attributes of objects and patterns. Using these language extensions, a *meta-level interface* (MLI) for the BETA language is described. Although focus in this report is on reflection and on using the added expressive power for specifying a MLI for BETA itself, the additions are very useful for programming in general, and thus adds considerably to the expressive power of the BETA language.

To the best of my knowledge, BETA is the first compiled and statically typed language for which a comprehensive meta-level interface has been designed.

1.1 Reflection and Meta-Level Architectures — Terminology

Along the lines of [Kiczales et al. 91], this report distinguishes *introspectional* reflection and *intercessory* reflection. *Introspection*[4] is the ability of a running program to look into and analyze its own current state. *Intercession*[5] is the ability to change the behaviour of certain primitives of the reflective system.

The term *reification*[6] denotes the task of converting an abstract concept into a concrete thing. For example, all imperative programming languages include the concept of procedure call, but in most cases it is impossible to manipulate a specific instance of the procedure call as data, e.g., passing references to procedure calls around or to change the general behaviour of procedure calls. By reifying the concept of procedure call, it becomes a concrete entity that may be manipulated by a reflectional program.

Relevant terminology also includes the terms *meta-level architecture* and *meta-level interface*. A *meta-level architecture* is a means for implementing a reflective system. Thus, the meta-level architecture provides the internal representation of the system needed for the system to be reflectional. The meta-level interface is an interface to the meta-level architecture.

A well-known example of a meta-level interface is the CLOS MOP [Kiczales et al. 91], providing an interface to the (meta-) objects in the Common Lisp implementation of the CLOS language.

1.2 Why Reflection?

Although most work on reflectional systems has been carried out in the context of dynamically typed programming languages, the problems attacked are of a more general nature, and thus exist in many different software systems. Kiczales and Lamping [Kiczales & Lamping 93] introduces the term *mapping dilemma* to describe one general class of problems that can be solved using reflectional techniques and meta-level interfaces.

So what is a mapping dilemma? When implementing substrate systems[7] such as programming languages, virtual memory systems, file systems, and windowing systems, one is often forced to choose between different possible implementations. The trouble is that there is often no single "right choice", as the best choice may depend on the application being built on top of the substrate system. Hence, the term "mapping dilemma" comes from the problem of choosing between different mappings from software abstraction to

[4]*introspect*: to look inside, to examine (one's own mind or its contents) reflectively : to engage in an examination of one's thought process and sensory experience. *Webster's dictionary*

[5]*intercession*: the act of interceding. *Webster's dictionary*

[6]*reify*: to convert into or regard as a concrete thing. *Webster's dictionary*

[7]Software abstractions used as building blocks for higher level systems.

software realization. The concept of mapping dilemma now leads to a new way of thinking about the design and implementation of substrate systems. Whenever a mapping dilemma crops up, we should still choose a feasible default solution, but we should also open up the implementation, allowing the user to change the default if it is not feasible for his particular purpose [Kiczales 94]. The result of "opening up" will be some sort of metaobject protocol, allowing the user to change and inspect the inner workings of the substrate system.

Exposing mapping dilemmas are just one specific way of making systems flexible and thus more useful. More generally we need systems that are open in order to allow tailoring. Introspectional and intercessory capabilities are exactly such openings that allow extensive tailoring to take place. However, as always there is a tradeoff between flexibility and complexity.

Some examples of non-language substrate systems with reflectional capabilities are the following:

- Virtual memory systems. Here mapping dilemmas include how to map virtual addresses to pages, and how to map pages to physical memory. What is the best policy depends on the application at hand. A classic example system providing some control over these aspects is Mach [Accetta et al. 86], through the concept of external pagers.
- In the CORBA [OMG 91] distributed object specification, the concept of dynamic invocation interfaces allows client programs to dynamically synthesize requests to servers based on server descriptions obtained from an interface repository. Hence, this is an example of an introspectional capability.
- The Emacs [Stallman 81] editor is an example of a meta-level architecture that has been around for quite a while. The success of Emacs is mainly due to its extreme customizability, allowing the user to change and replace parts of its implementation. Seen as a reflectional system, Emacs includes introspectional as well as intercessory capabilities.

In order to design open substrate systems it must be possible to explicitly denote the entities that are the subject matter of the substrate in question. The original motivations for this work stems from the design and implementation of substrates such as persistence, distribution, and interpretation libraries, where the subject matter is in fact primitives of the BETA language itself, i.e., objects, patterns and attributes. Thus, to be able to create flexible interfaces to these substrate systems, BETA needs reflectional capabilities.

1.3 Summary

I have briefly introduced the concepts of computational reflection, meta-level architectures and meta-level interfaces. Reflectional capabilities were subdivided into introspectional and intercessory capabilities. I argued that in general we should strive to create flexible and open systems, and that reflection and metaobject protocols are one way to do this.

1.4 Report structure

Chapter 2 describes the immediate background and problems leading to the work described in this report. Chapter 3 then describes selected aspects of the BETA programming language. A complete description of BETA may be found in [Madsen et al. 93b]. Chapter 4 describes the generalization of the BETA qualification concept, and gives a few examples of the added expressive power. Attribute references are described in chapter 5. On top of attribute references and generalized qualifications, the BETA meta-level interface, MLI, is described in chapter 6, with example applications in chapter 7. Chapter 8 outlines a meta-level architecture enabling efficient implementation of the BETA MLI. Finally, chapter 9 describes related and future work, and chapter 10 presents my conclusions.

Chapter 2

Background

This work has its offspring in systems programming in the Mjølner BETA system, including the implementation of a source level debugger, an object browser, a type-orthogonal persistent store and a distributed objects system [Brandt 94, Brandt & Madsen 94]. In addition, I have been involved in discussions relating to the design of a distributed object database, a distributed hypermedia [Grønbæk et al. 94] and an interpreter for the BETA language [Malhotra 94].

Work within these areas lead to the recognition of the applicability of a comprehensible meta-level interface for the BETA language. In the following sections, I describe a number of substrate systems whose implementation would be considerably eased by the provision of a meta-level interface.

2.1 Flexible Copying of Complex Object Structures

A number of substrate systems need the ability to manipulate generic complex object structures. Examples include persistence support, distributed object support, object based transaction support and generic object copying mechanisms. For such systems to be flexible, they need ways to talk about their subject matter, in this case including complex object structures.

For example, when copying complex objects, notions such as deep and shallow copying have become standard, since they are easy to describe and implement and furthermore seems to be reasonable defaults [Meyer 92]. However, a more generally useful copying mechanism should be tailorable, and thus allow the user to express exactly what parts of an object graph he wants copied. To describe a tailorable object copying mechanism, the programming language must support the expression of restrictions on the object graph under consideration. It must be able to explicitly talk about objects, object types, and references between objects. That is, the language must be able to denote the basic primitives of the programming language itself. This is exactly the purpose of reflectional language capabilities.

A number of special purpose systems have proposed special purpose solutions to these problems. For example, the POET [Robie & Witte 91] object oriented database system extends the C++ programming language with the `transient` keyword. Object attributes with storage type `transient` are not followed when saving persistent data to disk. A similar example is the `attach` keyword used in the Emerald [Black et al. 87] distributed and object based programming language. In Emerald, attached references are followed when moving objects between network hosts. That is, attach is effectively the opposite of the `transient` keyword, since the Emerald default is **not** to follow references.

Unfortunately, both of these solutions affect type orthogonality in the sense that only classes written explicitly with persistence or distribution in mind may benefit from the keywords `transient` and `attach`. Furthermore, they are special purpose constructs which cannot be used for anything but their specific intended usage. A better solution would be a general language construct allowing as special cases the expression of attachment as well as transience of object references. The concept of *attribute reference* described in this report is exactly such a general language construct.

In the implementation of the BETA persistent store and distribution library [Brandt 94], the problem of limiting object graph traversals in a type orthogonal way lead to the development of a tailorable graph traversal algorithm. This object graph traversal algorithm is available as the *ObjectSerializer* library, supporting flexible marshalling of object graphs. For each object reference ref met during object graph traversal from a set of root objects, the programmer may choose to follow or not to follow ref. The choice can be based on information regarding the identities, types and state of the objects connected by ref, as well as knowledge of the reference "kind"[1]. These informations, though sufficient for many purposes, do not allow an exact simulation of, e.g., the `transient` and `attach` keywords from POET and Emerald. The problem is that only the objects connected by a reference can be explicitly denoted, not the reference itself. This problem is the direct background leading to the concept of attribute references to be described in detail in this report.

Despite its limitations, the *ObjectSerializer* is an extremely important component in the implementation of the BETA persistent store, oodb and distributed object system. Thus, the idea of creating a tailorable marshalling engine has already paid of. Notice that the *ObjectSerializer* is an example of a substrate system with a meta-level interface. The user of the ObjectSerializer library may control several aspects of the algorithm by further binding virtuals and supplying different parameters to the basic algorithms supplied by the library. For example, the following aspects of the object marshalling process may be controlled by the user of the ObjectSerializer:

- The mapping between external and internal object identifications.
- The part of the object graph being marshalled.
- The serialization method used. By overriding the standard (general) serialization method for specific classes, the programmer may exploit type specific knowledge in order to optimize the serialization process.
- Location where results of the serialization process are saved and later fetched for unserialization. This of course includes "saving" to and "fetching" from a network connection.

It follows that, even if the subject matter of a substrate system is primitives of the BETA language itself, BETA may already be used to implement substrates with a MLI. However, the ObjectSerializer MLI is limited by the ability of BETA to talk about its own primitives, and the implementation of the ObjectSerializer relies on low level knowledge of the BETA runtime system, and thus depends on details of a specific BETA implementation. Even worse, the ObjectSerializer is not type safe. This might be acceptable if the ObjectSerializer had been a non-extensible black-box. However, type-safety of the ObjectSerializer depends on the correctness of the information supplied by its users. Furthermore, there is currently no way to enforce type-safety. Both of these problems are handled by the availability of a general BETA MLI, as the one described in this report.

Concerning substrate systems whose subject matter is not BETA primitives, BETA is of course as good as any OO language when it comes to the design and implementation of a MLI for those substrates.

[1] BETA objects contain three different kinds of references to other objects, namely static references, dynamic references, and origin references.

2.2 Distributed Object System

The implementation of a type independent object marshalling engine, (the ObjectSerializer), relies heavily on a number of prerequisites:

1. Detailed inside knowledge of the implementation of the Mjølner BETA system.
2. The presence of an ad-hoc interface providing limited introspectional capabilities.
3. Minor persistence- and ObjectSerializer specific additions to the introspectional ad-hoc interface.

The current implementation of distributed objects is *not* type independent, which is partly due to missing intercessory capabilities. To be able to implement a type independent distributed objects system, intercessory capabilities are needed in addition to the ad-hoc introspectional capabilities already present in the system. For example, message send must clearly be implemented differently in the local and remote cases. Furthermore, since BETA allows one object to directly access the substance attributes of other objects, attribute access will also need different implementations in the remote and local cases.

In Smalltalk [Goldberg & Robson 83], a dynamically typed language, the intercessory capabilities needed for a distributed objects system can be implemented by overriding *message not understood* [McCullough 87, Decouchant 86] in special *proxy* objects. A proxy object is a local representative of a remote object, the *principal*. Distributed BETA [Brandt 94], like several other distributed object systems, is based on the the notion of proxies. Unlike Smalltalk, however, BETA is statically typed, and therefore prevents *message not understood* at compile time. Thus, using *message not understood* is not an option in a statically typed language. The intercessory capabilities of the BETA MLI described in this report will greatly simplify the implementation of a type independent distributed objects system in BETA.

2.3 Program Debugging and Tracing

Another example where a MLI would be useful, is program debugging and tracing. A source-level debugger needs to access the state of a running program in order to represent program state to the programmer in high-level terms. In systems without a MLI, this is only possible through a detailed knowledge of the underlying implementation and, assuming a compiled language, usually only in a machine dependent way. This has two major implications:

1. At least the lower layers of the debugger needs to be rewritten for each new platform supported.
2. Only programmers with access to detailed implementation information are able to write a debugger. Such implementation information may be proprietary, meaning that only the company implementing the compiler and runtime system is actually able to write a debugger for the system. By supplying a feasible and non-proprietary MLI, anyone interested could implement a debugger.

Both of these problems could also be attacked by supplying a non-proprietary debugger interface that was fixed across different platforms and implementations. However, doing so actually amounts to designing a MLI specifically for debugging. Large parts of such a debugging MLI would have functionality in common with a general language MLI. By designing a general language MLI we therefore achieve that it can be used for purposes other than debugger implementation. Furthermore, the debugger specific MLI would become much smaller.

Program tracing in order to do, e.g., profiling could be supported in a reflective system by using the intercessory capabilities to redefine all methods to execute tracing code before entering the actual method body.

2.4 Interpretation

The BETA interpreter [Malhotra 94] is an embeddable interpreter allowing compiled BETA programs to be extended with interpreted code. Thus, the interpreter needs to access a mix of interpreted and non-interpreted objects and call methods in these objects. Again portability is hurt by the lack of a platform independent MLI.

Currently, the BETA interpreter runs only on a single platform, mainly due to problems of portability.

2.5 Summary

This section listed a number of substrate systems whose implementation would benefit from a general language MLI. The examples given were flexible object copying, distributed object systems, program debugging and tracing, and embedded interpretation. More examples and details will be described in section 7.

In general terms, problems solved by a MLI include missing expressiveness of the language when designing and implementing substrate systems needing to talk about the primitives of the language itself, and problems of portability of meta-level tools. A language MLI solves these problems by allowing a program to explicitly denote language primitives, and by providing portable access to the inner workings of a running system.

Chapter 3

The BETA Language

3.1 Composition of BETA Objects

A reflective system has the expressive power to describe parts of the system itself. An important part of any object oriented programming language is the set of primitives used to describe the state of single objects and the composition of complex objects.

In BETA, the most important abstraction mechanism is the *pattern* concept. The pattern is a generalization of concepts such as class, procedure, virtual procedure, and exception. For a complete description of the BETA language, see [Madsen et al. 93b].

A pattern `P` is declared as follows:

```
P: Super
   (# Decl1; Decl2; ... Decln
   enter In
   do Imp1; Imp2; ... Impm
   exit Out
   #)
```

where `Super` defines an optional super-pattern for `P`; `Decl1`, `Decl2`, `...`, `Decln` is a list of declarations of reference attributes or patterns; `In` defines optional input parameters of `P`-objects; `Imp1`, `Imp2`, `..`, `Impm` is a sequence of imperatives (statements) associated with `P`-objects; and `Out` defines optional output parameters of `P`-objects.

A simple BETA object is an instance of one of the basic patterns Integer, Char, Boolean, Real, etc. A complex object is composed of a number of references to other objects.

To be able to describe a BETA MLI, we should understand in detail the concepts to be reified by the MLI. One of these concepts is object composition, and BETA support for object composition is described in this section.

3.1.1 Reference Types

In general, references between BETA objects may be classified as to whether they are dynamic references, static references, or origin references.

Dynamic references, from now on referred to as *DynRefs*, may refer different objects at different points in time, or may have the special value `NONE`, referring no object. In this respect they are equivalent to C++ *pointers* and Smalltalk *instance variables*. In BETA, DynRefs are declared using the operator `^`, declaring a one-way relationship from the object containing the reference, to the object referred.

Static references, from now on referred to as *StatRefs*, is the way to describe whole/part hierarchies in BETA. In contrast to DynRefs, a statically referred object conceptually *belongs* to the object containing the reference. The statically referred object is created together with the referring object. Furthermore, a StatRef cannot change, i.e., it always refers the same object. In BETA, StatRefs are declared using the operator `@`, defining a two-way relationship between the objects. Given the whole, the part may as usual be found by following the reference, using dot notation. Reversely, the concept of object *location* [Madsen & Møller-Pedersen 92] gives a way to find the whole given a part. Thus, if the object `aa` is part of the object `a`, then **location**(`aa`) returns a reference to a. If `aa` is not a part of another object, then **location**(`aa`) returns `NONE`.

Origin references, from now on referred to as *OriRefs*, define the static environment of an object. BETA is a block structured language, and OriRefs capture the static block-structure of the BETA source at runtime. OriRefs are only declared implicitly by the BETA source. That is, an object being an instance of a nested BETA pattern `P.PP`, has an OriRef to an instance of the enclosing pattern `P`[1]. Notice that the concepts of location and origin are separate. The location of an object o is the object o' containing a StatRef to o. The origin of an object o is the object o' containing the *pattern* of which o is an instance.

The combination of inheritance and general block structure may lead to patterns whose instances have more than one origin, as exemplified by the `aContainer.Scan (# ... #)` pattern in figure 3.1: The pattern `Container.Scan` is defined inside the `Container` pattern. Thus, instances of `Container.Scan` have an OriRef to an instance of `Container`. Likewise, since the object denoted by `&aContainer.Scan (# ... #)` is defined inside the `main` object, it has an OriRef to `main`. Since `&aContainer.Scan (# ... #)` is, through inheritance, also an instance of `Container.Scan`, it has two OriRefs.

Like DynRefs, an OriRef defines a one-way relationship from the object to its origin. That is, an object knows its origin, but not vice versa. However, like StatRefs, an OriRef always refers the same object, and thus cannot be changed once the object has been created.

An example of the different reference types is shown in figure 3.1. The embedding of objects within objects symbolizes the concepts of StatRef and object location. The object `main` has a StatRef to the object `aContainer`, and therefore the object `aContainer` is shown embedded inside `main`. Dashed arrows symbolize OriRefs, and solid arrows symbolize DynRefs.

3.1.2 Summary

I have described the BETA primitives used to compose complex objects. Simple BETA objects are instances of the basic patterns Integer, Char, Boolean, etc.. Complex objects are composed of three different kinds of references to other objects. *DynRefs* are dynamic references allowed to point to different objects at different points in time. *StatRefs* are static references to part objects. The object containing the StatRef is denoted the *location* of the part object. The part object is created together with its location, and afterwards the StatRef is not allowed to change. Finally, *OriRefs* capture the block structure of the BETA source at runtime. An instance of a nested pattern has an OriRef to an instance of the statically enclosing pattern. The combination of block structure and inheritance may lead to objects with more than one OriRef.

The concepts of StatRef and DynRef described in this section will be generalized in later sections. However, apart from a statically referred object not always being created as part of the object from which it is statically referred, the description in this section remains valid.

[1]The *static link* between activation records of languages, such as Pascal, where procedures may be declared local to other procedures is a special case of the BETA origin reference.

```
main: @
  (# Container:
       (# element:< Object;
          Scan:
            (# current: ^element;
            do (for <<each element>> repeat
                  thisElement[]->current[];
                  INNER;
               for);
            #);
          ...
       #);
     aContainer: @Container;
  do &aContainer.Scan
     (#
     do current[]->...
     #);
  #);
```

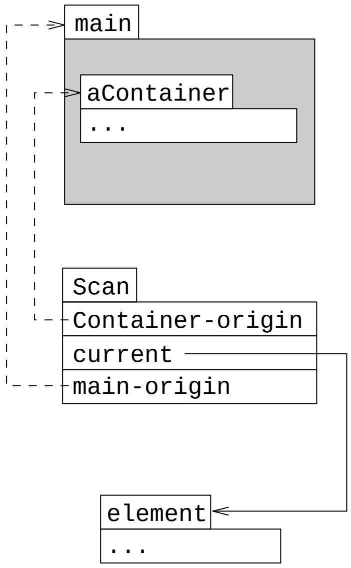


Figure 3.1: Object relationships.

In addition to the three different kinds of *object* references described above, a BETA object may contain static as well as dynamic *pattern* references. These will be described in section 3.3, after the description of the *qualification* concept in section 3.2.

3.2 Qualifications

The previous section described BETA support for the modelling of complex objects. It was noted that dynamic references, DynRefs, are allowed to refer different objects at different points in time. However, most DynRefs are not allowed to refer arbitrary objects. They are subject to *qualifications*.

A qualification limits the set of objects a DynRef is allowed to refer. Only objects that *qualify* to the qualification of the reference can be referred. At the same time, the qualification of an object reference tells the compiler what operations are applicable to the object referred. This is also true with regard to StatRefs, but here the qualification is furthermore used to create the part object referred by the StatRef.

In addition to qualified object references, BETA allows qualified virtual patterns and qualified dynamic pattern references. This section, however, focusses on qualified object references.

Typing depends heavily on BETA being one of very few statically typed and object-oriented languages that allow general nesting of classes. Simula is the only other language with these properties, although class nesting in Simula is more restricted than in BETA.

This section describes the BETA type system. More on this subject can be found in [Madsen et al. 93a].

3.2.1 Static Typing

A program in an object-oriented programming language can be said to be correctly typed if the program never accesses non-existing attributes of an object.

A language is said to be *strongly* typed if correct typing is enforced. If correct typing can be enforced at compile time, the language is said to be statically typed. Otherwise correct typing may be enforced by some combination of compile time and runtime checking. In a dynamically typed language, all type-checking is done at runtime.

BETA is strongly typed with most type checking done at compile time. However, assignments that can be statically neither accepted nor rejected are subject to a runtime type check [Madsen et al. 93a]. Due to the type system of BETA (like, e.g., Eiffel [Meyer 92]) being covariant[2], these runtime type checks cannot be completely removed in the general case.

Compile time type checking in BETA is based on explicit type constraints (qualifications) on attribute declarations. For example, consider the declaration of a dynamic object reference `c`:

```
c: ^Circle;
```

Here `c` is declared as a dynamic reference allowed to refer only instances of subpatterns of `Circle`, or instances of `Circle` itself. However, using `c`, only methods defined for the `Circle` pattern can be invoked on the object referred by `c`. Even though the object may have several other methods, they cannot be called using `c`, since they are not statically known.

An unconstrained declaration can be made using `Object` as qualification:

```
o: ^Object;
```

Note that the only operations allowed on object references qualified by `Object`, are those defined for all object types. Currently this includes only reference assignment and testing the actual type of the object referred by `o`.

Concerning reference assignment, the object reference `o` may for example be assigned to the object reference `c`, by writing `o[]->c[]`. The compiler is unable to statically neither accept nor reject this assignment, since it cannot statically deduce whether the assignment is type-correct. To handle this problem, the compiler insert a runtime type check[3].

To talk about qualifications, I shall use the functions **object**(), which returns the object that a reference actually denotes, and **qual**(), which returns the statically known qualification of an identifier, or the actual pattern of an object [Madsen et al. 93a]. The declaration

```
R: ^T
```

implies that **qual**(`R`) = `T`, whereas **qual**(**object**(`R`)) is the actual pattern of the object referred by `R`.

The notation $q \subseteq q'$ denotes that the qualification `q` is more restrictive than q' in the sense that all objects qualifying to q also qualify to q'. The set of objects qualifying to q is denoted **extension**(q). Thus, we shall define the $\subseteq$ relation on qualifications by

$$q \subseteq q' \quad \stackrel{def}{\Leftrightarrow} \quad \mathbf{extension}(q) \subseteq \mathbf{extension}(q')$$

[2]Usually, static typing of hierarchical type systems is ensured by enforcing contravariant relationships between super/sub-types in a type-hierarchy [Black et al. 87, Omohundro 93].

[3]Work on constraint based type inference [Oxhøj et al. 92] of BETA programs is being done, possibly leading to the ability to get rid of most runtime type checks, although probably at the expense of separate compilation.

```
   IntVal:
     (# value: @Integer; (* Integer part object. *)
        add: (* Pattern attribute *)
          (# amount: @Integer;
           enter amount
           do value+amount -> value
          #);
     #);
   i1: @IntVal; (* Static instance of IntVal. *)
   i2: @IntVal; (* Static instance of IntVal. *)
do 4 -> &i1.add; (* Add 4 to i1.value *)
   5 -> &i2.add; (* Add 5 to i2.value *)
```

Figure 3.2: Nested pattern example

```
  a1: ^i1.add; (* Dynamic reference to i1.add instance *)
  a2: ^i2.add; (* Dynamic reference to i2.add instance *)
do (* Assign reference to new i1.add instance to a1: *)
  &i1.add[] -> a1[]; (* OK *)
  (* Assign reference to new i2.add instance to a2: *)
  &i2.add[] -> a2[]; (* OK *)
  (* Assign reference to new i1.add instance to a2: *)
  &i1.add[] -> a2[]; (* WRONG *)
```

Figure 3.3: Illegal reference assignment

Given these definitions, the qualification constraint on the allowed values of a dynamic object reference R, can now be expressed as:

$$\mathbf{qual}(\mathbf{object}(\mathtt{R})) \subseteq \mathbf{qual}(\mathtt{R}) \tag{3.1}$$

3.2.2 Block Structure

BETA is a block structured language allowing general pattern nesting. This means that an object, in addition to other attributes, is allowed to have *pattern attributes*. As is the case with other object attributes, "the same" pattern attribute in different objects should of course be considered different, as illustrated by the example in figure 3.2.

The code fragment in figure 3.2 declares a pattern `IntVal`, and two instances of `IntVal`, `i1` and `i2`. Clearly the `value` attributes of `i1` and `i2` are different since they occupy different storage locations. Likewise, since instances of the pattern `add` nested inside `IntVal` are able to access the `value` attribute of the surrounding object, the patterns `i1.add` and `i2.add` are different. This means that a variable qualified by `i1.add` is not allowed to be assigned an instance of `i2.add`, as illustrated in figure 3.3.

