CHAPTER 26

The implementation of BETA

Ole Lehrmann Madsen

The purpose of this chapter is to describe the structure and techniques used in the BETA compiler and the run-time structure for compiled programs. The latter includes object formats, code formats, storage allocation and garbage collection.

The BETA compiler accepts one or more files each containing a fragment group as described in chapter 9 of this book. For each fragment form in a group an abstract syntax tree (AST) is constructed. The semantic analyzer, storage computation and code generator then all operate on these ASTs.

The metaprogramming system described in chapter 19 of this book is an integrated part of the BETA compiler. In addition the fragment system as described in chapter 9 is also integrated with the BETA compiler. The fragment system is in practice an integrated part of the metaprogramming system and when we in the following refer to the metaprogramming system this also includes the fragment system.

The main components of the compiler are the semantic analyzer and the code generator. They both operate on the ASTs as handled by the metaprogramming system. If the group files are created using a text-based editor, the metaprogramming system will parse the text and generate an AST. Alternatively the AST may have been constructed directly by means of the syntax-directed editor Sif (chapter 22). In this case the parsing phase is not needed.

The semantic analyzer adds semantic attributes to the AST in order to handle search for names, and for performing semantic checking. The symbol table is organized as an integrated part of the AST. It is thus possible to perform semantic checking and then later code generation.

The BETA compiler generates native machine code in the form of assembly code. A future version might generate object code directly and avoid generating assembly code.

The BETA compiler exists in several variants corresponding to different computer systems. Currently the following variants exist: Macintosh/MacOs, HP 9000/HP-UX, Apollo 3500/SR10, Sun-3/SunOS, Sun-4/SunOS, and DX-200. The DX-200 is a digital telephone switch developed by Telenokia. (See chapter 37 for details about the DX-200 implementation.) The Fragment System is used to handle the organization of the compiler into these different variants.

The organization of object code modules makes it possible to have object code for several computer systems on the same file system. When semantic analysis and code generation have been performed for e.g. HP 9000/300, a subsequent compilation for e.g. Sun-4 will only need to generate the Sun-4 code. The ASTs with integrated symbol table information will be reused by the compiler. This greatly simplifies the maintenance of several variants of the same program. For the compiler itself this has been of great importance.

The compiler handles separate compilation of fragment groups. Only those fragment groups which depend on fragment groups that have been changed are recompiled. As mentioned above, if a fragment has been semantically checked, but no code has been generated, only the code will be generated. Similarly assembly code may have been generated, but not yet assembled.

An implementation of BETA can be made using many of the techniques that have been developed for other object-oriented languages such as Simula [DMN70, SIMULA87], Eiffel [Mey86] and C++ [Str86a]. The generality of BETA, however, poses some special problems not found in other languages. Some of these problems are: (1) a pattern may be used as a procedure or class, etc.; (2) subpatterns combined with arbitrary block structure; and (3) the generality of virtual patterns including use of virtual patterns as virtual classes and the use of global patterns for qualifying a virtual pattern. The following sections describe how these problems have been handled.

The current implementation has not yet reached its final stage with respect to efficiency of the compiler and the produced code. There are still a number of improvements that can be applied, but due to limited resources, this has not yet been done. The current implementation is, however, acceptable for practical use and as such has been used in practice for many years. Work is still going on to improve the implementation.

Acknowledgment Several people have contributed to the implementation of BETA. The run-time structure of the current implementation was designed together with Lars Bak, who also implemented the garbage collector and other run-time routines. Knut Barra implemented the mark/sweep garbage collector used in the first implementation of BETA. The metaprogramming system and fragment system used by the compiler have been implemented by Claus Nørgaard. Ole Agesen, Peter Andersen, Svend Frølund, Kim Jensen Møller, Claus Nørgaard, Claus H. Pedersen, Tommy Thorn and Peter Ørbæk have all assisted with various parts of the implementation of BETA.

26.1 Organization of the compiler

The compiler is organized into the following modules:

AST modules Lexical analysis, parsing, abstract syntax trees (ASTs) and prettyprinting are handled using the Mjølner BETA metaprogramming system, the BETA grammar fragments, and the BOBS LALR(1) parser generator and prettyprinter. The use of the metaprogramming system, parser and prettyprinter is described in section 26.2.

Control module The control module performs dependency analysis of fragment groups and decides which groups to compile. Given a fragment group F, the control module constructs a dependency graph of all the fragment groups reachable from F. The predecessors of F in this graph will be all the fragments that may contain declarations used in F. A recursive traversal of the predecessors is performed in order to check if any of these have been changed/recompiled since the last compilation of F. If this is the case, F is recompiled otherwise no recompilation is made. Recompilation actually means semantic checking. Even if semantic checking is not necessary, it may still be necessary to generate assembly code. A fragment F may have been checked without the generation of assembly code or code for some other machine may have been generated when F was checked. In the case where machine code has already been generated, it may still be necessary to invoke the assembler on this code.

Semantic analyzer Decorates the AST with semantic attributes and performs semantic checking. The storage requirements for objects including allocation of relative addresses of object attributes are also performed by this module. The semantic analyzer is described in detail in section 26.3.

Code generator Generates assembly code. The code generator is described in detail in section 26.4.

Assembling and linking Calls the assembler and linker of the respective computer system.

26.2 Abstract syntax tree, parsing and prettyprinting

The compiler makes use of the Mjølner BETA metaprogramming system, which takes care of construction and manipulation of abstract syntax trees (AST). The principles of the metaprogramming system are described in chapter 19.

