MetaBETA

Model and Implementation

René W. Schmidt

Department of Computer Science
University of Aarhus
DK-8000 Aarhus C, Denmark

email: rws@daimi.aau.dk

Abstract

Object-oriented programming languages are excellent for expressing abstractions in many application domains. The object-oriented programming methodology allows real-world concepts to modelled in an easy and direct fashion and it supports refinement of concepts. However, many object-oriented languages and their implementations fall short in two areas: dynamic extensibility and reflection.

Dynamic extensibility is the ability to incorporate new classes into an application at runtime. Reflection makes it possible for a language to extend its own domain, e.g., to built type-orthogonal functionality. MetaBETA is an extension of the BETA language that supports dynamic extensibility and reflection. MetaBETA has has a metalevel interface that provides access to the state of a running application and to the default implementation of language primitives.

This report presents the model behind MetaBETA. In particular, we discuss the execution model of a MetaBETA program and how type-orthogonal abstractions can be built. This includes presentation of dynamic slots, a mechanism that makes is possible extend objects at runtime. The other main area covered in this report is the implementation of MetaBETA. The central component of the architecture is a runtime system, which is viewed as a virtual machine whose baselevel interface implements the functionality needed by the programming language.

Contents

List of Figures

List of Tables

1 Introduction

Reusability is a main ingredient in most scientific and technical disciplines. Scientific theories are building blocks for a deeper understanding of various systems, well-engineered technical devices serve as building blocks for larger systems, and software components should in general be able to form part of multiple end-user applications. The keywords are modularity, extensibility, and tailorability.

Structured programming turned modularity into the single most important program design principle. Extensibility and tailorability are major reasons for the recent success of object-oriented programming. It allows the programmer to model parts of the real-world and to reuse and refine previously created models. This makes object-oriented programming languages excellent for expressing abstractions in many application domains. However, many object-oriented languages and their implementations fall short in two areas: dynamic extensibility and reflection.

Dynamic extensibility is the ability to incorporate new classes into an application at runtime. Many languages only allow the use of extensibility at compile-time, i.e., by creating a new class by using subclassing. With dynamic extensibility, it is possible to use running applications as building blocks for more specialized applications. For example, a hypermedia application can be extended at runtime with support for a drawing application [Grønbæk & Malhotra 94]. Dynamic extensibility is normally achieved by dynamic loading and linking or by interpretation (e.g., emacs [Stallman 81]).

Reflection (or metalevel computation) makes it possible for a language to extend its own domain. Type-orthogonal functionality can be expressed directly in the language, i.e., functionality that depends on the *type* of an object. For example, functionality such as persistent storage, generic object copying, and distribution to be implemented. Common for these functionalities are that they all depend on metainformation and possibly implementation details. Without reflective capabilities, the programmer is forced to use source code preprocessing, special compilers, or new runtime systems to implement such functionalities [Brandt & Schmidt 96].

The lack of these two facilities are mainly due to the implementation of a language. By providing appropriate standardized interfaces as opposed to new language features both dynamic extensibility and reflection can be supported. The challenge is to find a consistent, expressive, and simple interface that can be supported across multiple operating systems and hardware architectures and to built an efficient implementation.

MetaBETA is an extension of the BETA language [Madsen et al. 93] that supports dynamic extensibility and reflection. MetaBETA has has a metalevel interface (MLI) that provides access to the state of a running application and to the default implementation of language primitives. MetaBETA can be viewed as an open implementation [Kiczales 92] of the BETA language, since certain implementation aspects are exposed.

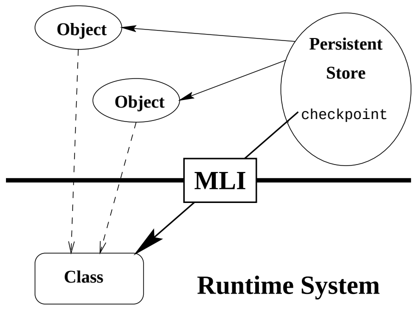


Figure 1: A Persistent Store using runtime metalevel information.

Figure 1 depicts a persistent store utilizing the MLI to access type information, thereby being able to serialize arbitrary objects to stable storage. The persistent store has references to a set of objects that are the roots of the object graph that is to be persistent. The `checkpoint` operation uses the MLI to access detailed information about the internal structure of the objects, thereby the persistent store can serialize objects in a type-independent way.

The MLI is runtime based, so the persistent store can serialize any object, independent of its class and creator. Access to source code is not necessary. The ability to serialize objects in an implementation independent manner is of major importance when implementing portable type-orthogonal abstractions. The MLI provides an efficient high-level typed interface to low-level implementation details.

Access to object layout is one requirement for implementing type-orthogonal abstraction. However, this is not always enough. For example, a persistent store implementation needs to associate additional state with the objects it works on, e.g., it needs to maintain a unique object-identifier for each object that is persistent. In a strictly class-based language it is not possible to add extra information to an object at runtime. To circumvent these limitations, it is often necessary to look up information in auxiliary data-structures or to sacrifice type-orthogonality [Schmidt 96]. In this report we present dynamic slots, which are a solution to this problem and allow type-orthogonal abstractions to be written in a simple and efficient way.

The other main area covered in this report is the implementation of MetaBETA. The implementation is based on a metalevel architecture that defines how MetaBETA programs are executed and how reflection and dynamic extensibility interacts. The central component of the architecture is a runtime system, which is viewed as a virtual machine whose baselevel interface implements the functionality needed by the programming language. The metalevel interface queries and alters the way the runtime system executes a MetaBETA program.

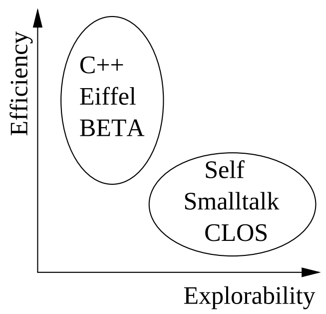


Figure 2: Efficiency versus explorability.

Viewing the runtime system as the central execution engine is similar to the view taken by dynamically typed[1] languages, such as Smalltalk [Goldberg & Robson 89], Self [Ungar & Smith 87], and CLOS [Kiczales et al. 91]. These languages are based on integrated programming environments that are responsible for the execution of programs, thereby supporting an exploratory programming style. This is in sharp contrast to most statically typed languages, where the development environment and program execution environment are separated. Statically typed languages, on the other hand, provide more readable programs through explicit type annotations, and they typically produce more efficient code. This difference is shown in Figure 2.

The Self project has demonstrated that advanced compiler technology can be used to significantly increase the performance of dynamically typed languages without sacrificing explorability, moving Self up the Efficiency axis. Conversely, the metalevel architecture can potentially move a statically typed language to the right on the Explorability axis, since runtime manipulations of programs are made possible. Hence, the architecture allows a clean separation between development environment and runtime environment, while narrowing the gap between explorability and efficiency.

The rest of the paper is organized as follows. The next section presents the approach taken by MetaBETA and introduces the metalevel interface and the metalevel architecture. Section 3 elaborates on how type-orthogonal abstractions can be constructed and presents dynamic slots in detail. Section 4 presents the implementation of MetaBETA. Section 5 relates MetaBETA to other reflective systems and languages. Section 6 outlines future research directions. Finally, Section 7 presents our conclusions.

[1] With dynamically typed we mean that most type-checks are done at runtime.

2 The MetaBETA Approach

The MetaBETA project was initiated to solve several extensibility and tailorability problems with the current BETA implementation. These problems was recognized during the design and implementation of a number of substrate systems for the Mjølner BETA system, including: type-orthogonal persistence [Brandt 94, Grønbæk et al. 94], distributed object system [Brandt & Madsen 94], an embedded interpreter [Malhotra 93], a source-level debugger, and object- and class-browsers.

These tools manipulate objects in a type-independent manner and currently depend on the memory layout of objects, as well as other implementation details. Such dependencies severely compromises portability and type-safety. A common trait of these tools is that they all work at the metalevel, using parts of a *running* BETA program as data. The design of MetaBETA was initiated to identify the core metalevel functionality needed by these systems and to support those needs through a type-safe metalevel interface.

The approach taken in MetaBETA is strongly inspired by the *open implementation* approach [Rao 91, Kiczales 92]. An open implementation is an implementation that not only provides a functional interface but also an implementational interface. It is based on the observation that often a framework might provide (close to) the right abstraction, but certain implementation decisions prevent one from using it. These implementation decisions are known as mapping dilemmas — they have to be made in the transition from an abstract model to a concrete implementation. One way to deal with these mapping dilemmas is to represent them explicit in the implementation. Thus, a framework should provide two interfaces, a *baselevel interface* that defines its behaviour, and a *metalevel interface* that exposes certain parts of its implementation [Kiczales & Lamping 93]. An example, of an open implementation is Mach's virtual memory system and external pagers [Accetta et al. 86]. The baselevel interface allows allocation of virtual memory, where as the metalevel interface (external pagers) makes it possible to control the mapping between physical and virtual memory. The MetaBETA metalevel interface can be viewed as an abstract interface into the implementation of the language and to the implementation of the executing program itself.

In the following we will first outline the design and rationale of the metalevel interface (MLI) to give a general understanding of it. A detailed description of the MLI can be found in [Brandt 95]. To support the metalevel interface, a metalevel architecture has been designed [Brandt & Schmidt 95, Brandt & Schmidt 96] that provides a conceptual model of metalevel programming and extensibility for MetaBETA. That model will be described in detail in Section 2.2.

2.1 The Metalevel Interface

The metalevel interface is a set of primitive operations that access and alter runtime behaviour of a program. They are implemented directly by an underlying runtime system.

Through the metalevel interface the programmer has access to low-level implementation details, such as object layout, e.g., the offset of an integer attribute in an object, and the implementation of method invocation. However, these low-level implementation details are abstracted into high-level concepts using first-class type-informations and runtime events. In this way, metalevel code is easier to reason about, portable, type-safe, and fits natural into the BETA programming language.

The metalevel interface is divided into three different categories: introspective, intercessory, and invocational. In this report we will limit our focus to the first two categories. Readers interested in an in depth description of the MLI are refered to [Brandt 95].

2.1.1 Reifying Type-Information

A BETA program execution consists of: objects, patterns, and attributes. An object is an instance of a pattern and a pattern is a collection of attributes. In order to do reflection, i.e., for a BETA program to talk about itself, patterns and attributes must be explicit represented at runtime in a similar way as objects are. They must be reified — be made into something concrete that can be referenced and named. Reifying patterns and attributes is the same as reifying type-information, since patterns denote types and attributes denote both methods and instance variables.

Patterns are already reified by *pattern references* in BETA. Pattern references make patterns first-class values. To name attributes, attribute references are introduced in MetaBETA. An Attribute reference is a dynamic reference to an attribute, i.e., it can point to different attributes at different points in time. This makes it possible to have explicit references to both objects, patterns, and attributes.

In order to make the metalevel interface as simple and elegant as possible, several extensions to the BETA type-system have been proposed. In particular, a broader range of pattern qualifications [Brandt & Knudsen 96] and attribute references as first-class values [Brandt 95]. These extensions to BETA are not strictly necessary to implement the MLI. In this report we will restrict ourself to standard BETA syntax and semantics. The benefits of this restriction is that the design can be implemented without requiring major changes to the current compiler, as well as showing that the existents of a metalevel interface does not require new language features. The drawback is that the metalevel interface is not as elegant and it is not possible to achieve the same degree of static type-checking.

```
AttributeRef:
  (#
     o: ^object;
     aChar:    (# c:@char    enter c do ... exit c #);
     aReal:    (# r:@real    enter r do ... exit r #);
     aInteger: (# i:@integer enter i do ... exit i #);
     aBoolean: (# b:@boolean enter b do ... exit b #);
     aRef: (# r:^object  enter r[] do ... exit r[] #);
     aStruct:  (# s:##object enter s## do ... exit s## #);
     exec: (# do ... #);
  enter o[]
  exit o[]
  #);
```

Figure 3: Simulating an attribute reference.

Attribute references will be simulated by using the `AttributeRef` pattern[2] shown in Figure 3. Given an `AttributeRef` object, `ar`, that names an integer attribute of the object, `o`, is it possible to assign a value to the attribute by executing: `o[]->ar; 7->ar.aInteger`.

The pattern encapsulates the implementation-specific offset of an attribute inside an object and provides type-safe access to the attribute. A valid `AttributeRef` object can only be manufactured by the metalevel interface. A programmer does not have access to the internal implementation of an attribute reference object. In the case of the simulated attribute access, all type-checking is done at runtime. However, as shown in [Brandt 95], is it possible to statically type-check attribute references

2.1.2 Introspective Capabilities

The introspective part of the MLI provides a way to obtain runtime access to class-definitions and type-annotations as they exist in the source code. Metainformation is represented as pattern and attribute references. The MLI allows extraction of complete object descriptions modulo the do-parts.

The main functionality of the invocational capabilities is the introspection of pattern and attribute references:

- **Introspecting Attribute References:** Given an attribute reference, the BETA MLI allows extraction of the name of the attribute, the qualification of the attribute, and the attribute kind. Qualifications are returned as dynamic pattern references that, e.g., can be compared against known patterns to locate attributes of specific types in an object.

- **Introspecting Pattern References:** Given a pattern reference, `p`, it is possible to scan the attributes that exist in instances of `p`. For each attribute of `p`, an attribute reference, `ar`, is returned.

Other parts of the MLI scans enter and exit parts, externalize patterns and attribute references, scans and externalize call-stacks, and maps patterns to an abstract syntax tree representation of the source code.