Conceptually, a pattern is thus identified by the source code describing the pattern, in combination with the object of which the pattern is an attribute – the *origin* of the pattern. Or said another way: A BETA pattern is a *closure*. As noted in section 3.1.1, a BETA pattern p may have several origins, and an instance of p will have the same set of origin references as p itself. For example, an instance of `i1.add` has an origin reference to the `i1` object, allowing it to access the attributes of `i1`. This is illustrated in figure 3.4. An example of an object having multiple origins was shown in figure 3.1, on page 16. To identify a BETA pattern uniquely,

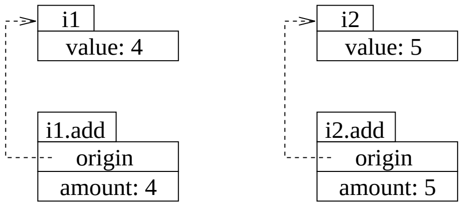


Figure 3.4: Objects after execution of code in figure 3.2

we therefore need two pieces of information:

1. An identification of the *source code* describing the pattern. In the BETA grammar, such a piece of source code is described by the syntactic category *ObjectDescriptor*.
2. The set of *origin objects* for the pattern.

3.2.3 Summary

This section briefly discussed issues of typing in the BETA language. BETA is a strongly typed language enforcing qualification constraints by a combination of compile time and runtime checking.

General blockstructure allows pattern nesting to an arbitrary level, resulting in BETA patterns being closures. A pattern is therefore defined by the source code, (the ObjectDescriptor), as well as the set of origin objects for the pattern.

Nested patterns being closures means that the set of origin objects of a pattern is significant, also in the case where the pattern is used as a qualification.

3.3 Pattern References

The notion of *dynamic pattern references*[4] make patterns first-class values by allowing them to be passed around as values.

Originally, dynamic pattern references were proposed in [Agesen et al. 89] with the primary purpose of enabling extensibility[5] in a statically typed language. Since then, dynamic pattern references have played an important role in the design of a BETA interpreter [Malhotra 94], a callable dynamic linker and loader [Agesen & Frølund 89], the design and implementation of distribution and persistence libraries for BETA [Brandt & Madsen 94, Brandt 94, Sloth & Hem 93] and finally as a general programming facility.

Pattern references can be considered the BETA generalisation of, e.g., function pointers in procedural programming languages such as C and Modula-2. However, in BETA, a procedure is a special case of the general pattern concept, and thus the BETA equivalent to a dynamic procedure reference becomes a dynamic pattern reference. Therefore, in addition to function references, a dynamic pattern reference also plays the role of a dynamic[6] *class* reference. In comparison, ordinary nested patterns as described in section 3.2.2, correspond to *static* pattern references.

[4]In the BETA book [Madsen et al. 93b] as well as in the original proposal [Agesen et al. 89], dynamic pattern references were called *pattern variables*. However, since this terminology tends to confuse people, pattern variables have, in this report, been renamed to *dynamic pattern references*.

[5]The ability to extend a running program with new classes.

[6]*Dynamic* means that it may refer different patterns at different points in time.

A dynamic pattern reference is declared as follows:

```
P: ##T
```

where `P` is the name of the dynamic pattern reference and `T` is its qualification. Thus `P` may be assigned any pattern which is `T` or any sub-pattern of `T`.

The distinction between dynamic and static *pattern* references parallels the distinction between dynamic and static *object* references, as illustrated below:

```
so: @foo(# ... #); (* Static object reference.   *)
do: ^foo;          (* Dynamic object reference.  *)
sp: foo(# ... #);  (* Static pattern reference.  *)
dp: ##foo;         (* Dynamic pattern reference. *)
```

We may think of qualifications (and thus patterns) as predicates inducing sets of objects. In this view, the main difference between an object reference and a pattern reference is that an object reference denotes a single object, while a pattern reference (static or dynamic) denotes an infinite *set* of objects.

Consider the declarations below.

```
1. Circle:
2.  (# draw: (# do ... #);
3.     outline: draw (# ... #);
4.     ...
5.  #);
6. c: @Circle;
7. cd: ##c.draw;
```

In line 1, the `Circle` pattern is declared, containing methods `draw` and `outline` as nested patterns (lines 2 and 3), i.e., static pattern references. `outline` is a sub-method of `draw`. In line 6, a static instance `c` of the `Circle` pattern is declared. Line 7 declares `cd` as a dynamic pattern reference qualified by `c.draw`, i.e., the pattern referred by `cd` can be `c.draw` or some subpattern of `c.draw`. For example, the value assigned to `cd` could be either `c.outline` or `c.draw`. What is important is that the value assigned to `cd` is a pattern whose extension is a subset of the extension of `c.draw`. Or more precisely, **pattern**(`cd`) $\subseteq$ `c.draw`, where the function **pattern**() returns the actual pattern referred by `cd`.

Below a value is assigned to `cd` followed by the creation and execution of an instance of the pattern referred by `cd`.

```
c.outline## -> cd##;
&cd;  (* Create and execute instance of c.outline. *)
```

Another way to assign a value to a pattern reference, is to retrieve the actual pattern of some object `o`, using the expression `o##`, as illustrated below:

```
   c: ^Circle;
   cp: ##Circle;
do &Circle[]->c[];
   c##->cp##;
```

Dynamic pattern references may be compared to test subclass relationships. For example, the following code tests whether the pattern referred by `p` is a subpattern of the pattern referred by `q`:

```
do (if p##<=q## then
      <<p is a subpattern of q>>
   if)
```

Thus, in BETA code, the `<=` operator corresponds to the $\subseteq$ operator introduced in section 3.2.1.

3.4 Summary

This chapter described selected aspects of the BETA object-oriented programming language.

BETA supports object composition through three different object reference types, namely static references, dynamic references, and origin references. In addition to object reference attributes, a BETA object may contain static or dynamic references to patterns. A dynamic pattern reference turns BETA patterns into first-class values, and can be used as a dynamic *class* reference, a dynamic *method* reference, or a dynamic *qualification* reference. Used as a dynamic qualification reference, it can be compared with other pattern references in order to determine subpattern relationships.

BETA is mainly statically typed, and induces static constraints, *qualifications*, on the values of reference attributes. Qualifications are enforced by a combination of compile time checking and runtime checking on potentially dangerous assignments. BETA typing is heavily influenced by support for general pattern nesting. For example, the set of origin objects for at pattern is significant even when the pattern is used as a qualification.

Chapter 4

Generalizing BETA

4.1 A Generalized Qualification Concept

Even though the BETA type system is extremely expressive when compared to most other statically typed OO languages, it is sometimes found to be more restrictive than really necesssary. This section introduces a generalized qualification concept, making BETA qualifications considerable more flexible, but without adding any new syntax.

First, I generalize the concept of BETA qualifications. Next, the semantics of the BETA operators `&` (object creation), `^` (dynamic object reference), and `@` (static object reference) are extended to cope with generalized qualifications. Finally, this section considers a few applications of generalized qualifications, demonstrating the gained expressive power.

With respect to the development of a MLI, two reasons exist for looking at the concept of qualification in detail:

1. The concept of qualification is to be reified by the MLI, and a detailed understanding is therefore required.
2. The MLI itself can make good use of more flexible qualifications.

Besides being too rigid, the currently allowed qualifications are actually sometimes not sufficiently rigid, since some of them cannot be efficiently enforced, neither at compile time nor at runtime. I briefly characterize the qualifications that should be allowed, leading to new possibilities as well as the need to abandon qualifications that are currently allowed. The net result will be an extension of BETA with substantially increased expressive power, but with no new syntax!

4.1.1 An Extensional View on Qualifications

What *is* a BETA qualification? Until this point I have been talking about qualifications as constraints that the programmer can put on attributes of BETA objects to constrain the values of these attributes. The constraints are then enforced by a combination of compile time and runtime type checking. Finally they are used to tell the compiler what attributes are present in a given object.

This section applies an extensional view to qualifications. More precisely, I think of a qualification as a predicate on objects. The set of objects induced by a qualification, the qualification *extension*, includes all

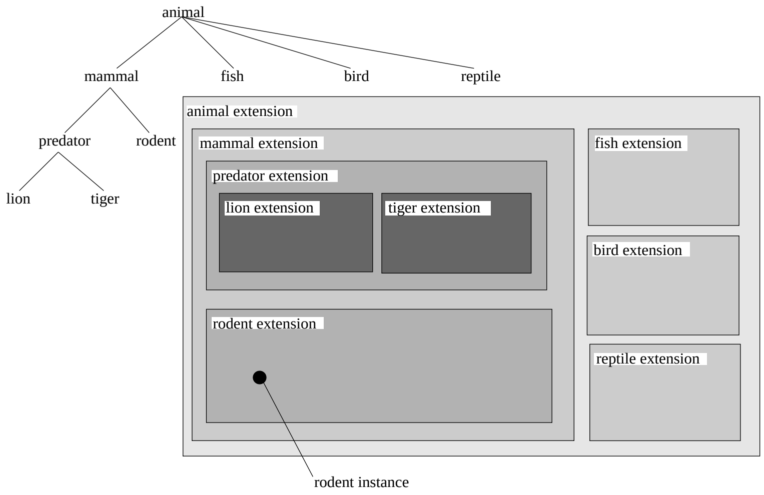


Figure 4.1: Classification hierarchy and corresponding extension sets

objects on which the predicate evaluates to true.

A BETA pattern p can be thought of as a qualification whose extension is the set of all instances of p and subpatterns of p. In most situations, current BETA allows *only* qualifications corresponding to patterns.

But in principle there is no reason why a qualification should always be equivalent to a specific pattern. BETA already allows a single special kind of qualification that does not uniquely correspond to a pattern. As described in the previous section, a BETA pattern p is defined by an ObjectDescriptor and the set of origin objects. By constraining only the ObjectDescriptor and not the origin objects, a qualification that corresponds to a *set* of patterns is obtained. This is already allowed in the case of qualified dynamic references, as shown by the following example:

```
Window:
  (# shape: (# ... #);
     ...
  #);
g: ^Window.shape;
```

In this example, the reference `g` is allowed to refer instances of any pattern `shape` nested inside *some* `Window` object `w`, but without constraining `w`.

All qualifications currently allowed in BETA have infinite extensions. Thinking of patterns as qualifications, the process of specialization corresponds to a stepwise narrowing of the extension of the qualification. This process is illustrated in figure 4.1. But no matter how much a pattern is specialized, the corresponding extension continues being infinite, since any pattern may have an infinite number of instances. Thus BETA, like all other class-based OO languages, only allows the expression of qualifications with infinite extensions. This is so since patterns only restrict the *structure* of the object itself, and the identities of the *origins*, but patterns do not restrict the identity of the object itself.

But in principle there is no reason why extensions of qualifications should always be infinite. In principle, we can imagine extensions of any size, infinite or finite. By explicitly listing a number of object names, any finite extension can be specified. However, I restrict attention to lists of length one, i.e., allowing the use of a single object name in place of a qualification. Notice that an object name in place of a qualification constrains the object id, as opposed to the structural constraints imposed by pattern qualifications.

	$n_1 \ldots n_{m-1}$: all objects	$n_1 \ldots n_{m-1}$: at least one pattern
n_m: object	full object qualification	partial object qualification
n_m: pattern	full pattern qualification	partial pattern qualification

Figure 4.2: Qualification categories for $q = n_1.n_2....n_m$

4.1.2 Taxonomy of Generalized Qualifications

The previous section introduced the idea of generalizing the set of qualifications allowed. This section divides qualifications into four main categories along two axes.

Syntactically, a BETA qualification is a path expression, i.e., a list of names separated by dots. Thus, a qualification q is generally of the syntactic form $n_1.n_2....n_m$. Each of the n_i is a name that may denote either a pattern attribute or an object reference attribute.

If n_m denotes a pattern attribute, q will be called a *pattern qualification*. If n_m denotes an object reference attribute, q will be called an *object qualification*.

If all $n_i,\ i < m$ denotes object reference attributes, q is called a *full qualification*. If some $n_i,\ i < m$ denotes a pattern attribute, q is called a *partial qualification*.

The result is four main categories of qualifications:

- Full pattern qualifications.
- Partial pattern qualifications.
- Full object qualifications.
- Partial object qualifications.

The four categories are depicted in figure 4.2.

Full object qualifications correspond to qualifications with a single element extension, i.e., they denote an object. Full pattern qualifications correspond to patterns. Partial pattern qualifications are different from full pattern qualifications by putting fewer restrictions on the origin set of objects referred. If a partial pattern qualification puts no restrictions on origins (recursively), it corresponds to an ObjectDescriptor. I.e., we may think of ObjectDescriptors as a special case of partial pattern qualifications. Finally, the concept of partial object qualification is best described by an example:

```
Person:
  (# name: @Text;
     ...
  #);
aPersonName: ^Person.name;
```

Here, the `aPersonName` reference is constrained by a partial object qualification to refer the name of *some* person, but without restricting the specific person whose name is referred.

In most cases, current BETA only allows full pattern qualifications, and in a single case (dynamic object reference attributes) also partial pattern qualifications.

The special object `NONE`, which can be interpreted as referring an object with no attributes, qualifies to all qualifications with infinite extension. Thus, `NONE` qualifies to all pattern qualifications and partial object qualifications. Given a full object qualification q, NONE qualifies to q if and only if $\textbf{extension}(q) = \{\texttt{NONE}\}$.

An attempt of a formalization of these concepts is given in appendix E

4.1.3 Generalized Semantics for Object Creation Operators

The BETA operators `&` and `@` are used to create new objects, i.e., new instances of BETA patterns. `&` is used in imperative parts to create new pattern instances. `@` is used in declarative parts to denote the creation of static part objects as part of their *location* (section 3.1). For example:

```
person:
  (# left_arm: @arm;
     right_arm: @arm;
     ...
  #);
p: ^person;
do &person[]->p[];
```

The creation of a new person object is done using the `&` operator. A reference to the new person object is then assigned to the dynamic object reference `p`. Along with the new person object, a number of static part objects, the limbs, are created. The creation of the limbs happens automatically, since they are declared using the `@` operator, which also binds the newly created limbs to the identifiers `left_arm` and `right_arm`.

The `&` operator takes a single qualification parameter q. The `@` operator takes two parameters: a qualification parameter q, and an identifier parameter id.

Current BETA requires that the q parameter given to `&` and `@` denotes a pattern, i.e., q must be a full pattern qualification. Given a qualification q, the semantics of both operators is to create a new instance of the pattern denoted by q. In addition, `@` binds the newly created object to the identifier parameter id.

What happens if the restrictions on the q parameter to `&` and `@` is relaxed? Given a qualification with a finite extension, the "create a new instance" semantics do not apply anymore, since all instances contained in a finite extension are by definition already there, and any new object would therefore not be part of that extension. Instead of object creation, I therefore generalize the semantics of `&` from "object creation" into:

$$\texttt{\&}(q) = \begin{cases} \text{new } q & \text{if } q \text{ is a full pattern qualification} \\ o & \text{if } q \text{ is a full object qualification and } \textbf{extension}(q) = \{o\} \\ \mathit{error} & \text{otherwise} \end{cases}$$

That is, applying `&` to a qualification q always returns an object that qualifies to q. But in case of q being a full object qualification, the returned object is no longer new.

The decision to disallow partial qualifications as parameter to `&` is more or less arbitrary. It could be argued that a better choice would be to create a new object along with the objects necessary to provide the environment of the new object. For example, `&Window.shape[]`, where `Window` and `shape` are both pattern names, could create a new window object `w`, followed by the creation of an instance of `w.shape`. In fact, the meta-level interface described in chapter 6 allows the meta-level programmer to do this, if required.

The "new object" operator, `&`, is evaluated each time the statement of which it is part is executed. The "static object reference" operator, `@`, is evaluated once, namely at creation time of the containing object. The result of evaluating `@`(id, q) is to bind the object returned from `&`(q) to the identifier id. In the terminology of the Kaleidoscope [Lopez et al.] constraint imperative programming language, `@` in the new interpretation is a "once" constraint enforced at object creation time. Afterwards, the binding of *id* is not allowed to change.

These generalized semantics of `@` interferes with the concept of object location as described in section 3.1.1. Since an object can now be statically referred from more than one object, the location of an object o can no longer be defined as the object o' containing a static reference to o. Instead, I therefore define $\mathbf{location}(o)$ as the object o' such that o' has a static refernce to o, and such that o was *created* as part of o'. This definition is no more than a more specific formulation of the old location concept.

The final BETA operator concerned with the qualification of object references is the "dynamic object reference" operator, `^`. Like `@`, `^` takes parameters *id* and q. Unlike `@`, the job of `^` is to continually enforce that the value bound to *id* qualifies to q. The initial value bound to *id* by $\hat{}(id, q)$, is `NONE`, unless `NONE` does not qualify to q. This initial behaviour can be expressed using a two-level constraint hierarchy [Borning et al. 92].

$$\begin{array}{ll} \textit{required} & \mathbf{qual}(\mathbf{object}(id)) \subseteq q \\ \textit{strong} & \mathbf{object}(id) = \texttt{NONE} \end{array}$$

After initialization, only the required constraint stays, and the strong constraint disappers. It is then the job of `^` to monitor changes to q and $\mathbf{object}(id)$ to ensure that $\mathbf{qual}(\mathbf{object}(id)) \subseteq q$ is always enforced. Notice that in the general case, `^` cannot do its job by monitoring assignments to *id* only, since the extension denoted by q could also change dynamically. Thus, in the terminology of Kaleidoscope, `^` is a *two-way always constraint*

The following sections consider the implications of these generalizations of the BETA qualification concept.

4.1.4 Simple Examples

As a first example, consider the following declarations:

```
Window: (# ... #);
w: ^Window;
foo:
  (# dw: ^Window; (* Dynamic window reference. *)
     sw: @Window; (* Window part object.       *)
     sa: @w;      (* Constant reference.       *)
     da: ^w;      (* Alias.                    *)
  #);
```

In this example, the DynRef `dw` and the StatRef `sw` behave in the usual way. That is, when an instance of `foo` is created, `dw` is initialized to NONE, and is afterwards allowed to refer different instances of the `Window` pattern. Also as usual, `sw` is initialized to refer a new instance of the Window pattern, and is afterwards not allowed to change.

Now consider the references `da` and `sa`, both qualified by the full object qualification `w`, which denotes a specific instance of the `Window` pattern. When an instance of `foo` is created, `sa` is initialized to point to the object referred by `w`, and is afterwards not allowed to change. `sa` becomes a *constant reference*. `da` is constrained to refer the object referred by `w`, but unlike `sa`, it must *always* do so. As a result, the semantics of `da` is that of an alias, and in fact can be implemented as such.

The previous example demonstrated how the generalized semantics have given us constant references and aliases for free. None of these could have been expressed in current BETA.

By thinking of full object qualifications as specializations of qualifications with infinite extensions, we achieve that virtual patterns can be specialized into a full object qualification. Consider the following example:

```
Person:
  (# fathertype:< Person;
     father: @fathertype;
  #);
makeSon:
  (# father: ^Person
  enter father[]
  exit &Person(# fathertype:: father #)[]
  #);
```

In this example, a `Person` is defined to have a static reference to his father. `Person` is an abstract pattern since it is impossible to create an instance without at the same time creating an infinite number of person objects due to the `father` attribute. However, by further binding `fathertype` into a full object qualification, it becomes possible to create an instance of person, and at the same time bind the `father` constant reference. These semantics could not have been implemented in ordinary BETA, which would have forced us to describe the father reference by a DynRef. Since people do not tend to change biological parents, a DynRef would not reflect the semantics of child-parent relationships.

But if we need a person object in order to create a person, how do we then create the first person? There are two answers to this chicken-and-egg problem: One is to make a person who is his own father. The second is to make a person with no father. In sharp contrast to real life, both possibilities are pretty easy in generalized BETA:

```
makeAdam1:
  (#
  exit &Person(# fathertype:: THIS(person) #)[]
  #);
makeAdam2:
  (#
  exit &Person(# fathertype:: NONE #)[]
  #);
```

Perhaps not the traditional version of that story, but ok for the purpose of illustration.

4.1.5 Allowed Qualifications

It must now be considered in more detail what kinds of qualifications should be allowed. The answer chosen here is that we should allow all qualifications that can be enforced by a combination of compile time type checking and occasional runtime type checks on potentially dangerous assignments. Generally, an assignment `fromexpr->toexpr` is potentially dangerous if all that is known at compile time is that **qual**(`toexpr`) $\subseteq$ **qual**(`fromexpr`). Consider the following example:

```
animal: (# ... #);
mammal: animal (# ... #);
a: ^animal;
m: ^mammal;
do ...;
   (*1*) a[]->m[];
```

The assignment at `(*1*)` assigns `a`, known to be an animal, to `m`, qualified by `mammal`. Since we only know that the set of mammals is a subset of the set of animals, i.e., $\mathbf{qual}(m) \subseteq \mathbf{qual}(a)$, the assignment is potentially wrong, and a runtime type check is needed.

In general, I demand the following from allowed qualifications:

1. A compact representation, i.e., any allowed qualification must be representable in a fixed amount of space.
2. It should be possible to enforce always constraints by checking only at potentially dangerous assignments to the qualified identifiers.
3. The runtime qualification check must be done in constant time.

A large number of qualifications not previous allowed are in fact possible, but it is also necessary to disallow certain kinds of constraints that were previously allowed. Fortunately, the new possibilities enabled by generalizing the concept of qualification more than outweigh what is lost by disallowing qualifications that cannot be efficiently checked.

As a simple example, consider the following qualification, which is currently allowed, but in fact should be disallowed because it cannot be efficiently enforced.

```
Window:
  (# shape: (# ... #);
  #);
w: ^Window;
s: ^w.shape;
```

The constraint to be enforced on `s` is $\mathbf{qual}(\mathbf{object}(\mathtt{s})) \subseteq$ `w.shape`, which cannot be done by monitoring `s` only, since **extension**(`w.shape`) changes if **object**(`w`) changes. We also need to monitor assignments to `w` in order to enforce the constraint on `s`. In the general case this requires either global information at compile time, or the maintenance of a constraint graph at runtime. None of these options are acceptable, and thus this kind of qualification must be abandoned. The current Mjølner BETA compiler only enforces the constraint $\mathbf{qual}(\mathbf{object}(\mathtt{g})) \subseteq$ `Window.shape`, and thus allows the following scenario without complaining:

```
do &Window[]->w[];
   w.shape[]->g[]; (* OK *)
   &Window[]->w[]; (* UPS. g is no longer correctly qualified *)
```

Another example which cannot be allowed, since it cannot be efficiently enforced, is the following partial object qualification:

```
Vehicle:
  (# owner: ^Person;
  #);
o: ^Vehicle.owner;
```

Unlike the previous example, this example is not allowed by the current BETA compiler. This example has the same property as the previous one, namely that the change of a dynamic reference may invalidate the qualification on `o`. In this case, a vehicle changing owner may result in the breaking of the qualification constraint on `o`. Furthermore, this qualification is impossible to check in constant time even at assignments to `o`.

In general, a qualification q is a path expression $n_1.n_2....n_m$, where each n_i denotes a pattern or object reference attribute. For a qualification q to be allowed as an always (^) qualification, it must either be a full object qualification (corresponding to the declaration of an alias), or all n_i must correspond to *static* object or pattern reference attributes. If all n_i are not static attributes, **extension**(q) is not fixed, and the enforcement of q thus requires monitoring of **extension**(q) in addition to monitoring of the qualified attribute itself.

Concerning once (@) qualifications, the above demand is not necessary, since such qualifications must not be continually enforced.

4.1.6 Summary

This section introduced generalized semantics of BETA qualifications.

Qualifications were described as predicates inducing sets of objects, the *extension* sets. Qualifications where subdivided along two axes: One axis determining whether the qualification is full or partial, and another axis determining whether it is an object or a pattern qualification. A full pattern qualification corresponds to a pattern, a partial pattern qualification relaxes the constraints on origin objects, a full object qualification corresponds to a qualification whose extension contains a single object, and finally a partial object qualification corresponds to an infinite set of part objects.

The "new" and "static" operators, & and @, were described as enforcing "once" constraints, and the "dynamic" operator, ^, was described as enforcing an "always" constraint.

A few uses of the generalized semantics were shown, adding aliases and constant references to the BETA language. The full implications of the generalization is outside the scope of this report, and will be described in a separate paper.

Finally, allowed qualification constraints must have a compact runtime representation, and be enforcable as a combination of compile time checking and simple runtime checks on potentially dangerous assignments. As a consequence, the use of path expressions with dynamic reference parts cannot be allowed as qualifications of dynamic object references ("always" constraints), unless the qualification is actually a full object qualification. In the latter case, the qualification can be implemented as an alias. These demands exclude at least one type of qualification currently allowed by the Mjølner BETA compiler, but also allows a number of new and interesting qualifications.

As a side remark, notice that two types of qualifications have been mentioned, namely those corresponding to structural constraints and those corresponding to identity constraints. The final possibility is constraints on the values of simple objects. Such constraints are the main subject of constraint imperative languages such as Kaleidoscope [Freeman-Benson & Borning 92, Lopez et al.] and Siri [Horn 92], but are outside the scope and aims of this report. Notice however, that full object qualifications are really constraints that fix the value of object references. In object oriented programming languages such as Smalltalk, where simple values are also objects, the framework described here therefore captures value constraints on simple objects as well. In BETA, simple values are not objects, and adding constraints on simple values therefore requires a separate mechanism. For example, an integer is an object that may contain, e.g., the value 5, but the value 5 is not an object itself.

4.2 Generalized Dynamic Pattern References

Dynamic pattern references were first introduced to turn patterns into first-class values. But in section 4.1 it was argued that patterns are just a special case of the more general concept of qualification. The BETA concept of qualification was extended from in most cases allowing only full pattern qualifications, into allowing full and partial qualifications as well as object qualifications.

This section describes the corresponding generalization of dynamic pattern references into dynamic *qualification references*. However, to avoid introducing too much new terminology, I shall stick to the term *pattern reference*.