The parse module is an LALR(1) parser generated using the BOBS-system [EJK+77]. The parser accepts a string of terminal and nonterminal symbols (a sentential form) generated from any nonterminal of the grammar. The parser constructs an AST for the string being parsed.

The Fragment System (chapter 9) allows in principle any sentential form of the grammar to be a fragment form (module). As mentioned above, the BOBS-parser is able to parse such sentential forms. The syntax-directed editor Sif allows for text editing of arbitrary sub-parts of a program. This also requires an incremental parser on arbitrary sentential forms.

For parsing a prettyprinter based on [Opp80] is being used. This prettyprinter algorithm has been extended to handle incremental changes to a screen. This facility is not used by the compiler, but is heavily used by the Sif editor.

As mentioned, the fragment system can handle arbitrary sentential forms as fragments. The compiler, however, has some limitations on this. Currently only the syntactic categories <Object-Descriptor>, <Attributes>, <MainPart>, and <DoPart> are supported by the compiler. This has turned out to be sufficient for most purposes. The reason that the compiler does not support arbitrary sentential forms is

efficiency. A fragment is the basis for separate compilation. If arbitrary sentential forms are to be supported it would not be possible to perform a complete static analysis of a fragment just as the generated code would have more overhead.

The metaprogramming system, BOBS parser and prettyprinter are all grammar based and can be used for any language with a context free syntax.

26.2.1 Example

The following BETA program[1] will be used as an example:

```
(#    P: (# V: ^ P #);
      X,Y: ^ P;
      Z: @ P(# do X[] → V[] #)
do X[] → Y[]
#)
```

The following grammar describing a subset of BETA is used to illustrate ASTs:

```
<desc> ::= <superOpt> '(#' <attList> <action> '#)'
<superOpt> ::? <ptnName>
<attList> ::* <attDecl> ';'
<attDecl> ::| <ptn> | <ref> | <part>
<ptn> ::= <nameList> ':' <desc>
<ref> ::= <nameList> ':' '^' <ptnName>
<part> ::= <nameList> ':' '@' <desc>
<action> ::= 'do' <impList>
<impList> ::* <imp> ';'
<imp> ::| <assign> | <activation>
<assign> ::= <refName> '[]' '→' <refName>'[]'
<activation> ::= <ptnName>
```

The AST for the example is illustrated in figure 26.1. The nodes corresponding to names are marked by either _d or _a corresponding to either a *declaration* or an *application* of a name. Nodes marked by e represent the empty string. Nodes marked by * represent a list of zero or more subtrees.

26.3 Semantic analysis

Semantic analysis is a major phase of the compiler. The purpose of the semantic analysis is to check whether or not the context sensitive syntax of a fragment is correct. This includes checking if all applications of names have been declared and are used in accordance with their declaration. The major parts of semantic check-

1. This program and the grammar are schematic and only for illustrative purposes.

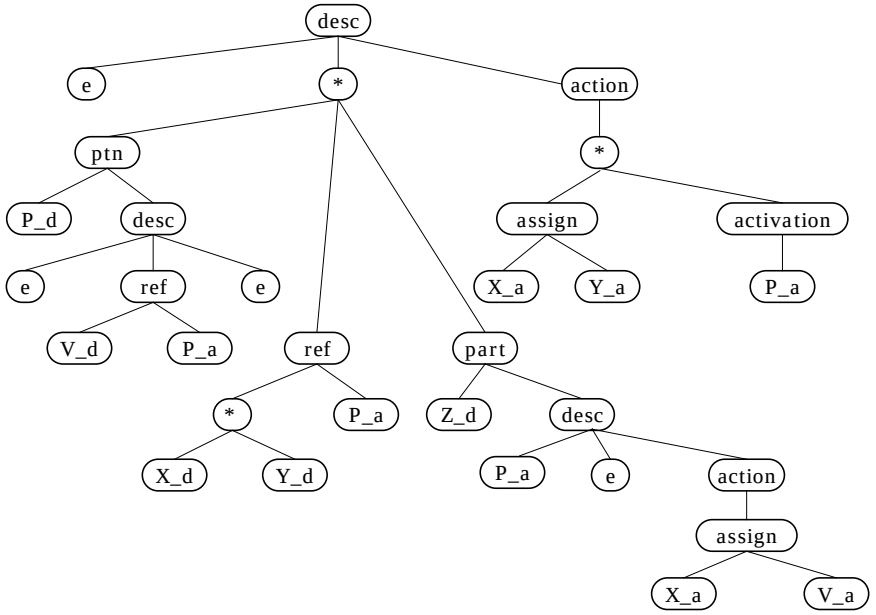


Figure 26.1 *AST for program example*

ing are symbol table organization and search for names, various semantic checks and handling of virtual patterns.

26.3.1 Symbol table organization

During semantic checking and storage computation, the AST is decorated with semantic attributes. Some of these attributes are references to other nodes in the AST. A node representing an application of a name will, e.g., have a reference to the node representing the corresponding declaration. This node in turn has a reference to the node representing the production which has generated it. This node will have all necessary semantic information about the name needed by the compiler. The symbol table of the compiler is thus the AST together with the semantic attributes. There is thus no need for a separate symbol table in the traditional sense.

The following is a list of some of the semantic attributes. The attributes marked by * have information about storage allocation and as such are not used by the semantic checker. They are, however, computed during the semantic checking. Some of the attributes are represented by locations in the AST whereas others are computed from the AST. In the following these two types of attributes are not distinguished.