For example, the `scanAttr` method in the MLI provides a way to scan all attributes of a pattern:

[2] In the pattern shown in Figure 3 is access to BETA repetitions left out for simplicity.

```
scanAttr:
  (# current: ^AttributeRef;
     o:^object;
  enter o[]
  do ... INNER ...
  #);
```

To print the names of all the attributes of the `AttributeRef` object,`ar`, we can write:

```
ar[]->MLI.scanAttr (#
do current[]->MLI.AttrName->screen.putline;
#);
```

This will print out the text strings `o`,`aChar`, `aReal`, and so on. The `MLI.AttrName` method returns the name of an attribute pointed to by an attribute reference.

2.1.3 Intercessory Capabilities

The intercessory capabilities opens up the runtime system, providing access to the implementation of language primitives. The MLI provides an abstract interface to the implementation of object creation, object execution, and object destruction using reflectors.

```
ExecReflector:
  (# type:< Object;
     onExec:<
       (# callNextReflector: (# do ... #);
       enter (method:^type, dp:##Object)
       do INNER
       #);
  #);
```

Figure 4: The reflector for object execution.

Runtime events are used to reify the implementation of these primitive operations. A reflector is an ordinary programmer-defined BETA object that can be hooked into the runtime system. It will be invoked whenever a given *event* occurs. MetaBETA defines events associated with object creation, object invocation, and object reclamation. Figure 4 shows the reflector for object execution. An instance of the pattern can be associated with a given do-part by calling the `ExecRegister` method in the MLI. The runtime system will then transfer the control of execution to the `onExec` method of the given reflector each time the given do-part is about to be executed.

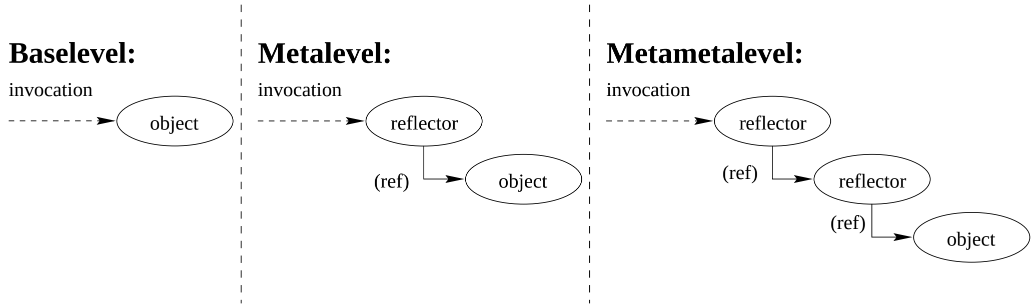


Figure 5: Reflectors and metalevels.

Figure 5 depicts the use of reflectors to control object invocation. Normally (baselevel), an object is invoked directly. If a reflector is associated with the invocation-event for a given object, the reflector is invoked instead of the object itself. The reflector is responsible for invoking the object, but is not required to do so. Because a reflector is an ordinary BETA object, it is possible to associate a reflector with a reflector-invocation-event, allowing, in effect, metametalevel computation and so on.

2.2 The Metalevel Architecture

Introducing a metalevel interface for a language means that much more information must be managed at runtime. For example, information about class definitions and the class hierarchy. Also, because the MLI allows the behaviour of language primitives to be changed at runtime, a flexible runtime structure is needed. The metalevel architecture is a structuring of a compiled and statically typed language that allows just that — by viewing the language runtime system as a virtual machine.

Informally, a runtime system is the part of a language implementation that is responsible for executing a program. For dynamically-typed languages and semi-interpreted languages, such as Smalltalk and CLOS, the runtime system is an active entity that is responsible for the execution. The runtime system interprets source code, performs method look-up, maintains variable bindings, performs memory management, and checks for runtime errors. For most statically-typed languages no active runtime system is used. The responsibility of executing the program is delegated to the compiler, which directly outputs executable code. The runtime system is a passive entity, that includes certain functionalities that the compiled program needs at runtime. For an object-oriented language this would typically include object instantiation, runtime type checking, and garbage collection.

The metalevel architecture reintroduces the active runtime system for a statically-typed and compiled language. Thereby gaining a runtime structure that not only focuses on the functional aspects of a program, but also on the implementational aspects, i.e., the metalevel aspects. This is achieved by moving responsibilities from the compiler and linker to the runtime system.

2.2.1 The Virtual Machine Model

We think of the runtime system as a virtual machine extending the underlying operating system. In the same way as binary programs may be considered data to be executed by the CPU, we can think of compiled programs as data to be executed by the runtime system. The MetaBETA runtime system in concert with the hardware and operating system defines a complete MetaBETA virtual machine that takes compiled MetaBETA programs as data and executes them. Conceptually, the runtime system is responsible for executing the compiled code. In practice, it hands the machine-code part of the compiled program to the underlying processor for fast execution.

This architecture marks a departure from the efficient-code-generator view of a compiler. The compiler is instead viewed as a supplier of information to the runtime system about a program. This information is a representation of the source code that allows efficient execution and, at the same time, maintains a mapping to the source code, i.e., the metalevel information.

The virtual machine view of the runtime system induces a different system structure than for traditional statically-typed languages. For a traditional compiled language (Figure 6a), the application runs directly on top of the operating system, supported by the runtime system and libraries. The runtime system, operating system, and libraries are black box implementations, i.e., they can only be accessed through functional (baselevel) interfaces. Libraries also do not have any direct contact with the runtime system and are therefore not able to access runtime information. In the metalevel architecture (Figure 6b), the runtime system is the central component. Its main responsibility is loading, linking, and running the application. By having control over these aspects, it can provide detailed information about the current state of the program execution to the application and libraries through its metalevel interface. The runtime system and libraries can also make use of a metalevel interface provided by the operating system [Kiczales & Lamping 93].

The definition of a runtime system as a virtual machine for the MetaBETA language pins down the contents of the baselevel interface as the needs defined by the language and its compiler, e.g., memory management and runtime type-checks. The runtime system should be designed to put as few restrictions as possible on the ways it can be extended, i.e., it should be minimal in the sense that only functionality that cannot be implemented as libraries are located in the runtime system. In this sense, the runtime system is similar to a micro-kernel operating system.

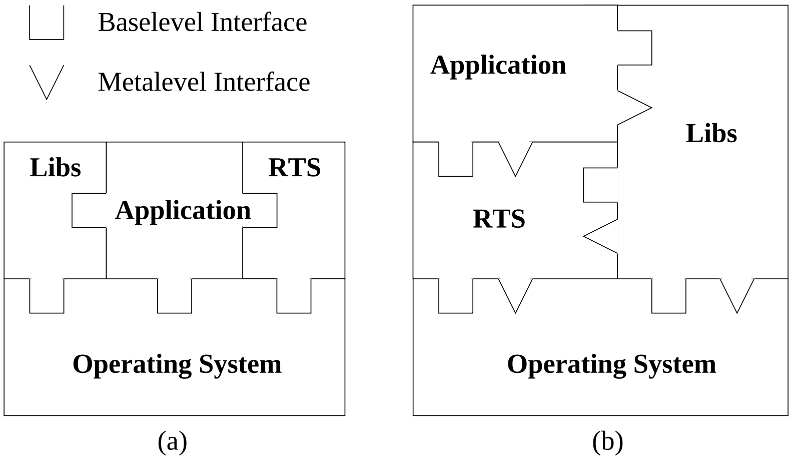


Figure 6: The traditional and metalevel system structure.

2.2.2 Executing Metalevel Code

Executing a method in the MLI means invoking a primitive operation defined in the runtime system. Hence, all metalevel computations are carried out in the runtime system. The result of the computation is then lifted back up into the current MetaBETA execution, typically in the form of a pattern or attribute reference. In other words, the metalevel is isomorphic, the result of executing a MLI operating in MetaBETA is also in the domain of MetaBETA. By having an isomorphic metalevel interface, we avoid a tower of reflective languages each describing the program execution at more specialized level [Masuhara et al. 92].

2.2.3 An Extensible Execution Model

The runtime system in the metalevel architecture is the only executing program seen by the operating system. To execute a compiled BETA program, the runtime system is started from the operating system command-line with parameters that specify the application's main pattern. This is in contrast to a traditional compiled language where the compiler generates a monolithic executable image for each application.

MetaBETA program executions are based on an extensible model, similar to the models described by [Agesen et al. 89, Malhotra 93]. The runtime system defines the outer-most block-level for all program executions. To execute an application, the runtime system is *extended* with the patterns describing that particular application. The `LoadFragment` method in the runtime system extends the program execution with new substance, i.e., MetaBETA source fragments.

The outer-most block-level object is shown in Figure 7. It contains the `BVM` part object that defines the interface to the virtual machine, which includes the metalevel interface and an interface to dynamic loading of code. The extensibility point is explicitly denoted in the source code with the `SLOT` primitive.

Extensibility in this model means increasing the name space of the current program execution. The `LoadFragment` does not return any values. Access to newly loaded patterns is done through the MLI. This design reflects that dynamic extensibility is a metalevel operation. The way extensibility and reflection interacts is illustrated in the do-part of Figure 7. Firstly, `LoadFragment` is called to extend the current program execution with a fragment which name is specified on the command-line. This operation increases the set of visible names in the `Execution` object. Secondly, the MLI is used to instantiate and execute a specific pattern nested inside the `Execution` object.

A direct benefit of this organization, is that invoking `LoadFragment` does not necessarily loads any coded. That can be postponed until it is first accessed via the MLI. This is in contrast to many systems where dynamically extensibility and dynamic loading are synonymous. Separating extensibility and dynamic loading provides a simpler programming model and can potentially increase code reuse. Firstly, source code organization and dynamic loading is orthogonal. The MetaBETA loader can dynamically load code in units of an object-descriptor, with no constraints where it is defined. For example, already existing source-code

```
Execution:
(#
   Object: (# do INNER #);

   BVM: @
    (#
       MLI: @(# ... #);

       LoadFragment:
          (# name: ^text;
          enter name[]
          #);
       ...
    #);

    <<SLOT lib:attributes>>
do
   (* Load and execute the pattern given as command-line argument *)
   2->arguments->BVM.LoadFragment;
   (3->arguments,this(Execution))->BVM.MLI.execNamedAttribute;
#);
```

Figure 7: The Virtual Machine object.

can exploit fine-grained dynamic linking without being restructured. Secondly, the MLI is used to uniformly access the metainformation. Instead of having the dynamic loader defining its own interface to access newly loaded code, the MLI is used to access this code. Thirdly, loading on demand can be used for metalevel information, so only minimal overhead is paid for programs that does not use the MLI.

2.3 Summary

In this section we presented an overview of MetaBETA, i.e., the metalevel interface and the metalevel architecture for BETA. The MLI makes the BETA language an open implementation, where the behaviour of language primitives can be changed at runtime and detailed information about object layout can be retrieved.

The metalevel architecture provides an extensible execution model for MetaBETA, by clearly defining the responsibilities of the runtime system. In particular, the runtime system is responsible for loading and linking code. By doing this, it has detailed knowledge of code and object layout. This information can be used to load metainformation on demand and to insert reflection hooks by patching executable code.

3 On the Implementation of Type-Orthogonal Abstractions

One of the goals of MetaBETA is to be able to construct type-orthogonal abstractions directly in the language, i.e., abstractions that work on any object with no regard to its type. For example, persistence, distribution, and the ability to insert profiling and debugging hooks.

Consider the design and implementation of object persistence. Object persistence is a concept that allows: (i) objects to be checkpointed to stable storage during the execution of a program, and (ii) to reinstantiate the objects from their checkpointed form into a possibly new program execution. Object persistence should ideally be modelled, using one or more classes, in such a way that it is possible to apply persistence to all objects. Modelling the ability to be persistent by a subclass relationship violates the type-orthogonality.

Instead, persistence can be modelled as a runtime association from a *PersistentStore* object to the object to be persistent. Hence, an object is persistent if it is in the transitive closure of the objects associated with the PersistentStore. However, the PersistentStore still needs to associate additional state with the objects it works on, e.g., it needs to maintain a unique object-identifier for each object that is persistent. In a strictly class-based language is it not possible to add extra information to an object at runtime. This can have several negative consequences when implementing type-orthogonal abstractions: (i) unnecessary limitations, such as a class is required to be of a certain (sub)type to be used by a particular abstraction; (ii) unsatisfactory performance, because the extra information must be found in auxiliary data-structures; or (iii) the design of special-purpose programming languages, that include tailored support for a specific feature, e.g., the Emerald [Jul et al. 88] language is designed and implemented with compiler and runtime support for distribution.

A metalevel interface makes it possible to express functionality that depends on the type of an object, i.e., in contrast to only on the state of an object. Another requirement when implementing type-orthogonal abstractions is to express the functionality in a manner that does not rely on implementation details or require changes to the language implementation, i.e., the language implementation must be *open* for extensions. Several languages, including CLOS [Kiczales et al. 91], Open C++ [Chiba 95], and MetaBETA provides such capabilities.

However, as illustrated above, some type-orthogonal abstractions may need to associate additional information to the objects they work on. A type-orthogonal facility to extend objects is needed to allow for maximum flexibility and efficiency of metalevel programs. In this section, we propose a mechanism for dynamically extending objects at runtime with additional attributes, called *dynamic slots*[3]. Dynamic slots are orthogonal to the class hierarchy. A dynamic slot can be added to any object without regard to its type. Dynamic slots are static declared in the source code, hence statically type-checking is possible, but storage is not allocated until their first use.

The rest of this section is organized as follows. The next section describes dynamic slots in detail, and gives several real-word examples of how they are applicable. Section 3.2 presents how they can be implemented efficiently in a statically-typed language. Section 3.3 evaluates our prototype implementation of dynamic slots for BETA. Finally, Section 3.4 summarizes our conclusions.