In current BETA, dynamic pattern reference values are obtained using the `##` operator on object names or pattern names. The expression `o##` returns the actual pattern of the object `o`, i.e., it returns **qual**(`o`). Likewise, `p##` returns a reference to the pattern `p`. In both cases, a reference to a full pattern qualification is obtained.

In order to obtain references to partial qualifications and full object qualifications as well, we shall change and generalize `##` as follows.

> The expression $\mathtt{n}_1.\mathtt{n}_2....\mathtt{n}_m$`##` always evaluates to a reference to the qualification denoted by the path expression $\mathtt{n}_1.\mathtt{n}_2....\mathtt{n}_m$.

In the case of a full object qualification `o`, this actually changes the value of the expression `o##` from returning **qual**(`o`), into returning the full object qualification `o` itself. To obtain the actual pattern of an object `o`, the `GetObjQual` pattern is provided by the MLI, described in section 6.

4.3 Summary

This chapter generalized the BETA notion of qualifications. In addition to full pattern qualifications, BETA was generalized to support partial pattern qualifications, partial object qualifications, and full object qualifications. Object reference operators `^` and `@`, and the object creation operator `&`, were generalized accordingly. It was demonstrated how the generalization resulted in support for aliases and constant references was added to the language. Further benefits will become evident during the description of the concept of attribute references, and the description of the meta-level interface. Notice that the generalization of BETA described in this chapter was achieved without the addition of new syntax.

Finally, the concept of dynamic pattern reference was generalized to support the new generalized notion of qualifications.

Chapter 5

Attribute References

Dynamic pattern references, as described in previous chapters, turn patterns and qualifications into first class language citizens, thus allowing them to be passed as method parameters. Doing so allows the programmer to explicitly talk about qualifications. However, dynamic pattern references do not allow the programmer to explicitly talk about references between objects. Only the objects and patterns themselves can be denoted in current BETA, not their attributes.

Attribute references, whose syntax and semantics will be described in this chapter, allows the programmer to explicitly talk about attributes, pass attributes as method parameters, and apply them to objects. Thus, attribute references *reify* the concept of attributes, and allow computed attribute access. In contrast, current BETA only allows access to attributes known at compile time. Using attribute references it is possible to parameterize a method with the object attributes it is going to access.

Computed attribute access should of course be type-safe, i.e., it should be possible to efficiently type-check the use of attribute references either statically or at runtime. Therefore, attribute reference declarations include qualifications, allowing the compiler and programmer to make assumptions concerning the attribute referred by a particular attribute reference.

5.1 Denoting Attributes

In general, BETA imperatives denote attributes through path expressions of the form $\mathtt{n}_1\mathtt{.n}_2\mathtt{...n}_m\mathtt{.A}$. The semantics of a path expression depends on the trailing characters:

- If the expression is followed by `[]`, i.e., the full expression is $\mathtt{n}_1\mathtt{.n}_2\mathtt{...n}_m\mathtt{.A[]}$, it means "reference to the *object* denoted by $\mathtt{n}_1\mathtt{.n}_2\mathtt{...n}_m\mathtt{.A}$". If $\mathtt{n}_1\mathtt{.n}_2\mathtt{...n}_m\mathtt{.A}$ is a full object qualification, this is equivalent to **object**($\mathtt{n}_1\mathtt{.n}_2\mathtt{...n}_m\mathtt{.A}$). **object**() was introduced in section 3.2.1.
- If the expression is followed by `##`, it means "reference to the *qualification* denoted by $\mathtt{n}_1\mathtt{.n}_2\mathtt{...n}_m\mathtt{.A}$", as described in section 4.2. If $\mathtt{n}_1\mathtt{.n}_2\mathtt{...n}_m\mathtt{.A}$ is a dynamic pattern reference, this is equivalent to **pattern**($\mathtt{n}_1\mathtt{.n}_2\mathtt{...n}_m\mathtt{.A}$). **pattern**() was introduced in section 3.3.
- If the expression is not followed by anything, its meaning depends on whether $\mathtt{n}_1\mathtt{.n}_2\mathtt{...n}_m\mathtt{.A}$ denotes a full pattern qualification or a full object qualification, but in any case includes the actual evaluation (dopart execution) of some object.

To be able to talk about the attributes themselves, this chapter introduces a fourth possibility, which is to use the trailing character `:`. Thus, the expression $\mathtt{n}_1\mathtt{.n}_2\mathtt{...n}_m\mathtt{.A:}$ means "reference to the *attribute* denoted by $\mathtt{n_1.n_2...n_m.A}$".

The concept of attribute references, as described in this chapter, is introduced to allow BETA objects to have attributes whose value is references to other attributes. Thus, like dynamic object references support dynamic references to *objects*, and dynamic pattern references support dynamic references to *qualifications*, so do attribute references support dynamic references to *attributes*.

Given an attribute expression $\mathtt{n_1.n_2...n_m.A:}$, its *location* is the qualification $\mathtt{n_1.n_2...n_m\#\#}$. Notice that the "same attribute" may be denoted through attribute expressions with different locations. Consider the following example:

```
foo: (# t: ^Text #);
f: @foo;
```

Relative to this code, the attribute expressions `foo.t:` and `f.t:` can abstractly be said to denote the same attribute, which is the `t` attribute of the `foo` pattern. However, the location of `foo.t:` is the full pattern qualification `foo##`, whereas the location of `f.t:` is the full object qualification `f##`. Consequently, more is known about `f.t:` than about `foo.t:`.

The **location**() function[1] returns the location of an attribute expression.

5.2 Attribute Reference Syntax

The syntax for the declaration of an attribute reference is:

```
attr_ref: .loc_qua*attr_qua
```

The leading dot means that `attr_ref` is an attribute reference. The dot symbol is chosen since it is also used to access attributes of remote objects, and is therefore associated with the "attribute" meaning already.

The syntax includes three qualifications:

1. `loc_qua`, the *location qualification*, constraining the set of objects on which `attr_ref` can be applied.
2. The *kind qualification*, shown as `*` in the declaration of `attr_ref`, constrains the attribute reference to refer specific kinds of attributes.
3. `attr_qua`, the *attribute qualification*, constrains the qualification of the attribute referred by `attr_ref`. For example, an attribute reference may be qualified to refer only attributes referring text objects.

The full syntax of attribute references is shown in appendix A.

Consider the example in figure 5.1, declaring the patterns `Person` and `Family`, the attribute reference `aFamilyPerson`, a static `Family` object `aFamily`, and a dynamic `Person` reference `aPerson`. The declaration of `aFamilyPerson` may be read as: "`aFamilyPerson` is a reference to a dynamic object reference from `Family` objects to `Person` objects".

Families have attributes `mother`, `father`, and `child`, each of which are dynamic references to `Person` objects. The attribute reference `aFamilyPerson` has location qualification `Family##`, kind qualification "dynamic

[1] Notice that this is not the same function as the **location**() introduced in section 3.1.1, although their semantics are similar. In this chapter, **location**() refers to the location part of an attribute reference, whereas in section 3.1.1, **location**() refers to the location of a part-object.

```
   Person: (# ... #);
   Family:
     (# mother: ^Person;
        father: ^Person;
        child: ^Person;
     #);
   aFamilyPerson: .Family^Person;
   aFamily: @Family;
   aPerson: ^Person;
do Family.mother:->aFamilyPerson;        (* 1 *)
   aFamily.(aFamilyPerson)[]->aPerson[]; (* 2 *)
```

Figure 5.1: First Attribute Reference Example

object reference", and attribute qualification `Person##`, and is therefore qualified to refer any of the `Family` attributes, since they are all dynamic object references from `Family` objects to `Person` objects.

The example assigns the attribute expression `Family.mother:` to the attribute reference `aFamilyPerson`. Notice that, in the assignment, `aFamilyPerson` is *not* followed by a trailing colon. This is because the expression `aFamilyPerson:` would mean the `aFamilyPerson` attribute itself, whereas `aFamilyPerson` means the *value* of the attribute reference `aFamilyPerson`.

At `1`, `aFamilyPerson` refers the `mother` attribute of the `Family` pattern. Applying `aFamilyPerson` to the family object `aFamily` thus returns a reference to the mother in `aFamily`. At `2`, `aPerson[]` therefore refers the same person as `aFamily.mother[]`. In this example, all type-checking can be done at compile-time because of the qualifications of the attribute reference. The details of attribute reference type-checking are described in section 5.3.

Syntactically, the only difference between ordinary dot notation and application of an attribute reference to access an object, is that the name of the attribute reference is enclosed in parentheses. As usual in expression syntax, parentheses are used to express evaluation precedence rules. In this case we may think of the parentheses as denoting that computation of the attribute referred is followed by access to that attribute. Furthermore, the scope of the attribute reference name within the parentheses is not the remote object accessed, but the local scope of the expression.

5.3 Type Checking Attribute References

An important demand on the attribute reference concept is that it should be as type safe as possible. That is, to retain strong typing, attribute references themselves must be strongly typed. Attribute references are therefore amenable to static type checking where possible, and runtime type checking otherwise. To allow static type checking, attribute references are qualified, i.e., constrained with respect to the attributes they may refer. This section describes the constraints that must be enforced by the compiler to ensure strong typing of attribute references.

To denote the attribute referred by an attribute reference, I introduce the **attribute**() function, whose purpose is similar to the purpose of **object**() introduced in section 3.2.1, and the purpose of **pattern**() introduced in section 3.3. Given an attribute expression `attr_ref:` denoting an attribute reference attribute, **attribute**(`attr_ref:`) returns the actual attribute referred by `attr_ref`. Thus, in BETA code, the expression `attr_ref`, being the value of the attribute reference, has the meaning **attribute**(`attr_ref:`).

5.3.1 Location Qualification

The *location qualification* of an attribute reference constrains the objects on which the attribute reference can be applied. The purpose of location qualifications therefore is to allow the compiler to statically check the validity of the application of an attribute reference `attr_ref` to some object `o`. That is, to statically check the validity of an expression `o.(attr_ref)`.

The location qualification deserves its name due to its purpose of constraining the location of attribute expressions that can be assigned to an attribute reference. Below, some notation to describe location qualifications is introduced:

The **locqua**() function returns the location qualification of an attribute reference attribute, and is undefined on attributes that are not attribute references. For example, the declaration

```
attr_ref: .LocQua^AttrQua
```

implies that **locqua**(`attr_ref:`) = `LocQua##`. `attr_ref` may therefore be applied on objects `o` qualifying to `LocQua##`. In general, an attribute reference `attr_ref` may be applied to an object `o`, iff

$$\texttt{o\#\#} \subseteq \mathbf{locqua}(\texttt{attr\_ref:}) \tag{5.1}$$

Notice that the lefthand side of rule 5.1 is a full object qualification (section 4.2). I.e., 5.1 is equivalent to

$$\texttt{o} \in \mathbf{extension}(\mathbf{locqua}(\texttt{attr\_ref:}))$$

To cope with unconstrained location qualifications, rule 5.1 will be extended in section 5.4.1.

Notice the difference between **locqua**() and **location**(). **locqua**() is a statically known qualification, whereas **location**() of an attribute expression is in general not known until runtime. For example, after execution of

```
do Family.mother:->aFamilyPerson
```

the following becomes true:

$$\mathbf{location}(\mathbf{attribute}(\texttt{aFamilyPerson:})) = \texttt{Family\#\#}$$

whereas

$$\mathbf{locqua}(\texttt{aFamilyPerson:}) = \texttt{Family\#\#}$$

is statically known from the declaration of `aFamilyPerson`. As another example, notice that

$$\mathbf{location}(\texttt{aFamilyPerson:}) = \texttt{CurrentObject\#\#}$$

where `CurrentObject##` is the full object qualification whose single element is the object containing the `aFamilyPerson:` attribute itself, in this case the (implicit) object at the outermost block-level of figure 5.1.

In general, expressions assigning values to attribute references are of the form:

$$\texttt{n}_1\texttt{.n}_2\texttt{...n}_m\texttt{.A:->attr\_ref}$$

Assuming the assignment is legal, **location**(**attribute**(`attr_ref:`)) will afterwards be $\texttt{n}_1\texttt{.n}_2\texttt{...n}_m\texttt{\#\#}$. However, for the assignment to be legal, it must be ensured that all objects on which we are allowed to apply `attr_ref` do indeed have the attribute denoted by $\texttt{n}_1\texttt{.n}_2\texttt{...n}_m\texttt{.A:}$. This is the case iff

$$\mathbf{locqua}(\texttt{attr\_ref:}) \subseteq \texttt{n}_1\texttt{.n}_2\texttt{...n}_m\texttt{\#\#}$$

since, due to rule 5.1 and transitivity of $\subseteq$, this implies that

$$\texttt{o\#\#} \subseteq \texttt{n}_1\texttt{.n}_2\texttt{...n}_m\texttt{\#\#}$$

is true for all objects `o` on which it is legal to apply `attr_ref`.

Finally, the location qualification of an attribute reference `attr_ref` therefore expresses the following constraint on the location of the attribute referred by `attr_ref`:

$$\textbf{locqua}(\texttt{attr\_ref:}) \subseteq \textbf{location}(\textbf{attribute}(\texttt{attr\_ref:})) \tag{5.2}$$

Rule 5.2 should be compared to expression 3.1 introduced on page 18 to describe the allowed values of an object reference attribute `R`:

$$\textbf{qual}(\textbf{object}(\texttt{R:})) \subseteq \textbf{qual}(\texttt{R:})$$

Notice that these expressions, although similar, describe opposite constraints: The location qualification of an attribute reference is a *lower bound*, (with respect to $\subseteq$), on the location of attributes referred, whereas the qualification of an object reference is an *upper bound* on the type of objects referred. This may at first seem strange, but is a logical consequence of the informal definition of the location qualification as describing the set of objects on which an attribute reference can be applied.

5.3.2 Kind Qualification

The *kind qualification* decides whether an attribute reference `attr_ref` is allowed to refer, e.g., dynamic object reference attributes, static object reference attributes, or one of the other attribute kinds supported. BETA is a language with very few keywords, but nevertheless includes 11 kinds of *explicitly* declared object attribute kinds and four kinds of implicitly declared object attribute. Explicitly declared attribute kinds are:

- Object references: dynamic (`^`) and static (`@`).
- Component references: dynamic (`^|`) and static (`@|`). A component is an object including a call stack, i.e., it may serve as a coroutine or a process.
- Repetitions (arrays) of dynamic (`[]^`) and static (`[]@`) object references.
- Repetitions of dynamic (`[]^|`) and static (`[]@|`) component references.
- Repetitions of dynamic pattern references (`[]##`).
- Pattern references: Static and dynamic. Static pattern references may be further divided into non-virtuals, virtuals, virtual further bindings and final bindings, but that does not affect the definition of attribute references as described in this report.

Implicitly declared attribute kinds are:

- The origin references caused by the block structure, and described in section 3.1. Most of the time, origin references can be treated like static object references, since they too cannot be changed once an object has been created.
- Inserted items, i.e., blocks statically nested in the dopart of an object. Apart from being anonymous, i.e., having no name, an inserted item can be handled like a part-object, i.e., like a static object reference attribute[2].
- Index variables in for-loops. The index variable in a for-loop gives rise to an integer-attribute of the object.

[2]Actually, inserted items are a bit more tricky than that. The problem is that they, in principle, only exist during execution of their dopart.

- Labelled imperatives give rise to an implicit object attribute whose value is in principle a stackpointer and a code-address to restore in the case of `restart` or `leave` from the label.

Inserted items, index variables, and labelled imperative attributes are special by only being (logically) available at specific points in time during the execution of the object.

Finally, the introduction of attribute references has added an extra kind of attribute, which is then going to be the 16th attribute kind! I.e., including attribute references, BETA includes 12 explicit kinds of object attributes and 4 implicit kinds of object attributes!!

The **kindqua**() function returns the kind qualification of an attribute reference attribute, and is undefined for attributes that are not attribute references. For example, **kindqua**(`aFamilyPerson:`) = `dynref`.

The **kind**() function returns the actual kind of an attribute. For example, **kind**(`Family.mother:`) = `dynref`, and **kind**(`aFamily:`) = `statref`.

The kind qualification of an attribute reference imposes the following constraint:

$$\textbf{kindqua}(\texttt{attr\_ref:}) = \textbf{kind}(\textbf{attribute}(\texttt{attr\_ref:})) \tag{5.3}$$

5.3.3 Attribute Qualification

The *attribute qualification* constrains the qualification of the attribute referred. The function **attrqua**() returns the attribute qualification of an attribute reference attribute, and is undefined on other kinds of attributes. Thus, the declaration

```
attr_ref: .LocQua^AttrQua
```

implies that **attrqua**(`attr_ref:`) = `AttrQua##`, and allows `attr_ref` to refer attributes that are qualified by `AttrQua##`. The **qual**() function, introduced in section 3.2.1, returns the qualification of an attribute which is *not* an attribute reference attribute, and is undefined on attribute reference attributes. Thus, the attribute qualification of an attribute reference enforces the following rule:

$$\textbf{qual}(\textbf{attribute}(\texttt{attr\_ref:})) \subseteq \textbf{attrqua}(\texttt{attr\_ref:}) \tag{5.4}$$

From expressions 5.4 and 3.1 (page 18), it follows, for object references attributes, that

$$\textbf{qual}(\textbf{object}(\textbf{attribute}(\texttt{attr\_ref:}))) \subseteq \textbf{attrqua}(\texttt{attr\_ref:})$$

and **attrqua**(`attr_ref:`) is therefore a statically known upper bound on the type of the object referred by the attribute referred by `attr_ref`. The same holds for pattern reference attributes, where **attrqua**(`attr_ref:`) is an upper bound on the qualification referred by the attribute referred by `attr_ref`. Rule 5.4 does not apply to attribute references referring attribute references. Such meta attribute references will be described in section 5.5.

5.3.4 Example

Consider the following, statically legal, example:

```
   Vehicle: (# driver: ^Person #);
   Car: Vehicle (# codriver: ^Person #);
   CarAttr: .Car^Person;
do Vehicle.driver:->CarAttr;
```

The attribute expression `Vehicle.driver:` is assigned to the attribute reference `CarAttr`. The assignment is statically legal, since all attributes of `Vehicle` are attributes of all objects in **extension**(`Car##`), and because the attribute qualification and attribute kind are clearly valid. More precisely, the assignment is statically legal since it is statically known that

$$
\begin{array}{lcl}
(5.2) & & \mathbf{locqua}(\texttt{CarAttr:}) \subseteq \mathbf{location}(\mathbf{attribute}(\texttt{CarAttr:})) \\
& \Leftrightarrow & \mathbf{locqua}(\texttt{CarAttr:}) \subseteq \mathbf{location}(\texttt{Vehicle.driver:}) \\
& \Leftrightarrow & \texttt{Car\#\#} \subseteq \texttt{Vehicle\#\#} \\
& \Leftrightarrow & \mathbf{true} \\
\text{and} & & \\
(5.3) & & \mathbf{kindqua}(\texttt{CarAttr:}) = \mathbf{kind}(\mathbf{attribute}(\texttt{CarAttr:})) \\
& \Leftrightarrow & \mathbf{kindqua}(\texttt{CarAttr:}) = \mathbf{kind}(\texttt{Vehicle.driver:}) \\
& \Leftrightarrow & \text{dynref} = \text{dynref} \\
& \Leftrightarrow & \mathbf{true} \\
\text{and} & & \\
(5.4) & & \mathbf{qual}(\mathbf{attribute}(\texttt{CarAttr:})) \subseteq \mathbf{attrqua}(\texttt{CarAttr:}) \\
& \Leftrightarrow & \mathbf{qual}(\texttt{Vehicle.driver:}) \subseteq \mathbf{attrqua}(\texttt{CarAttr:}) \\
& \Leftrightarrow & \texttt{Person\#\#} \subseteq \texttt{Person\#\#} \\
& \Leftrightarrow & \mathbf{true}
\end{array}
$$

The following is an example of a statically illegal assignment to an attribute reference:

```
   VehicleAttr: .Vehicle^Person;
do Car.codriver:->VehicleAttr;
```

This example is illegal since not all `Vehicle` objects have the `Car.codriver:` attribute. Formally, it is illegal because the assignment violates rule 5.2:

$$
\begin{array}{cl}
& \texttt{Car\#\#} \subset \texttt{Vehicle\#\#} \\
\Leftrightarrow & \mathbf{location}(\texttt{Car.codriver:}) \subset \mathbf{locqua}(\texttt{VehicleAttr:}) \\
\Leftrightarrow & \mathbf{location}(\mathbf{attribute}(\texttt{VehicleAttr})) \subset \mathbf{locqua}(\texttt{VehicleAttr:})
\end{array}
$$

when in fact the opposite should be true for the assignment to be legal.

5.3.5 Strong Typing

Rules 5.2, 5.3, and 5.4 describe the constraints on the values of attribute references, i.e., the attributes an attribute reference is allowed to refer. As shown through examples in the previous section, the compiler is in some situations able to statically deduce the correctness of assignments to attribute references. In other situations, however, the validity of an assignment may only be checkable at runtime. This would, for example, be the case when assigning a weakly qualified attribute reference to a strongly qualified attribute reference, as illustrated below:

```
   VehicleAttr: .Vehicle^Person;
   CarAttr: .Car^Person;
do ...; CarAttr->VehicleAttr;
```

This assignment requires a runtime type check to ensure that the attribute denoted by `CarAttr`, which is statically known to be an attribute of all `Car` objects, is also an attribute of all `Vehicle` objects. Like the compile time checks, runtime type checks inserted by the compiler must ensure that the rules 5.2, 5.3, and 5.4 are enforced.

Likewise, the application of an attribute reference, say `o.(attr_ref)`, must be checked. If it can be statically deduced that the application is in accordance with rule 5.1, it is statically known that `o` does in fact have the attribute referred by `attr_ref`.

Finally, it must be checked whether the operation attempted is compatible with **kindqua**(`attr_ref:`). For example, changing the value of a static object reference through an attribute reference should of course be prohibited. However, most such checks are equal to the usual validity checks performed by the compiler, and can be done at compile-time.

One check, however, needs to be postponed until runtime. The reason is that BETA, for efficiency and garbage collection reasons, does not treat simple attributes such as integers, chars, booleans, etc., as fullblown objects, and therefore does not allow dynamic references to such attributes[3]. Currently, attribute references with kind qualification `@` and location qualification `Object` *are* allowed to refer such simple attributes, and therefore the following example requires a runtime check on the application of the `intref` attribute reference:

```
   foo: (# i: @Integer #);
   aFoo: @foo;
   intref: .@;
   o: ^Object;
do foo.i:->intref;
   aFoo.(intref)[]->o[]; (* ILLEGAL!! Requires runtime check *)
```

To get rid of this runtime check, attribute references would need to distinguish simple attributes from other attributes, which may, for reasons of efficiency, prove necessary anyway.

5.3.6 Explicit Runtime Checks

To allow explicit qualification checks, the meta-level interface, (MLI), described in chapter 6, provides the following patterns:

- `MLI.Qual`: Returns the qualification of an attribute expression denoting an attribute which is not an attribute reference attribute. For example, `attr_ref->MLI.Qual` returns **qual**(**attribute**(`attr_ref:`)).
- `MLI.Loc`: Returns the location of an attribute expression. For example, `attr_ref->MLI.Loc` returns **location**(**attribute**(`attr_ref:`)).
- `MLI.Kind`: Returns the kind of an attribute expression. For example, `attr_ref->MLI.Kind` returns **kind**(**attribute**(`attr_ref:`)).
- `MLI.LocQua`: Returns the location qualification of an attribute reference attribute. For example, `attr_ref:->MLI.LocQua` returns **locqua**(`attr_ref:`).
- `MLI.KindQua`: Returns the kind qualification of an attribute reference attribute. For example, `attr_ref:->MLI.KindQua` returns **kindqua**(`attr_ref:`).
- `MLI.AttrQua`: Returns the attribute qualification of an attribute reference attribute. For example, `attr_ref:->MLI.AttrQua` returns **attrqua**(`attr_ref:`).

These functions of the meta-level interface allows the meta-level programmer to explicitly ensure that his code will never violate any of the BETA type-rules. In any case, this will be prevented by implicit runtime checks inserted by the compiler.

[3]Although dynamic references to simple values could, in principle, be implemented using an extra indirection in this special case!

5.4 Unconstrained Attribute References

Highly generic code may need the ability to declare attribute references that are able to refer any attribute. In this section, the attribute reference concept is therefore extended to allow the declaration of such unconstrained attribute references.

An attribute reference declaration includes three constraints: the location qualification, the kind qualification, and the attribute qualification. To declare a fully unconstrained attribute reference, each of these constraints must specify "no constraint". How to do this is the subject of sections 5.4.1, 5.4.2, and 5.4.3.

5.4.1 Unconstrained Location Qualification

The standard BETA way of expressing "no constraint" is to use the `Object` pattern as a qualification. However, in the case of the location qualification of an attribute reference `attr_ref`, the usual interpretation of `Object` would imply that `attr_ref` could be applied on all objects, and that `attr_ref` could only refer attributes known to be part of any object. Such attribute references would be useless, since the `Object` pattern has no attributes at all.

The `Object` pattern is the *top* of the BETA qualification hierarchy, and what is needed here is the *bottom*. Instead of adding a bottom pattern to BETA[4], two different ways of expressing "no location constraint" will be allowed:

1. Simply omitting the location qualification, i.e., by allowing `loc_qua` to be the empty string.
2. Specifying `Object` as the location qualification, thus imposing an alternative interpretation of `Object` when used in the location qualification part of an attribute reference declaration.