NameDcl A node representing a declaration of a name has the following attributes:

- sort: The sort of a NameDcl is the production used for generating the declaration of the name. The sort of a NameDcl gives the semantics of a name. In the

example grammar the sorts are ptn, ref and part. Examples of sorts in the BETA grammar are PatternDecl, SimpleDecl, RepetitionDecl, RepetitionIndex, etc.
- off*: The offset is the relative address from the start of an object generated according to the Object-Descriptor containing this NameDcl.
- direct*: A reference attribute may be either a static (part) object or a reference to a separate object. The attribute direct has information about whether or not the corresponding object attribute should be allocated in-line or as a reference to a separate object.

NameApl A node representing an application of a name has the following attributes:

- dclRef: This attribute is a reference to the node (NameDcl) representing the corresponding declaration of the name.
- descriptor: This attribute is a reference to the node representing the object descriptor associated with this name.
- on,pn*: These attributes represent the path to the object descriptor containing the corresponding NameDcl. The declaration of the NameApl is located in the ObjectDescriptor found by following the origin attribute (see below) on times and then following the superpattern chain pn times.

Object-Descriptor A node representing an object-descriptor has the following semantic attributes:

- origin: The origin is a reference to the nearest enclosing node representing a BETA construct that defines a scope for declarations. The possible node types for BETA are ObjectDescriptor, LabelledImp, and ForImp.
- descId: This attribute holds a unique identification of the object descriptor used for generation of entry-points in the assembly code.
- size: The size attribute holds the fixed size of an object generated according to the object descriptor. The fixed size of an object is the part of the size that may be computed at compile-time. The sizes of dynamic repetitions and virtual part-objects are not known at compile-time.

Note that there are more semantic attributes than have been described above.

Example
The tree in figure 26.2 is the AST from section 26.2.1 decorated with semantic attributes. The following attributes are illustrated:

- dclRef: The curved lines from nameApl nodes to nameDcl nodes like the one from X_a to X_d show the dclRef reference.
- origin: The curved lines from a desc node to an enclosing desc node show the block structure.
- sort: The fat lines from a nameDcl node to a node of the form ref, part or ptn show sort nodes.

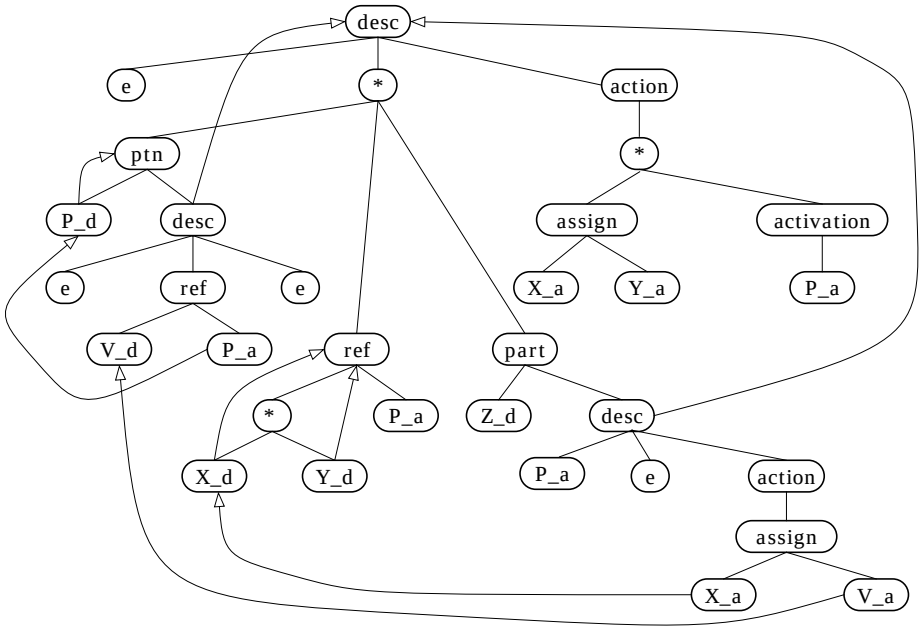


Figure 26.2 *AST decorated with attributes*

In the example, only some of the semantic attributes are shown.

Search for names Each desc node defines an *environment* of names. All nameDcl nodes within such an environment must define different names. The environment of a desc node is the set of nameDcl nodes located in the subtree rooted by the desc node except those nameDcl nodes in subtrees rooted by other desc nodes. There are two search algorithms for names:

- **Attribute-search**: The search for a name associated with a desc node is done as follows. First the name is searched in the environment associated with the desc node. If the name is not found, then an attribute-search is performed on the desc node associated with a possible superpattern. Note that an attribute search is performed for the superpattern, which means that the name is searched for at all superpatterns.
- **Global-search**: A global search for a name associated with a desc node is performed by making an attribute-search at the desc node. If the name is not found, then a global-search is performed at the desc node associated with the possible enclosing desc node (obtained by following the origin reference). Again note the recursive nature of the search, which implies that all enclosing desc nodes may be searched.

A search for a nameApl, X, is performed as a global-search of the desc node containing the nameApl.

For a remote name of the form R.X, the search for X is performed as a local-search in the desc node associated with R.

Checking For a nameApl, various checks must be performed. One example is that a name used as a superpattern must be declared as a pattern. The checking may be carried out by following the nameDcl reference and then the sort reference. The sort has all the information necessary to carry out the checking.