3.1 Dynamic Slots

A dynamic slot is an attribute that conceptually exists in all objects and for which storage is only allocated if the dynamic slot is used, i.e., if a value is assigned to it. Conceptually, we can view all classes as having a common superclass, typically called *Object*. Adding behaviour or substance to this common superclass will affect all classes and therefore all objects in the system. Hence, creating a dynamic slot is semantically equivalent to adding a new object reference to class Object. A dynamic slot exists in all objects independently of whether an object was created before or after the dynamic slot was created.

Each dynamic slot is represented as an instance of the `DynSlot` class, shown in Figure 8. It contains three methods: `set`, `get`, and `init`. An object reference is assigned to a dynamic slot by the `set` method. It takes as parameters: an object, `o`, which contains the slot we want to access, and the new value for the slot, `value`. For example, a BETA assignment of the form: `newVal[]->o.mySlot[]` is written as `(o[], newVal[])->mySlot.set` for a dynamic slot The value of a dynamic slot can be read by using the `get` method, which given an object, returns the value of the slot.

Before a new dynamic slot can be used, it must be initialized with the `init` method. In a sense, `init` is a metaobject method (or class-method) that adds the slot-definition to class Object and initializes the slot to `NONE` for all objects. The following code fragment shows a declaration of a dynamic slot `psSlot` that is of type: reference to a `psInfo` class.

```
psInfo: (# OID: @integer #);
psSlot: @DynSlot(# type::psInfo #);
```

In the example, the virtual `type` parameter is specialized to `psInfo`. This tells the compiler to statically check that a value assigned to `set` is at least a `psInfo` object, and that it can safely assume that a value returned by `get` is at least a `psInfo` object. Thus, no runtime type-checking is needed for a dynamic slot. Dynamic slots are statically typed and dynamically allocated.

Notice that it is the ability to have generics that makes it possible to implement the dynamic slot interface using a class and still have static type-checking. In BETA, this is done using a virtual class. In C++ [Stroustrup 93], the template mechanism could be used. The drawback of modelling dynamic slots as a

[3]Slots are similar to instance variables in Smalltalk [Goldberg & Robson 89], and dynamic object references in BETA. We use the term *dynamic slot* to disambiguate them from standard BETA references.

```
DynSlot:
  (# type:< Object;                   (* virtual class *)

     set: (#                          (* set method *)
      o: ^object;
      value: ^type;
      enter (o[], value[])
      do ...
      #);

     get: (#                          (* get method *)
      o: ^object;
      value: ^type;
      enter o[]
      do ...
      exit value[]
      #);

     init: (# do ... #);              (* init method *)
  #);
```

Figure 8: Declaration of the DynSlot class.

class is that access to dynamic slots and ordinary BETA attributes is different. In a language that supports operation overloading, e.g., C++, this could potentially be avoided. Another approach could be to extend the language definition with a special syntactical construct for dynamic slot declaration. Attribute references could be used in BETA to refer to the dynamically created attributes. Our approach is similar to LISP property lists [Steele 84], which also uses special access methods.

3.1.1 Examples

To further motivate dynamic slots, we will look at three examples where they are applicable. Based on these examples, we will argue that the kind of runtime extensibility provided by dynamic slots are indeed applicable and useful in a compiled language, and that the same kind of functionality cannot easily be provided at compile/link time.

The first two examples are based on the current implementations of object persistence [Brandt 94] and distribution [Brandt & Madsen 94] for the Mjølner BETA System. The last example describes how dynamic slots make it possible to extend an existing framework in a way that is not possible by using subclassing alone.

Persistent Storage

The persistence model for BETA is based on reachability. A set of objects are registered with the persistent object store (PStore) as persistent roots. When the *checkpoint* operation is invoked on the PStore, the transitive closure of the root objects is serialized to stable storage. The PStore is completely type-orthogonal, i.e., it can serialize any object with no regard to its type.

To make it possible to reconstruct objects anywhere in memory when objects are restored from their serialized form, object references are converted to unique object IDs (OIDs) during serialization. Each object is assigned an OID by the PStore. Converting an object reference into an OID requires a linear scan of a list containing pairs of object references and OIDs. An operation whose execution time is proportional to the number of objects that have been serialized. This is because it is, in general, impossible to hash on an object reference in a garbage-collected language, since most garbage collectors move objects around in memory. Also, the OIDs cannot be stored directly in an object, because the object can be of an arbitrary type. The linear scan is a major source of overhead in the current implementation.

By using dynamic slots, the OID for an object can be stored directly with the object, so the linear scan is replaced by a constant time operation. Thus, the time complexity is asymptotically improved. Figure 9 shows how an OID look-up method (`getOID`) can be implemented with dynamic slots.

```
PStore:
  (# psInfo: (# OID: @integer #);        (* extra information *)
     psSlot: @DynSlot(# type::psInfo #); (* dyn. slot for PStore *)
     next_OID: (# ... #);                (* returns a new OID *)

     (* getOID method *)
     getOID:
       (# info: ^psInfo;
          o: ^object;
       enter o[]
       do o[]->psSlot.get->info[];       (* access dyn. slot of o *)
          (if info[]=NONE then
              &psInfo[]->info[];         (* create new psInfo *)
              next_OID->info.OID;
              (o[],info[])->psSlot.set;  (* assign to dyn. slot *)
          if);
       exit info.OID                     (* return OID *)
       #);

      (* Init method *)
      init: (# do psSlot.init; #);
   #);
```

Figure 9: Using dynamic slots to store object identifiers.

The main advantage of using dynamic slots in this situation comes from the fact that objects are extended on demand. Dynamic slots will only be allocated for objects that are persistent. This same property is difficult, if not impossible, to achieve through compile or link time program modification. It is not known at compile-time which objects are persistent, since any object reference could potentially be handed to a PStore. A link-time analysis can determine if the persistent store is linked in, and in that case conservatively extend *all* objects with an OID attribute. Even if the link-time analysis would be able to come up with an accurate minimal set of objects that possible could be persistent during a program execution, it might be more efficient to use dynamic slots. For example, in the case of an interactive program where the persistent store is only used under very rare circumstances. The implementation using link-time analysis will then on average require a larger memory overhead than the implementation using dynamic slots.

Distribution

The distribution library provides transparent access to remote objects, i.e., objects located in another process that possibly exists on another physical machine. Transparency is provided by proxies [Shapiro 86]. A remote object is represented by a proxy object in the caller's address-space, which forwards all calls to the remote object. For the caller, the proxy is indistinguishable from the remote object.

A request is forwarded to the remote object by serializing the method invocation along with its parameters and sending it to the process where the remote object resides. This serialization is similar to the PStore implementation. In addition to an OID, the distribution library also needs to maintain state information that tells if an object is acting as a proxy and which communication channel to use.

In contrast to the PStore where OIDs are only needed for the checkpoint operation, the distribution library needs the information on every call on a remote object. Therefore, a linear scan to locate the extra state information is unacceptable. Instead the current implementation requires a distributed object to be a subclass of the `remoteable` class, that contains the extra state information. In effect, the implementation is trading transparency away for efficiency. Dynamic slots can be used in a similar way in the distribution library as they were used in the PStore example, thereby removing the transparency limitation, while retaining efficiency.

Extending Frameworks

The last two examples discussed how dynamic slots are practical when implementing type-orthogonal abstractions. This example is more general than the preceeding two. It outlines how dynamic slots can be used to solve an extensibility problem in object-oriented software engineering.

Object-oriented software development is known for its support for extensibility and reuse. New subclasses can be derived to specialize and extend the behaviour of a system. For example, a graphical framework might describe a general shape with the class `Shape`. More specific shapes (e.g., triangles, squares, etc.) can easily be added to the system by subclassing `Shape`.

But what if we want to add some behaviour to all shapes in the framework? Then we have to modify the `Shape` class and recompile the entire toolkit. Unfortunately, this is only possible if the source code for the toolkit is available, which is typically not the case if the framework is supplied by a third-party vendor. Recompilation of an entire toolkit might also not be an option if the current version of the library is frozen.

Again, dynamic slots can be used to provide the extra substance needed in class `Shape` and in all its subclasses. No recompilation is necessary, the extra slots are created at runtime. Of course, using dynamic slots will have a higher runtime overhead than making the change directly in the `Shape` class, but it does provide a fairly efficient and direct solution to extending frameworks without having access to the source code. In the case where the source code is available, it makes it possible to quickly prototype a change without recompiling the entire toolkit.

3.2 Implementation

Since the main motivation behind dynamic slots is to enhance the performance of certain types of abstractions, an efficient implementation is important. However, of equal importance is that the implementation does not impose significant overhead on ordinary programs. Our requirements to the implementation are:

- Dynamic slots should not have any impact on the performance of applications not using them, i.e., you only pay for what you use.

- Dynamic slots are available on all objects, i.e., they are orthogonal to the type-system. Also, their use does not require special compiler-switches during source code compilation. These requirements ensure that, e.g., persistence is available on all objects, no matter if the code for an object was written with persistence in mind or not. This is especially critical when third-party libraries are used.

- The use of dynamic slots must scale. All dynamic slots are conceptually defined in class Object. Thus, the total number of slots in a system is the number of created slots times the number of objects. The space overhead must be proportional to the number of slots in *use*, not to the conceptual total number of dynamic slots.

We have integrated dynamic slots into the BETA programming language. The object layout used by the Mjølner BETA System is fairly standard for a garbage-collected object-oriented language. A similar implementation can be used for languages such as Eiffel [Meyer 92] and Modula-3 [Nelson 91]. In the following, the implementation will be described in more details.

3.2.1 Representing Dynamic Slots

The representation of dynamic slots at runtime must be space-efficient, so only slots in use take up space. Secondly, it must be efficient to add a dynamic slot to an object and to look it up. For each object, we store the non-zero slot values in a linked list. The `init` method assigns a unique dynamic-slot ID (DID) to each slot, which is used to identify a slot. The `set` operation adds a new element to the list, or removes an element if the `NONE` value is assigned. The `get` operation scans the list for an element with a particular DID.

Figure 10 shows an object with three dynamic slots stored in a linked list. The object has an attribute, *FirstSlot*, that points to the first dynamic slot. A slot only takes up space if it contains a value that is different from `NONE`, i.e., the memory overhead is proportional to the number of slots in use. Also, a slot can be easily located by following the pointer that is stored in the last word of the object.

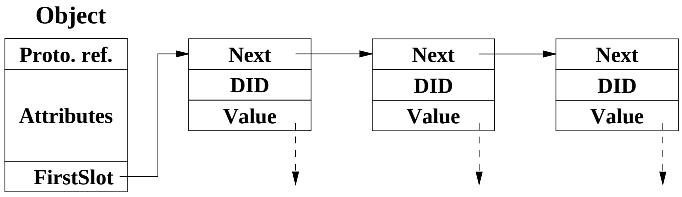


Figure 10: Representation of dynamic slots in an object.

Application	Compiler	Sif	Factory
Avg. IOA object size	32b	26b	27b
IOA allocation	1.89Gb	150Mb	373Mb
Avg. AOA object size	46b	65b	35b
AOA allocation	4400Kb	660Kb	2500Kb
AOA to IOA ratio	0.2%	0.4%	0.7%

Table 1: Allocation Statistics.

Access to a dynamic slot will naturally be slower than for statically allocated attributes, because it requires a traversal of a linked list instead of a direct memory access. We expect only a few slots to be associated with each object, so the overhead of scanning through a linked list will be small. If a large number of slots are in use for individual objects, it might be feasible to store the dynamic slot values in a hash table instead. The dynamic slot implementation could at runtime choose the optimal representation of the slot values to maximize its performance.

The Self system [Hölzle & Ungar 94] allows objects to be dynamically extended during runtime using the `addSlot` method. They allocate a new object with the additional slot, copy the contents of the old object to the new object, and update all pointers to point to the new object. This approach aim at fast access at the cost of a high setup time, since it might require recompilation of code and a scan of the heap to update pointers. In contrast, representing dynamic slots as a linked list aim at medium setup cost and a medium access time. Also, this scheme can easily be integrated into most compiled language implementation and requires no modifications to the compiler.

3.2.2 Extending Objects

The above representation of dynamic slots requires an extra word for the *FirstSlot* reference in all objects. Unfortunately, a trivial implementation which statically extends all objects with the extra word will add a significant time and space overhead to a program execution.

The memory behaviour of several BETA programs have previously been studied [Grarup & Seligmann 93] and is summarized in Table 1[4]. The table shows memory statistics for the BETA Compiler compiling itself (Compiler), an interactive editor for BETA source code (Sif), and a discrete-event simulation system (Factory). Based on these numbers, extending each object with an extra word will yield an allocation overhead from 12% to 16%, which for the Compiler means that it will request an additional 240Mb of memory. This increased memory allocation will result in a minimum of 480 additional IOA collections[5], thereby significantly degrade performance.

The ideal solution would be to only extend objects which use dynamic slots with the FirstSlot attribute. To approximate this behaviour, we have adopted a solution where the garbage-collector is modified to dynamically extend objects with the extra word at runtime. The worst-case space overhead can then never

[4]The current BETA implementation uses a generational-based garbage-collector, where copy-collection is used for the *Infant Object Area* (IOA), and a mark-and-compact collector is used for the *Adult Object Area* (AOA). For a more detailed discussion see [Grarup & Seligmann 93] or [Schmidt 96].