Seen isolated, these possibilities are semantically equivalent. However, the special interpretation of `Object` as a location qualification allows attribute references with an unconstrained virtual location qualification. For example, we may specify:

```
foo:
  (# locQua:< Object;
     a: .locQua^Text;
  enter a
  do INNER
  #);
```

stating that `a` is an attribute reference qualified to refer dynamic text reference attributes in any pattern. Using only this knowledge, it is necessary to runtime check all applications of `a`, but since the `locQua` virtual can be strengthened in subpatterns of `foo`, static type checking of applications of `a` may be possible in subpatterns of `foo`. For example, the following code may be completely compile-time checked:

```
Person: (# address: ^Text; ... #);
bar: foo
  (# locQua:: Person; p: ^locQua;
  enter p[] do p.(a)[]->screen.putline;
  #);
aPerson: @Person;
do (Person.address:,aPerson[])->&bar
```

[4]A role the special value `NONE` is very close to playing already, as described in section 4.1.2.

To allow application of attribute references whose location qualification is `Object`, rule 5.1 from page 34 is extended as follows:

$$\begin{array}{ll} & (\mathbf{locqua}(\texttt{attr\_ref:}) \neq \texttt{Object\#\#}) \ \mathbf{and} \ (\texttt{o\#\#} \subseteq \mathbf{locqua}(\texttt{attr\_ref:})) \\ \mathbf{or} & (\mathbf{locqua}(\texttt{attr\_ref:}) = \texttt{Object\#\#}) \ \mathbf{and} \ (\texttt{o\#\#} \subseteq \mathbf{location}(\mathbf{attribute}(\texttt{attr\_ref:}))) \end{array} \tag{5.5}$$

Notice that rule 5.5 implies that a runtime check on the application of attribute references with unconstrained location qualification will always be necessary, since **attribute**(`attr_ref:`) denotes the runtime value of `attr_ref`.

5.4.2 Unconstrained Kind Qualification

The kind qualification of an attribute reference `attr_ref` decides which kind of attribute it is allowed to refer. For example, the code

```
statattr: .loc_qua@attr_qua;
dynattr: .loc_qua^attr_qua;
```

states that the kind qualification of `statattr` is "static object reference", and the kind qualification of `dynattr` is "dynamic object reference". To allow the declaration of a fully unconstrained attribute reference, I introduce the following syntax:

```
generic_attr: .loc_qua?
```

The question mark denotes that `generic_attr` is not constrained to refer any specific attribute kind, and that the runtime value of **kind**(**attribute**(`generic_attr:`)) is therefore statically unknown. Due to the unknown attribute kind, generic attribute references can only be applied to produce other attribute references, as illustrated in figure 5.2. In addition, generic attribute references may be assigned to more qualified attribute references, subject to runtime type checks.

```
   queue:
     (# element:< Object;
        first: ^element;
        ...
     #);
   persons: @queue (# element:: Person #);
   families: @queue (# element:: Family #);
   queueAttr: .queue?;
   p,q: ##Object;
do queue.first:->queueAttr;
   persons.(queueAttr):->MLI.Qual->p##;  (* p## = Person##   *)
   persons.(queueAttr):->MLI.Loc->q##;   (* q## = persons##  *)
   families.(queueAttr):->MLI.Qual->p##; (* p## = Family##   *)
   families.(queueAttr):->MLI.Loc->q##;  (* q## = families## *)
```

Figure 5.2: Retrieving the qualification of an object attribute.

Omitting or using `Object` as the location qualification of a generic attribute reference leads to a totally unconstrained attribute reference, allowed to refer any attribute:

```
any_attr1: .Object?
any_attr2: .?
```

Note that generic attribute references are not allowed to declare an attribute qualification. The reason is that the qualification has no obvious interpretation if the attribute referred is itself an attribute reference attribute, i.e., if `generic_attr` is an attribute reference to an attribute reference. For the same reason, the **qual**() function was described as being undefined on attribute reference attributes.

5.4.3 Unconstrained Attribute Qualification

The attribute qualification (`attr_qua`) of an attribute reference declaration can be omitted, and is semantically equivalent to using `Object` as attribute qualification. As opposed to using `Object` as location qualification, this does not impose any special interpretation of the location qualification.

5.5 References to Attribute References

Current BETA includes 15 different "kinds" of object attributes. In preceding sections, the syntax and semantics of attribute references referring these existing attribute kinds was described, along with the syntax of attribute references allowed to refer any kind of attribute.

However, by adding attribute references to the language, we have added several new kinds of attributes. Since the aim is to be able to explicitly talk of any kind of attribute using attribute references, it seems as if we are now forced to add yet another attribute reference kind in order to describe attribute reference attributes themselves.

The parallel problem in object-oriented systems based on meta-classes, e.g., Smalltalk, is the description of meta-classes, leading to meta-meta classes and infinite regress. In such systems, the problem is solved by arranging for a meta-class to describe itself. A similar arrangement is needed for attribute references in order to avoid the need for an infinite number of attribute kinds.

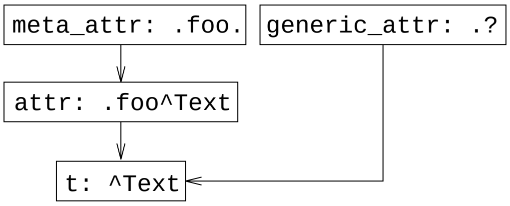
Thus, the final kind of attribute reference is the *meta attribute reference*, constrained to refer other attribute references, including meta attribute references themselves. The syntax of meta attribute references is as follows:

```
meta_attr: .loc_qua.
```

As with generic attribute references, described in section 5.4.3, meta attribute references are not allowed to include an attribute qualification, and for the same reason: There is no obvious interpretation of the attribute qualification in this case.

The reason why *both* generic attribute references and meta attribute references are needed, is that generic attribute references cannot be applied to objects to get the value of the attribute referred. However, given a meta attribute reference `meta_attr`, it can be applied to an object to get the value of the attribute reference referred. This is illustrated below:

```
   foo: @
     (# t: ^Text;
        attr: .foo^Text;
     #);
   meta_attr: .foo.;
   generic_attr: .?;
do foo.t:->foo.attr;
   foo.attr:->meta_attr;
   foo.(meta_attr)->generic_attr;
```



Thus, the purpose of the meta attribute reference is to allow meta-level code to follow chains of attribute references. Assumingly, this is not going to be a common case, but must be supported for completeness.

5.6 Full Object Location Qualifications

Until now, an attribute reference `attr_ref` has been described as the identity of some attribute of objects qualifying to **locqua**(`attr_ref:`). I.e., to actually apply `attr_ref`, an object is needed as well. Notice that this separates attribute references from, e.g., C pointers, that are simply the address of some memory cell. That is, C pointers are the address of a specific attribute in a specific object, whereas attribute references in general only identifies the attribute, not the object.

But iff **location**(**attribute**(`full_ref:`)) is a full object qualification (section 4.1.2), rule 5.5 implies that there is in fact only one object on which `full_ref` can be applied. In effect, `full_ref` is simply a reference to a specific attribute of a specific object. Fortunately, the information present in an attribute reference allows us to retrieve the value of the attribute referred, by first retrieving the single object on which it can be applied, and then apply the attribute reference. This is illustrated by the `FullIntRefValue` pattern below:

```
FullIntRefValue:
  (# ref: .@Integer; value: @Integer;
     loc: ##Pattern; o: ^Object;
  enter ref
  do ref->MLI.Loc->loc##;
     &loc[]->o[];          (* extension(loc##) = {o} *)
     o.(ref)->value;
  exit value
  #);
```

To understand `FullIntRefValue`, notice that the location of `ref` is assumed to be a full object qualification, and that the singular object of its extension is therefore returned by the expression `&loc[]` (section 4.1.3).

The implication of attribute references whose location is a full object qualification, is that attribute references, as a special case, can be used to monitor the value of any attribute in any specific object. For example, this may be useful to implement constraint systems, monitoring some set of attributes to ensure that their values are in some specific relation to one another. However, this could also be achieved by simply supplying an attribute reference *and* an object to the constraint system.

5.7 Comparing Attribute References

Dynamic *pattern* references may be compared using the operators `=`, `<=` and `>=`. In the case of pattern references, the semantics of the `<=` operator is as follows:

$$\begin{aligned} & \texttt{p\#\#} <= \texttt{q\#\#} \\ \Leftrightarrow\ & \texttt{p\#\#} \subseteq \texttt{q\#\#} \\ \Leftrightarrow\ & \mathbf{extension}(\texttt{p\#\#}) \subseteq \mathbf{extension}(\texttt{q\#\#}) \end{aligned}$$

Likewise, attribute references can be compared. Two attribute expressions `a1:` and `a2:` are comparable if and only if they refer the same attribute. That is, when applied to the same object, comparable attributes will always yield the same value.

The relationship between comparable attributes is defined by means of their locations, as described below:

$$\begin{aligned} & \texttt{a1:<=a2:} \\ \Leftrightarrow\ & \mathbf{location}(\texttt{a1:}) \subseteq \mathbf{location}(\texttt{a2:}) \end{aligned} \tag{5.6}$$

That is, `a1:` is less than `a2:` iff `a1:` and `a2:` denote the same attribute, and the location part of `a1:` is a subqualification of the location part of `a2:`.

If attributes `a1:` and `a2:` are incomparable, the expressions `a1:<=a2:`, `a1:=a2:`, and `a1:>=a2:` all evaluate to false.

If `a1:<=a2:`, we may speak of `a1:` as a *sub-attribute* of `a2:`. This terminology is consistent with speaking of `p1##` as a sub-pattern of `p2##` iff `p1## <= p2##`.

As a result of the comparison operators, attribute references do not only make it possible to denote specific attributes, they also allow the expression of sets of objects on which they should be applied. For a simple example, consider the following code:

```
   foo: (# t: ^Text #);
   f: @foo;
   attr: .foo^Text;
do foo.t:->attr;
   (if f.t:<=attr then
       (* f.t: is a sub-attribute of foo.t: *)
       f.t[]->screen.putline;
   if)
```

Since `f.t:` is a sub-attribute of `foo.t:`, the code above will print out the text referred by `f.t:`. Thus, the attribute reference `attr` has been used to point out a set of text attributes for printing. Used together with the ability of scanning object attributes, as provided by the meta-level interface of chapter 6, this ability is a very useful property of attribute references.

The examples in section 7 will demonstrate how this can be applied in a more realistic setting to point out object attributes that should receive special treatment.

5.8 Summary

This chapter described the novel language concept of attribute references. An attribute reference turns attributes into first class language citizens, i.e., attribute references reify the concept of attributes.

Attribute references are qualified, making them in some cases amenable to static type checking. In other situations, where static checking is not possible, attribute references allow dynamic checks to be inserted by the compiler. Thus, type safety of the programming language is retained.

Finally, attribute references can be compared, thus allowing a program to test whether two attribute references in fact refer the same attribute, and whether they are sub-attributes of one another.

Attribute references are a natural extension to the BETA language, which already included the ability to explicitly denote objects and patterns. With the advent of attribute references, BETA is now also able to explicitly talk about attributes.

To the best of my knowledge, attribute references is currently the only general language mechanism for a statically typed language, reifying the concept of attributes in a type-safe way, and thus turning attributes into first-class language citizens.

Chapter 6

The Meta-Level Interface

Previous chapters motivated the need for reflectional capabilities, described selected aspects of the BETA language, and generalized the BETA notions of qualifications and pattern references. Finally, the concept of attribute references was added to the language.

This chapter describes the meta-level interface itself, providing comprehensive introspectory and intercessory capabilities. These capabilities are provided without compromising type-safety of the language, and thus without breaking any assumptions made by the compiler. Type-safety of the meta-level interface is mainly a consequence of the type-safety provided by the attribute reference concept, as described in chapter 5.

The meta-level interface described in this chapter will be available to BETA programs through the globally visible object MLI, provided by the basic BETA environment.

Finally, notice that the current MLI is a pure paper design, although based on experience from the implementation of several meta-level applications in the Mjølner BETA System, including persistence, distribution, object-browsers and a source-level debugger. As a paper design, it is likely to change during implementation and use. Implementation work has recently been initiated, as described in [Brandt & Schmidt 95].

6.1 Object Introspection

The `GetObjQual` pattern shown in figure 6.1 returns the actual type of an object, i.e., the pattern of which the object is an instance. `GetObjQual` thus corresponds to the **qual**() function (section 3.2.1) evaluated on objects.

Notice that if `o` is a singular object, i.e., an instance of a pattern with exactly one instance, the value returned

```
GetObjQual:
  (# o: ^Object; value: ##Object;
  enter o[]
  do ...
  exit value##
  #);
```

Figure 6.1: `GetObjQual`: Retrieving the pattern of an object.

```
GetObjLocation:
  (# o: ^Object; value: ^Object;
  enter o[]
  do ...
  exit value[]
  #);
```

Figure 6.2: `GetObjLocation`: Retrieving the location of a part object.

from `GetObjQual` is the full object qualification `o##`. For example:

```
Car: (# engine: @ (# ... #)#);
aCar: @Car;
```

Here, `aCar.engine[]` denotes a singular object, and `(aCar.engine[]->GetObjQual)` = `aCar.engine##`, i.e., a full object qualification. However, `aCar[]` is not a singular object, since there can be other instances of the `Car##` pattern. Thus, `(aCar[]->GetObjQual)` = `Car##`, i.e., a full pattern qualification.

The `location` of a part object may be retrieved by calling the `GetObjLocation` pattern shown in figure 6.2, thus corresponding to the **location**() function evaluated on objects (see section 3.1.1). If the parameter to `GetObjLocation` is not a part of another object, `GetObjLocation` returns `NONE`.

Given an object `o`, it can be converted into the full object qualification `o##`, or into `o[]->GetObjQual`, i.e., the pattern of which `o` is an instance. Thus, the remaining introspectional capabilities are targeted at pattern and attribute references.

6.2 Pattern Reference Introspection

As described in section 4.1.2, qualifications correspond to expressions of the form $\mathtt{n}_1\mathtt{.n}_2\mathtt{....n}_m\mathtt{\#\#}$, and may be subdivided into four categories: In one dimension, subdivision is done according to whether $\mathtt{n}_m$ denotes a pattern or object reference attribute (pattern qualification versus object qualification). In the other dimension, subdivision is done according to whether all $\mathtt{n}_1$ to $\mathtt{n}_{m-1}$ denote object reference attributes, (in which case it is a full qualification), or whether at least one of them denotes a pattern attribute (in which case it is a partial qualification).

Given a qualification $\mathtt{p\#\#} = \mathtt{n}_1\mathtt{.n}_2\mathtt{....n}_m\mathtt{\#\#}$, we may think of it as being composed of two parts:

1. The origin `p'##` of `p##`, where `p'##` usually corresponds to the expression $\mathtt{n}_1\mathtt{.n}_2\mathtt{....n}_{m-1}\mathtt{\#\#}$
2. n_m:, which is an attribute of objects `o` such that $\mathtt{o\#\#} \subseteq \mathtt{p'\#\#}$.

The main introspectional interface to pattern references is the `PatRefInfo` pattern, shown in figure 6.3 on page 47. `PatRefInfo` is designed to reflect the decomposition described above, i.e., it supports decomposition of pattern references into primitive parts by allowing a pattern reference $\mathtt{p\#\#} = \mathtt{n}_1\mathtt{.n}_2\mathtt{....n}_m\mathtt{\#\#}$ to be split into another pattern reference $\mathtt{p'\#\#} = \mathtt{n}_1\mathtt{.n}_2\mathtt{....n}_{m-1}\mathtt{\#\#}$, and an attribute reference corresponding to $\mathtt{n}_m\mathtt{:}$. Decomposition can be continued until a full pattern or object qualification has been obtained.

Given a pattern reference `p##` as enter parameter, `PatRefInfo` calls one of three virtuals depending on the type of `p##`:

```
PatRefInfo:
  (# p: ##Object;
     OnPatternQual:<
       (# GetOrigin:
            (# onFullPatternQual:<
                 (# ori: ^Object enter ori[] do INNER #);
               onPartialPatternQual:<
                 (# ori: ##Object enter ori## do INNER #);
            do ...; (* Calls OnFull- or OnPartialPatternQual. *)
            #);
          AsAttribute:
            (# OnStaticPat:<
                 (# att: .: enter att do INNER #);
               OnDynamicPat:<
                 (# att: .## enter att do INNER #);
               OnInserted:<
                 (# do INNER #);
            do ...; (* Calls one of the above virtuals *)
            #);
          GetPrefix:
            (# pref: ##Object do ... exit pref## #);
       do INNER;
       #);
     OnFullObjectQual:<
       (# GetObject:
            (# o: ^Object do &p[]->o[] exit o[] #);
       do INNER;
       #);
     OnPartialObjectQual:<
       (# GetOrigin:
            (# ori: ##Object do ... exit ori## #);
          AsAttribute:
            (# OnStatic:<
                 (# att: .@ enter att do INNER #);
               OnDynamic:<
                 (# att: .^ enter att do INNER #);
               OnStaticComp:<
                 (# att: .@| enter att do INNER #);
               OnDynamicComp:<
                 (# att: .^| enter att do INNER #);
            do ...; (* Calls one the four virtuals. *)
            #);
       do INNER;
       #);
  enter p##
  do ...; (* Calls the appropriate virtual *)
  #);
```

Figure 6.3: `PatRefInfo`: Introspection pattern references.

```
AsObjectDescriptor:
  (# p: ##Object; value: ##Object
  enter p##
  do ...
  exit value##
  #);
```

Figure 6.4: `AsObjectDescriptor`: Obtaining the ObjectDescriptor corresponding to a pattern.

- If `p##` is a pattern qualification, `OnPatternQual` is called. `OnPatternQual` allows the programmer to
  - Get the origin of `p##`. If `p##` is a full pattern qualification, the origin is an object. Otherwise, i.e., if `p` is a partial pattern qualification, the origin is another pattern reference.
  - Get a reference to the attribute of the origin corresponding to `p##`. This is possible unless `p##` correponds to an anonymous (inserted) pattern.
  - Get the prefix, i.e., the superpattern of `p##`.
- If `p##` is a full object qualification, it is possible to retrieve the single object in **extension**(`p##`). Notice that this may be implemented using the `&` operator, as described in section 4.1.3.
- If `p##` is a partial object qualification, it may again be split into an origin pattern and an attribute.

The next section adds further functionality to `PatRefInfo`. In addition to the functionality of `PatRefInfo`, all pattern references may be converted into ObjectDescriptors, using `AsObjectDescriptor`, shown in figure 6.4. Semantically, `AsObjectDescriptor` strips the pattern reference from information concerning specific origin objects. Thus, the ObjectDescriptor can be seen as the canonic representative of the source code describing the structure of a pattern. In case of full object qualifications, `AsObjectDescriptor` returns the ObjectDescriptor of which the object is an instance.

6.2.1 Superpatterns and Subpatterns

Given a reference to a pattern `p##`, class-browsers and other tools may need information on the immediate superpattern of `p##`, or, in BETA terms, the *prefix* of `p##`. Reversely, such tools may need to retrieve the list of immediate subpatterns of `p##`.

Thus, we would like the MLI to be able to deliver super and subpattern information. Given a pattern `p##`, the prefix of `p##` is always well-defined, and a feasible meta interface was shown in figure 6.3 as the `GetPrefix` method inside `PatRefInfo.OnPatternQual`.

But what about the list of subpatterns? Exactly what subpatterns do we mean? At least the following possibilities exist:

- The subpatterns whose code is currently loaded into the process.
- All subpatterns contained in some database schema.
- The list of all subpatterns ever created.

The latter possibility is probably rather unpractical, but in any case this question cannot be answered without taking the programming environment into consideration. We clearly need scope control, in order to choose

```
PatRefInfo:
  (# ...
     OnPatternQual:<
       (# ...
          ScanSubPatterns:
            (* Calls INNER for each subpattern of p in scope. *)
            (# scope: ^ScopeInfo;
               thisSub: ##Object;
            enter scope[]
            do ... INNER ...
            #);
       #);
     ...
  #);
```

Figure 6.5: `ScanSubPatterns`: Scanning Subpatterns.

between the possibilities, but how to specify the scope of a subpattern scan depends on the programming and runtime environments.

A tentative meta interface for scanning subpatterns is shown as the `ScanSubPatterns` pattern in figure 6.5, extending `PatRefInfo.OnPatternQual`. The exact nature of `ScopeInfo` is not described.

6.2.2 External Representation of Pattern References

To save dynamic pattern reference values on secondary storage, or in general to communicate such values between different processes, it is necessary to convert a pattern reference into an external representation and vice versa. In general, however, the definition of a qualification depends on a number of objects, and information on these objects must somehow be saved along. On the other hand, an important goal of the MLI is to allow flexible persistence and distribution mechanisms to be built on on top, and therefore the MLI itself should not define a general object persistence mechanism.

This problem is solved by a tailorable mechanism allowing pattern references to be saved to and restored from a stream. Users of this mechanism, shown in figure 6.6 as the patterns `SavePatternRef` and `GetPatternRef`, must implement a set of virtuals to supply a way to save and restore object identities. To avoid the need of saving object references, the pattern reference may be converted into an ObjectDescriptor before being saved onto a stream.

The values written to the stream clearly depends on the runtime environment in which the metaobject protocol is to be implemented. This subject will be considered in section 8. For now, we'll assume that the runtime environment allows the conversion of a pattern reference into a sequence of simple values, and vice versa.

6.2.3 Scanning Pattern Attributes

An important need is the ability to scan the attributes of a pattern, and a mechanism to do so is shown in figure 6.7. For each explicit attribute kind, a method is provided to scan the attributes of a pattern. Each `Scan*` pattern executes INNER for each attribute of the relevant kind in the pattern `p##` entered as parameter.

```
SavePatternRef:
  (# saveObjectRef:< (# o: ^Object enter o[] do INNER #);
     p: ##Object; s: ^Stream;
  enter (p##,s[])
  do ...
  #);
GetPatternRef:
  (# getObjectRef:< (# o: ^Object do INNER exit o[] #);
     s: ^Stream; p: ##Object;
  enter s[]
  do ...
  exit p##
  #);
```

Figure 6.6: `SavePatternRef`: External representation of patterns and qualifications.

```
AttrScan: (# locQua:< Object; p: ##Object enter p## do INNER #); (* ABSTRACT! *)
ScanDyn:         AttrScan (# current: .locQua^    do ... #);
ScanStat:        AttrScan (# current: .locQua@    do ... #);
ScanDynComp:     AttrScan (# current: .locQua^|   do ... #);
ScanStatComp:    AttrScan (# current: .locQua@|   do ... #);
ScanDynRep:      AttrScan (# current: .locQua[]^  do ... #);
ScanStatRep:     AttrScan (# current: .locQua[]@  do ... #);
ScanPatRep:      AttrScan (# current: .locQua[]## do ... #);
ScanDynCompRep:  AttrScan (# current: .locQua[]^| do ... #);
ScanStatCompRep: AttrScan (# current: .locQua[]@| do ... #);
ScanDynPat:      AttrScan (# current: .locQua##   do ... #);
ScanStatPat:     AttrScan (# current: .locQua:    do ... #);
ScanAttrVar:     AttrScan (# current: .locQua.    do ... #);
ScanAll:         AttrScan (# current: .locQua?    do ... #);
```

Figure 6.7: Scanning explicit pattern attributes.

```
ScanOri:      AttrScan (# current: .locQua@                      do ... #);
ScanInserted: AttrScan (# current: .locQua@                      do ... #);
ScanIndices:  AttrScan (# current: .locQua@Integer               do ... #);
ScanLabels:   AttrScan (# current: .locQua@ActivationRecord do ... #);
```

Figure 6.8: Scanning implicit pattern attributes.

As described in section 5.3.2, BETA, in addition to the 12 explicit attribute kinds, include 4 implicit kinds of attributes, namely origins, inserted items, index-variables, and those arising from labelled imperatives. These attributes are handled by the the patterns shown in figure 6.8. `scanOri` and `scanInserted` qualifies `current` as a static object reference attribute, since BETA does not include a specific syntax for origin- and inserted item references, and since these references, like static references, are not allowed to change dynamically. `ScanIndices` qualifies current as a static integer attribute since index-variables, (from for-loops), are always integer valued. The implicit attributes originating from labelled imperatives are qualified as static[1] references to an `ActivationRecord`. The `ActivationRecord` pattern, which will be described in section 6.5.1, includes information corresponding to a stackpointer and a code-address. Notice that the set of attributes scanned by `ScanInserted`, `ScanIndices`, and `ScanLabels` only exist in specific execution states. I currently consider it an open problem how to ensure that these attributes are not accessed in objects that are not in a feasible state of execution.

6.2.4 Enter and Exit Parameters

This section describes how to scan enter and exit parameters of BETA patterns. This ability is useful in situations where method calls need to be synthesized by some generic mechanism. More specifically, distributed object systems, debuggers, querying mechanisms, and embedded interpreters are examples of systems that may benefit from this ability.

In BETA, the formal parameters of a method pattern is specified as a list of attribute denotations and patterns. That is, formal parameters are attributes of some object, or they are patterns. The attributes denoted are usually attributes of the method object itself, but this is not necessarily the case. For example, the following code is legal:

```
foo:
  (# o: ^Object;
     bar: (# enter o[] do ... #);
  #);
```

Here, the method pattern `bar` has a formal enter parameter corresponding to an attribute of the enclosing pattern `foo`. In general, the formal enter and exit parameters of a method pattern `m` may be attributes of any object reachable from an instance of `m`. Due to this generality, the scanning of attributes corresponding to formal enter and exit parameters is not as simple as the scanning of pattern attributes in the previous section.