Consider the assignment X[]→V[]. Here X and V must be references and their qualifications must be compatible. By following the nameDcl and sort references, the sorts of X and V may be obtained and used to verify that both sorts are ref. From the sort node (a ref node), the qualifications may be obtained by following the nameDcl and sort references giving the ptn nodes defining the qualifications of X and V. These nodes have all the information needed to check the correctness of the qualifications.

Generality
Some of the techniques used for construction of integrated symbol table and AST are language independent and may be handled by a grammar based system. Such a grammar based system should support the following:

1. A lexeme defining a name must be defined as either a declaration or an application. This corresponds to NameDcl and NameApl as used above.
2. It should be possible to mark certain productions as defining unique environments of name applications. This corresponds to the above environment concept defined for desc nodes. Within such an environment, the system may then automatically check that all declarations are unique.
3. It should be possible to mark sort nodes. The sort of a name declaration may then be defined to be the nearest enclosing sort node in the AST. In most hand-coded compilers, the symbol table has definitions of datatypes which define the meaning of names. The structures of such datatypes are very close to the AST nodes which would be marked for the corresponding grammar.
4. The name application nodes should contain a semantic attribute that may refer to the corresponding declaration. Similarly, the environment nodes should have a semantic attribute referring to the enclosing environment attribute.

The above elements would be a very simple addition to a grammar based system like the Mjølner BETA metaprogramming system and BOBS parser generator. It would provide a simple, but useful automation of part of the semantic checking.

The search for names and the various semantic checks cannot be automated without turning to more powerful formalisms such as attribute grammars as described in part IX. In the Mjølner BETA metaprogramming system it might be possible to extend the automatically generated patterns to include virtual procedures for search of names and semantic checking.

26.3.2 Virtual patterns

The semantic checking of virtual patterns is the most complicated part of the semantic analysis. Consider the following example:

```
A:   (#   F: ... #); AA: A(# G: ... #);
T:   (#   V:< A;
          X: @ V   {1}
     do   V;       {2}
          X.F      {3}
     #);
TT: T(#   V::< AA;
     do   V;       {4}
          X.G      {5}
     #)
```

The instantiations of V at the points {1}, {2} and {4} are simple to handle since only the declaration/binding of the virtual pattern V is needed. These instantiations are similar to virtual procedure calls in Simula (and C++ and Eiffel). The use of V in this way makes it an example of what in BETA terminology is called *virtual procedure patterns*. Since V is also used to instantiate a named object X, V is also an example of a *virtual class pattern*.

The remote names in {3} and {5} using X are more complicated to handle. In order to check the legality of remote names such as X.F and X.G, the actual object descriptor associated with X must be known. In the main part of the pattern T, the actual descriptor associated with X is the one of A. In the main part of TT the actual descriptor is the one associated with AA. That is, the actual descriptor associated with X is not a static property of the declaration of X, but in the above example it depends on the application of X. (As we shall see later, it is not even a static property of the application.)

The checking of the remote name in {3} might take place as follows:

1. Search for X giving the declaration X:@V.
2. Search for V giving the declaration V:<A.
3. Search for A giving the pattern A.
4. Search for the attribute F in the descriptor associated with A giving the declaration F: ...

Consider now the remote name in {5}. Here it will not suffice to perform the above sequence, since this will lead to the descriptor associated with A, and the attribute G is not defined here. Instead we have to go through the following steps:

1. Search for X giving the declaration X:@V.
2. Search for V giving the declaration V:<A.
3. Find the actual binding of V relative to X.G. This will result in the virtual binding V::<AA.
4. Search for AA giving the pattern AA.
5. Search for the attribute G in the descriptor associated with AA giving the declaration G: ...

The difficult part is step 3.

26.4 Synthesizer

The synthesizer traverses the AST and produces assembly code. The synthesizer is divided into two parts: a machine independent part and a machine dependent part.

The machine dependent part is an object with attributes defining an abstract BETA machine. This object has attributes such as registers, memory addresses and operations on registers and memory addresses. The registers and memory addresses are organized in a subclass hierarchy representing different kinds of registers and addresses.

The operations on the BETA machine are the usual kinds of load/store, arithmetic operations, control operations, etc.

The BETA machine is organized as an interface fragment defining the abstract attributes of the machine object. For each machine being supported by the compiler there is a set of implementation fragments corresponding to the machine. Currently such implementation sets exists for Macintosh, HP-9000, Apollo 3500, Sun-3, Sun-4, and DX-200. These sets are not completely disjoint. The Unix based Motorola 680x0 implementations share some fragments.

The design of the abstract BETA machine is a compromise between generality, efficiency and ease of porting the compiler to a new machine type.

- Generality of the BETA machine makes the compiler more machine independent. The ultimate BETA machine would be very close to BETA itself.
- A very general BETA machine will either result in inefficient code or require a lot of work on the implementation parts of the machine fragments.
- Portability has been a major design goal. The BETA machine has been designed such that the body part of an operation just consists of emitting some assembly code emulating the operation. This makes it easy to port the compiler since the major part of the compilation is done in the machine independent part.

The experience with porting the Mjølner BETA System to a new machine type is that this requires from 3 to 6 months of work depending on the machine type. This includes porting the code generator, porting the run-time system, interfacing to the operating system and the window system. The type of CPU on the machine, the operating system and window system of course make a difference between machine types. Porting the code generator to a new type of CPU is often less time consuming than porting libraries using Unix and X Windows to Macintosh.