[5]Assuming that the size of the IOA area is 512Kb, which is the default. In Section 3.3 actual measurements results will be shown.

be worse than statically extending all objects with an extra word. An object in IOA can easily be extended by provoking an IOA garbage-collection, and then extending it when it is copied to To-Space. Objects are lazily extended when the first dynamic slot value is assigned to them. In our implementation, we set a bit in the prototype[6] indicating that an object must be extended, thereby extending all instances of a specific class at a time. This simplifies the implementation, because all objects of a given class have the same size, and it amortizes the cost of the IOA garbage-collection over all objects of a given class.

Extending objects in AOA is non-trivial, because these objects are garbage-collected using a mark & sweep algorithm that needs several passes of AOA in order to locate live objects and compact storage. Provoking AOA garbage-collections may introduce several disruptive breaks in the program execution, due to its potentially large size. Also, little garbage is likely to be found. To avoid this, we notice that only about 0.2% to 0.7% of all allocated objects are promoted to AOA. Thus, eagerly adding the extra word to all objects that are moved into AOA, will only introduce an insignificant allocation overhead. In fact, based on the numbers from Table 1, an allocation overhead between 0.02% to 0.07%, which in the case of the Compiler translates to a little less than 400Kb. For Sif and Factory, the additional memory overhead is 40Kb and 300Kb, respectively.

We have modified the garbage-collector in the Mjølner BETA System to dynamically extend objects during IOA collections, and to eagerly extend objects when they are promoted to AOA. These changes required only minor modifications to the existing implementation. Extending objects during IOA collections introduces one complication. It is possible that the To-Space runs full during an IOA collection if most objects stay alive and get extended at the same time. We handle this case by promoting the rest of the live objects to AOA.

3.3 Performance

To evaluate dynamic slots we use three benchmarks. First, we measure the performance of the dynamic slot primitives to see how they perform compared to ordinary BETA attributes. This measures the raw speed of the implementation. Second, we examine the runtime overhead of adding dynamic slot support to the BETA runtime system, i.e., what is the impact on ordinary applications. Third, we have implemented a version of the persistent store that uses dynamic slot and measured the speed-up in comparison with the original implementation and a "static" implementation.

All the measurements have been performed on a 75MHz dual-processor SUN SPARC-Station 20 with 175Mb of memory. The machine was lightly loaded and had approximately 50Mb of free memory at the time of the measurements. The times measured are the total user + system time for a given operation as reported by the Solaris 2.4 operating system. All times are given as an average over a large number of executions.

3.3.1 Microbenchmark

Table 2 shows the performance of the three dynamic slot primitives. In the benchmark, the object was already extended with the FirstSlot reference and the linked list of dynamic slots only contained one element. Hence, the measured results are the best-case performance of the primitives. The `set` operation will provoke an IOA garbage-collection once for each object of a new type it is invoked on. Performing an IOA garbage-collection takes on average 18 msecs.

Primitive	Time
init	1.1
set	4.1
get	2.3

Table 2: Performance of the dynamic slot primitives (in μsecs).

An access to a statically declared integer attribute takes approximately 0.15 μsecs, so dynamic slots

[6]There exist a prototype for each pattern in the Mjølner BETA implementation. It is used to store immutable class-information, such as object size, virtual dispatch table, etc.

sacrifice more than an order of magnitude in performance. An invocation of a null-method takes 0.9 μsecs, so a large percent of the overhead is due to the method invocation. This could be eliminated by inlining the `set` and `get` primitives.

3.3.2 Runtime Overhead Benchmark

Extending the BETA runtime system with dynamic slot support introduces two kinds of runtime overhead: i) objects in AOA are eagerly extended with the FirstSlot reference, so the memory requirements are increased, and ii) the garbage-collector must check if an object has been extended, and in that case follow the FirstSlot reference when marking live objects.

Since the effect of adding dynamic slots will be most apparent on long running batch-programs, we use the BETA compiler as a benchmark. Three versions of the compiler are used, the standard BETA Compiler (Standard), a version that eagerly extend *all* objects with the extra FirstSlot attributes (Trivial), and one that only eagerly extend the objects in AOA (Dynamic). By comparing, the Trivial and the Dynamic implementation we see the effect of avoiding extending objects in AOA.

	Standard	Trivial		Dynamic	
		Total	Overhead	Total	Overhead
Exec. time	352s	386s	9.7%	360s	2.3%
Mem. usage	1.84Gb	1.89Gb	2.7%	1.89Gb	2.7%
IOA GCs	5770	7256	+1486	5770	0
AOA GCs	17	20	+3	18	+1

Table 3: Implementation Overhead for the BETA Compiler.

Table 3 shows the runtime statistics of compiling an approximately 28,000 lines program (the compiler itself) with the three versions of the BETA compiler. The Trivial version shows a significantly changed garbage-collection behaviour compared to both the Standard version and the Dynamic version. Thus, avoiding changing the size of objects in IOA is an important optimization. The Dynamic implementation shows an execution time overhead of 2.3% which we find acceptable. The extra 2.7% of memory usage does require an extra AOA garbage-collection. The average time for an AOA garbage-collection is 0.26 secs, which means that most of the execution time overhead is spent in the dynamic slot handling code in the garbage-collector.

Our current implementation of the dynamic slot handling is not optimal and could be improved slightly, but that will require two minor compiler changes. The implementation extends an object with two words, because BETA objects are double-word aligned. The last word in, on average, half of the objects is not used, and therefore could be used to store the FirstSlot reference. This will cut the memory requirements in half, as well as avoiding an IOA garbage-collection to extend objects which have an unused word. If the compiler indicated in the prototype that there is an unused word, this could easily be implemented.

Another possible optimization is the handling of the FirstSlot reference. The garbage-collector must follow the FirstSlot reference in order to locate all live objects. This is currently implemented by explicit checks in the garbage-collector code, thereby introducing an overhead even when dynamic slots are not used. Instead, the *dynamic reference table* in a prototype could be extended at runtime. It contains the offsets of all reference attributes in an object. In the BETA implementation, the dynamic reference table is represented as a null-terminated list of offsets. If the compiler was changed to make a double-null terminated list, the list could be extended by overwriting the first terminating null. Using this scheme, following the FirstSlot reference will introduce no performance overhead for applications that do not use dynamic slots, at the expense of slightly larger prototypes.

3.3.3 Persistent Store Benchmark

Finally, to evaluate the performance increase provided by dynamic slots when implementing type-orthogonal abstractions, we have modified the existing implementation of a persistent store [Brandt 94] to use dynamic slots. We compare that implementation against the original type-orthogonal implementation and against a "static" implementation. In the "static" implementation the OID attribute is pre-allocated for all objects so it can be accessed directly. This is done by requiring all persistent objects to be a subtype of a special PersistentObject class. In effect, the "static" implementation is trading type-orthogonality for speed.

The time complexity of the checkpoint operation is proportional to the number of objects, n, and the total number of object references, m, since it both has to visit all objects as well as following all references. For the original implementation, a linear scan is required to translate an object reference to an OID, so the time complexity is $O(n + mn)$, or simply $O(mn)$. In the dynamic slot implementation (and "static" implementation), an object reference can be translated into an OID in constant time, so the time complexity is $O(m + n)$, or simply $O(m)$, when m dominates n. Thus, we would expect an asympotical speed-up by using dynamic slots.

To evaluate the effect of the reference density, we have measured the time to checkpoint a single linked list, where there is one reference in each object ($m = n - 1$), and we have measured the time to checkpoint a fully connected graph, where all objects have references to the others ($m = n^2$). For each structure, both the time to checkpoint it the first time (1), where OIDs have to be assigned, and the time to checkpoint it a second time (2), where all objects have been assigned OIDs, are measured.

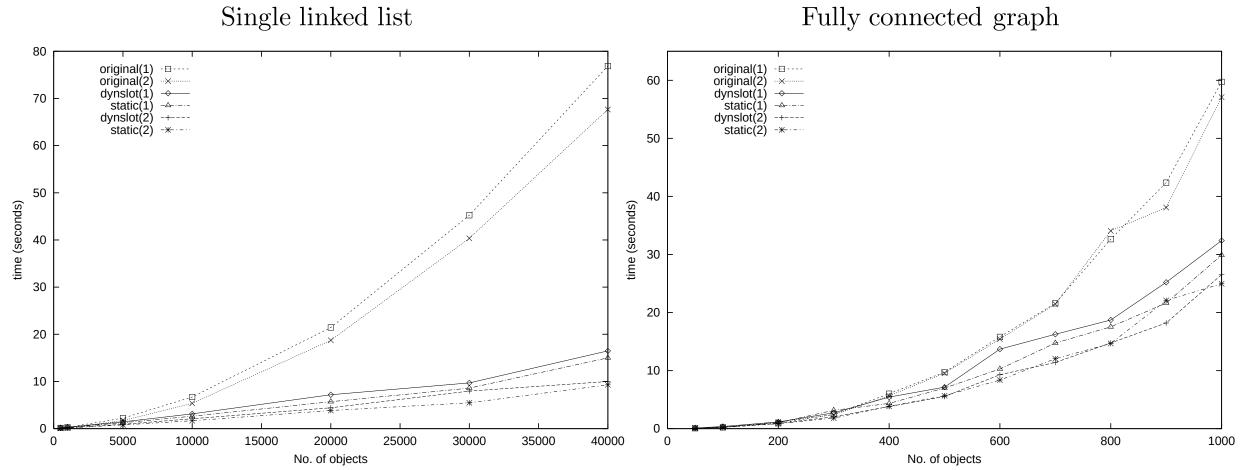


Figure 11: Checkpointing benchmark.

The results are shown in Figure 11. For the single linked list the size of the data structure is approximately 1 MB for 40,000 objects, and the fully connected graph the size is 3.8 MB for 1,000 objects. The minor fluxturations in the graphs are due to different garbage-collection behavior, e.g., the swap in original(1) and original(2) for the fully connected graph at 800 objects.

As expected, we get an asympotic speed up by using dynamic slots, since the linear scan is avoided. For the single linked list, it is apperant that the dynamic slot implementation exhibits a linear time complexity in the number of objects, whereas the original implementation exhibits a higher-order time complexity. Furthermore, the dynamic slot implementation provides a significant performance increase for checkpointing object structures for which OIDs have already been assigned. The dynamic slot implementation performs close to the optimal speed exhibited by the "static" implementation. They are only separated by a small constant factor. This shows that type-orthogonality and speed does not have to rule each other out.

3.4 Summary

We have argued that the concept of dynamically extensible objects can easily be incorporated into a statically-typed and class-based language. Dynamic slots allow new substance to be added to an object at runtime. They extend the expressive power by providing a new dimension of extensibility in addition to the class hierarchy. By using this mechanism, type-orthogonal abstractions can be implemented in a straight forward and efficient manner.

We have shown that strong typing can be maintained, since dynamic slots are statically declared in the source code. Dynamic slots thereby seamlessly integrate into the BETA type system, and do not give rise to potential runtime errors. Furthermore, a simple and efficient implementation strategy was presented. Measurements of our prototype implementation showed both little overhead for ordinary applications, as well as efficient access to the dynamic slot primitives.

4 The MetaBETA Implementation

In the previous sections we have described MetaBETA from a functional viewpoint, focusing on the metalevel interface, the execution model, and how MetaBETA supports construction of type-orthogonal abstractions. Now we shift the focus to the MetaBETA implementation. This section will describe the major components in the implementation and how they work together. Space limitations prevents us from going into all the details.

4.1 Overview

In the metalevel architecture, the runtime system is responsible for dynamic loading and linking of code, implementing the metalevel interface, and provide efficient execution of MetaBETA programs. To reconcile the capability to perform metalevel computation with efficient execution, the implementation is based on loading on demand and runtime code optimizations.

By loading metalevel information on demand, a program that does not utilize the MLI will only pay minimal time and space overhead from loading and storing the metalevel information. Furthermore, the runtime system optimizes and patches the executing code based on the current runtime state to ensure optimal performance. For example, inserting a reflective hook on object invocation might affect all objects of a particular type. However, by dynamically patching the code, this performance degradation will only be in effect during the use of the reflective hook. This is in contrast to compile-time detection of reflective hook and statically compiling them into the code [Kirby et al. 94, Chiba 95].

The key enabling technology is a MetaBETA intermediate representation (MIR). An intermediate representation is used by compiles to separate the code that handles the source language from the code that handles the target language, i.e., to separate the compiler front-end and back-end. The MIR is an intermediate representation for MetaBETA programs. It defines the interface between the architecture-independent MetaBETA compiler (front-end) and the architecture-dependent runtime system (back-end).

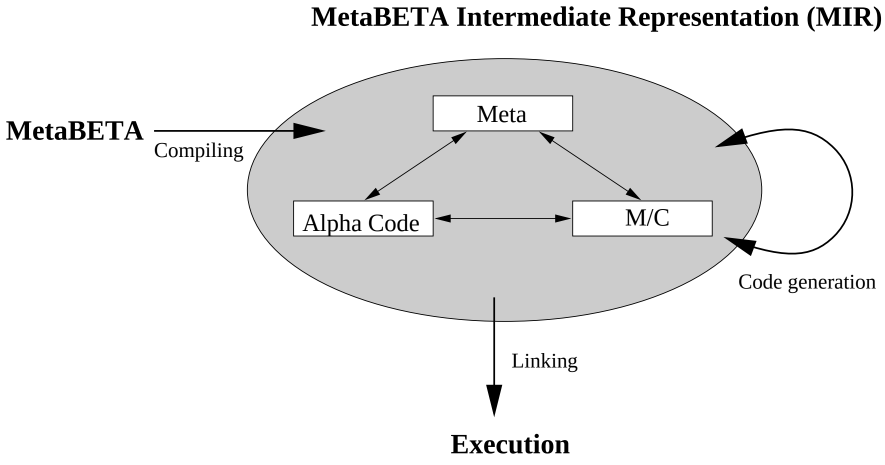


Figure 12: The MetaBeta Intermediate Representation (overview).