This problem is solved by providing a mechanism which scans the parameters of a pattern, and describes each enter and exit parameter as a *list* of attribute references. Since all enter/exit parameters are reachable from the method object, they can be accessed by applying the attribute list to a sequence of objects, starting in the method object itself. In the example above, the formal enter parameter `o[]` can be described as the attribute list `(bar.origin:,foo.o:)`, and the additional information that the parameter is an object reference, as opposed to an enter parameter denoting the evaluation of `o`. The latter possibility is shown

[1]These references are not really static, but their value are only allowed to change when the labelled imperative is executed.

```
ScanEnterParams:
  (# locQua:< Object;
     OnAttributeList:<
       (# isreference: @Boolean;
          ScanAttributeList:<
            (# OnDynRef:< (# att: .locQua^ enter att do INNER #);
               OnStatRef:< (# att: .locQua@ enter att do INNER #);
               ... (* Other attribute kinds. *)
            do <<Calls appropriate virtual for each list element>>
            #);
       enter isreference do INNER; scanAttributeList;
       #);
     OnInsertedItem:< (# ii: ##Object enter ii## do INNER #);
     p: ##Object;
  enter p##
  do (for <<each enter parameter>> repeat
        (if <<current is inserted item>> then <<Call OnInsertedItem>>
        else <<Call OnAttributeList>>
        if)
     for)
  #);
ScanExitParams:
  (# <<Like scanEnterParams, but scans exit parameters.>> #);
```

Figure 6.9: Scanning enter and exit parameters of patterns.

below:

```
foo:
  (# onBarEntry: @(# enter ... #);
     bar: (# enter onBarEntry do ... #);
  #);
```

Here, the enter parameter denotes that the `onBarEntry` object must be executed, as opposed to the previous example, where `o[]` denoted a simple object reference enter parameter.

The final possibility is that an enter parameter may be given in the shape of a pattern, or, more precisely, an inserted item:

```
WindowName:
  (# enter (# name: ^Text; enter name[] do <<set name of window>>#)
     exit (# name: ^Text; do <<get name of window>> exit name[] #)
  #)
```

Such inserted item parameters are provided as pattern references by the parameter scanning mechanism.

The concept of enter/exit parameters applies to all kinds of qualifications, and thus the enter and exit parameters can be scanned on any pattern reference. The meta-level interface for parameter scanning are the patterns `scanEnterParams` and `scanExitParams`, shown in figure 6.9.

```
GetMPSObjectDescriptor:
   (# p: ##Object; od: ^MPS.ObjectDescriptor;
   enter p##
   do ...
   exit od[]
   #);
```

Figure 6.10: `GetMPSObjectDescriptor`: Obtaining the source code corresponding to a pattern reference.

6.2.5 Patterns and Source Code

In the Mjølner BETA Meta Programming System (MPS), source code is represented and manipulated as abstract syntax trees (AST's). It would thus in both principle and practice be possible to map a pattern reference denoting a pattern qualification onto the AST node corresponding to the ObjectDescriptor. In fact, this is exactly what objectbrowsers and debuggers in the Mjølner BETA System do today. Given an AST that was annotated by the compiler, these tools have access to all information that was available to the compiler.

While this use of MPS is acceptable for the debugger tool, it is too heavy weight for general and efficient meta programming. Furthermore, only information needed by the compiler is efficiently accessible through the compiler annotated AST. More is needed for meta programming in general. Finally, the use of MPS requires access to the source code of a program, which is not an acceptable requirement when shipping applications.

Even though MPS is not a feasible alternative to the metaobject protocol described in this report, a number of tools may benefit from the possibility of mapping pattern references onto AST nodes. Example tools include integrated programming environments, and distributed object systems needing to generate code for new platforms on the fly, as objects move between different hardware.

The `GetMPSObjectDescriptor` pattern shown in figure 6.10 provides a way to obtain a reference to the source code, given a pattern reference.

6.2.6 Summary

This section presented the introspectional interface to pattern references. The part of the MLI allowing introspection of pattern references was provided by the patterns `PatRefInfo`, `SavePatternRef`, `GetPatternRef`, `Scan*` patterns for attribute scanning, `ScanEnterParams`, `ScanExitParams`, and `GetMPSObjectDescriptor`. Using these patterns, meta-level code can get detailed information on the pattern referred by some pattern reference, and is able to exchange pattern references between separate processes.

6.3 Attribute Reference Introspection

This section describes the introspectory interface to attribute references. The concept of attribute references was introduced in chapter 5, which also, in section 5.3.6, briefly described some of the functionality of the introspectory interface to attribute references. The main instrospectory interface to attribute references is shown in figure 6.11 on page 54.

- `Qual` returns the qualification of the attribute referred by the enter parameter `a`, unless `a` refers an attribute reference attribute, in which case the `undefined` exception is raised.

```
Qual:
  (# a: .?; value: ##Object; undefined:< Exception;
  enter a do ... exit value##
  #);
Loc:
  (# a: .?; value: ##Object;
  enter a do ... exit value##
  #);
Kind:
  (# a: .?; value: @Integer;
     onDynRef:< (# a: .^; enter a do dynref->value; INNER #);
     onStatRef:< (# a: .@; enter a do statref->value; INNER #);
     ... (* Other possibilities. *)
  enter a
  do <<Call appropriate virtual>>
  exit value
  #);
Name:
  (# a: .?; value: ^Text;
  enter a do ... exit value[]
  #);
LocQua:
  (# a: ..; value: ##Object;
  enter a do ... exit value##
  #);
KindQua:
  (# a: ..; value: @Integer;
  enter a do ... exit value
  #);
AttrQua:
  (# a: ..; value: ##Object;
  enter a do ... exit value##
  #);
AttrCanonify:
  (# a: .?; value: .?;
  enter a do ... exit value #)
  #);
StatRefIsSimple:
  (# a: .@; value: @Boolean;
  enter a do ... exit value
  #);
```

Figure 6.11: Attribute reference introspection.

- `Loc` returns the location of the attribute referred by `a`.
- `Kind` returns the kind of the attribute referred by `a`.
- `Name` returns the name of the attribute referred by `a`.
- `LocQua` returns the location qualification of the attribute reference attribute referred by `a`.
- `KindQua` returns the kind qualification of the attribute reference attribute referred by `a`.
- `AttrQua` returns the attribute qualification of the attribute reference attribute referred by `a`.
- `AttrCanonify` returns the minimal attribute which is comparable to `a`. Attribute comparison was described in section 5.7. Thus, the value returned by `AttrCanonify` is an attribute corresponding to `a`, but with the location replaced by the ObjectDescriptor of the location of `a`.
- `StatRefIsSimple` returns whether `a` refers a simple attribute, i.e., an integer, shortint, real, char, or boolean.

Example usage:

```
       Person: (# address: ^Text #);
       att: .Person^Text;
       p: ##Object;
       kind: @Integer;
       name: ^Text;
       aPerson: @Person;
    do Person.address:->att;
       att->MLI.Qual->p##;      (* p## = Text##     *)
       att->MLI.Loc->p##;       (* p## = Person##   *)
       att->MLI.Kind->kind;     (* kind = dynref    *)
       att->MLI.Name->name[];   (* name = "address" *)
       att:->MLI.LocQua->p##;   (* extension(p##) = {current_object} *)
       att:->MLI.KindQua->kind; (* kind = dynref    *)
       att:->MLI.AttrQua->p##;  (* p## = Text##     *)
       aPerson.address:->MLI.AttrCanonify->att; (* att = Person.address: *)
```

6.3.1 External Representation of Attribute References

An external representation of attribute references is provided by the patterns `SaveAttributeRef` and `GetAttributeRef` shown in figure 6.12. As with `SavePatternRef`, the user of `SavAttributeRef` must supply a way to save object identities on the stream, or, to avoid object references, must canonify the attribute reference using `AttrCanonify`.

6.3.2 Summary

This section presented the introspectional interface to attribute references. Using this interface, meta-level code can get detailed information on the attribute referred by some attribute reference, and is able to exchange attribute references between separate processes.

```
SaveAttributeRef:
  (# saveObjectRef:< (# o: ^Object; enter o[] do INNER #);
     a: .?; s: ^Stream;
  enter (a,s[])
  do ...
  #);
GetAttributeRef:
  (# getObjectRef:< (# o: ^Object; do INNER exit o[] #);
     s: ^Stream; a: .?;
  enter s[]
  do ...
  exit a
  #);
```

Figure 6.12: `SaveAttrRef`: External representation of attribute references.

```
grammar:
  (# symbol:< (* Virtual pattern ! *)
       (# ...
       #);
  #);
gram: ^grammar;
sym: ^grammar.symbol;
do &grammar[]->gram[];
   &gram.symbol[]->sym[];
```

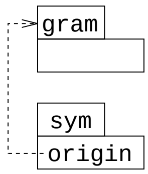


Figure 6.13: The origin problem of object copying.

6.4 Cloning Objects

An important goal of the MLI is to allow generic functionality such as object copying, persistence, and distribution to be implemented on top. However, general pattern nesting and creation time binding of origins and static references, (see section 4.1.3), complicates object copying considerably. The problems are illustrated below.

To see the problem imposed by pattern nesting, consider the example in figure 6.13, in which two objects are created. One object, `sym`, has an origin reference to the other object, `gram`. Now assume that generic object copying code is to make a copy of the object graph rooted in the symbol `sym`. I.e., since `sym` has an origin reference to `gram`, we would like to create a copy of the object graph containing the objects `sym` and `gram`.

A tempting but erroneous way to implement the creation of a copy of the `sym` object would be[2]:

```
&(sym[]->MLI.GetObjQual)[]->symcopy[]
```

Since `sym` has origin in `gram`, this would result in `symcopy` having origin in `gram`, and not in `gramcopy`. On the other hand, we cannot allow origin references to change once an object has been created. Doing so might

[2]Notice that if `sym[]` denoted a singular object, this expression would be equivalent to `sym[]->symcopy[]`, reflecting that the only object of exactly the same type as `sym[]` would be `sym[]` itself.

```
main: @
  (# g: ^grammar;
     g1: @grammar;
     g2: @grammar (# symbol:: (# ... #)#);
     gsym: (# s: @g.symbol #);
     gs,gscopy: ^gsym;
  do g1[]->g[]; &gsym[]->gs[]; g2[]->g[];
     (* Attempt to copy gs: WRONG!! *)
     &(gs[]->MLI.GetObjQual)[]->gscopy[];
  #);
```

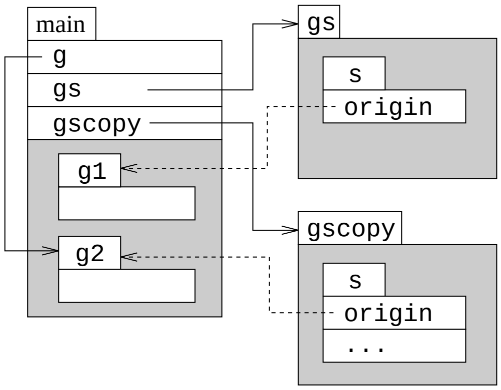


Figure 6.14: The static reference problem of object copying.

break qualification constraints on dynamic object references. This is illustrated by the tentative example below:

```
   g1,g2: @grammar;
   s1: ^g1.symbol;
do &g1.symbol[]->s1[];
   g2[]->s1.origin[]; (* ILLEGAL!! *)
```

In this example, an instance of `g1.symbol` is created and assigned to `s1`. The problem arises from the subsequent change of origin in the object referred by `s1`, which breaks the constraint

$$\textbf{ori}(\textbf{object}(\texttt{s1:}), \texttt{grammar.symbol\#\#}) = \textbf{object}(\texttt{g1:})$$

on `s1`. The notation $\textbf{ori}(o, od)$ denotes the origin of the object o with respect to the ObjectDescriptor od, and is introduced in appendix E.

In summary, we need a mechanism supporting the creation of an object clone, but with some control of the set of origins of the clone.

A similar problem arises from creation time binding of static object references. Consider the execution of the code in figure 6.14. First, while **object**(`main.g:`) = **object**(`main.g1:`), an instance **object**(`main.gs:`) of the `main.gsym##` pattern is created. Notice that `gsym##` has a part object of type `g.symbol##`. Then, the object reference `g:` is reassigned to refer the grammar **object**(`g2:`). Finally, an attempt is made to create a copy of **object**(`gs:`). However, as a result, **object**(`gscopy.s:`), supposed to become a a copy of **object**(`gs.s:`), in fact ends up being an instance of `g2.symbol##`, whereas **object**(`gs.s:`) is an instance of `g1.symbol##`. Notice that `g1.symbol##` has fever attributes than `g2.symbol##`.

Again, the problem is the lack of a mechanism supporting object creation, with control of the creation time binding of static attributes.

The `PendingObject` mechanism is introduced to handle this problem. A `PendingObject` corresponds to an object instance whose origin references and static references have not yet been fully bound. `PendingObject` splits the, usually atomic, operation of object creation into three steps:

1. Creation of a PendingObject `po` as an instance of some ObjectDescriptor `od##`.
2. Binding of origin references and static references inside `po`. However, if the static reference is qualified by a pattern qualification whose corresponding ObjectDescriptor `pod##` is uniquely determined at

creation time[3]. the static reference will be bound to a pending instance of `pod##` in step 1. Otherwise, the user of PendingObject is required to supply a feasible binding for the static reference.

3. Release of a reference to the completed object `o`.

The main reason for delaying release of `o` is that an object should be considered unsound until all origins and static references have been bound. For example, the code generated by the Mjølner BETA compiler assumes that origins and static references are never `NONE`. Furthermore, since the PendingObject abstraction is the only way to change bindings of origins and static references, it provides a natural way of ensuring that origins and static references are not changed once the completed object `o` has been released.

Origin references are bound in step 2, i.e., before the object itself is released. If the origin object has not yet been created, this demands the creation of the origin object before the pending object can be completed.

When all origins and static references in a pending object have been bound to complete objects, the object is complete[4], and a reference can be released, unless one of the attribute bindings does not comply to its qualification. In that case, the `QualError` exception will be raised.

The interface to the `PendingObject` mechanism is shown in figure 6.15. `CreatePendingObject` creates a new pending object as an instance of the ObjectDescriptor `od##`. To bind origins and static references inside a pending object, thereby completing the object, the `CheckStatOrOriRef` and `CheckStatRepRef` operations of the `PendingObject` pattern should be used.

In addition to the binding of origins and static references, the `&` operator binds repetition (array) attributes to repetitions of some default size. However, since BETA repetitions can be dynamically extended, the `PendingObject` abstraction is simplified by assuming that repetition attributes of objects created with `CreatePendingObject` are empty, i.e., have size 0. A special repetition case are repetitions of complex objects. These are handled by `CheckStatRepRef`.

6.4.1 Summary

This section described the `PendingObject` abstraction, offering a way to create instances of BETA ObjectDescriptors with detailed control of the binding of origins and static references. `PendingObject` splits the, usually atomic, operation of object creation into a sequence of steps, offering more control of creation time binding of origins and static references. Such an interface is needed to allow generic object copying mechanisms to be implemented on top of the MLI.

As with the rest of the MLI, an important part of the `PendingObject` abstraction is to ensure type-safety, i.e., taking care not to break assumptions made by the compiler.

6.5 Callstack Introspection

The final runtime entity to be reified is the concept of callstacks. In BETA, *component* objects are objects with their own callstack. Component objects can be used as coroutines or threads, depending on the way they are started.

Figure 6.16 shows an introspectional interface to execution stacks. The interface assumes that the callstack of some component is only introspected if the component is not currently executing, i.e., the component is

[3]This corresponds to the case where the size of the static part object is known at compile time. In this case, the Mjølner BETA compiler inlines the part objects in the containing object.

[4]Due to possible cycles in the object graph consisting of origins and static references, the object `o` is complete if it can be proven to be complete under the assumption that `o` is complete.

```
CreatePendingObject:
  (# asComponent:< BooleanValue; (* Create as component? *)
     od: ##Object; (* ObjectDescriptor *)
     po: ^PendingObject; (* New pending object. *)
  enter od## do ... exit po[]
  #);
PendingObject:
  (# CheckStatOrOriRef: (* Check StatRef or OriRef attribute. *)
      (# OnMissing:< (* Called if attr has not yet been bound to an object. *)
           (# bindToPending: (* Bind attr to pending object po. *)
                (# po: ^PendingObject enter po[] do ... #);
              bindToComplete: (* Bind attr to complete object o. *)
                (# o: ^Object enter o[] do ... #);
           do INNER
           #);
         OnPending:<  (* attr is bound to the pending object po. *)
           (# po: ^PendingObject; enter po[] do INNER #);
         OnComplete:< (* attr is bound to the complete object o. *)
           (# o: ^Object; enter o[] do INNER #);
         attr: .@; (* Canonic non-simple Stat- or OriRef. *)
      enter attr do <<Call feasible virtual>>
      #);
    CheckStatRepRef: (* Check repetition of complex objects. *)
      (# attr: .[]@;
         OnMissing:< (* Called if attr has not yet been bound. *)
           (# bindToPending: (* Bind to pending repetition of range [low..high]
                              * and element type od##. *)
                (# od: ##Object; low,high: @Integer
                enter (od##,low,high) do ...
                #);
           do INNER
           #);
         OnPending:< (* attr is bound to pending repetition. *)
           (# GetElm:
                (# index: @Integer; po: ^PendingObject;
                enter index do ... exit po[]
                #);
           do INNER
           #);
      enter attr do <<Call feasible virtual>>
      #);
    IsComponent: BooleanValue (# do .., #);
    GetObject: (* Get reference to the complete object. *)
      (# StillPending:< Exception;
         QualError:< Exception (# att: .? enter att do INNER #);
         OnComponent:< (# o: ^|Object; enter o[] do INNER #);
         OnObject:< (# o: ^Object; enter o[] do INNER #);
      do ... (* Calls one of the four virtuals. *)
      #);
  #)
```

Figure 6.15: PendingObject: An abstraction of incomplete objects.

```
ScanStack:
  (# comp: ^|Object; (* Component reference *)
     current: ^activationRecord;
  enter comp[]
  do <<Call INNER for each activation record>>
  #);
ActivationRecord:
  (# Object: (# o: ^Object do ... exit o[] #);
     Pattern: (# p: ##Object; do ... exit p## #);
     Imp: (# imp: ^MPS.node do ... exit imp[] #);
  #);
```

Figure 6.16: `ScanStack`: Scanning component stacks.

assumed to be in a suspended state. If a component is introspected, it will be descheduled until introspection is complete. As a result, a thread is not able to introspect its own callstack.

6.5.1 Scanning a Callstack

A callstack may be thought of as a sequence of activation records. In BETA, an activation record includes information on the currently executing object (remember that method activations are also object instances), the return address belonging to that activation record, and temporary register contents.

To support implementation of a cross-platform introspectory interface to callstacks, we shall use the notion of *busstops* [Jul 94]. A busstop corresponds to a code address where the execution of a component may be suspended, either because of a procedure call, or due to thread preemption. For each busstop, the compiler must be able to generate information to allow a callstack to be moved between different address spaces, possibly on different architectures.

The introspectory interface to callstacks is the `ScanStack` pattern shown in figure 6.16. Notice that `ScanStack` provides information on the currently executing object and the pattern whose dopart is currently executing. Both are needed to identify the active dopart.

To get more detailed information, a reference to the source code corresponding to the current busstop may be obtained. Such information is useful in order to implement a source-level debugger.

6.5.2 External Representation of Call Stacks

To allow callstacks to be saved on secondary storage, or exchanged between different address spaces, an external representation is needed. The interface to save the state of a callstack on a stream corresponds to the interfaces used to save pattern and attribute references on a stream, and is shown in figure 6.17.

6.5.3 Summary

This section introduced the reflectional interface to component stacks. This interface is needed to exchange stacks between different address spaces, implement in-memory transactions, and implement source-level debugging on top of the MLI.

```
SaveStack:
  (# saveObjectRef:< (# o: ^Object; enter o[] do INNER #);
     comp: ^|Object; s: ^Stream;
  enter (comp[],s[])
  do ...
  #);
RestoreStack:
  (# getObjectRef:< (# o: ^Object; do INNER exit o[] #);
     comp: ^|Object; s: ^Stream;
  enter (comp[],s[])
  do ...
  #);
```

Figure 6.17: `SaveStack`: External representation of callstacks.

6.6 Intercession

The capabilities of the meta-level interface described so far have been mainly introspectional. I.e., they allow programs to look into their current state and get information on a number of aspects.

Intercessory capabilities, on the other hand, make it possible to reach in and change the behaviour of certain parts of the system. The implementation of intercessory capabilities is inherently more difficult than the implementation of introspectional capabilities, since the implementation of intercession immediately forces tradeoffs between generality and efficiency. While implementation of introspection is mostly a matter of maintaining an adequate amount of runtime type information, intercession may mean interception of variable access and method calls in a general yet efficient way.

This section introduces meta-level interfaces that allow meta-level code to redefine what happens when instances of some qualification are executed, when assigning or reading the value of simple object attributes, when objects are created, and when objects are destroyed.

6.6.1 Method Call

In BETA, method objects, ("activation records"), only differ from other objects in the way they are used. To understand the mechanism of method call, assume the following code:

```
foo:
  (# a,b,c,d: @Integer;
  enter (a,b)
  do a->d; b->c;
  exit (c,d)
  #);
x,y: @Integer;
do (5,10)->&foo->(x,y);
```

The method call `(5,10)->&foo->(x,y)` is executed in the following steps:

1. Allocate an instance[5] `m` of the pattern `foo`. If `foo` had been a virtual pattern, the allocation step would

[5] It is possible to execute an already existing object. The only difference in that case is that the allocation step is skipped.

```
ExecReflector:
  (# methodType:< Object; (* Qualification of objects reflected upon. *)
     method: ^methodType; (* The current method object.              *)
     when: ##Object;      (* The current dopart.                      *)
     continue: (# do ... #); (* Continue execution of method.        *)
  enter (method[],when##)
  do INNER
  #);
AddExecReflector:
  (# who: ##Object;    (* Qualification of objects to reflect upon. *)
     when: ##Object;   (* When to reflect.                          *)
     reflector: ##ExecReflector; (* The reflector.                  *)
     reflId: @Integer; (* Identity of the new association.          *)
  enter (who##,when##,reflector##)
  do ...
  exit reflId
  #);
RemoveExecReflector:
  (# reflId: @Integer;
  enter reflId
  do ...
  #);
```

Figure 6.18: `ExecReflector`: Intercepting method calls.

include a table lookup to retrieve the exact type of `foo`.

2. If an actual input list (`(5,10)` above) is specified, evaluate it and assign the resulting list of values to the enter list of `m`. In the example, the values 5 and 10 are assigned to `m.a` and `m.b`.

3. Execute the dopart of `m`.

4. If an actual output list (`(x,y)` above) is specified, assign the result of evaluating the formal exit list (`(c,d)` above) of `m`, to the result of evaluating the actual outputlist. In the example, the values 10 and 5 are assigned to `x` and `y`.

The `ExecReflector` mechanism described below allows interception of step 3. An `ExecReflector` may be considered a *meta-pattern*, monitoring execution of other patterns. The super pattern of all reflectors is the `ExecReflector` pattern, shown in figure 6.18.

A programmer wanting to redefine method dispatch for some pattern may associate that pattern with a reflector by calling `AddExecReflector`, also shown in figure 6.18. The `who` parameter to `AddExecReflector` is the qualification of objects to reflect upon. For example, assume that we want to monitor calls to the `Multiply` method of `Calculator` instances:

```
Calculator:
  (# Multiply:
       (# a,b: @Integer; c: @Integer;
       enter (a,b)
       do a*b->c
       exit c
       #);
  #);
```

To monitor calls to the Multiply method of the Calculator pattern, we need to define a subpattern of ExecReflector, and associate the new reflector pattern with the qualification Calculator.Multiply##:

```
  MultReflector: MLI.ExecReflector
    (# methodType:: Calculator.Multiply;
    do '%d+%d='->screen.putformat (# do method.a->d; method.b->d #);
       continue; (* Continue execution of the reflected method. *)
       '%d\n'->screen.putformat (# do method.c->d #);
    #);
  multId: @Integer;
do (Calculator.Multiply##,Calculator.Multiply##,MultReflector##)
     ->MLI.AddExecReflector
     ->multId;
```

As a result, execution of objects m qualified by Calculator.Multiply## is redirected into the execution of an instance of MultReflector, which prints the arguments to Multiply, executes the multiplication, and finally prints the result.

The ExecReflector mechanism owes much flexibility to the possibility of giving any qualification as the who parameter to AddExecReflector. For example, if only Multiply calls to a specific calculator instance aCalc should be monitored, aCalc.Multiply## could be used as the who parameter to AddExecReflector. Or, if execution of a specific object should be monitored, a full object qualification could be used.

If multiple reflectors are associated with the same pattern, they are called in the opposite order of the order in which they were associated with the pattern, thus allowing a chain of reflectors to be called before execution of the method object itself.

Reflection may be turned off by calling RemoveExecReflector with the reflId returned from the call to AddExecReflector.