26.5 Runtime organization

In the following we shall use patterns described as follows:

```
P1: (# ... #);
P2: P1(# ... #);
...
Pn: Pn-1(# ... #)
```

Pn may be a singular descriptor instead of a pattern as shown in the following example:

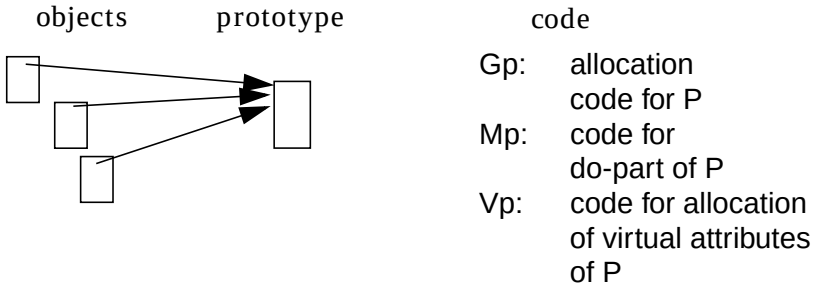
```
Pn: @ Pn-1(# ... #)
```

In addition to describing a static item, Pn-1 may describe static components, and singular action objects.

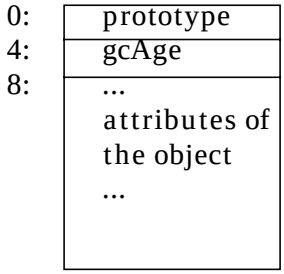
The patterns P1, P2,..., Pn are not necessarily declared together but may be declared at arbitrary points in a collection of BETA fragments. Of course the declarations are assumed to be correct with respect to the BETA semantics.

The name P will be used as an alias for Pn.

A BETA program execution consists of code to be executed, prototypes and objects generated during program execution. The following figure illustrates the run-time elements corresponding to the object-descriptor for P:



Objects generated during program execution are allocated in a heap. There are several object types. They all have the following storage layout:



The protoType attribute is a reference to a prototype description for the object which is described in section 26.5.5. The gcAge attribute is used by the garbage collector.

The following object types exist:

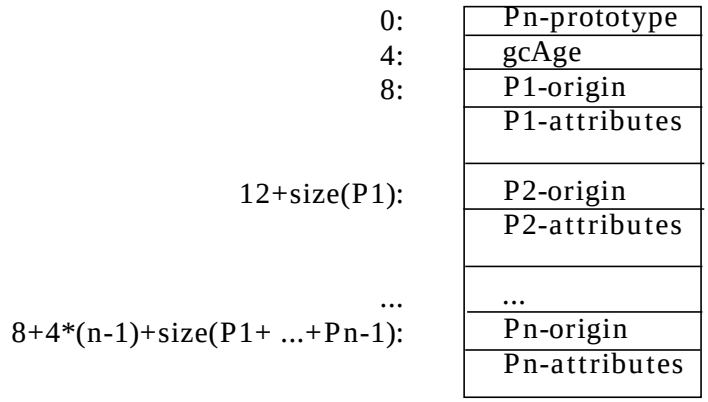
1. **Item-object**: This is the object type generated as instances of user-defined patterns/singular descriptors.
2. **Repetition-object**: This is the object type generated for a repetition object.
3. **Struc-object**: A struc-object represents the result of an evaluation T##, where T is a pattern. A struc-object for T## has the information necessary for generating an instance of T.

4. **Head-object**: For each descriptor used as a component, an item-object corresponding to that descriptor is generated. In addition a special head-object is generated. The head-object includes an item-object and a reference to a component-stack-object.
5. **Component-stack-object**: A component-stack-object is associated with each head-object. It is used for storing the call/return stack and evaluation stack of a suspended component (thread).

The different object types will be further explained below.

26.5.1 Object types

Item-object
An instance of Pn of kind item will have the following storage layout:



The origin attribute is a reference to the statically enclosing object. This is a standard technique for implementing block structure in languages such as Pascal and Simula. An attribute at an outer block level is then accessed by following the origin attribute until the given block level is reached.

In Simula all subclasses of a given class must appear at the same block level. It is therefore only necessary to have one origin attribute for an object. BETA does not have this restriction. It is therefore necessary to allocate an origin attribute for each level of superpatterns. This means that an origin attribute exists for each of the patterns P1, P2,..., Pn. If, however, a pattern Pi+1 is declared at the same block level as its superpattern Pi (i>0), then there is no origin allocated for Pi+1. This means that if all the patterns P1, P2,..., Pn are declared at the same block-level (obeying the Simula restriction), then only one origin attribute is allocated for the object (as in Simula).

If the object does not refer to attributes in enclosing objects, then it would be possible to avoid allocating any origin attribute at all. This optimization is currently not implemented.

Repetition-object
A repetition-object of the form

```
R: [range] <spec>
```

where <spec> may be @integer, @boolean, @char or ^P, has the following storage layout:

0:	rep-prototype	special rep.prototype
4:	gcAge	gcAge
8:	low	currently always 1
12:	high	range of the repetition
16:	R[1]	rep. elm. no. 1
20:	R[2]	rep. elm. no. 2
...	...	...
12+range*4:	R[range]	rep. elm. no. range

The above repetition-object shows a repetition where each element has been allocated four bytes of storage. A similar repetition-object where each element occupies one byte of storage will be implemented for boolean and char repetitions. Repetitions where <spec> has the form @T for a non-simple T have not been implemented, but this is straightforward to do.