Three representations of the original MetaBETA program is stored in the MIR: i) metainformation (meta), ii) do-part information (Alpha code), and iii) optional machine-code (M/C). Figure 12 shows a high-level view of the MIR. It is possible to move between the representations by following links in the data-structure. The MetaBETA compiler generates the metainformation and the Alpha code. The machine-code is generated either explicit with a code-generation tool or on demand by the runtime system.

The different representations of the compiled program are stored seperatly, so the can be loaded *independently* of each other. For example, if only the machine-code part is loaded, then the metalevel architecture degenerates into a traditional compiled language implementation. If only the Alpha code is loaded, we have a system with just-in-time code generation, similar to the Java runtime system [Arnold & Gosling 96]. Finally, the metainformation will only be loaded on demand to avoid both time and space overhead for ordinary programs.

This model has several benefits:

- The metalevel interface can be implemented since all the metainformation is stored in the MIR.
- Code structure and dynamic linking are orthogonal. The unit of dynamic linking is not restricted to a source file, but can be as fine-grained as an object-descriptor[7]. This increases code reuse, since dynamic linking does not dictate a certain source code structuring.
- Extensibility and tailorbility are supported. The MIR representation corresponds to all the loaded code in the system. By extending the MIR representation at runtime, possibly by loading extenstions from disk, dynamic extensibility is achived. In fact, the runtime system must contain a MIR loader to bootstrap a program. This loader can also be used to extended a program execution at runtime.
- Faster program development by doing loading, linking, and code generation on demand.
- Architecture-independent code format. The metainformation and Alpha code part of the MIR is the same across all platforms. This increase portability of the code, and dynamic code exchange between different machines is possible. Furthermore, it saves disk space and minimize the number of compilations that are necessary in an heterogenous environment.
- Runtime code optimization is supported. The runtime system generates architecture-dependent code at runtime. Hence, it can generate code that is optimized for special metalevel operation. Furthermore, it has a global view of the loaded code, so it can utilize close-world assumptions[8].

4.2 Tools

The MetaBETA programming environment consists of 4 tools. The two most important are the compiler and the runtime system itself.

$$\begin{array}{ll} \text{Compiler:} & \text{MetaBETA} \rightarrow \text{MIR}_{meta,alpha} \\ \text{RTS:} & \text{MIR} \rightarrow \text{Execution} \end{array}$$

The Compiler only generates the metainformation and optimized Alpha code. Another tool is the static code-generation tool that generates architecture-dependent machine code and incorporates it into the MIR.

$$\text{MGEN:} \quad \text{MIR} \rightarrow \text{MIR}$$

It might not always be practical or desirable to distribute an application as a large number of MIR files — one for each source file in the application. For this purpose, a MIR linker exist that takes a number of MIR files and merge them together into one MIR file. Notice, that merging the MIR files into a single large file will not have any effect on dynamic loading and linking.

$$\text{MLINK:} \quad \text{MIR}^* \rightarrow \text{MIR}$$

The use of *MGEN* and *MLINK* is optional. They are merely stand-alone utilities for some of the functionality that exist in the runtime system.

[7] An object-descriptor is the syntactical category in the BETA grammar that looks like `(#...#)`.

[8] However, it might have to undo some optimizations later, if new code it loaded

4.3 The MetaBETA Intermediate Representation

Most compiled language systems use a static exchange format between the compiler and the execution engine. For example, the current BETA implementation outputs an executable image. This image is loaded by the operating system (e.g., UNIX). The operating system patches the image a few places at load-time and then transfer control to the entry-point in the image. The exchange format is static. It is not to be manipulated at runtime.

The MetaBETA approach is different. Instead of defining a static exchange format between the compiler and the runtime system, we have chosen a dynamic and more object-oriented approach. The MIR is defined entirely with high-level BETA abstractions. This means the a compiled MetaBETA fragment is represented as a collection of interrelated objects — in contrast to a bytestream in a traditional compiled languages. The design of the MIR is driven by the needs of the runtime system, so the structure is close to the internal representation used by the runtime system. This approach has been chosen because it satisfies several of our design goals:

- Fine-grained dynamic loading is supported by a persistent store with lazy fetch of objects.
- We can utilize the benefits of object-oriented programming when implementing the compiler and runtime system. The metainformation, Alpha code, and machine code can be manipulated using BETA abstractions allowing, e.g., code-manipulation functionality to be written directly in BETA.
- The external format is defined automatically and implicitly by the BETA representation, so experimentation with different representations is easy. An important capability for a research prototype.

A possible negative consequence of this is that the code size might increase, because the persistent store cannot directly use semantic information to reduce the size of the external format. However, this is not an inherit limitation of the model.

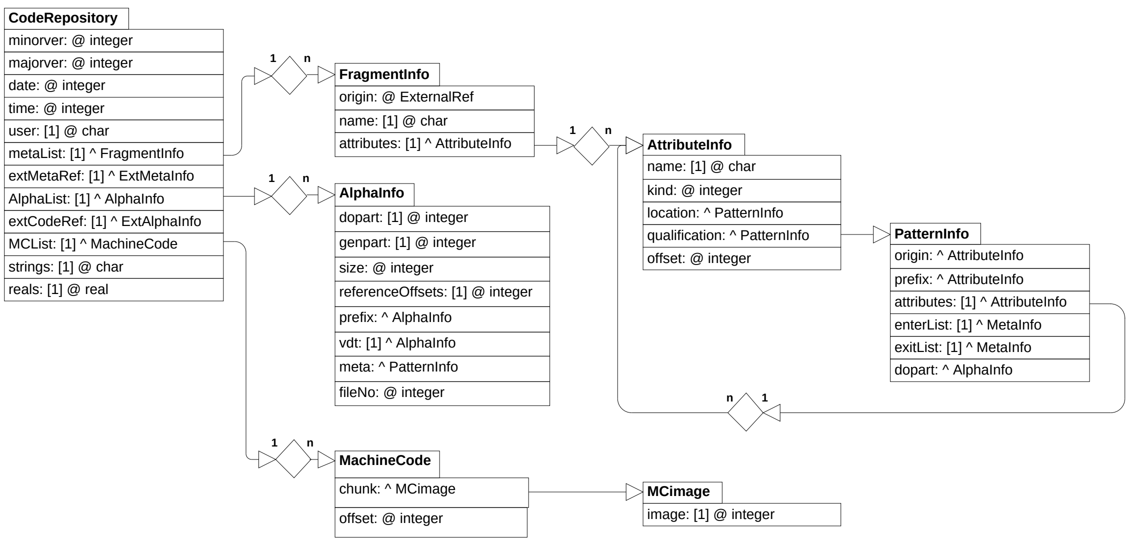


Figure 13: A high-level description of the MIR representation. The names in boldface are classes, others are attributes of that specific class. An arrow represent a relation. An arrow that goes through a diamond is an one-to-many relation.

Figure 13 outlines the layout of a MIR store. It is split into three parts, one for metalevel information, one for Alpha code, and one for the (optional) precompiled machine-code. The representation is defined by

11 patterns. The seven main patterns are shown in the figure. The rest of the patterns are mainly concerned with handling external references. Appendix A lists the BETA source code for the MIR representation. The following subsections will explain the three parts in more detail.

4.3.1 Metainformation

The metainformation part of the MIR contains all the information needed to implement the introcessory part of the MLI. It is rooted by `metaList` in the `CodeRepository` object and is organized in an object structure the mimics the fragment and block-level structure of the original MetaBETA source code. The patterns `FragmentInfo`, `PatternInfo`, and `AttributeInfo` describe a fragment, a pattern, and an attribute respectivly. The main object in the store (the singular instance of `CodeRepository`) corresponds to a fragment group.

A `FragmentInfo` object contains: external origin information, the name of the fragment, and a list containing the attributes defined in the fragment. The external origin information is a reference to the `PatternInfo` object in another MIR store, where the attributes logically belongs, i.e., the object-descriptor that contains the `SLOT` annotation to be filled out.

A `PatternInfo` object describes an object-descriptor. Logically, the *origin* and *prefix* of a pattern denotes another pattern. However, we have chosen to stored them as a reference to the defining `AttributeInfo` objects. This makes it easy to reconstruct source code at runtime. From an `AttributeInfo` object, the `PatternInfo` object can always be found be following its *location* reference. The `PatternInfo` object also contains information about enter and exit-parts, the list of attributes, and the do-part (i.e.,by a reference to an `AlphaInfo` object).

An `AttributeInfo` object describes an attribute of a given object-descriptor. It contains the name of the attribute, the kind, the location, the qualification, and the offset. The kind denotes if it is, e.g., a part object (`@`), or a dynamic reference (`^`). The location is a reference to the `PatternInfo` object that describes the enclosing pattern. The qualification is an reference to the type of the attribute. If the attribute is simple, e.g., an integer, it is not used. Finally, it contains the machine-dependent offset of the attribute in the runtime representation of an object.

In the current design, it is the compiler's responsibility to assign offsets to attributes. Thus, the compiler defines the object layout. This makes code generation faster, but requires the same object layout to be used across all platforms. However, this approach could be changed. The compiler could assign logical numbers to attributes, and then the runtime system could map between the logical numbers and physical offsets at runtime.

The metainformation can have external references to `AttributeInfo` objects and `PatternInfo` objects in anoter MIR. As already noted, this is the case for the `FragmentInfo` object. The other places are the *origin* and *prefix* attributes of `PatternInfo`, and the *qualification* attribute of `AttributeInfo` (in the case of a descriptor slot). This external information is stored in the `extMetaRef` list in the `CodeRepository` object.

4.3.2 Alpha Code

The `AlphaInfo` object describes a runtime MetaBETA object in detail. This includes the size of the object, the reference table (used by the garbage collector), the virtual dispatch table, the INNER dispatch table, etc.

The do-part is represented as a list of bytecodes for an architecture-independent language: Alpha. The Alpha code is a low-level architecture-independent representation of MetaBETA do-parts. It has the following properties:

- Close to RISC [Patterson 85]. It is possible to quickly generate efficient code for contemporary microprocessor architectures.
- BETA specific. Complex actions, such as object invocation, INNER calls and attribute assignment are represented as one operation.
- Minimal. A few number of general operations to make the code generation as simple as possible. For example, no I/O operations exists in the Alpha code.

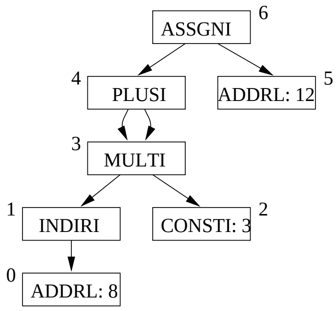
- Supports code optimizations. Common subexpression elimination, constant folding, dead-code elimination, inlining, loop unrolling, etc. can be simply and efficiently implemented.

To satisfy these requirements, we have based our format on a DAG[9] with simple operators, such as arithmetic and assignment, as nodes. There as a total of 40 different opcodes. Some of these opcodes have type-suffixes, making a total of 70 operations. Our design is greatly inspired by the design and implementation of the lcc compiler [Fraser & Hanson 91a, Fraser & Hanson 91b].

The DAG structure has several nice properties that made it our Alpha representation of choice. First, it does not assume any specific register model or number of registers. Second, it is relatively simple to generate efficient RISC code from the DAG [Fraser & Hanson 91a]. Third, basic optimizations, such as common subexpression elimination, constant folding, dead code elimination, can easily be carried-out by the compiler, and the DAG representation contains enough information to support more advanced optimizations, such as inlining, at runtime [McConnell 93]. The last two capabilities are not supported very well by stack-code which was another alternative. Stack-code is used by the Java virtual machine [Arnold & Gosling 96].

The Alpha code contains mainly low-level RISC-like primitives, except for 6 BETA specific primitives. The high-level primitives encapsulate architecture-dependent properties, such as object creation, invocation of a method, attribute access, etc. This has several benefits: i) the Alpha code is potentially smaller than machine-dependent RISC code, since garbage-collector write-barriers, entry and exit-code for method invocation is abstracted away, and ii) it makes the Alpha code very portable, by giving each specific platform a great degree of freedom to implement register allocation, garbage collection, and method invocation, and finally, iii) the intercessory part of the MLI exactly changes the way the high-level primitives behave. Thus, compiling a do-part on which a reflector is associated only means changing the code-generation template of one Alpha operator.

DAG



Linearized DAG

no.	opr.	kids	arg.
0:	ADDRL		8
1:	INDIRI	0	
2:	CONSTI		3
3:	MULTI	1 2	
4:	PLUSI	3 3	
5:	ADDRL		12
6:	ASSGNI	4 5	

Figure 14: Alpha Code representations of the BETA expression: `(i*3)+(i*3)->c`.

Figure 14 shows an example of a BETA expression that has been converted into an Alpha DAG and into a linearized Alpha DAG. In the example, the Alpha operators are suffixed with an `I` since the code works on integer values. We assume that the `i` variable is store at offset 8 and the `c` variable is stored at offset 12 relative to the current object. The operation `ADDRC` returns the address of a cell relative to the current object. `INDIR` reads a value from an address, in this case an integer. `MULT` and `PLUS` performance multiplication and addition, respectivly. Finally, `ASSIGN` assigns a value to a given address.

The DAG is linearized and converted into bytecodes before it is put into the `AlphaInfo` object. The linearized version of the DAG is also shown in Figure 14. The complete set of Alpha operators is listed in Appendix B.