As the name of the when parameter to AddExecReflector suggests, its purpose is to describe *when* to reflect, whereas the who parameter describes the objects whose execution should be reflected. The reason why both are needed, is the INNER mechanism. Consider the following example:

```
foo: (# do 'foo '->screen.puttext; INNER #);
bar: foo (# do 'bar '->screen.puttext #);
barReflector: MLI.ExecReflector
  (#
  do 'barReflector '->screen.puttext; continue
  #);
reflId: @Integer;
do (bar##,foo##,barReflector##)->MLI.AddExecReflector->reflId;
&bar;
reflId->MLI.RemoveExecReflector;
screen.newline;
(bar##,bar##,barReflector##)->MLI.AddExecReflector->reflId;
&bar;
reflId->MLI.RemoveExecReflector;
```

Running this example would print the following lines:

```
barReflector foo bar
foo barReflector bar
```

It follows that the `when` parameter points out the dopart at the beginning of which reflection should take place. For this to be meaningful, the following relationship must hold between the `when` and `who` parameters to `AddExecReflector`:

```
who## <= when##
```

`who##` may denote a full object qualification, in which case only that particular object is reflected upon. However, `when##` as a full object qualification only makes sense iff the object reflected upon is singular, i.e., it is the only object with its particular type. For example, to specify reflection on the most specific dopart of any object, the following call to `AddExecReflector` can be used:

```
o: ^Object;
GenericReflector: MLI.ExecReflector
  (#
  do ...; continue; ...
  #);
do (o##,o[]->MLI.GetObjQual,GenericReflector##)
  ->MLI.AddExecReflector
  ->reflId;
```

since `MLI.GetObjQual` only returns a full object qualification if `o` is really a singular object.

The ability to reflect on the execution of several doparts of the same object is useful to implement a general object distribution mechanism in BETA, as described in [Brandt 94]. The reason is that the execution of `INNER` could result in the need to move the executing object itself.

6.6.2 Variable Access

The previous section described a meta-level interface for altering the execution of specific doparts. This section describes an interface for altering access to simple variables. In this section, variables of type integer, char, boolean, real and short are considered simple as BETA maps these types directly to simple types of the underlying machine. Furthermore, access to dynamic object reference attributes may be reflected upon. By

```
AccessReflector:
  (# locQua:< Object; (* Location qual of attributes reflected upon.*)
     onReadDynRef:<
       (# attrQua:< Object;
          o: ^locQua; att: .locQua^attrQua; value: ^attrQua
       enter (o[],att) do INNER
       exit value[]
       #);
     onWriteDynRef:<
       (# attrQua:< Object;
  o: ^locQua; att: .locQua^attQua; value: ^attrQua
       enter (o[],att,value[]) do INNER
       #);
     onReadInt:<
       (# o: ^locQua; att: .locQua@Integer; value: @Integer
       enter (o[],att) do INNER
       exit value
       #);
     onWriteInt:<
       (# o: ^locQua; att: .locQua@Integer; value: @Integer
       enter (o[],att,value) do INNER
       #);
     (* Read/Write virtuals for other simple attributes. *)
  #);
AddAccessReflector:
  (# att: .?; (* Attribute on which to reflect.     *)
     reflector: ##AccessReflector; (* The reflector *)
     reflId: @Integer
  enter (att,reflector##)
  do ...
  exit reflId
  #);
RemoveAccessReflector:
  (# reflId: @Integer
  enter reflId
  do ...
  #);
```

Figure 6.19: `AccessReflector`: Intercepting simple attribute access.

associating a simple attribute with an `AccessReflector` shown in figure 6.19, meta level code may change what happens when a simple attribute is read or written.

As an example, assume that we wanted to count the number of read and write accesses to the `value` attribute in instances of the `Counter pattern`. Using an `AccessReflector`, this may be achieved as follows:

```
    Counter: (# value: @Integer #);
    trackCounter: @
      (# readCount,writeCount: @Integer;
         countAccessReflector: MLI.AccessReflector
           (# locQua:: Counter;
              onReadInt::
                (#
                do o.(att)->value; (*!*)
                   readCount+1->readCount;
                #);
              onWriteInt::
                (#
                do value->o.(att); (*!*)
                   writeCount+1->writeCount;
                #);
           #);
         init:
           (#
           do (Counter.value:,countAccessReflector##)
                ->MLI.AddAccessReflector;
           #);
     #);
  do trackCounter.init;
```

Notice that the qualifications in this example ensure that the assignments marked `(*!*)` do not need any extra runtime type checking, since the qualifications on object and attributes references are sufficient.

6.6.3 Object Allocation

This section describes a meta-level interface that allows the programmer to change the default behaviour of object allocation. The `AllocReflector` pattern shown in figure 6.20 is the super pattern of all meta-patterns monitoring object allocation. Only allocation of autonomous objects can be monitored. Thus, the allocation of an object as part of another object does not trigger the `AllocReflector` mechanism.

For example, the code below initializes reflection on the allocation of `Person` objects, such that allocation of a `Person` is replaced by the allocation of a `specialPerson`:

```
AllocReflector:
  (# objType:< Object;   (* Qual of objects reflected by this reflector. *)
     newqual: ##objType; (* The actual qualification to instantiate.     *)
     new: ^objType;      (* The new object.                              *)
     default:            (* Default value for new.                       *)
       (# new: ^objType do ... exit new[] #);
  enter newqual##
  do INNER
  exit new[]
  #);
AddAllocReflector:
  (# reflected: ##Object; (* Qualifications to reflect upon.             *)
     reflectOnSubs: @Boolean; (* Reflect on subpatterns of reflected## ? *)
     reflector: ##AllocReflector; (* The reflector.                      *)
     reflId: @Integer;
  enter (reflected##,reflectOnSubs,reflector##)
  do ...
  exit reflId
  #);
RemoveAllocReflector:
  (# reflId: @Integer
  enter reflId
  do ...
  #);
```

Figure 6.20: `AllocReflector`: Intercepting object allocation.

```
     SpecialPerson: Person (# ... #);
     PersonAllocReflector: MLI.AllocReflector
       (# objType:: Person;
       do &SpecialPerson[]->new[]
       #);
     reflId: @Integer;
     aPerson: ^Person;
  do (Person##,FALSE,PersonAllocReflector##)
       ->MLI.AddAllocReflector
       ->reflId;
     &Person[]->aPerson[];
     (* aPerson is now a SpecialPerson ! *)
```

The `reflectOnSubs` parameter to `AddAllocReflector` determines whether only allocation of direct instances of `reflected##` should trigger the `AllocReflector`, or whether allocation of subpatterns of `reflected##` should trigger as well.

The `newqual` enter parameter to `AllocReflector` is the qualification which was the parameter to the `&` operator. Thus, in the example above, the value of the `newqual` parameter given to `PersonAllocReflector` was `Person##`. `newqual` is needed to ensure that the object allocated is actually an instance of `newqual##`, or of some subpattern of `newqual##`.

Finally, the `AllocReflector.default` pattern may be used by subpatterns of `AllocReflector` to retrieve the default value of `new`. Notice that in the event of a chain of AllocReflectors, this default value may be

```
GCReflector:
  (# reflId: @Integer; (* Identity of association. *)
  enter reflID
  do INNER
  #);
AddGCReflector:
  (# o: ^Object; (* Object to trace.           *)
     reflector: ##GCReflector; (* The reflector.   *)
     reflId: @Integer; (* Identity of association. *)
  enter (o[],reflector##)
  do ...
  exit reflId
  #);
RemoveGCReflector:
  (# reflId: @Integer; (* Identity of association. *)
  enter reflId
  do ...
  #);
```

Figure 6.21: `GCReflector`: Monitoring object destruction.

different from a new instance of `newqual##`.

6.6.4 Garbage Collection

BETA is garbage collected, so objects are never explicitly destroyed. The MLI allows programs to trace garbage collection of specific objects, but does not allow garbage collection of an object to be prevented, since this would put unreasonable demands on the garbage collector implementation. Likewise, the meta-level interface does not allow GC-tracing of instances of arbitrary qualifications, since this would be hard to implement with a garbage collector of the copying type. These demands leads to the meta-level interface shown in figure 6.21. For example, the following code initiates monitoring of garbage collection of a specific `Person` object:

```
   PersonGCReflector: MLI.GCReflector
     (#
     do 'Person %d garbage collected\n'->screen.putformat
        (# do reflId->d #);
     #);
   aPerson: ^Person;
   reflId: @Integer;
do &Person[]->aPerson[];
   (aPerson[],PersonGCReflector##)
     ->MLI.AddGCReflector
     ->reflId;
```

6.7 Summary

This section has described the BETA meta-level interface, MLI. The MLI provides introspectory, invocational, and intercessory capabilities. Introspectory capabilities allow a running BETA program to obtain detailed meta-level information on runtime entities such as objects, patterns, attributes and stacks, and to exchange these entities between separate processes. The `PendingObject` abstraction provided a flexible mechanism for the creation of new objects. `PendingObject` is an invocational mechanism, allowing access to parts of the runtime system that are usually hidden behind the, usually atomic, object creation primitive. Intercessory capabilities allows reflection on object creation and destruction (garbage collection), method dispatch, and remote attribute access.

The main goal of the MLI is to allow generic, i.e., type-independent, code, such as persistence, distribution, and others to be implemented in the BETA language itself, thereby providing portability and generality. The MLI builds on top of and relies heavily on the language concepts of pattern (qualification) references and attribute references. The concept of pattern references was generalized in chapter 4. The concept of attribute references was described in chapter 5.

The current design of the MLI, as described in this chapter, is a pure paper design, and is therefore likely to change during implementation and use. The next section outlines a few examples of how the MLI may be used, whereas chapter 8 outlines a meta-level architecture to support the implementation of the meta-level interface.

Chapter 7

Example Applications

This chapter briefly describes a number of usages of the meta-level interface described in the previous chapter.

7.1 Copying Objects

The first example usage of the MLI is the implementation of a generic object copying pattern. Two different copying algorithms are shown. First, a generic "shallow copy" is implemented, corresponding to the Eiffel `standard_copy` method [Meyer 92]. Second, a tailorable copying algorithm allowing the user of the algorithm to decide what dynamic reference attributes should be followed. Default for this second algorithm is a deep copy of the object graph reachable from the root object entered. I.e., default corresponds to Eiffel `deep_copy`.

7.1.1 Shallow Copying

An implementation of shallow object copying implementation is shown in appendix C, as the `CopyShallow` pattern. It follows from this example that even a shallow copy in block-structured and object oriented languages is in the worst case a pretty complicated matter. This is mostly due to static part objects and because dynamic references may refer parts that have already been copied.

`CopyShallow` enters an object reference `obj`, and creates a copy of every object connected to `obj` via static references. More precisely, `CopyShallow` creates a pending copy of each object in the static object cluster $\langle \texttt{obj} \rangle$ (see appendix B). The copying runs in a number of steps:

1. Create pending copies of all objects in $\langle \texttt{obj} \rangle$. This is done by starting in the root object of $\langle \texttt{obj} \rangle$, $\langle \texttt{obj} \rangle_{root}$, and then create pending copies of all objects reachable through static references. At the same time, these static references are bound in the pending copies. Static references with full object qualifications are not followed, since the objects referred by these are not part of $\langle \texttt{obj} \rangle$.

2. Bind origin references in the pending copies. Since it is now known that all the pending copies have been created, it is also known that an origin reference can be bound to either an object that is not copied, or to a pending copy that has already been created. The same goes for the constant references that were not followed during step 1.

3. Since all static references in pending copies have been bound, the real objects can be released.

4. Copy remaining attributes. Dynamic reference attributes are copied while ensuring that references to copied objects are updated to refer the copy. I.e., the invariant enforced by the algorithm is that

```
do main[]->CopyShallow->maincopy[];
```

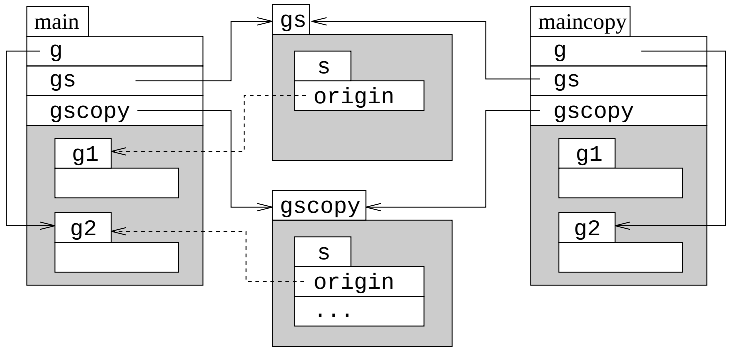


Figure 7.1: The result of shallow object copying of the `main` object from figure 6.14 on page 57.

references that refer the same object in the original must refer the same object in the copy, even if this means updating dynamic references.

For example, in the object graph in figure 6.14 on page 57, we may want to create a shallow copy of the `main` object. The code to do this, and the result of the shallow copy, is shown in figure 7.1. Notice that since `g` and `g2` refer the same object in `main`, they refer the same object in `maincopy`. Remember that solid arrows symbolize dynamic object references, dotted arrows symbolize origin references, and embedding of objects within objects symbolize object location and static references.

For simplicity, `CopyShallow` does not consider the copying of all attribute kinds. The following attribute kinds have been left out:

- Dynamic pattern references and attribute references. Notice that since pattern and attribute reference values depend on references to objects that may have been copied, the values of pattern and attribute references in the copy may have to be updated in the copying process. This is not done by `CopyShallow`, but may be achieved by writing pattern and attribute references to an external representation using `MLI.SavePatternRef` and `MLI.SaveAttributeRef`, and then reading them back using `MLI.GetPatternRef` and `MLI.GetAttributeRef`, while ensuring that references to copied objects are replaced by references to the copies.
- Static repetitions of complex objects. These must be handled by step 1.
- Component objects. Component stacks could be copied using the same technique as was outlined above for dynamic pattern and attribute references.
- Implicit attributes apart from origins. Logically, these attributes only exist during execution of the object. Since `CopyShallow` does not take the execution state of objects into consideration, there is therefore no need to copy implicit attributes, apart from origin references.

7.1.2 Tailorable Copying

The tailorable copying algorithm implemented by the `CopyDeep` pattern shown in appendix D is an extension to the shallow copying algorithm, allowing the programmer to control the set of reachable objects that should be copied. Default is to copy the full transitive closure of the object given as parameter.

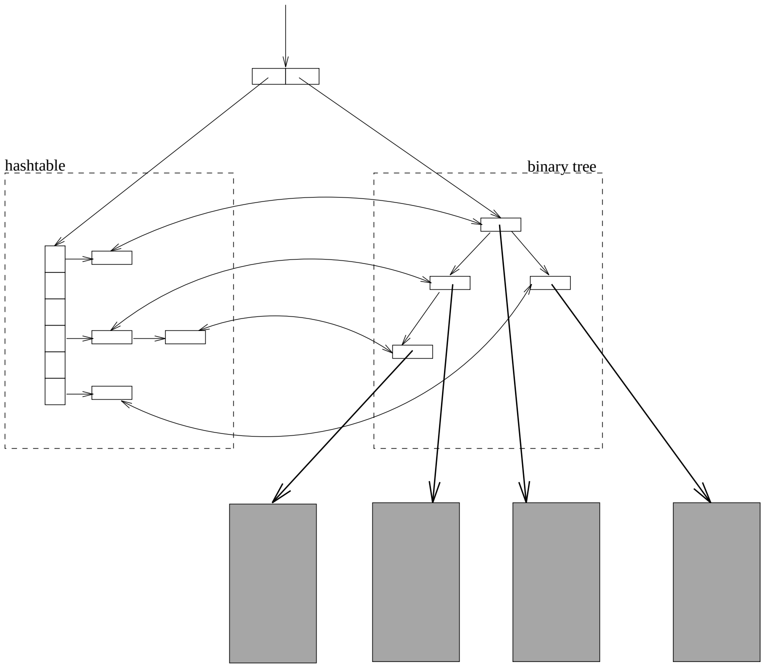


Figure 7.2: Dual datastructure for efficient multiway lookup

`CopyDeep` only differs from `CopyShallow` through the implementation of the `FindCopiedSet` method, and only code differing from `CopyShallow` is therefore shown in appendix D. `FindCopiedSet` is the method which determines the set of objects to be copied.

`CopyDeep` provides two different ways of controlling the set of objects copied:

- Based on object type. By further binding the `SkipObject` virtual, CopyDeep can be made to ignore objects qualifying to some set of qualifications.
- Dynamic object references can be skipped. By further binding the `SkipDynRef` virtual, CopyDeep can be made to skip specific object references, i.e., not copy the object referred.

Note that if it is decided to copy an object o, all of $\langle o \rangle$ is copied, since objects in $\langle o \rangle$ conceptually belong together. This is an inherent part of the BETA conceptual framework and semantics, and cannot be tailored.

For one example usage of the tailorable object copying algorithm, consider a dual data structure for efficient multiway lookup as illustrated in figure 7.2. Such a data-structure may, e.g., be useful in order to provide fast access to a sorted collection of data. The tree maintains the ordering, while the hastable allows fast access. Assume that for some reason we want to make a copy of the datastructure without copying the huge data indexed by the structure. For example, separate processes may need a private copy of the search-structure, but without copying the data indexed by the structure. If the contents of the table were known to be instances of the pattern `blob`, we may furtherbind `SkipObject` to limit the part of the object graph copied as follows:

```
multiway_table:
  (# elementType:< Object;
     tree_node:
       (# hash: ^hash_node;
          left, right: ^tree_node;
          data: ^elementType;
       #);
     ...
  #);
blob_table: multiway_table (# elementType:: blob #);
table, copy: ^blob_table;
do ...;
table[]->CopyDeep
(# SkipObject:: (# do (obj##<=blob##)->value #);
#)->copy[];
```

On the other hand, if the table contents were heterogeneous or even unknown, it would be more convenient or even necessary to express the limitation using attribute references:

```
   table, copy: ^multiway_table;
do ...;
   table[]->CopyDeep
   (# SkipDynRef:: (# do (attr<=multiway_table.tree_node.data:)->value #);
   #)->copy[];
```

7.1.3 Remarks on the Copying Algorithms

One of the things that can be learned from the implementation of the copying algorithms, is that object copying in a block structured language like BETA is a complicated process. However, the mere fact that it is possible to describe such an algorithm indicates that attribute references and the meta-level interface described in this report are worthwhile. Without them it would not even have been possible to implement such algorithms in a safe way, let alone parameterizing the deep copy algorithm. The complexity of these algorithms also suggest that they do not belong in the language runtime system, as is tempting with respect to optimization. The trouble is that the complex object relationships present in BETA means that even defining a simple thing as “shallow copy” is not straightforward, and thus a number of alternative choices exist.

Something akin to the tailorable object copying algorithm has already been implemented in the *ObjectSerializer* described in [Brandt 94], and used as a building block for persistence and distribution in BETA. However, first of all it is implemented in an unsafe, low-level way requiring detailed knowledge of the object layout, garbage collector and runtime system in general. Secondly it is not possible to express parameterization corresponding to that of `SkipDynRef`.

Most of the examples in the following sections are very akin to object copying algorithms, but hints at the large number of useful applications of attribute references and the MLI.

7.2 Persistence

Using the MLI described in this report, it would be possible to rewrite the current implementation of BETA persistence in a safe way. Since saving objects on secondary storage and later retrieval may be seen as a split of the object copying process into two, many of the issues involved have already been described in the previous section on object copying.

An example of an implementation of persistence on top of a MLI may be found in [Paepcke 88], describing an implementation of persistence in CLOS.

However, a persistence implementation purely on top of the meta-level interface is probably not able to achieve performance comparable to that of hardware assisted persistence such as that of the Texas persistent store [V. Singhal & Wilson 92]. But even in such a system, the MLI would be useful in providing a flexible persistence mechanism, since attribute references and the MLI provide the expressive power needed to describe the necessary interfaces. For example, the ability to speak of attributes is useful in order to express clustering policies, implement schema-evolution tools, object-browsers, export and import persistent objects from external database formats and so on.

7.3 Object Movement and Attachment

The distributed programming language Emerald [Black et al. 87] includes the `attach` keyword used to specify what objects should move along when an object is moved between network hosts. In Emerald, attachment is part of the type declaration, and it is therefore impossible to change attachment dynamically. In contrast, by using attribute references, attachment may be changed on the fly. To see how, assume the existence of a `RegisterAttachedAttribute` pattern used to register attributes that should be attached:

```
RegisterAttachedAttribute: (# a: .^ enter a do ... #)
```

Using `RegisterAttachedAttribute`, attribute attachment may be expressed as follows:

```
   foo: (# criticalData: ^bar ... #);
do foo.criticalData:->RegisterAttachedAttribute;
```

The effect of this code would be to ensure that whenever a foo object is moved between different network hosts, the object referred by the `criticalData` attribute is moved along.

In general it can be argued that optimizations such as the Emerald `attach` keyword does not belong in the class-descriptions. Class-descriptions are supposed to describe the application area and not optimization details concerning the application program itself. Optimization issues may change between platforms, but the classes describing the application domain does not. Therefore, it is better to keep optimization issues as separate "meta-level modules". The concept of attribute references and the meta-level interface support such separation of concerns. And, even more importantly, allows such optimizations to be implemented without changing the compiler. Thus, the language itself has become extensible and tailorable.

7.4 General Distribution Implementation

As outlined in [Brandt 94], the implementation of a fully general distributed object system for BETA involves the movement of method objects at different points in time. In some situations, object movement should occur when the `INNER` imperative is executed. Using intercession as described in section 6.6, the needed functionality may be implemented. However, this functionality depends on the ability to scan implicit attributes such as index variables, inserted items, and labelled imperative attributes, whose existence depends on the current execution state of the object. A better understanding of how to handle such attributes is still needed, as mentioned in section 6.2.3.

7.5 Constraints

Constraint programming is a declarative programming paradigm allowing the programmer to declare constraints on program variables to be maintained automatically by the constraint system. Examples of constraint programming languages are Kaleidoscope [Lopez et al.] and Siri [Horn 92] and the CLP family of languages [J. Jaffar et al. 92].

A constraint framework, Gleipner, has been implemented for the BETA programming language [Knudsen & Middelfart 94]. To use the Gleipner framework to constrain, e.g., integer values, the programmer must use a special integer class, `cInteger`. This means that attributes of pre-existing patterns cannot be constrained using Gleipner. Using attribute references and the introspective and intercessory facilities of the BETA MLI, a version of Gleipner could be implemented to support constraints on attributes of arbitrary objects.

7.6 Conclusion

This chapter has described a few potential uses of the BETA meta-level interface described in this report. However, what is important to notice is not the particular examples, but their common nature: They are all examples of functionality that has traditionally been an inherent part of a programming language design. In a programming language with a meta-level interface, these functionalities can be implemented in the language itself, keeping the basic language implementation small and elegant while allowing advanced programmers to add generic functionality. I.e., a programmer may write code *for* the language, in addition to writing code *in* the language.

To a large extend, a language meta-level interface may be compared to the interface of an operating system micro-kernel. In both cases, the aim is to define a minimal interface on top of which virtually every other functionality may be implemented. The downside is that operating system micro-kernels are still to prove that they can deliver all this functionality as efficiently as monolithic kernels, the latter having the advantage of more tight integration. The same prblem applies to language meta-level interfaces, in the sense that even if all functionality may logically be implemented on top, the performance of the resulting system may be poor.

However, even though it is likely that certain functionalities, e.g., distribution and persistence, may need more direct support from the runtime system than that provided by a general-purpose language MLI, such an MLI will still be useful due to the provided increase in expressive power. That is, in addition to providing the means to *implement* generic functionality, the language meta-level interface provides the expressive power to *describe* the interfaces to such functionality, which may often be independent of the choice of implementation technique. For example, concerning persistence, such examples were described in section 7.2. Likewise, the example of object attachment described in section 7.3 is independent on the underlying implementation of the basic distribution primitives.

Chapter 8

Implementation Outline

In this report, implementation issues have been widely ignored. Now it is time to consider how we may implement the MLI. A programming environment allowing the implementation of reflection and more is outlined below.

A number of relevant questions to be answered are listed below:

1. What kind of runtime environment is needed to support the MLI described?
2. How do we minimize or avoid the overhead paid by non-reflectional programs?
3. What generality is wanted and needed in an implementation?
4. What are the part technologies involved?

In the following sections, I briefly outline answers to these questions, while postponing the exact details of the implementation to future work. Implementation work is under progress, and the first results have been reported elsewhere [Brandt & Schmidt 95].

8.1 Existing Programming Environments

Existing programming systems supporting reflection include SELF [Sun 93] CLOS [Kiczales et al. 91] and OpenC++ [Chiba 93].

SELF and CLOS are fully dynamic and include general and flexible reflectional capabilities, while OpenC++ is more restricted.

The runtime environment outlined in the next section have goals in between CLOS/SELF and OpenC++. It will be less flexible than CLOS/SELF when it comes to exploratory capabilities. On the other hand it will (hopefully) be a lot more flexible and general than OpenC++. As a consequence, efficiency is likely to end up in between. Furthermore, a specific goal not induced by the implementation of the MLI, but rather of the intended uses of the MLI, is a distributed programming environment supporting general object persistence and distribution. This goal was outlined in [Brandt 94].

In the case of dynamically typed languages such as SELF and CLOS, there is usually no distinction between the programming environment and the runtime environment. For the implementation of introspection, this means that nothing special must be done to ensure that all available information is also present at runtime.

Since it is there for the compiler, it is there for the reflectional parts of a running program. Furthermore, the compiler or interpreter is present at runtime and may be necessary in order to implement reflection.

The price of intercessory capabilities, e.g., redefining what happens when some object receives a specific message, of course differs between languages. In CLOS and SELF, message dispatch is inherently slower[1] than in most statically typed languages, and so the extra overhead added by reflection is relatively smaller. Since it is easier to accept a small relative overhead than it is to accept a large relative overhead, the result is more general reflectional capabilites in these languages than in OpenC++. In the SELF environment, incremental optimizations at "runtime" may even be used to totally remove overhead. We shall se how a corresponding idea may be applied to implement reflection in BETA with no overhead for non-reflectional programs.