Struc-object
A struc-object is generated from an expression like

```
T##
```

where T is a pattern. The corresponding struc-object has the form:

struc-prototype
gcAge
T-origin
T-prototype

A struc-object for T## has all the necessary information to create an instance of T: the prototype for T and the origin for T. The origin is necessary because of the block structure of BETA. Consider the following example:

```
F: ##T0;
P: (# T: T0(# ... #) do T## → F## #);
P1: @ P; P2: @ P;
...
P1; &F; {an instance of P1.T is created}
P2; &F; {an instance of P2.T is created}
```

The T## struc-object created from execution of T##→F## in the do-part of P will have P1 as its origin when P1 is executed and P2 as its origin when P2 is executed.

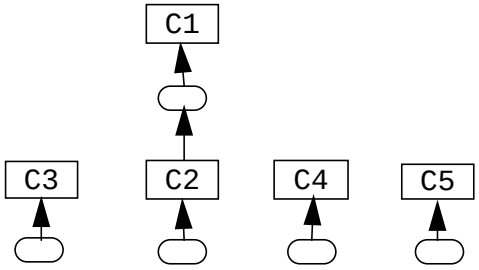
Component-object
A component-object is allocated for objects of kind component which may be declared as follows:

```
C: @ | T
```

A component is the basis for an independent execution stack in the form of either a coroutine or a concurrent process. A component-object includes a reference to the calling component, a reference to its component-stack-object and a T-item as described above.

Component-stack-object
A component-stack-object contains the execution state of a suspended component (coroutine). A program execution includes one or more components where each component has an associated stack of items. The stack of items is the set of items constituting the current thread of items that have been called from the component.

The main program is a component. A component may attach another component. The state of execution may be illustrated by the following picture:



The picture illustrates a state of execution with five components. The components C3, C4, and C5 are suspended. The main program is C1. The item on top of the C1 stack has attached C2. The objects on the active stacks C1 and C2 are called the operating chain. Each object on a stack has a structural attribute called the *return link*. The return link consists of a *dynamic reference* to the calling object and a *code pointer* in the calling object from where the call was made. For a more detailed description of components as execution stacks see [MMN93].

The return link attributes are not implemented as attributes of the objects for efficiency reasons. Instead the call stack of the underlying CPU is used. The return link attributes of the objects on the operating chain are stored on the call stack. This is similar to what is done for procedural languages like C and Pascal. When a component executes a suspend, the return link attributes are copied from the call stack to a component-stack-object associated with the component. In the above picture, the return link attributes for C3, C4 and C5 are stored in such component-stack-objects, whereas they are stored on the call stack for C1 and C2. When a component is attached, the return link attributes are copied back to the call stack.

The call stack is also used to store intermediate values computed during computation of expressions and other evaluations. Such intermediate values are also copied to the component-stack-object.

26.5.2 Simple objects and dynamic references

Simple objects such as instances of integer, boolean, and char are allocated four bytes of memory and instances of the pattern real are allocated eight bytes of memory. Such objects are not handled as item-objects. The reason for this is efficiency. There is no logical reason for not treating such objects in the general way. For the current implementation it was decided to give up generality in order to obtain improved efficiency. It should be said that no measurements of performance have been carried out and it is not known how performance will be affected by using the general implementation.

The basic BETA environment has predefined patterns IntegerObject, CharObject, etc. corresponding to all the simple objects. By using these patterns the full generality may be obtained at the expense of extra overhead.

Dynamic references are allocated as four bytes of storage. A reference is a virtual memory address. The value NONE is represented by a zero.

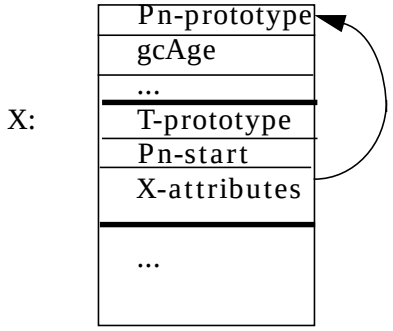
A first implementation of persistent objects has recently been carried out (chapter 12). When a persistent object is stored, all objects that can be reached from this object are also stored. Similarly when a persistent object is fetched from secondary storage, all objects that can be reached are also fetched. The reason for this is to assure that references are always well defined. In practice it may not be desirable to fetch all objects that can be reached, since this may result in copying a large number of objects into primary memory. Instead it will be possible to group persistent objects that should be read in together. This may then result in references that refer to objects in secondary memory. In order to distinguish such references from virtual memory addresses, it is considered to use negative numbers for referring to disk objects. When a reference is accessed, it must be checked whether or not the reference is a virtual memory reference or a disk reference. Since the current implementation tests for NONE references, the extended test can be performed without additional run-time overhead. Of course, if the reference is a disk reference, some code must be executed to bring the disk object into primary memory before execution can proceed.

26.5.3 Part-objects

A part-object like X in Pn

```
Pn: Pn-1 (# ...; X: @ T; ... #)
```

is allocated in-line in the Pn-object:



The gcAge of the X-object has a reference to the Pn-object of which it is a part.

It is possible to obtain a reference to a part-object as shown in the following example:

```
R: ^Pn;
S: ^T:
...
R.X[] → S[]
```

The reference S will now refer to the X-object which is located as part of a Pn-object. Such a reference inside the middle of another object requires special consideration by the garbage collector. The gcAge of the X-object is used by the garbage collector to find the object of which X is a part. The gcAge is NONE for objects which are not part-objects.