4.3.3 Machine-code

The fundamental model in the MetaBETA implementation is that all machine-code is generated on demand by the runtime system — the compiler only generates metainformation and Alpha code. Unfortunately,

[9] Directed Acyclic Graph.

generating all machine-code on demand will require an unncessary huge start-up overhead everytime an application is loaded. To avoid this overhead, it is possible to precompile Alpha code to architecture-dependent machine-code and store it in the MIR.

Machine-code is generated in units corresponds to an object-descriptor (i.e., an `AlphaInfo` object). This unit contains the prototype, the generation code (G-code), and the do-part code (M-code) [Madsen 93a]. These units can be static linked and stored in an `MCimage` (see Figure 13). For each object-descriptor, a `MachineCode` object exist that has a reference to the `MCimage` that contains the machine-code, and an offset to where it is located in the image.

The MIR representation does not dictate the granularity of static linking. It can be as fine-grained as every object-descriptor exist in its own image to as coarse-grained as an entire application exist in a single image. If an application is compiled and static linked into one image, then only references to functionality in the runtime system has to be patched at load-time. Thus, the start-up time will be comparable to a traditional compiled application.

4.4 The Runtime System

The runtime system is the only program that can be directly executed by the operating system. Once it is loaded and started executing, it loads the MIR representation of a MetaBETA program and starts to execute it. The main responsibilities of the runtime system are: i) implementation of the metalevel interface, and ii) dynamic loading, linking and code-generation. The following sections describe these parts in more details.

4.4.1 Implementing the Metalevel Interface

The introspective capabilities of the MLI is implemented by browsing the metainformation structure provided by the MIR. Consider the example from Section 2.1.2: `ar[]->scanAttr(# do ... #)`. Given the object reference, `ar`, the prototype of the object can be found, and the prototype contains a pointer to the corresponding `PatternInfo` object. The implementation of `scanAttr` iterates over all attributes, i.e., the `attributes` list. An `AttributeRefs` object is represented internally with a pointer to an `AttributeInfo`. The `AttributeInfo` objects are hidden from the application programmer, so the runtime system can guarantee that the metalevel information is consistent.

During the runtime system's initialization phase, `PatternInfo` and `AttributeInfo` objects representing the basic patterns defined by the runtime system gets created. This includes objects for patterns, such as integer, real, the `Execution` object (see Figure 7), as well as the patterns used to describe the MIR representation itself. This creates an interesting metacircularity. In the metainformation, a pattern is described by an `PatternInfo` object. Hence, the `PatternInfo` pattern must be described by an `PatternInfo` object. It is, in fact, describing itself.

The intercessory capabilities will be implemented by a combination of patching and recompilation of do-parts in the case of reflection on object allocation and object invocation. Consider the case where a reflective hook is inserted on a do-part. Then the runtime system patches the code for that do-part, so it either: i) invokes the reflective hook directly (i.e., a jump to the reflective code), or ii) patches the calling code to call a new version of the do-part where the reflective code has been inlined. In [Brandt & Schmidt 95] preliminary performance measurements of the performance degradation due to runtime patching of code is reported.

The tradeoff between the two approaches are between the setup cost and the invocation cost. If the reflective hook is in effect for a long time, a high setup cost and a low invocation cost is bedst. Otherwise, a low setup cost and a relativly high invocation cost might be best. The runtime system is responsible for chosing the bedst approach to achive the optimal performance of MetaBETA programs. However, in general no algorithm exist for deciding which approach is bedst, i.e., we have a mapping-dilemma when we go from the model to the implementation. This can be solved by providing the runtime system with a default policy and the ability to modify it. The application programmer can then replace or fine-tune the policy when needed, or use a policy that is based on profiling information (similar to [Grove et al. 94]).

Reflection of object reclamation will be handled by extra checks at garbage-collection time, similar to the implementation of the DOT (Debugger Object Table) in the current BETA implementation.

4.4.2 Dynamic Loading, Linking, and Code Generation

The MIR representation is stored as persistent objects using the BETA persistent store [Brandt 94]. Loading of the code is simply done by opening the persistent store. The persistent store supports lazy fetch of objects, which means that objects are brought in from disk when a reference is followed. Hence, loading on demand is implemented by the persistent store.

Once a MIR persistent store is loaded, it has to be linked into the current program execution. In the following, we will outline how the linking phase and code-generation phase is implemented. We will describe how the MIR design allows only the parts of the MIR that is used to be loaded into a program execution.

The `LoadFragment` method dynamically extend a program execution with new patterns, i.e., it extends the metainformation data-structure. This is done as follows:

1. Open the persistent store containing the code to load.
2. Verify that the origins of the fragments in the MIR is already loaded. Otherwise, signal an error.
3. For all external origins, i.e., `FragmentInfo` objects, extend the corresponding `PatternInfo` objects. All new attributes will then be accessible through the metalevel interface.

The objects that describe the metalevel information are loaded in on demand. In order to minimize the loading of metalevel information, the runtime system uses a table, `MetainfoUpdateTable`, to delay updates of the metainformation until it is brough in from disk. The invariant is that the loaded metainformation is up-to-date. The persistent store can keep this invariant by checking the table every time it fetches a new `AttributeInfo` or `PatternInfo` object. The `LoadFragment` method delays updates by storing update information in the `MetainfoUpdateTable`.

The `LoadFragment` method only changes the metainformation part of the MIR. The Alpha and M/C objects are fetched when the code is accessed through the MLI or implicitly though a source code dependency. This is handled implicitly by the persistent store, since `LoadFragment` can setup lazy references to the objects.

The external references in the architecture-dependent code is handled in a similar way. If the do-part for the pattern P has a reference to a the do-part for the pattern Q (i.e., it calls it), and the do-part for Q is not loaded yet, then the compiler will generate a *thunk* for Q. The thunk is a piece of code that will trap to the runtime system and load the do-part for Q. In this way, lazy linking of machine-code is achieved. The runtime system does the following steps when is handles a thunk:

1. Check if the machine-code has been loaded. In that case, it patches the calling code and returns.
2. Checks if the Alpha code exist. In that case, it generates machine-code, patches the calling code, and returns.
3. Otherwise, it invokes `LoadFragment` for that file containing Q, and starts at 1. again.

Recall, that the optional pre-compiled machine-code in the MIR format is stored in *images* that contains code for one or more patterns. By putting more than one pattern in an image, these patterns can be statically-linked and the use of thunks are not neccessay. Hence, the trade-off between statically-linking and dynamic-linking is made by the programmer.

4.5 Summary

In this section, we outlined the implementation of the runtime system for MetaBETA. At the basis for the implementation, is a MetaBETA-specific intermediate representation that is used by the both the compiler and the runtime system. The compiler generates code directly into this format. Exchange of information between the compiler and runtime system can be done directly or through a persistent store.

The design phase has recently been finished, and the implementation of the MetaBETA compiler and the runtime system is underway. Currently, a version of the MetaBETA compiler that outputs the metainformation part of the MIR is working.

5 Related Work

The design and implementation of MetaBETA relates to many earlier and current research efforts in the reflective languages and systems area. Earlier papers on MetaBETA have discussed work related to the metalevel interface [Brandt 95, Brandt & Schmidt 96], so in this report we will restrict ourselves to work related to dynamic slots and the MetaBETA implementation.

5.1 Dynamic Slots

The dynamic slot concept presented here for a class-based language has previously materialized itself in a number of other languages and systems, including Self, CLOS, and LISP. Dynamic slots are also related to a number of reflective systems.

Prototype-based languages use a similar mechanism as dynamic slots to extend objects. In Self [Ungar & Smith 87] objects can be extended with new slots at runtime by using the `addSlot` method. Dynamic slots were strongly inspired by that feature. However, adding dynamic slots to a class-based language is not an attempt to make it more prototype-based, since we are not sacrificing or limiting any expressibility of the original language. Dynamic slots provide an extra dimension in which objects can be extended that is orthogonal to the class-hierarchy. This is particularly useful when implementing type-orthogonal abstractions.

Interestingly, the Self system itself does not use `addSlot` in a way similarly to dynamic slots. Instead, they have implemented a primitive that assign an immutable hash value to an object. The extra word we extend BETA objects with, could also be used to store a unique and immutable hash value. However, this will require the programmer to explicitly create a hash table to store all (object,value) relations[10]. We find the *extensible object* metaphor to be a more direct and general concept.

LISP also contains a mechanism similar to dynamic slots. A LISP symbol has an associated property list, where (key,value) pairs can be stored. The mechanism is not as general as dynamic slots, since it is only available on symbols as compared to on all objects. Also, static type-checking is not an issue in LISP. In LISP implementations, property-list are typically stored in an external hash table hashed on the symbol's hash value, i.e., not with the symbol. Dynamic slots have also been proposed for CLOS [Kiczales et al. 91], where they denote statically declared attributes but for which memory is allocated on demand. These are very similar to the dynamic slots presented in this report, except that we allow dynamic slots to be declared outside a class definition. That is crucial to support type-orthogonal abstractions.

Metalevel interfaces have been constructed for several statically-typed languages, including Open C++ [Chiba 95], Napier88 [Kirby et al. 94], and BETA [Brandt & Schmidt 96]. They all support the construction of type-orthogonal abstractions directly in the language itself. In principle, all type-orthogonal abstractions can associate additional information to the objects they work on. A type-orthogonal facility to extend objects is therefore needed to allow maximum flexibility and efficiency of metalevel programs. In Open C++ and Napier88 this is supported by compile-time modification of the source code. As argued in Section 3.1.1, we do not believe that all such modification can be satisfactorily done at compile time. The metalevel interface for BETA is entirely a runtime entity. It does not require access to the source code for the objects it works on. Hence, a mechanism for runtime extensibility of objects is mandatory.

5.2 Reflective Languages

Open C++ [Chiba 95] is a reflective version of C++. Like C++, Open C++ is a statically-typed and compiled language. Its metalevel interface is based on a metaobject protocol (MOP) that controls how classes are compiled. For example, a metaobject can change how method calls are implemented and add substance to a class. Similar to what can be done with the intercessory capabilities and dynamic slots in MetaBETA.

However, for Open C++ this is resolved at compile-time. Thus, the Open C++ metalevel programmer has to distinguish between compile and runtime values. The MetaBETA approach provides greater flexibility and simplicity, since it does not require access to source code and recompilation to apply metalevel code, and all manipulation are done with runtime values. Furthermore in MetaBETA reflective code does not need

[10] Notice that, e.g., a persistent store could not use the hash value directly as an OID, because OIDs must stay unique across program executions.

to be in effect during an entire program execution. Also, MetaBETA is based on an active runtime system, that provides runtime extensibility and introspective capabilities. That is not an integral part of the Open C++ MOP.

CodA [McAffer 95] is a language-independent architecture for implementing reflection in a programming language. Its model is based on reification on primitive operations, such as message send, message delivery, message receive, etc. It is similar in nature to the intercessory part of the MetaBETA MLI, but somewhat more fine-grained. An implementation of the CodA architecture for Smalltalk has shown that this model can be used to parallelize a large single-treaded program with only a few modifications to the original source code. CodA was implemented by modifying the implementation of the Smalltalk interpreter on top of the Smalltalk virtual machine. However, where as in the CodA Smalltalk implementation not much emphasis was put on efficiency, that is a major goal for the MetaBETA implementation.

CLOS (Common Lisp Object System) [Kiczales et al. 91] was one of the first successful languages with a MOP. It is implemented by a metalevel (or reflective) interpreter, that has a reification interface (or metalevel interface). The reflective interpreter approach is typically used when the implementation language is close to the implemented language, e.g., CLOS is implemented in Common Lisp. Relying on interpretation of a source program makes it possible to implement very powerful and flexible metalevel interfaces and to facilitate extensibility. However, it has the drawback that the interpretational overhead can be very large, typically many orders of magnitudes compares to a compiled system. The MetaBETA metalevel architecture is similar in spirit to the reflective interpreter approach, but the implementation language used by the runtime system is not MetaBETA, but Alpha code. The runtime system rely on runtime code optimizations to reconcile a flexible metalevel interface with efficient program execution.

ABCL/R3 [Masuhara et al. 95] is also based on a reflective interpreter approach similar to CLOS. In order to gain performance, they use an approach, where the reflective interpreter is partial evaluated with the source program. In effect, specializing the compiler with respect to the metalevel computation, and thereby getting rid of the metalevel interpretation. This is possible because their metalevel interface is based on metaobjects that control execution of baselevel objects, and these metalevel objects are statically represented in the source code. Hence, the compiler can statically determine which metalevel objects are associated with which baselevel objects.

The MetaBETA MLI has no metaobjects. Reflection is on a finer granularity, i.e., language primitives. In general, this makes it impossible for the MetaBETA compiler to predict which methods or classes will be reflect upon. Therefor the runtime system uses runtime code generation and optimization to gain performance. In this way, we avoid a statically analysis of an entire application to determine where reflection is used.

5.3 Reflective Operating Systems

Apertos [Yokote 92] is a reflective operating system built around a metaobject protocol (MOP). The MOP allows user-level programs to tailor most aspects of the operating system, including message passing, thread scheduling, and virtual memory usage. In the first version, a variant of C++ was used to write extensions to Apertos. Currently, a special language, Cognac [Murata et al. 94], is being implemented that is special designed to utilize the Apertos MOP.

The MetaBETA language and the runtime system has a similar relation as Cognac and Apertos. The reflective capabilities of the MetaBETA language is abstracting the reflective capabilities of the runtime system, thereby providing a uniform programming model. However, the MetaBETA runtime system is not intended as a general operating system. Instead of being an operating system, the runtime system is the glue between the language and the underlying OS.