In OpenC++, reflectional capabilities are restricted to what is explicitly pointed out in the class descriptions. This is used by the compiler to insert extra code on method calls that may be reflected upon. The problem with this approach is primarily one of orthogonality and flexibility.

8.2 A Distributed Programming Environment

The basic idea is to build the programming environment around a distributed, possibly replicated object oriented database. Basically, the oodb will replace the traditional file system, and thus contain source code and objects derived from the source code. Thus, objects managed by the oodb include:

1. Source code, proabably in the shape of abstract syntax trees (AST's).
2. Checked AST's, i.e., source code that has been analyzed and annotated by the compiler.
3. Binary code objects.
4. Version information.
5. Different kinds of information that may be needed at runtime by some programs, but may be unneccesary for other programs. This includes information enabling reflectional capabilities.
6. Debugging information.

By replacing the passive file system by an active agent (the oodb), a number of problems are solved, and several new possibilities arise. Some of these are listed below:

1. All system objects are automatically assigned unique object id's (OID's), thereby solving problems of referential integrity described in [Brandt 94]. This is paramount for the secure implementation of persistence and distribution.
2. A process needing some kind of derived object may request it from the oodb. This eases the secure implementation of fine-grained dynamic link. Furthermore, information that would otherwise have to be linked into all programs may be dynamically fetched from the oodb on demand. As already noted, this includes information needed to implement the MLI.
3. The presence of an oodb enables replication of immutable objects. In large distributed environments this may lead to considerable performance gains.

[1] Although type inference [Agesen & Ungar 94] and incremental optimization techniques [Hölzle & Ungar 94] limits the cost considerably.

4. Derived objects may be created on demand. For example, binary code requested from the oodb may be created by asking a compiler to create it from source code already present in the oodb. This could happen if a persistent object is to be accesssed from a new platform for the first time, thus requiring the binary code of the accessed object to be linked in. Notice that the process needing the code does not even have to be aware of the demand-driven code generation, except for the delay when obtaining the code.

5. Different versions of the same code may be available for different purposes. For example, a non-reflectional program may be dynamically linked with a version of binary code not supporting reflection. If the need for reflection arises, it would be possible to exchange the non-reflectional version with a reflection-prepared version. Thus, a program not using reflection would not have to pay the price in neither time nor space.

6. Tight integration between a version management tool and the rest of the programming environment eases the implementation of versioning and configuration management considerably.

8.3 Enabling Technologies

First of all an object oriented database is clearly needed. Such an oodb is already available in the Mjølner BETA System. Since the oodb was originally designed to support cooperative applications, a number of the features supported are directly relevant for the implementation of a distributed programming environment. Concerning efficiency, the current oodb may have difficulties fulfilling the demands of an integrated distributed runtime environment. To address this problem, work is being done to provide more efficient persistence and locking mechanisms.

Next, a distribution library is needed. The distribution library described in [Brandt 94, Brandt 93] is general enough for a prototype distributed programming environment. It is, for example, already used to implement the oodb.

Finally, a very important component is a fine grained dynamic linker. The BETA system already includes a prototype of a callable dynamic linker and loader named Quark[2]. Quark supports dynamic link at the level of single object files. A more fine-grained link would be nice, but for a beginning, single object files will do. Quark is not able to remove or replace code that was previously linked in dynamically, and extension in this direction is probably needed. This is by no means trivial[3], and is an interesting project in its own right. Work is being done on this subject.

[2] Created by Bjorn Freeman-Benson.

[3] Consider the replacement of active code.

Chapter 9

Related and Future Work

9.1 Related Work

CLOS [Kiczales et al. 91] includes a comprehensive meta-level interface, (the CLOS MOP, or metaobject protocol), which allows the meta-level programmer to inspect and change several primitives of the basic programming language, including redefiniton of slot access, multiple inheritance semantics, and replacement of meta-classes. Much functionality of the BETA MLI was inspired by the functionality provided by the CLOS MOP. However, BETA is statically typed, whereas CLOS is dynamically typed, leading to a considerably different style of meta-level interface. Most notably, the BETA MLI allows, but does not force, the meta-level programmer to express static typing constraints, thereby leading to more readable and possibly more efficient meta-level code.

ABCL/R [Masuhara et al. 92], AL-1/D [Okamura et al. 93], and Self [Hölzle & Ungar 94] are further examples of dynamically typed programming languages with reflectional capabilities. As with CLOS, the most notable difference between the meta-level interfaces of these languages, as compared to the BETA MLI, is the issue of static versus dynamic typing. Dynamic typing means that method lookup in ABCL/R, AL-1/D, and Self is basically implemented by a runtime table-lookup. In BETA, static type information allows procedure calls to be implemented either by direct subroutine calls, or by using a single-level indirection jump through the virtual table of the target object. In both cases, the destination address is resolved at link time. In order to retain efficiency, the implementation of the BETA MLI will therefore use dynamic replacement of code objects and dynamic linking techniques.

OpenC++'s [Chiba 93] concept of *reflect methods* was the direct inspiration for the `reflector` pattern of the BETA MLI. However, OpenC++ does not have introspective capabilities, and reflect methods must be known at compile time. The BETA design aims at larger orthogonality, i.e., it allows reflection on patterns that were written without reflection in mind.

The C++ concept of *pointers to members* may be considered a restricted version of attribute references. However, the absence of runtime type information and restriction to function members, renders C++ pointers to members considerably less useful than attribute references.

Octopus [Farkas & Dearle 93] provides a reflective language mechanism for generic manipulation of complex object structures. Like the BETA MLI, the Octopus mechanism enforces static typing constraints, but does not allow the meta-level programmer to express such constraints. Concerning the type of functionality provided, an octopus is similar to the `PendingObject` abstraction described in section 6.4. Octopus's provides only limited intercessory capabilities, through replacement of function pointers in singular objects.

The Mjølner BETA Meta Programming System (MPS) [Madsen & Nørgård 93], is a grammar based library for managing abstract syntax trees (AST's). Virtually all tools of the Mjølner BETA System (MBS) is built on top of MPS. Here, "meta" stems from the primary usage of MPS, namely as a foundation for programming tools such as the compiler, the syntax directed editor, the debugger and so on. I.e., MPS is used for writing programs that manipulate programs.

In the introduction, a reflectional system was defined to include "an internal representation of itself". The internal representation should be causally connected to the current state of the reflectional system. In MPS, a program works on a representation of another program, and is therefore not reflectional. Even if the compiler compiles itself, the AST manipulated is not causally connected to the process doing the compilation. So, MPS does not turn BETA into a reflectional programming system.

9.2 Future Work

Clearly, future work includes the design and implementation of a distributed runtime environment allowing a practical implementation of reflection. Initially the plan is to make a prototype implementation feasible for experimenting with reflection, persistence and distribution. On top of such a prototype, a large body of experiments will be possible. Some of the possibilities have already been outlined in chapter 7. Implementation work is underway, as described in [Brandt & Schmidt 95].

The job of a meta-level interface is to expose certain parts of the language implementation in a controlled manner. Clearly, a MLI can only be designed with a detailed understanding of the language being exposed. During the work described in this report, a formal semantics for BETA would have been an immense help, and should be given a high priority. Especially concerning the complex issue of qualification constraints is a more detailed understanding needed. Appendix E contains an ad-hoc description of of a few aspects of generalized BETA qualifications. However, more work is clearly needed.

Also, more work is needed on the reification of implicit object attributes of the kind that only exist during execution of an object. Such attributes were described in section 5.3.2, and the scanning of these attributes was described in section 6.2.3.

Chapter 10

Conclusion

When I originally began the writing of this report, the primary goal was to give a more explicit formulation of some vague ideas on attribute references. These ideas had primarily developed while designing and implementing the *ObjectSerializer* library and trying to understand why BETA despite its huge expressive power did not allow me to express what I really wanted. The provisional answer that I came up with was that I needed a way to talk about attributes, and that BETA did not allow me to do so. Thus, my first attempt to describe the notion of attribute references was part of [Brandt 94], also describing the design, implementation and first uses of the ObjectSerializer.

Apart from being more precise about what attribute references really are, the goal was to demonstrate their feasibility through a number of detailed examples. By working with these examples, a more detailed understanding of the BETA type system and its limitations emerged. Most significantly, it was increasingly clear that the rigid type system of current BETA did not allow me to express the typing constraints that I needed. Thus, I needed to extend BETA to allow less rigid typing constraints. The result was the generalized qualifications described in chapter 4, including the notions of partial versus full qualifications, and object versus pattern qualifications. As a side effect, my work on attribute references therefore added aliases, constant references, and in general a much more flexible type-system to BETA *without* adding any new syntax.

This generalization of the BETA type system is the first major contribution of the work reported here.

During my work on attribute references, I attended the tutorial "Open Implementations: A Metaobject Protocol Approach" given by Gregor Kiczales, John Lamping and Andreas Paepcke at the OOPSLA'94 conference. Apart from being an extremely interesting tutorial, it made me realize what I was really doing: I was actually designing the introspectional parts of a meta-level interface for BETA.

Introspection would solve most of my problems with the ObjectSerializer, but it did not help solving some of the principal problems of the implementation of a fully general distributed object system as outlined in [Brandt & Madsen 94]. However, the right intercessory protocol, allowing the redefinition of method dispatch and variable access, could provide significant help in this area too.

As a consequence I started looking into the literature on reflection. Most of what I found emerged from the LISP community and was extremely theoretical in nature. In any case, it was not directly applicable to BETA, being a compiled and statically typed language. Finally, "The Art of the Metaobject Protocol" [Kiczales et al. 91] gave me a practical understanding of what was really going on, and while reading a description of the OpenC++ [Chiba 93] system by Chiba, I realized that BETA could express reflection on method dispatch even better than OpenC++, using the orthogonality and larger expressive power of BETA as compared to C++. Direct inspiration from OpenC++ lead to the intercessory parts of the BETA MLI described in section 6.6.

Thus, the second major contribution of this report, is a proposal for a meta-level interface for BETA. To the best of my knowledge, BETA is currently the only statically typed language for which a comprehensive meta-level interface, including introspective as well as intercessory facilities, has been designed.

A very important building block for the meta-level interface is the notion of attribute references that originally initiated this work. Attribute references provide *typed* references to instance and pattern attributes, and allow BETA programs to explicitly talk about such attributes and pass them as parameters in method calls. In effect, attribute references provide type-safe computed attribute access. Since "standard" BETA already allows the programmer to explicitly talk about objects and patterns (through dynamic pattern references), the addition of attribute references is a natural extension of the language, providing a huge increase in expressive power.

Attribute references are therefore the third major contribution of this report.

A number of possible application areas have been pointed out in section 7. Before it is possible to apply the MLI for real applications, we need an implementation of the MLI itself. In section 8, an outline of a distributed programming and runtime environment was described. The implementation of such an environment has already been initiated. Most of the enabling technologies are already available in the Mjølner BETA System, and should allow a relatively quick implementation of a first prototype. On top of the prototype, experiments with reflection may then be carried out. Such experiments would in principle be possible without the distributed environment, but in order to solve problems of referential integrity in BETA distribution and persistence, the distributed environment is needed anyway. The recognition of the great number of new possibilities and interesting research directions enabled by a truly distributed programming environment as the one outlined in section 8 may also be considered a contribution of the work I have been carrying out since the beginning of my Ph.D. study.

Finally, as with all programming language abstractions, the real feasibility of the BETA MLI and attribute references described here can only be proven in practice. Hopefully the future will bring the experience needed.

10.1 Acknowledgements

First of all, I would like thank Rene Wenzel Schmidt and Jørgen Lindskov Knudsen for numerous stimulating discussions and comments on the subjects of this report. Without Rene and Jørgen, the work described here had never reached its current level of maturity. Furthermore, Mads Nielsen and Erik Ernst have provided valuable comments along the way.

Bibliography

[Accetta et al. 86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. T. Jr., , and M. Young. Mach: A New Kernel Foundation for Unix Development. In *Proceedings of the 1986 Summer USENIX Conference*, pages 93 – 113, July 1986.

[Agesen & Frølund 89] O. Agesen and S. Frølund. Dynamic Link and Load as a Basis for Implementing Extensibility. Technical report, Department of Computer Science, University of Aarhus, 1989.

[Agesen & Ungar 94] O. Agesen and D. Ungar. Sifting out the gold. In *OOPSLA'94 Conference Proceedings*, pages 355 – 370. ACM, ACM Press, October 1994.

[Agesen et al. 89] O. Agesen, S. Frølund, and M. Olsen. Persistent and Shared Objects in BETA. Master's thesis, Department of Computer Science, University of Aarhus, 1989.

[Black et al. 87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), January 1987.

[Borning et al. 92] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. In *Lisp and Symbolic Computation*, pages 223 – 270, September 1992.

[Brandt & Madsen 94] S. Brandt and O. Madsen. Object-Oriented Distributed Programming in BETA. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Lecture Notes in Computer Science 791*, pages 185 – 212. Springer-Verlag, 1994.

[Brandt & Schmidt 95] S. Brandt and R. Schmidt. The Design of a Meta-Level Architecture for the BETA Language. Presented at META'95: Workshop on Advances in Metaobject Protocols and Reflection at ECOOP '95, August 1995.

[Brandt 93] S. Brandt. *The BETA Distribution Library. Manual.* Department of Computer Science, University of Aarhus, Nov 1993. Draft Version.

[Brandt 94] S. Brandt. Implementing Shared and Persistent Objects in BETA — Progress Report. Technical report, Department of Computer Science, University of Aarhus, 1994.

[Chiba 93] S. Chiba. Designing an Extensible Distributed Language with a Meta-Level Architechture. In O. Nierstrasz, editor, *ECOOP'93 - Object-Oriented Programming*, volume 707 of *LNCS*, pages 482 – 501. Springer-Verlag, July 1993.

[Decouchant 86] D. Decouchant. Design of a Distributed Object Manager for the Smalltalk-80 System. In *OOPSLA'86 Proceedings*, 1986.

[Farkas & Dearle 93] A. Farkas and A. Dearle. Octopus: A Reflective Language Mechanism for Object Manipulation. In *Proceedings of the Fourth International Workshop on Database Programming Languages*. Springer-Verlag, 1993.

[Freeman-Benson & Borning 92] B. Freeman-Benson and A. Borning. Integrating constraints with an object oriented language. In *ECOOP'92 proceedings*, number 615 in LNCS, pages 268 – 286. Springer-Verlag, 1992.

[Goldberg & Robson 83] A. Goldberg and D. Robson. *Smalltalk-80. The Language and its Implementation.* Addison-Wesley, Publishing Company, Reading, MA, 1983.

[Grønbæk et al. 94] K. Grønbæk, J. Hem, O. Madsen, and L. Sloth. Hypermedia Systems: A Dexter-Based Architecture. *Communications of the ACM*, 37(2):64 – 74, Feb 1994.

[Hölzle & Ungar 94] U. Hölzle and D. Ungar. A Third Generation Self Implementation: Reconciling Responsiveness with Performance. In *OOPSLA'94 Conference Proceedings*, pages 229 – 243, October 1994.

[Horn 92] B. Horn. Constraint patterns as a basis for object oriented programming. In *OOPSLA'92 proceedings*, pages 218 – 234. ACM, 1992.

[J. Jaffar et al. 92] S. M. J. Jaffar, P. Stuckey, and R. Yap. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3):339 – 395, 1992.

[Jul 94] E. Jul. Private conversation regarding work on heterogeneous Emerald, May 1994.

[Kiczales & Lamping 93] G. Kiczales and J. Lamping. Operating Systems: Why Object-Oriented? In *Proceedings of IWOOOS'93*, 1993.

[Kiczales 94] G. Kiczales. Why are black boxes so hard to reuse? (towards a new model of abstraction in the engineering of software). Invited Talk at OOPSLA'94, October 1994.

[Kiczales et al. 91] G. Kiczales, J. Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Knudsen & Middelfart 94] J. Knudsen and J. Middelfart. Supporting constraints in beta. Presented at NWPER'94: Nordic Workshop on Programming Environment Research, Lund, Sweden. Available from http://www.daimi.aau.dk/ beta/Papers/BETAconstraints/BETAconstraints.html, June 1994.

[Lopez et al.] G. Lopez, B. Freeman-Benson, and A. Borning. Kaleidoscope: A constraint imperative programming language. Draft of August 3, 1993.

[Madsen & Møller-Pedersen 92] O. Madsen and B. Møller-Pedersen. Part Objects and Their Location. In B. M. G. Heeg and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems – TOOLS 7*, 1992.

[Madsen & Nørgård 93] O. Madsen and C. Nørgård. An Object-Oriented Metaprogramming System. In J. Knudsen, O. Madsen, B. Magnusson, and M. Löfgren, editors, *Object-Oriented Environments.* Prentice Hall, Sep 1993.

[Madsen et al. 93a] O. Madsen, B.Magnusson, and B. Pedersen. Strong typing of object-oriented languages revisited. In J. Knudsen, O. Madsen, B. Magnusson, and M. Löfgren, editors, *Object-Oriented Environments.* Prentice Hall, Sep 1993.

[Madsen et al. 93b] O. Madsen, B. MøllerPedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison Wesley, June 1993.

[Maes & Nardi 88] P. Maes and D. Nardi, editors. *Meta-Level Architectures and Reflection.* North-Holland, 1988.

[Malhotra 94] J. Malhotra. *Tailorable Systems: Design, Support, Techniques, and Applications. (Daimi PB-466).* PhD dissertation, Department of Computer Science, University of Aarhus, January 1994.

[Masuhara et al. 92] H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa. Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently. In A. Paepcke, editor, *OOPSLA'92 Conference Proceedings*, pages 127 – 144. ACM Press, October 1992.

[McCullough 87] P. McCullough. Transparent forwarding: First steps. In *OOPSLA'87 Proceedings*, pages 331 – 341, 1987.

[Meyer 92] B. Meyer. *Eiffel, The Language.* Prentice Hall, 1992.

[Okamura et al. 93] H. Okamura, Y. Ishikawa, and M. Tokoro. Metalevel Decomposition in AL-1/D. In *Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS)*, November 1993.

[OMG 91] OMG. *The Common Object Request Broker Architecture and Specification*, Dec 1991. Document number 91.12.1, Revision 1.1.

[Omohundro 93] S. Omohundro. The Sather Programming Language. *Doctor Dobb's Journal*, 1993.

[Oxhøj et al. 92] N. Oxhøj, J. Palsberg, and M. Schwartzbach. Making type inference practical. In O. Madsen, editor, *ECOOP'92. European Conference on Object-Oriented Programming*, number 615 in LNCS, pages 329 – 349. Springer-Verlag, June/July 1992.

[Paepcke 88] A. Paepcke. PCLOS: A flexible implementation of clos persistence. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object-Oriented Programming*, number 322 in LNCS, pages 374 – 389, 1988.

[Robie & Witte 91] J. Robie and B. Witte. *The POET handbook.* Berlin, Germany, 1991.

[Sloth & Hem 93] L. Sloth and J. Hem. Object Oriented Model for the Distributed Object Oriented Database. Technical report, EuroCODE, Workpackage WP2 - Deliverable D2.2, 1993.

[Smith 84] B. Smith. Reflection and semantics in Lisp. In *Proceedings of the 1984 ACM Principles of Programming Language Conference*, pages 23–35. ACM, December 1984.

[Stallman 81] R. Stallman. Emacs: The extensible, customizable, self-documenting display editor. In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 147–156, 1981.

[Sun 93] Sun Microsystems Laboratories, Inc., 2550, Garcia Avenue, Mountain View, CA 94043 USA. *SELF 3.0*, 1993.

[V. Singhal & Wilson 92] S. K. V. Singhal and P. Wilson. Texas: An Efficient, Portable Persistent Store. In A. Albano and R. Morrison, editors, *Persistent Object Systems, San Miniato 1992*, 1992.

Appendix A

Attribute Reference Syntax

Extensions to the BETA grammar to include attribute reference syntax is shown below. Only the affected parts of the grammar are shown.

```
<referenceSpecification> ::| <StaticItem>
                           | <DynamicItem>
                           | <StaticComponent>
                           | <DynamicComponent>
                           | <VariablePattern>
                           | <AttributeReference>
                           ;

<AttributeReference> ::= '.' <LocQua:AttrRefQuaOpt> <KindAndAttrQua> ;

<KindAndAttrQua>     ::=   | <StaticOrOriginKind>
                           | <DynamicItemKind>
                           | <StaticComponentKind>
                           | <DynamicComponentKind>
                           | <DynamicPatternKind>
                           | <StaticPatternKind>
                           | <StaticRepetitionKind>
                           | <DynamicRepetitionKind>
                           | <PatternRepetitionKind>
                           | <StaticComponentRepetitionKind>
                           | <DynamicComponentRepetitionKind>
                           | <AttributeKind>
                           ;

<StaticOrOriginKind>             ::= '@'    <AttrQua:AttrRefQuaOpt> ;
<DynamicItemKind>                ::= '^'    <AttrQua:AttrRefQuaOpt> ;
<StaticComponentKind>            ::= '@|'   <AttrQua:AttrRefQuaOpt> ;
<DynamicComponentKind>           ::= '^|'   <AttrQua:AttrRefQuaOpt> ;
<DynamicPatternKind>             ::= '##'   <AttrQua:AttrRefQuaOpt> ;
<StaticPatternKind>              ::= ':'    <AttrQua:AttrRefQuaOpt> ;
<StaticRepetitionKind>           ::= '[]@'  <AttrQua:AttrRefQuaOpt> ;
<DynamicRepetitionKind>          ::= '[]^'  <AttrQua:AttrRefQuaOpt> ;
<PatternRepetitionKind>          ::= '[]##' <AttrQua:AttrRefQuaOpt> ;
<StaticComponentRepetitionKind>  ::= '[]@|' <AttrQua:AttrRefQuaOpt> ;
<DynamicComponentRepetitionKind> ::= '[]^|' <AttrQua:AttrRefQuaOpt> ;
<AttributeKind>                  ::= '.' ;
<GenericKind>                    ::= '?' ;

<AttrRefQuaOpt>      ::? <AttributeDenotation> ;

<AttributeDenotation>::| <NameApl>
                       | <Remote>
                       | <ComputedRemote>
```

```
                        | <Indexed>
                        | <ThisObject>
                        | <RemotePrimitive>
                        | <AttrRefRemote>
                        | <AttrRefComputedRemote>
                        ;
<AttrRefRemote> ::= <AttributeDenotation> '.' '(' <NameApl> ')' ;
<AttrRefComputedRemote> ::= '(' <Evaluations> ')' '.' '(' <NameApl> ')' ;

<Transaction> ::| <ObjectEvaluation>
                | <ComputedObjectEvaluation>
                | <ObjectReference>
                | <EvalList>
                | <StructureReference>
                | <Primitive>
                | <Address>
                | <AttributeEvaluation> ;

<AttributeEvaluation> ::= <AttributeDenotation> ':' ;
```

Appendix B

Static Object Clusters

The object copying algorithms shown in the following appendices are built around the concept of "static object clusters". Informally, a static object cluster is simply a set of objects connected by static references, i.e., StatRefs. When copying an object o, it is necessary to copy all objects in the static object cluster to which o belongs. Below, the concept of static object cluster is defined.

Formal definition of Static Object Clusters

Given a graph $G_o = (O, R)$ consisting of objects (the nodes) and references (the edges), this section describes the grouping of objects into equivalence classes defined by their static connections. The equivalence classes may be used as nodes in a new super graph G_s. The graph G_s consists of atomic entities, i.e., entities that should not be split into smaller parts[1]. As an example of the equivalence classes, consider the object graph in figure 6.14 on page 57, consisting of 7 distinct objects, and three equivalence classes. One equivalence class is the set $\{\texttt{main}, \texttt{main.g1}, \texttt{main.g2}\}$, the second is the set $\{\texttt{gs}, \texttt{gs.s}\}$, and the third is the set $\{\texttt{gscopy}, \texttt{gscopy.s}\}$.

Given an arbitrary BETA object o, it will be either autonomous or a static part of some other BETA object o'. Autonomous objects are those whose *location* reference (see section 3.1.1) is `NONE`. Now, given o we may repeatedly follow location references until we reach a well-defined autonomous object. Call this object $\langle o \rangle_{root}$. To describe the way objects belong together in static clusters, we shall define the equivalence relation $\equiv_{stat}$ as follows:

$$o \equiv_{stat} o' \overset{def}{\Leftrightarrow} \langle o \rangle_{root} = \langle o' \rangle_{root}$$

$\equiv_{stat}$ divides the set O of all BETA objects into equivalence classes. Given some object $o \in O$, let $\langle o \rangle$ denote the equivalence class to which o belongs, and let $\langle o \rangle_{root}$ be the canonic representative of $\langle o \rangle$.

The intuition behind $\equiv_{stat}$ is a wish to regard clusters of statically connected objects as wholes during traversal of the object graph G_o. The result is that the equivalence classes of $\equiv_{stat}$ are the atomic units with regard to many object graph traversal algorithms.

As noted in section 3.1.1, static references is the way BETA supports whole/part composition. The "part" corresponds to an object o. The "whole" corresponds to the static object cluster $\langle o \rangle$.

The relation $\rightarrow_{ref}$ may now be extended to the equivalence classes of $\equiv_{stat}$ by:

$$\langle o \rangle \rightarrow_{ref} \langle o' \rangle \overset{def}{\Leftrightarrow} \langle o \rangle \neq \langle o' \rangle \ \wedge\ \exists o'' \in \langle o \rangle, o''' \in \langle o' \rangle \,.\, o'' \rightarrow_{ref} o'''$$

[1] At least not logically. In an implementation, it may be convenient to split these entities into smaller parts.