The gcAge is also used for implementing the location concept as described in [MM92], i.e.

```
S.loc →
```

The object X is only allocated in-line if the associated descriptor (T) is not defined by (1) a virtual pattern, (2) a descriptor slot or (3) a descriptor with a main-part or do-part slot in it. In these cases the size of the object cannot be determined at compile-time. Instead the object is allocated off-line and the object has a reference to the X-object.

26.5.4 Pattern attributes

For simple non-virtual pattern attributes, no storage is allocated in an object or its prototype.

For a virtual pattern, a location is allocated in the prototype of the prototype. The technique is quite similar to the one used for Simula. Due to the generality of virtuals in BETA, the implementation is slightly more complicated in the general case. This will be further explained below.

For pattern variables a reference to a struc-object is allocated in the object.

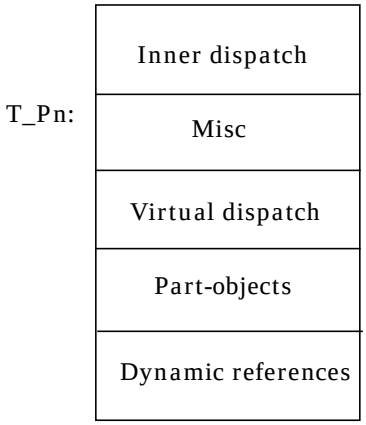
26.5.5 Prototypes

The first location of an object is the prototype of the object. The prototype determines the object type as described above and necessary run-time information about the object. An example of run-time information is the size of the object. For repetition-objects, struc-objects, head-objects and component-stack-objects the run-time information is located in the object itself or is fixed. The size of a repetition-object may be computed from its range: the size of a struc-object is always 16 bytes, etc.

For item-objects the first location of the object is a reference to a prototype table describing the run-time information of the object. This includes the size of the object, a dispatch table for virtual patterns and one for implementing inner.

For each pattern/descriptor there is an item-object prototype. The prototype is generated by the compiler. Assume that a Pn-object has virtual pattern attributes V1,V2, ... Vm, which include all virtuals declared in P1, P2, ...,Pn.

The prototype descriptor for a Pn-item has the following form:

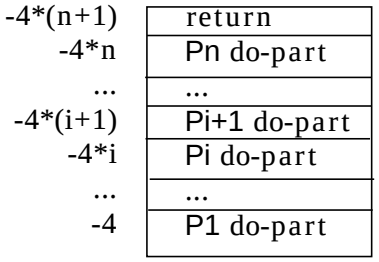


The prototype reference of an object refers to location zero of the prototype, which is the start of the misc-table. The inner-dispatch table is referred by negative offsets. The following information is stored in the prototype:

Misc table The following information is stored in the misc-table: the fixed size of the object, the offset of the origin attribute for Pn, a reference to the prototype for the superpattern of Pn, a reference to the node in the AST for the corresponding object descriptor, and various other information.

Inner dispatch table The code for inner has to be more general than in Simula. In Simula at most one inner is allowed at the outermost statement level. In BETA there can be several inners in the do-part of a descriptor, just as an inner may appear in singular objects (internal blocks) in the do-part. The code for inner is generated as a subroutine call indirectly via the inner dispatch table in the prototype. The subclass level of the descriptor determines the appropriate entry in this table.

For Pn this inner dispatch table contains:



The execution of inner in the do-part of Pi in a Pn-object is implemented as follows:

```
jumpSubroutine [this[0] - 4*(i+1)]
```

The expression this[0] gives the address of the prototype for the Pn-object. The expression [this[0] - 4*(i+1)] then gives the code address of the do-part of Pi+1.

Note that execution of inner in the do-part of Pn in a Pn-object will result in a jump subroutine to the label return, which results in execution of return from subroutine, i.e., execution of the empty action.

Virtual dispatch table The virtual dispatch table is used for generation of instances of virtual patterns. For Pn, in this table there is an entry-point corresponding to each virtual attribute of Pn as shown in the left part of the following figure:

The labels Vp1, Vp2,... Vpm are the entry points for the code that generates instances of the virtual pattern attributes V1, V2,..., Vm of Pn as shown in the right part of the above figure. This code computes the origin for the code and performs the instantiation.

Part-objects table This table holds information about the location of all part/static objects of the object. This table is used during generation of an object and during garbage collection.

Dynamic reference table This table holds information about all locations in an object which holds a reference. This table is used during garbage collection.

26.6 Code layout

For each descriptor, the following code is generated:

```
Gp:   save return information
      <allocation code>
      return
Mp:   save return information
      <do part code>
      return
```

The allocation code is executed when an instance of descriptor is generated. The do-part code is executed when the object is executed.

Control structures The code for control structures resembles that found in most other languages.

Evaluations and procedure calls Simple infix expressions like a+b*c are handled as in most languages. The general evaluations are more complicated to handle. Consider the following example:

```
F1: (# X,Y,Z: @ integer enter(X,Y) do ... exit(Y,Z) #);
F2: (# A,B,C: @ integer enter(A,B) do ... exit C #);
Q: @ integer;
...
(E1,E2) → F1 → F2 → Q
```

The following code is generated:

1. An instance of F1 is generated.
2. The evaluation E1 is evaluated and assigned to X of the F1 instance.
3. The evaluation E2 is evaluated and assigned to Y of the F1 instance.
4. The do-part of the F1 instance is executed.
5. An instance of F2 is generated.
6. The exit element X of the F1 instance is evaluated and assigned to A of the F2 instance.
7. The exit element Y of the F1 instance is evaluated and assigned to B of the F2 instance.
8. The do-part of F2 is executed.
9. The exit element C of the F2 instance is evaluated and assigned to Q.