Spin [Bershad et al. 95] is an extensible micro-kernel that allows user-level programs to inject code into the kernel, thereby avoiding the context-switching and IPC[11] cost that is a problem for traditional micro-kernels. Spin uses an event model similar to the intercessory part of the MetaBETA MLI. It is possible to associate a code fragment to a certain event in the kernel. The code-fragment is written in a safe subset of Modula-3. The injected kernel extensions are inlined and optimized in the context of the existing code in order to provide optimal performance. The MetaBETA runtime system is using its ability to do dynamic

[11] Inter-Process Communication

loading, linking, code generation in a similar fashion.

5.4 Other Systems

The MetaBETA execution model is very similar to the implementation of dynamic typed languages, such as Smalltalk [Goldberg & Robson 89] and Self [Ungar & Smith 87]. They are all based on an underlying virtual machine which is responsible for executing programs. In particular, the Self implementation, where lots of effort has been devoted to runtime code generation and runtime optimization of code [Chambers & Ungar 91, Hölzle & Ungar 94]. Many of the techniques used in the Self implementation is possible to apply to the MetaBETA implementation. However, since BETA is a statically-typed language, we believe that efficient execution times can be achieved without relying on the same amount of aggressive code optimizations.

Statically-typed languages, such as Simula (ORM) [Magnusson 93] and Trellis/OWL [O'Brien et al. 87] are also based on an active runtime system and an integrated programming environments. In contrast to these systems, the MetaBETA architecture does not define or rely on an integrated program development environment. Instead we are trying to provide the necessary hooks and functionality, by opening up the runtime system, so a development environment can be built on top.

By separating the development environment from the runtime system, MetaBETA is not suspect to the problems of extracting an application from the development environment [Agesen & Ungar 94]. In fact, the BETA fragment structure [Madsen 93b] can be used in the MetaBETA model too.

The Alpha code format was directly inspired by the work done on the lcc C compiler [Fraser & Hanson 91b, Fraser & Hanson 91a] and on Tree Based code optimizations [McConnell 93]. The experience from the implementation of the lcc compiler has shown that efficient code can be generated fast for leading RISC and CISC architectures. Furthermore, the representation has been used for dynamic code generation [Engler & Proebsting 94].

6 Future Work

The work on MetaBETA was initiated to provide an open implementation of the BETA language, since functionality such as debuggers and persistent stores was not possible to express directly. This has lead to the design of a metalevel interface and the metalevel architecture, for reflective BETA programs. In this report the first realization of MetaBETA has been described.

The prototype implementation of MetaBETA is underway. When it is operational there is a number of research directions that will be interesting to pursue. The main directions can be classified as:

- **Conceptual**: Evaluate and refine the metalevel interface and architecture.
- **Implementational**: Improve and streamline the implementation, so it gets more time/space efficient, utilize resources better, and has increased functionality.
- **Applicational**: Use MetaBETA to implement applications and frameworks in order to get real world experience with both the model and the implementation.

These main directions are often related. For example, a change in the conceptual model is likely to require a change in the implementation, and development of applications in MetaBETA will give valuable feedback on both the conceptual model and implementation. We will in the following describe several future work projects that has been identified. They will be grouped according to the above classification.

6.1 Conceptual

Unifying the metalevel interface and the metaprogramming interface. The current Mjølner BETA System contains a metaprogramming system (MPS) [Madsen & Nørgård 93], where BETA programs are manipulated as abstract syntax trees (AST). The MPS defines the common representation that links the tools in the Mjølner BETA system together, i.e., the syntax-directed editor (Sif), the object-oriented case-tool (Freja), the application-builder (Frigg), the debugger (Valhalla), and the compiler itself.

The information provided by MPS is basically the same as the introspective part of the metalevel interface provides. Both interfaces allows browsing of the attribute parts of an object-descriptor. The difference is that the MPS model is based on files with static source code, where as the MLI is based on instantiated patterns at runtime.

It would be interesting to combine these two interfaces, so the distinction between static source code manipulations and dynamic source code manipulations would vanish. One possible way of attacking this problem, would be to redesign MPS to use partial pattern qualifications [Brandt & Knudsen 96]. Partial pattern qualifications precisely captures the idea of manipulating patterns without having to instantiate them.

Extend the metalevel interface to include do-parts. The current metalevel interface does not provide an abstract representation of the do-parts of BETA objects. The functionality of the current interface is skewed towards the needs of object persistence and distribution. For completeness, it should be possible to introspect the do-parts. For example, debuggers and profilers could use such an interface to insert break and watch-points. Incorporate the abstract syntax tree into the BIR format, and have an runtime reification of AST nodes is a possible approach. However, it is not clear how this can be integrated into the current metalevel interface.

Incorporating dynamic code modifications. The current metalevel architecture is based on runtime extensibility. It would be interesting to extended it to also include runtime code modifications, much in the same way as the Smalltalk [Goldberg & Robson 89] and Self [Ungar & Smith 87] implementations allow. This requires both careful design of a model, an interface, and an implementation. Some of the problems are: since the metalevel architecture does not define an interactive environment, what happens in case of inconsistencies, how fine-grained or coarse-grained should it be possible to do modifications, and how can this be implemented efficiently.

Incomplete Program Executions. Loading code on demand makes it possible to execute incomplete programs, i.e., where certain methods, classes, etc. are not implemented yet. As long as the executing code does not refer to any of the unimplemented code everything works as expected. In the case of a reference to some unimplemented code, an exception method could be called in the runtime system. The exceptions will be similar to the so called *checked runtime errors* in Modula-3 [Nelson 91].

6.2 Implementational

Code Optimizations. In the MetaBETA implementation it is possible to utilize and experiment with runtime code optimizations. The Alpha code is accessible for the runtime system and can be used to implement optimizations such as inlining, dead-code eliminations, etc. The Self implementation has shown impressive performance improvements by using dynamic profiling and code optimizations [Hölzle & Ungar 94]. Runtime code optimizations provide new possibilities that are not present at compile-time. In particular, to optimize code across source code boundaries. For example, `INNER` chains and descriptor-slots can be inlined. The runtime system can utilize a close-world assumption, since it knows all the code that is loaded at any given point in time. Optimizations such as static class hierarchy analysis would be possible [Dean et al. 95].

Verification of Alpha Code. The success of the Java [Arnold & Gosling 96] language as the internet programming language of choice is mainly because it is possible to verify that a Java program is safe at the byte-code level, where the definition of safe can be type-correct, utilize limited amounts memory, limited amount of threads, does not access the file system, etc.

While the Java byte-code is based on a stack-machine, the MetaBETA Alpha code is based on a DAG structure. It would be interesting to investigating how an efficient DAG verifier can be built. Such an verifier is an important tool if Alpha code is, for example, exchanged across the network between unthrusting parties.

Garbage Collecting of Code. The extensibility model used for MetaBETA might require large amounts of code being loaded into an executable image. In order to keep memory usage minimal, it would be necessary to detect unreachable code, i.e., code which is not possible to execute anymore. One scenario, where such a situation could happen, is a World Wide Web browser based on MetaBETA. A page might contain some MetaBETA code that is downloaded and executed. The representation of the page has a pattern reference to the down loaded code. Once, that page is removed from memory, the code is not executable anymore, and should therefore be flushed from memory.

Tight Integration with an Operating System. The runtime system acts as an operating system for compiled MetaBETA program. It provides memory management, threads, and disk-storage abstractions. However, instead of being a full-blown operating system, it should be a thin glue layer, that makes the underlying operating system MetaBETA aware. Several operating system are under development which lets user applications get fine-grained control of system resources [Bershad et al. 95, Engler et al. 95]. In effect, these systems provide a metalevel interface that allows the operating system to be customized. Integrating the MetaBETA runtime system tightly with the operating system can provide more efficient use of the system resources, e.g., combining user-level and system-level thread-scheduling [Anderson et al. 92], and combining garbage-collection and virtual memory management.

6.3 Applicational

Implementing distribution and persistence. The experience from designing and implementing the current version of object persistence and distribution has been a major factor in the design of both the metalevel interface and architecture. Reimplementing object persistence and distribution in MetaBETA to get an implementation that has no implementational dependencies, is completely type-orthogonal, and better performance has high priority.

Integrated Programming Environment. The runtime system can be viewed as the engine running a MetaBETA program. However, another alternative is as a toolbox to manipulate executing MetaBETA program. One goal is to extend the MetaBETA runtime system, so it allows the construction of an integrated programming environment on top, similar to what exist for Smalltalk and Self. One approach, is to port the Mjølner Development Environment (Ymer, Sif, Freja, Valhalla) to MetaBETA and investigate how they can utilize the metalevel interface.

7 Conclusion

MetaBETA extends the BETA language with a standardized metalevel interface for program extensibility and reflection. In contrast to interfaces for ordinary frameworks and libraries, the functionality provided by the metalevel interface cannot be implemented in the language itself — the functionality must be implemented with lower level primitives. The provision of the metalevel interface significantly increases the expressive power of the BETA language.

The design of the metalevel interface is based on a conceptual model of execution — the metalevel architecture. The changes that a programmer can apply to a program execution through the metalevel interface is not limited to have local effect (e.g., change the state of one or more objects), but can change the way all objects behaves. In MetaBETA such changes are modelled as runtime hooks. A hook is a small piece of code that is injected into the runtime system, which conceptually executes a program. Furthermore, queries about the structure of a program is expressed in terms of type-information. A natural and concrete substance for a BETA programmer.

Another power of the conceptual model is that it is based on an imperative view of execution. It is not based on an abstract mathematical/philosophical view of reflection, i.e., it shields the user for abstractions such as reflective towers and metacircularity. This has two benefits:

- The conceptual gab between baselevel and metalevel computation is small, making the MLI a potential tool for all programmers. This means that the metalevel interface can be employed *every time* an abstraction that works on the metalevel is implemented. On the other hand, the metalevel interface should also *only* be used in that case. Metalevel programming is not necessarily restricted to superusers.

- The metalevel architecture is so concrete that it can form a basis for an implementation. The implementation with an active runtime system is directly derived from the model.

The implementation provides several challenges for a statically-typed and compiled language. In particular, combining efficient execution with a flexible metalevel interface, that allows the implementation of most language primitives to be overridden at runtime. Our approach has been to combine technologies that have been developed from a wide variety of projects, instead of trying to (re)invent a new scheme. For example, we use ideas from dynamic languages, the persistent store area, and advanced compiler and operating system research. Given the synergy of all these previous research efforts, we are able to design a new advanced implementation of a compiled language that meets both our efficient and flexibility requirements and to have relative high confidence that our approach will succeed.

Acknowledgement

Many people have contributed with helpful discussions and remarks. Foremost, Søren Brandt, who originally initiated the work on MetaBETA (before we actually decided to call it MetaBETA). Many of the ideas presented in this report has been developed in collaboration with Søren or through some of our many discussions. Ole Lehrmann Madsen (my advisor) and Jørgen Lindskov Knudsen have provided many valuable suggestions and inside information about language design. Finally, a thanks to the other members of the DEVISE research group at the University of Aarhus: Alexandre Sousa, Elmer Sandvad, Michael Christensen, Michael H. Lassen, Erik Ernst, Søren Pingel, Lennert Slot, Henrik Bærbak Christensen, Peter Andersen, and Mads Torgersen. They all had the patience to listen to my ideas and the nerve to tell me what the thought about them.

References

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

[Agesen & Ungar 94] O. Agesen and D. Ungar. Sifting Out the Gold: Delivering Compatc Applications from an Exploratory Object-Oriented Programming Environment. In *Proceedings of the Ninth Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Portland, Oregon, 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, April 1989.

[Aho et al. 88] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, principles, techniques, and tools.* Addison-Wesley, March 1988.

[Anderson et al. 92] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[Arnold & Gosling 96] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley Publishing Company, Reading, MA, March 1996.

[Bershad et al. 95] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the spin operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 267–284, December 1995.

[Brandt & Knudsen 96] S. Brandt and J. L. Knudsen. Patterns and Qualifications in BETA: A Case for Generalization. In *Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP)*, July 1996.

[Brandt & Madsen 94] S. Brandt and O. L. 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, January 1994.

[Brandt & Schmidt 95] S. Brandt and R. W. Schmidt. Runtime Reflection in a Statically Typed Language. Technical report, Department of Computer Science, University of Aarhus, November 1995.

[Brandt & Schmidt 96] S. Brandt and R. W. Schmidt. The Design of a Metalevel Architecture for the BETA Language. In C. Zimmermann, editor, *Metaobject Protocols*. CRC Press Inc, Boca Raton, Florida, May 1996.

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

[Brandt 95] S. Brandt. Reflection in a Statically Typed and Object Oriented Language – A Meta-Level Interface for BETA. Technical report, Department of Computer Science, University of Aarhus, 1995.

[Chambers & Ungar 91] C. Chambers and D. Ungar. Making Pure Object-Oriented Languages Practical. In *Proceedings of the sixth Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Phoenix, Arizona, October 1991.

[Chiba 95] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1995.

[Dean et al. 95] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Staic Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP)*, August 1995.

[Engler & Proebsting 94] D. R. Engler and T. A. Proebsting. DCG: An Efficient, Retargetable Dynamic Code Generation System. In *The 6th International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, October 1994.

[Engler et al. 95] D. R. Engler, M. F. Kaashoek, and J. O'Tool. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, December 1995.

[Fraser & Hanson 91a] C. W. Fraser and D. R. Hanson. A Code Generation Interface for ANSI C. *Software-Practice and Experience*, 9(21):963–988, Sep 1991.

[Fraser & Hanson 91b] C. W. Fraser and D. R. Hanson. A Retargetable Compiler for ANSI C. *SIGPLAN Notices*, 10(26):29–43, Oct 1991.