Note that, since StatRefs always connect objects within the same equivalence class, references between equivalence classes are by definition either OriRefs or DynRefs.

The super graph $G_s = (O_s, R_s)$ of static object clusters may now be defined by:

$$\begin{aligned} O_s &= \{\langle o \rangle | o \in O\} \\ R_s &= \{(\langle o \rangle, \langle o' \rangle) \mid \langle o \rangle \rightarrow_{ref} \langle o' \rangle\} \end{aligned}$$

Appendix C

Shallow Copying

This appendix shows the code for a shallow copying algorithm as described in section 7.1.1. For simplicity, the algorithm does not consider all attribute kinds. However, in most cases, the additions needed to handle these extra attribute kinds are trivial.

```
CopyShallow:
  (#
     Copies: @list
       (* The Copies table keeps track of the association of objects and
        * their copies. *)
       (# element::
            (# obj: ^Object; pcopy: ^MLI.PendingObject; copy: ^Object #);
          getCopy:
            (* If a copy of obj has been finished, getCopy calls "OnFinished" with a
             * reference to the copy. If a copy of obj is pending, getCopy calls
             * "OnPending" with the pending copy. Otherwise, getCopy calls NotFound. *)
            (# OnFinished:< (# found: ^Object enter found[] do INNER #);
               OnPending:< (# found: ^MLI.PendingObject enter found[] do INNER #);
               NotFound:< Object;
               obj: ^Object;
            enter obj[]
            do scan
                 (#
                 do (if obj[]=current.obj[] then
                        (if current.copy[]<>NONE then current.copy[]->OnFinished
                        else current.pcopy[]->OnPending
                        if);
                        leave getCopy;
                    if);
                 #);
               NotFound;
            #);
          CreateAndRegisterPendingCopy:
            (# obj: ^Object; pcopy: ^MLI.PendingObject;
               new: ^element;
            enter obj[]
            do &element[]->new[];
               obj[]->getObjectDescriptor->od##->MLI.CreatePendingObject->pcopy[];
               (obj[],pcopy[])->(new.obj[],new.pcopy[]);
               new[]->append;
            exit new.pcopy[]
            #);
          RegisterPendingCopy:
```

```
      (# obj: ^Object; pcopy: ^MLI.PendingObject;
         new: ^element;
      enter (obj[],pcopy[])
      do &element[]->new[];
         (obj[],pcopy[])->(new.obj[],new.pcopy[]);
         new[]->append;
      #);
  #);

getObjectDescriptor:
  (* Returns the ObjectDescriptor of which obj is an instance. *)
  (# obj: ^Object; od: ##Object;
  enter obj[]
  do od[]->MLI.GetObjQual->MLI.AsObjectDescriptor->od##
  exit od##
  #);

findStaticRoot:
  (* Returns the root object of the static object cluster to which
   * obj belongs. *)
  (# obj, loc: ^Object;
  enter obj[]
  do while:
      (if (obj[]->MLI.GetObjLocation->loc[])<>NONE then
         loc[]->obj[]; restart while;
      if)
  exit obj[]
  #);

FindCopiedSet:
  (* Ensure pending copies are created for all static parts of obj: *)
  (# obj: ^Object; pcopy: ^MLI.PendingObject;
  enter (obj[],pcopy[])
  do obj[]->getObjectDescriptor->MLI.ScanStat
     (# constobj: @obj;    (* Constant reference to obj.                *)
        locQua:: constobj; (* For static check of "constobj.(current)". *)
        part: ^Object;     (* Current part object.                      *)
     do (if (not current->MLI.StatRefIsSimple) and
            constobj[]=(constobj.(current)[]->part[]->MLI.GetObjLocation)
        then
            (* Current is not a simple attribute, and it is part of the same
             * static cluster as obj. *)
            current->pcopy.CheckStatOrOriRef
            (# OnMissing::
                 (# po: ^MLI.PendingObject;
                 do (* Create and bind pending copy of thispart *)
                    part[]->Copies.CreateAndRegisterPendingCopy->po[]->bindToPending;
                    (part[],po[])->FindCopiedSet;
                 #);
               OnPending::
                 (#
                 do (* Register in Copies. *)
                    (part[],po[])->Copies.RegisterPendingCopy;
                    (part[],po[])->FindCopiedSet;
                 #);
            #);
        if)
     #);
  #);

BindOriginAndConstRefs:
  (* Bind all origin references and constant references, thereby
   * preparing pending objects for release. *)
  (#
  do Copies.Scan
     (# (* Aliases: *) obj: ^current.obj; pcopy: ^current.pcopy;
     do
```

```
        (* Bind origins. Notice that all origins are known to be missing. i.e.,
         * not bound, at this stage *)
        obj[]->getObjectDescriptor->MLI.ScanOri
        (# constobj: @obj; locQua:: constobj;
        do current->pcopy.CheckStatOrOriRef
           (# OnMissing::
                (#
                do constobj.(current)[]->Copies.getCopy
                   (# OnPending:: (# do pcopy[]->bindToPending #);
                      NotFound:: (# do constobj.(current)[]->bindToComplete #);
                   #);
                #);
           #);
        #); (* End of MLI.ScanOri *)

        (* Then bind constant references: *)
        obj[]->getObjectDescriptor->->MLI.ScanStat
        (# constobj: @obj; locQua:: constobj;
        do (if (not current->MLI.StatRefIsSimple) and
               constobj[]<>(constobj.(current)[]->part[]->MLI.GetObjLocation)
           then
               (* Current is a constant reference *)
               current->pcopy.CheckStatOrOriRef
               (# OnMissing::
                    (#
                    do constobj.(current)[]->Copies.getCopy
                       (# OnPending:: (# do pcopy[]->bindToPending #);
                          NotFound:: (# do constobj.(current)[]->bindToComplete #);
                       #);
                    #);
               #);
           if);
        #); (* End of MLI.ScanStat *)

     #); (* End of Copies.Scan *)

     (* All origins and static references in all pending copies
      * have been bound, and the pending objects can be released: *)

     Copies.Scan
     (#
     do current.pcopy.GetObject (# OnObject:: (# do o[]->current.copy[] #)#);
     #);

  #);

CopyRemainingAttributes:
  (* Copy all attributes that have not already been setup. I.e., all simple
   * static attributes and all dynamic attributes. *)
  (#
     Copies.scan
     (# (* Aliases: *)  obj: ^current.obj; copy: ^current.copy;
     do
        (* Copy simple static attributes: *)
        obj[]->getObjectDescriptor->MLI.ScanStat
        (#
        do (if current->MLI.StatRefIsSimple then
              (if current->MLI.Qual
              //Boolean## then  (# a: .@Boolean  do current->a; obj.(a)->copy.(a) #)
              //Char## then     (# a: .@Char     do current->a; obj.(a)->copy.(a) #)
              //ShortInt## then (# a: .@ShortInt do current->a; obj.(a)->copy.(a) #)
              //Integer## then  (# a: .@Integer  do current->a; obj.(a)->copy.(a) #)
              //Real## then     (# a: .@Real     do current->a; obj.(a)->copy.(a) #)
              if);
           if)
        #);
```

```
          (* Copy dynamic references: *)
          obj[]->getObjectDescriptor->MLI.ScanDyn
          (#
          do obj.(current)[]->Copies.getCopy
             (# OnFinished:: (# do found[]->copy.(current)[] #);
                NotFound::   (# do obj.(current)[]->copy.(current)[] #);
             #);
          #);

          (* Simple static repetitions: *)
          obj[]->getObjectDescriptor->MLI.ScanStatRep
          (#
          do
             (* For primitive repetitions, we may use the language
              * primitive for repetition copying: *)
             (if current->MLI.Qual
              //Boolean## then  (# a: .[]@Boolean  do current->a; obj.(a)->copy.(a) #)
              //Char## then     (# a: .[]@Char     do current->a; obj.(a)->copy.(a) #)
              //ShortInt## then (# a: .[]@ShortInt do current->a; obj.(a)->copy.(a) #)
              //Integer## then  (# a: .[]@Integer  do current->a; obj.(a)->copy.(a) #)
              //Real## then     (# a: .[]@Real     do current->a; obj.(a)->copy.(a) #)
              else
                 (* This is a static repetition of complex objects, and is
                  * not handled by this simple version of CopyShallow. *)
             if);
          #);

          obj[]->getObjectDescriptor->MLI.ScanDynRep
          (#
          do obj.(current).range -> copy.(current).new;
             (for i: obj.(current).range repeat
                  obj.(current)[i][]->Copies.Getcopy
                  (# OnFinished:: (# do found[]->copy.(current)[i][] #);
                     NotFound::   (# do obj.(current)[i][]->copy.(current)[i][] #);
                  #);
             for);
          #);
       #);
    #);

   obj,root,objcopy: ^Object;
enter obj[]
do
   (* Initialization: *)
   Copies.init;

   (* Locate the static root object of which obj is a part: *)
   obj[]->findStaticRoot->root[];

   (* Create pending copies of static object cluster rootet in root: *)
   (root[],root[]->Copies.CreateAndRegisterPendingCopy)->FindCopiedSet;

   (* Bind origins and constant references. Constant references are
    * static references with a full object qualification. *)
   BindOriginAndConstRefs;

   (* Copy remaining attributes: *)
   CopyRemainingAttributes;

   (* Get the finished copy of obj: *)
   obj[]->Copies.getCopy (# onFinished:: (# do copy[]->objcopy[] #)#);

exit objcopy[]
#);
```

Appendix D

Tailorable Deep Copying

This appendix shows the code for a tailorable deep copying algorithm as described in section 7.1.2. Only code differing from that of the previous appendix on shallow copying is shown.

```
CopyDeep:
  (# SkipObject:< BooleanValue (# obj: ^Object enter obj[] do INNER #);
     SkipDynRef:< BooleanValue (# attr: .^ enter attr do INNER #);

     ...

     (* StatClasses is the set of static object cluster roots that have been pointed
      * out to be copied. *)

     StatClasses: @Set;

     (* TraversalQueue is a queue of static object cluster roots that should be copied
      * and scanned for dynamic references. All objects in TraversalQueue are also
      * in StatClasses. *)

     TraversalQueue: @queue;

     scheduleForCopy:
       (* Schedules root, which is the root of a static object cluster, to
        * be copied. *)
       (# root: ^Object;
       enter root[]
       do root[]->StatClasses.insert; root[]->TraversalQueue.append;
       #);

     HandleStatClass:
       (* Ensure pending copies are created for all static parts of obj (recursively).
        * Then check what dynamic references should be followed. *)
       (# obj: ^Object; pcopy: ^MLI.PendingObject;
       enter (obj[],pcopy[])
       do (* Create and register pending copies: *)
          obj[]->getObjectDescriptor->MLI.ScanStat
          (# constobj: @obj;    (* Constant reference to obj.               *)
             locQua:: constobj; (* For static check of "constobj.(current)". *)
             part: ^Object;     (* Current part object.                     *)
          do (if (not current->MLI.StatRefIsSimple) and
                 constobj[]=(constobj.(current)[]->part[]->MLI.GetObjLocation)
             then
                 (* Current is not a simple attribute, and it is part of the same
```

```
              * static cluster as obj. *)
             current->pcopy.CheckStatOrOriRef
             (# OnMissing::
                  (# po: ^MLI.PendingObject;
                  do (* Create and bind pending copy of thispart *)
                     part[]->Copies.CreateAndRegisterPendingCopy->po[]->bindToPending;
                     (part[],po[])->HandleStatClass;
                  #);
                OnPending::
                  (#
                  do (* Register in Copies. *)
                     (part[],po[])->Copies.RegisterPendingCopy;
                     (part[],po[])->HandleStatClass;
                  #);
             #);
          if)
       #);

       (* Check what dynamically referred objects should be copied along. *)
       obj[]->getObjectDescriptor->MLI.ScanDyn
       (# constobj: @obj; locQua:: constobj;
          ref, root: ^Object;
       do obj.(current)[]->ref[];
          (* Check wheter ref has already been scheduled for copying: *)
          (if not ref[]->findStaticRoot->root[]->StatClasses.has then
             (* Not scheduled. Check whether we should skip it. *)
             (if true
             //current->SkipDynRef
             //ref[]->SkipObject then
             else
                (* Do not skip. Schedule for copying: *)
                root[]->scheduleForCopy;
             if)
          if)
       #);
    #);

   FindCopiedSet:
     (# root: ^Object;
     do
        while:
          (if not TraversalQueue.empty then
             TraversalQueue.remove->root[];
             (root[],root[]->CreateAndRegisterPendingCopy)->HandleStatClass;
             restart while;
          if);
     #);

do (* Initialization: *)
   Copies.init; StatClasses.init; TraversalQueue.init;

   (* Locate the static root object of which obj is a part: *)
   obj[]->findStaticRoot->root[];

   (* Create pending copies of all objects to be copied: *)
   obj[]->findStaticRoot->root[]; root[]->scheduleForCopy; FindCopiedSet;

   (* Bind origins and constant references. Constant references are
    * static references with a full object qualification. *)
   BindOriginAndConstRefs;

   (* Copy remaining attributes: *)
   CopyRemainingAttributes;

   (* Get the finished copy of obj: *)
   obj[]->Copies.getCopy (# onFinished:: (# do copy[]->objcopy[] #)#);
#);
```

Appendix E

Formal Description of Qualification Semantics

This section introduces some formal notation for speaking about qualifications. The notation is introduced along with a number of examples explaining the semantics of BETA qualifications. A full semantics of BETA qualifications, although clearly valuable during the design of a MOP, is outside the scope of this report.

Please notice that this section is extremely ad-hoc, and only tries to give a taste of the semantics of generalized BETA qualifications. More work is clearly necessary to provide semantics feasible for implementation purposes.

Assume in the following that `p` and `q` denote patterns, `o` and `b` denote objects, `q` and `r` denotes qualifications, `od` denotes an ObjectDescriptor (i.e., a partial pattern qualification) and `id` denotes an object reference attribute (static of dynamic).

The functions **object**() and **qual**() were introduced in section 3.2.1. **object**(id) returns the object currently bound to `id`. **qual**(id) returns the statically known qualification of `id`, whereas **qual**(o) returns the actual pattern of which `o` is an instance.

To illustrate the definitions to follow, consider the code below:

```
p: (# pp: (# <<PP>> #)#);
o: @p; (* Instance of pattern p *)

q: (# qq: o.pp (# <<QQ>> #)#);
b: @q; (* Instance of pattern q *)

stat_qq: @b.qq; (* Instance of pattern b.qq  *)
dyn_qq: ^b.qq;  (* Dynamic reference to instance of b.qq *)
```

The function **odesc**(p) returns the ObjectDescriptor of the pattern `p`. I.e., **odesc**(`p`) basically strips origin information of `p`. In the example above, **odesc**(`b.qq`) corresponds to the source code `o.pp (# <<QQ>> #)`, denoted by the path expression `q.qq`.

The relation $\subseteq$ is used to relate qualifications, and is defined as follows:

$$q \subseteq q' \quad \overset{def}{\Leftrightarrow} \quad \textbf{extension}(\mathtt{q}) \subseteq \textbf{extension}(\mathtt{q'})$$

To define $\subseteq$ precisely therefore requires a precise definition of the **extension**() function. For now, we shall be content with a few examples:

1. $\texttt{b.qq} \subseteq \texttt{o.pp}$ holds because `b.qq` is a subpattern of `o.pp`.
2. $\mathbf{odesc}(\texttt{b.qq}) \subseteq \mathbf{odesc}(\texttt{o.pp})$ holds because all objects qualified by $\mathbf{odesc}(\texttt{b.qq})$ are also qualified by $\mathbf{odesc}(\texttt{o.pp})$.
3. $\texttt{b.qq} \subseteq \mathbf{odesc}(\texttt{b.qq})$
4. $\texttt{b.qq} \subseteq \texttt{q.qq}$ holds and is equivalent to item 3, since $\mathbf{odesc}(\texttt{b.qq}) = \texttt{q.qq}$.
5. $\mathbf{qual}(\mathbf{object}(\texttt{dyn\_qq})) \subseteq \texttt{b.qq}$ holds because this is exactly what is guaranteed by the qualification on the reference `dyn_qq`.
6. $\mathbf{object}(\texttt{dyn\_qq}) \subseteq \texttt{b.qq}$ holds because $\mathbf{object}(\texttt{dyn\_qq})$ is a full object qualification with the extension $\{\mathbf{object}(\texttt{dyn\_qq})\}$, which, according to item 5, is a subset of $\mathbf{extension}(\texttt{b.qq})$.
7. $\texttt{NONE} \subseteq \texttt{b.qq}$ holds because, by definition, $\{\texttt{NONE}\} \subset \mathbf{extension}(\texttt{b.qq})$

The function **ori**() returns the origin of an ObjectDescriptor, a pattern, or an object. If the parameter(s) to **ori**() has no origin, `NONE` is returned. The following cases must be considered:

- $\mathbf{ori}(\texttt{od}) = \texttt{od'}$ means that the ObjectDescriptor `od` is nested inside the ObjectDescriptor `od'`. For example,

$$\begin{array}{rcl} \mathbf{ori}(\mathbf{odesc}(\texttt{p.pp})) & = & \mathbf{odesc}(\texttt{p}) \\ \mathbf{ori}(\mathbf{odesc}(\texttt{q.qq})) & = & \mathbf{odesc}(\texttt{q}) \end{array}$$

- $\mathbf{ori}(p) = o$ means that the most specific origin of the pattern `p` is the object `o`, i.e. the pattern `p` is nested inside `o`. For example,

$$\begin{array}{rcl} \mathbf{ori}(\texttt{b.qq}) & = & \mathbf{object}(\texttt{b}) \\ \mathbf{ori}(\texttt{o.pp}) & = & \mathbf{object}(\texttt{o}) \end{array}$$

- An object `o` has an origin reference for each of the ObjectDescriptor `od` of which it is an instance, i.e. for all members of the set $\{od \mid \texttt{o\#\#} \subseteq \texttt{od}\}$. The function **ori**() returns the origin of an object with respect to a given pattern or ObjectDescriptor. Thus, $\mathbf{ori}(o, od) = o'$ means that `o` has the origin `o'` with respect to the ObjectDescriptor `od`. For example,

$$\begin{array}{rcl} \mathbf{ori}(\mathbf{object}(\texttt{stat\_qq}), \mathbf{odesc}(\texttt{b.qq})) & = & \mathbf{object}(\texttt{b}) \\ \mathbf{ori}(\mathbf{object}(\texttt{stat\_qq}), \mathbf{odesc}(\texttt{o.pp})) & = & \mathbf{object}(\texttt{o}) \end{array}$$

Since `stat_qq` statically refers an instance of `b.qq`, and thus is a *type-exact* reference, we know that $\mathbf{odesc}(\mathbf{qual}(\mathbf{object}(\texttt{stat\_qq}))) = \mathbf{odesc}(\texttt{b.qq})$. On the other hand, `dyn_qq` is a dynamic reference whose qualification `b.qq` expresses the following constraint on $\mathbf{object}(\texttt{dyn\_qq})$

$$\mathbf{odesc}(\mathbf{qual}(\mathbf{object}(\texttt{dyn\_qq}))) \subseteq \mathbf{odesc}(\texttt{b.qq})$$
$$\mathbf{ori}(\mathbf{object}(\texttt{dyn\_qq}), \mathbf{odesc}(\texttt{b.qq})) = \mathbf{object}(\texttt{b})$$

which is equivalent to the $\mathbf{qual}(\mathbf{object}(\texttt{dyn\_qq})) \subseteq \texttt{b.qq}$ constraint.

The final notion to be introduced here is that of *origin chain*. Since ObjectDescriptors can be nested to arbitrary levels, $\mathbf{ori}(od)$ can have an origin itself and so on, leading to a chain of origins. Likewise, an object has a chain of origins for each ObjectDescriptor of which it is an instance. The function **orich**() returns the origin chain of an object. The origin chain of an object is a set of pairs, where each pair consists of an object and a *BlockLevel* value. It is defined as follows:

$$
\begin{array}{lcl}
\mathbf{orich}(o, od) & = & \mathbf{oc}(o, od, 1) \\
\mathbf{oc}(o, od, bl) & = & \{(\mathbf{ori}(o, od), bl)\} + \mathbf{oc}(\mathbf{ori}(o, od), \mathbf{ori}(od), bl + 1) \\
\mathbf{oc}(\mathtt{NONE}, \mathtt{NONE}, \_) & = & \emptyset
\end{array}
$$

$(o', bl) \in \mathbf{orich}(o, od)$ means that `o'` is the object reached by following origin references `bl` levels out, starting in `o`, as seen in the perspective of the ObjectDescriptor `od`. As an example, consider the following code:

```
A: (# B: (# C: (# #)#)#);
anA: @A;
aB: @anA.B;
aC: @aB.C;
aD: @
   (# E: aB.C (# #);
      anE: @E;
   #);
```

In this example we have that

$$
\begin{array}{lcl}
\mathbf{orich}(\mathbf{object}(\mathtt{aC}), \mathbf{odesc}(\mathrm{A.B.C})) & = & \{(\mathbf{object}(\mathtt{aB}), 1), (\mathbf{object}(\mathtt{anA}), 2)\} \\
\mathbf{orich}(\mathbf{object}(\mathtt{aD.anE}), \mathbf{odesc}(\mathrm{A.B.C})) & = & \{(\mathbf{object}(\mathtt{aB}), 1), (\mathbf{object}(\mathtt{anA}), 2)\} \\
\mathbf{orich}(\mathbf{object}(\mathtt{aD.anE}), \mathbf{odesc}(\mathrm{aD.E})) & = & \{(\mathbf{object}(\mathtt{aD}), 1)\}
\end{array}
$$

To illustrate the use of **orich**(), consider the following example of a partial pattern qualification:

```
Window:
  (# shape:
       (# draw: (# ... #);
       #);
  #);
w: @Window;
d: ^w.shape.draw;
```

In this example, the immediate origin of **object**(`d`) is only constrained to be a specific instance of `w.shape`, while the origin two levels out is constrained to be the window instance `w`. Thus, the constraint on `d` can be described as follows:

$$
\mathbf{qual}(\mathbf{object}(\mathtt{d})) \subseteq \mathbf{odesc}(\mathtt{w.shape.draw})
$$
$$
(\mathbf{object}(\mathtt{w}), 2) \in \mathbf{orich}(\mathbf{object}(\mathtt{d}), \mathbf{odesc}(\mathtt{w.shape.draw}))
$$

Pattern qualifications and `BlockLevel`

Partial pattern qualifications do not restrict one or more origins of the object (pattern) referred by the qualified object (pattern) reference. But since BETA allows nesting of patterns to arbitrary depth, even a partial qualification `pq` may restrict some member of the origin chain (see section E) with respect to $\mathbf{odesc}(pq)$.

What distinquishes a full qualification from a partial qualification is that a full qualification constrains the element of the origin chain with blocklevel value one, whereas a partial qualification restricts some element of the origin chain with a blocklevel larger than one, or no elements of the chain. Notice that restriction of some element in the origin chain also restricts all successive elements of the origin chain.

We shall think of a pattern qualification `pq` as consisting of three pieces of information:

1. An ObjectDescriptor, **odesc**(pq).
2. An integer, **blocklevel**(*pq*).
3. An `origin` reference, **ori**(pq) to an object which is an instance of the ObjectDescriptor declared **blocklevel**(pq) block levels relative to **odesc**(pq).

Full pattern qualifications are simply special cases of partial pattern qualifications, with a **blocklevel**() value of one.

Consider the following example of a partial pattern qualification:

```
Window:
  (# shape:
       (# draw: (# <<DRAW>> #);
       #);
  #);
w: @Window;
d: ^w.shape.draw;
```

The partial pattern qualification `w.shape.draw` is represented as follows:

$$\begin{array}{lcl}
\mathbf{odesc}(\texttt{w.shape.draw}) & = & \texttt{(\# <<DRAW>> \#)} \\
\mathbf{blocklevel}(\texttt{w.shape.draw}) & = & 2 \\
\mathbf{ori}(\texttt{w.shape.draw}) & = & \texttt{w}
\end{array}$$

The constraint on `d` can be described as follows:

$$\mathbf{qual}(\mathbf{object}(\texttt{d})) \subseteq \mathbf{odesc}(\texttt{w.shape.draw})$$

$$\begin{array}{l}
(\mathbf{ori}(\texttt{w.shape.draw}), \mathbf{blocklevel}(\texttt{w.shape.draw})) \\
\qquad \in \ \mathbf{orich}(\mathbf{object}(\texttt{d}), \mathbf{odesc}(\texttt{w.shape.draw}))
\end{array}$$

A partial qualification with a **blocklevel**() of zero means that the origin chain is unconstrained.

Object Qualifications

Full object qualifications are easily described, since they effectively correspond to a, possibly computed, object reference.

Concerning partial object qualifications, consider the following example:

```
Person:
  (# name: @Text;
     ...
  #);
aName: ^Person.name;
```

The identifier `aName` is qualified to refer the name of some person, which means that $\mathbf{qual}(\mathbf{object}(\texttt{aName})) = \texttt{Text}$ is a necessary but not sufficient demand. But the referred text object should also be statically referred by the `name` attribute of some person object. More precisely, the qualification constraint on `aName` can be expressed as follows:

$$\begin{gathered}
\mathbf{qual}(\mathbf{object}(\texttt{aName})) = \texttt{Text} \\
\mathbf{qual}(\mathbf{location}(\mathbf{object}(\texttt{aName}))) \subseteq \texttt{Person} \\
\mathbf{object}(\mathbf{location}(\mathbf{object}(\texttt{aName})).\texttt{name}) = \mathbf{object}(\texttt{aName})
\end{gathered}$$