If e.g. F1 is a reference to an object instead of a pattern, then step 1 is not present. Similarly if F2 is a reference and not a pattern, then step 5 is not present.

The enter/exit elements may be arbitrary evaluations instead of simple integer objects as in the above example. Consider the example

```
F: (#   X: @ (# A,B: @ integer enter (A,B) do ... #);
        T: (# U: @ integer enter U do ... #)
   enter(X,T) do ... #)
...
((E1,E2),E3) → F
```

For this evaluation the following code is generated:

1. An instance of F is generated.
2. The evaluation E1 is evaluated and assigned to X.A.
3. The evaluation E2 is evaluated and assigned to X.B.
4. The do-part of X is executed.
5. An instance of T is generated.
6. The evaluation E3 is evaluated and assigned to U of the T-instance.
7. The do-part of the T-instance is executed.
8. The do-part of the F-instance is executed.

Address computation For most block structured languages a unique block level and offset can be associated with the declaration of a variable. This is not possible in BETA due to the possibility of allowing subpatterns of a pattern to be defined at arbitrary block levels. Remember that this is not allowed in Simula, which can use the above mentioned addressing scheme. Consider the following example:

```
(# T:   (#   X: @ Integer;
             B: (# Y: @ Integer do X1→Y #);
        do   111→X2
        #);
   TT: T(# do 222→X3 #);
   A:  (#   TTT: T (# do 333→X4 #);
            TTTT: TTT(# do (# do 444→X5 #)#)
        #)
#)
```

In this example the names T, TT and A are declared at block level 0, the names X, B, TTT and TTTT are at block level 1, and the name y is at block level 2. The five different applications of X have been marked with a subscript in order to distinguish them.

In most block structured languages the address of X would be block-level 1 and an offset within the T-object, say 12. For the X attributes denoted by X1, X2, and X3 this is sufficient. This will, however, not work for X4 and X5 since the X attributes denoted here are not at block level 1.

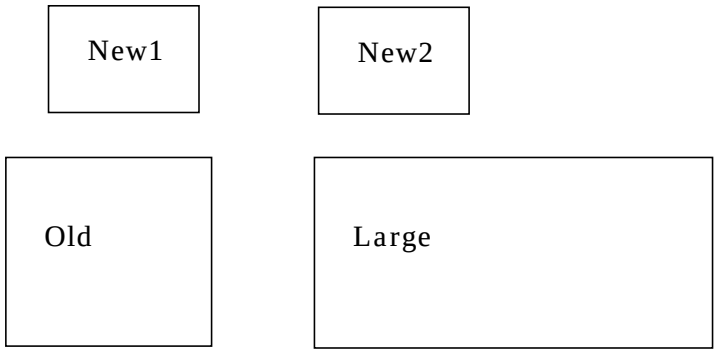
In [Kro79] a scheme for handling BETA addresses was proposed. This scheme was, however, very complicated. In the current BETA implementation a much more simple addressing scheme is used.

For each application of a name, X, there is a fixed distance of block levels to the enclosing object containing X as an attribute. The number of block levels to traverse is computed in the ON semantic attribute described for nameApl in section 26.3.1. The PN semantic attribute is the number of superpattern levels to traverse to reach the attribute. The OFF attribute is the relative offset of the attribute. The offset is the same for all applications of X.

For a given nameApl, X, path(X) is the pair (on,pn) that is the search path from X to its declaration. For the above example, we have that path(X1)=(1,0), path(X2)=(0,0), path(X3)=(0,1), path(X4)=(0,1) and path(X5)=(1,2).

26.6.1 Storage management

Objects are allocated dynamically in a number of heaps. A garbage collector is used to collect memory space occupied by objects that are no longer referred to. The garbage collector is based on generation based scavenging [Ung84] and mark/sweep [Tho76]. The following memory areas are used:



Initially new objects are allocated in New1. When New1 is full, live objects in New1 are copied to New2. Then the roles of New1 and New2 are swapped and allocation of new objects takes place in New2. When New2 is full, live objects in New2 are copied to New1, etc.

When objects have survived a number of garbage collections, they are copied from the New areas to Old. When Old becomes full, a mark/sweep is performed on Old. If only a small portion of Old is reclaimed, the size of Old is dynamically extended.

The Large area is used for allocation of large value repetitions. Since the objects in Large are big, a free-list of unused memory is kept. The free-list is then used for allocation of new objects. This may result in a fragmentation of Large. When a new object cannot be allocated using the free-list, a compaction is made in order to reclaim possible fragmented areas. If the compaction only reclaims a small amount of memory, then Large is dynamically extended.

Various other tables for keeping track of objects referring from Old to New or Large, etc. are also used by the memory management.

The memory management and garbage collection are controlled by various parameters. These include the size of the New areas, the size of blocks for extending Old and Large, the age an object must have in order to be moved to Old, etc.

26.7 Conclusion

The current implementation of BETA has reached a stage where BETA can be used for industrial programming. A number of improvements of the implementation can still be made. These include a faster compiler, and a better code generation. Also incremental compilation, interpretation and dynamic linking are facilities that will enhance the usage of the Mjølner BETA System. A number of these activities are currently taking place.