[Goldberg & Robson 89] A. Goldberg and D. Robson. *Smalltalk-80. The Language*. Addison-Wesley, Reading, MA, 1989.

[Grarup & Seligmann 93] S. Grarup and J. Seligmann. Incremental Mature Garbage Collection. Master's thesis, Department of Computer Science, University of Aarhus, August 1993.

[Grønbæk & Malhotra 94] K. Grønbæk and J. Malhotra. Building Tailorable Hypermedia Systems: the embedded-interpreter approach. In *Processings of the 8th Conference On Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1994.

[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, February 1994.

[Grove et al. 94] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-Guided Receiver Class Prediction. In *Proceedings of the 9th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, October 1994.

[Hecht 77] M. S. Hecht. *Flow Analysis of Computer Programs*. North-Holland, 1977.

[Hölzle & Ungar 94] U. Hölzle and D. Ungar. A Third Generation Self Implementation: Reconciling Responsiveness with Performance. In *Proceedings of the 9th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 229 – 243, October 1994.

[Jul et al. 88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Kiczales & Lamping 93] G. Kiczales and J. Lamping. Operating Systems: Why Object-Oriented? In *Proceedings of International Workshop on Object-Orientation in Operating Systems (IWOOS)*, 1993.

[Kiczales 92] G. Kiczales. Towards a new model of Abstraction in Software Engineering. In A. Yonezawa and B. C. Smith, editors, *Proceedings of International Workshop on Reflection and Meta-level Architecture (IMSA)*, pages 1–11, November 1992.

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

[Kirby et al. 94] G. Kirby, R. Connor, and R. Morrison. START: A Linguistic Reflection Tool Using Hyper-Program Technology. In *Proceedings of the 6th International Workshop on Persistent Object Systems*, pages 346 – 365, September 1994.

[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, September 1993.

[Madsen 93a] O. Madsen. The Implementation of BETA. In J. Knudsen, O. Madsen, B. Magnusson, and M. Löfgren, editors, *Object-Oriented Environments*. Prentice Hall, September 1993.

[Madsen 93b] O. Madsen. The Mjølner BETA fragment system. In J. Knudsen, O. Madsen, B. Magnusson, and M. Löfgren, editors, *Object-Oriented Environments*. Prentice Hall, September 1993.

[Madsen et al. 93] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison Wesley, Reading, MA, 1993.

[Magnusson 93] B. Magnusson. The Mjølner Orm. In J. Knudsen, O. Madsen, B. Magnusson, and M. Löfgren, editors, *Object-Oriented Environments*. Prentice Hall, September 1993.

[Malhotra 93] J. Malhotra. Dynamic Extensibility in a Statically-Compiled Object-Oriented Language. In *Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS)*, Kanazawa, Japan, November 1993.

[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, *Proccedings of the 7th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 127 – 144, October 1992.

[Masuhara et al. 95] H. Masuhara, S. Matsuoka, K. Asai, and A. Yonezawa. Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation. In *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1995.

[McAffer 95] J. McAffer. Meta-Level Programming with CodA. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP)*, August 1995.

[McConnell 93] C. D. McConnell. *Tree-Based Code Optimization*. PhD dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 1993.

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

[Murata et al. 94] K. Murata, R. N. Horspool, E. G. Manning, Y. Yokote, and M. Tokoro. Cognac: a Reflective Object-Oriented Programming System using Dynamic Compilation Techniques. In *Proceedings of the annual conference of Japan Society of Software Science and Technology 1994 (JSSST)*, October 1994.

[Nelson 91] G. Nelson, editor. *System Programming with Modula-3.* Prentice Hall Series in Innovative Technology, 1991.

[O'Brien et al. 87] P. D. O'Brien, D. C. Halbert, and M. F. Kilian. The Trellis Programming Environment. In *Proceedings of the Second Conference On Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1987.

[Patterson 85] D. Patterson. Reduced Instruction Set Computers. *CACM*, 28(1), January 1985.

[Rao 91] R. Rao. Implementational reflection in Silica. In P. America, editor, *Proceedings of 5th European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, pages 251–267. Springer-Verlag, 1991.

[Schmidt 96] R. W. Schmidt. Dynamically Extensible Objects in a Class-Based Language. Technical report, Department of Computer Science, University of Aarhus, Marts 1996.

[Shapiro 86] M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of The Sixth International Conference on Distributed Computing Systems.*, pages 198 – 204, May 1986.

[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.

[Steele 84] G. L. Steele. *Common LISP: The Language.* Digital Press, 1984.

[Stroustrup 93] B. Stroustrup. *The C++ Programming Language.* Addison Wesley, second edition, 1993.

[Ungar & Smith 87] D. Ungar and R. B. Smith. Self: The Power of Simplicity. In *Proceedings of the Second Conference On Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1987.

[Yokote 92] Y. Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In A. Paepcke, editor, *Proceedings of the seventh Conference On Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 414 – 434, October 1992.

A The MetaBETA Intermediate Representation

The following fragment, `codeformat.bet`, defines the MetaBETA Intermediate Representation (MIR).

```
ORIGIN '~beta/basiclib/v1.4/betaenv';
(*
 *  codeformat.bet
 *
 *  Defines the Beta Virtual Machine code format as a set
 *  of BETA patterns. This is also called a code repository.
 *
 *  Written by Rene W. Schmidt.
 *
 *  Last updated: Apr. 1996
 *)

---lib:attributes---

(* CodeFormatHeader
 * ================
 *)
CodeRepository:
  (#
     (* code repository information
      *)
     minorver: @integer;
     majorver: @integer;
     date:  @integer;
     time:  @integer;
     user:  [1]@char;

     (* Metalevel information
      *)
     metaList: [1]^FragmentInfo;
     extMetaRef: [1]^ExtMetaInfo;

     (* Code
      *)
     codeList: [1]^CodeInfo;
     extCodeRef: [1]^ExtCodeInfo;

     (* string/real constant tables
      *)
     strings: [1]@char;
     reals:   [1]@real;
#);

(* Fragmententation Information
 * ============================
 *
 * Describes fragment and slots. Only attribute slots
 * are described, since all other slots are bound
 * at compile time.
 *
 *)

FragmentInfo:
  (# origin: @ExternalRef;

     name: [1]@char;
     attributes: [1]^AttributeInfo;
  #);


(* Meta Information
```

```
 * ================
 *
 * The patterns: MetaInfo, PatternInfo, AttributeInfo
 * describes this part.
 *)

MetaInfo: (# #);                    (* common superpattern *)

PatternInfo: MetaInfo
  (#
     origin: ^AttributeInfo;
     prefix: ^AttributeInfo;

     attributes: [1]^AttributeInfo;

     enterList: [1]^MetaInfo;        (* can both be simple variables, *)
     exitList:  [1]^MetaInfo;        (* -and inserted items *)

     dopart: ^CodeInfo;
  #);

AttributeInfo: MetaInfo
  (#
     name: [1]@char;
     kind: @integer;                 (* defined in attrkind.bet *)
     location: ^PatternInfo;
     qualification: ^PatternInfo;
     offset: @integer;
  #);


(* Compiled Code
 * ==============
 *
 * The following patterns contains the compiled code.
 * Note that the representation is self-contained, i.e.,
 * it can be dynamic linked without loading the
 * metainformation
 *
 *)
AlphaInfo:
  (#
     (* Intermediate Representation
      *)
     dopart:  [1]@integer;  (* opcodes are defined in bvmopcodes.bet *)
     genpart: [1]@integer;

     (* Object layout
      *)
     size: @integer;
     referenceOffsets: [1]@integer;
     prefix: ^CodeInfo;              (* used for IDT *)
     vdt: [1]^CodeInfo;              (* virtual dispatch table *)
     meta: ^PatternInfo;             (* meta-info link *)

     (* Runtime information
      *)
     fileNo: @integer;
  #);


(* External Information
 * ====================
 *
 * Defines external references in the store. Externals are
 * stored as a negative index into this table. The persistent
 * store genereates an exception in the case of negative references,
```

```
 * and looks it up in this table to see if it can resolve it. This
 * is an extened use of lazy fetching of objects.
 *
 * External references are stored as a name of a code file,
 * the canonical pattern (and attribute) we are refering,
 * and the kind of object needed.
 *)

ExternalRef:
  (# name: [1]@char;  (* name of store *)
     no: @integer;    (* entry in store *)
  #);

ExtCodeInfo:
  (# kind: @integer;
     store: @ExternalRef;
  #);


ExtMetaInfo:
  (# name: [1]@char;
     fragno: @integer;
     patno: @integer;
  #);

ExtMetaPrefixInfo: ExtMetaInfo(# pi: ^PatternInfo #);
ExtMetaQuaInfo:    ExtMetaInfo(# ai: ^AttributeInfo #);

ExtKindPrototype: (# exit 1 #);
ExtKindGenCode:   (# exit 2 #);
ExtKindDoCode:    (# exit 3 #);
```

B Specification of the Alpha Code

This appendix describes the abstract machine-code (Alpha-code) that is used by the MetaBETA compiler and runtime system to represent do-parts. The Alpha-code is, in contrast to MetaBETA code, a low-level representation that can easily be translated into machine-specific instructions and on which machine-specific optimizations can be applied.

Suffixes	Type
R	Real
I	Integer
C	Char
B	Bool
P	Any Pointer

Table 4: Type modifiers for Alpha opcodes.

A BETA do-part is represented as a *Directed Acyclic Graph* (DAG) where Alpha operators are the nodes in the graph. This eliminates the need for doing register-allocation in the Alpha code, thereby making it less architectural-dependent and faster to generate. An example of an Alpha DAG is shown in Figure 14.

The Alpha-code consist of 40 opcodes, where 19 of these opcodes have type modifiers (see Table 5). This gives a total of 70 different operators. The type-modifiers are shown in Table 4. Depending on the operation, the type modifier either specifies the resulting type or the type of the arguments.

Execution of Alpha code consist of a depth-first traversal of the DAG, execution each node only once. Hence, if a node is refered more than one (e.g., a common subexpression) it is only calculated once. The execution of Alpha code assumes to implicit registers: *CurrentObject* and *NextObject*, which respectively refer the current execution object and the object that is being manipulated or called.

The operators are divided into five different groups:

1. **Variable Access.** Objects are accessed by using the operations `ADDRC` and `ADDRN`. They return an address relative to *CurrentObject* or *NextObject* respectively. `INDIR` fetches a value from a given address, and `CNST` returns a constant. `ASSGN` assigns a value into a given address.

2. **Conversion.** These three operators convert between integers, reals, and characters.

3. **Calculation.** This group defines all value manipulation operations, i.e., basic arithmetic, boolean arithmetic, predicates for testing, and bit-manipulation operations.

4. **Control Flow.** Control flow within a do-part is represented with the high-level control operators: `REGION`, `LEAVE`, `RESTART`, and `IF`. They reflect that MetaBETA does not have any goto-imperative, so the control-flow is tree-structured [Hecht 77]. This simplifies code generation and optimization. `COMP` is used to string statements together.

5. **High-level.** The Alpha code abstracts the implementation of `INNER` (`INNER`), calling conventions (`EXEC`, `RET`, `RETI`), object allocation (`ALLOC`), and trap to the runtime system (`TRAP`). The `ALLOC` operator assigns the newly allocated object to the *NextObject* register. `EXEC` pushes *CurrentObject* on a stack, and assigns *NextObject* to *CurrentObject*, before it executes *NextObject*. `RET` and `RETI` restore the previous value of *CurrentObject* and returns to the calling code. `TRAP` is used to call the built-in functionality in the runtime system, e.g., the meta-level interface.

The choice of operations are based on the lcc C compiler [Fraser & Hanson 91a, Fraser & Hanson 91b], which uses a similar set of instructions. However, the control-flow instructions are borrowed from CORTL [McConnell 93]. C. McConnel shows in his Ph.D. thesis that these high-level control-primitives leads to simpler optimization algorithms than traditional *Control Flow Graphs* [Aho et al. 88].

Group	Kids	Args	Opr.	Type	Description
1		offset	ADDRC	P	address relative to Current object
		offset	ADDRN	P	address relative to New object
		val	CNST	BIR	defines a constant
	1		INDIR	BCIPR	fetch
	2		ASSGN	BCIPR	assignment
2	1		CVR	IC	convert from Real
	1		CVI	CR	convert from Integer
	1		CVC	IR	convert from Character
3	1		NEG	IR	negation
	2		ADD	CIPR	addition
	2		SUB	CIPR	subtraction
	2		MUL	IR	multiplication
	2		DIV	IR	division
	2		MOD	I	modulus
	2		AND	B	boolean AND
	2		OR	B	boolean OR
	1		NOT	B	boolean NOT
	2		EQ	IR	Equal predicate
	2		NEQ	IR	Not equal predicate
	2		GT	IR	Greater Than predicate
	2		GTE	IR	Greater Than or Equal predicate
	2		LT	IR	Less Than predicate
	2		LTE	IR	Less Than or Equal predicate
	2		BAND	I	bitwise AND
	2		BOR	I	bitwise OR
	2		BXOR	I	bitwise XOR
	1		BNOT	I	bitwise NOT
	2		LSH	I	left shift
	2		RSH	I	right shift
4	1		REGION		control enter/exit point
	1		LEAVE		jump to end of a region
	1		RESTART		jump ro start of region
	2		IF		branch if true
	2		COMP		composition of statments
5	1	no.	ALLOC	CP	create object (static)
	1	no.	EXEC	CP	execute do-part of object
	1		INNER	P	do INNER call
			RET		return from call
			RETI		return from INNER
		no.	TRAP		trap to built-in function

Table 5: The Alpha operators.