Contents

Part I

The gbeta Virtual Machine

Chapter 1

Introduction

This thesis documents our work performed at the DAT6 semester, Department of Computer Science, Aalborg University. The result of this work is a virtual machine which executes gbeta bytecodes. We have implemented a virtual machine based on a format for gbeta bytecodes provided to us by the developer of the experimental, strongly-typed, object-oriented language called gbeta, namely Erik Ernst. gbeta is a generalization of the language BETA (see, [KLM93], amongst others) with features such as a general mechanism for merging patterns allowing for expressiveness in the form of, e.g., propagating pattern combinations and the facility to create class-like patterns at run-time.

A gbeta virtual machine already exists, but we created one with better performance. This work was initiated at the DAT5 semester, and resulted in the release of the first ever stand-alone gbeta virtual machine. We have continued our work on the previous release of our virtual machine focusing on how to improve its performance. The new release is called gVM.

The work on our previous virtual machine release was focused on correct semantics; i.e., that bytecodes were executed correctly on our virtual machine. Much of this work has been reused in gVM. Consequently, we refer to [JWJ00] for a complete understanding of gVM. In this thesis only new features are described. The new features of gVM are compiling gbeta bytecodes to Java source code, making use of information about statically determined patterns, and using a cache for fast access to statically determined patterns already created. The intent of these new features is to increase performance.

Implementation of the new features in gVM and subsequent benchmarking results combined with the theoretical discussions about the differences between gbeta and BETA in terms of performance enable us to conclude on the limitations of gbeta. In particular, the propagation feature and the facility to create patterns dynamically in gbeta are obstacles to getting better performance, as discussed in this thesis. We have implemented the capability to use knowledge about statically determined patterns. The intent was to avoid merging pat-

terns at run-time in all cases (merging is not needed if patterns are statically determined).

The intent of this thesis is to discuss the fundamental differences between gbeta and BETA, and determine what can and cannot be done to improve on the performance of gbeta. The implementation of bytecodes which use statically determined patterns is important in this regard.

In fact, the goals of this semester are:

- to create a virtual machine that is faster than the previous virtual machine release
- to use the implementations of the new features in an attempt to document what is expensive in gbeta compared to BETA

Firstly, we aim to create a virtual machine that is faster than the previous one. In order to achieve this we opted for using a caching strategy and compilation; i.e., compiling gbeta bytecodes into Java source code. Secondly, in a combined effort to both improve the performance of gVM and understand the fundamental differences between gbeta and BETA in terms of performance, we opted for using knowledge about statically determined patterns.

1.1 Contents of Chapters

In the next section we discuss the expressiveness of gbeta and its impact on performance. We use the implementation of BETA as a reference on how we can possibly improve performance. Other ideas about how we can improve performance are also discussed. Chapter 3 introduces the new features of gVM and discusses how they improve gVM's performance. In Chapter 4 we discuss how the new features are implemented. A number of benchmark programs and a methodology for benchmarking BETA binaries, gVM in interpretation mode, gVM in compilation mode, and gbeta, respectively, and the benchmarking results are presented in Chapter 5. Finally, we conclude in Chapter 6.

Chapter 2

gbeta and BETA

or Why gbeta is Slower than BETA

In this chapter we elaborate on the differences between gbeta and BETA – for tutorials on programming in gbeta and BETA we refer to [Ern01b] and [MPN93], respectively. In the following section we discuss the expressiveness of gbeta and that of BETA and compare those aspects which are relevant to an understanding of performance issues in gbeta; we do this by discussing propagation [Ern99b], but the ability to create class-like patterns [Ern99a] at run-time is also relevant.

In Section 2.1.1 we continue the elaboration on performance issues in gbeta and BETA. The discussion in this section will be focused on the differences between the gbeta and BETA implementations. Further elaborations on the implementation of gbeta can be found in [JWJ00] and [Ern99a] whereas [KLM93] documents the implementation of the Mjølner BETA System in version 4.1 which is used by us for comparison with gbeta.

Finally, we discuss various ideas to improve performance in gVM inspired by the aforementioned implementation of BETA.

2.1 gbeta versus BETA

The inheritance mechanism in gbeta differs from the one in BETA in that a mixin-based inheritance mechanism is used – see page 13 in [JWJ00] for a definition of *mixin* as used in gbeta and see pages 39–45 in [Ern99a] for more on mixins. This allows for more expressiveness compared to the non-mixin-based, single-parent inheritance mechanism used in BETA. In fact, inheritance in BETA is possible only by prefixing the definition of a pattern (class) with the name of a super-pattern (super-class). In gbeta, however, inheritance is achieved by linearization of lists of mixins (patterns) as described in [Ern99b] and elaborated on in the next section.

The major difference in expressiveness between gbeta and BETA and the most significant in terms of impact on performance is the support for propagating pattern combinations in gbeta. In gbeta, however, it is also possible to specialize existing objects and defer the creation of class-like patterns until run-time.

The combination of the use of linearization, the use of virtual attributes, and the use of `INNER` statements makes it possible to propagate combinations of patterns (see page 11 in [Ern99b]). We will describe propagation of patterns by the use of an example.

Basically, what makes it possible to propagate patterns is the `MERGE_ptn` bytecode (see [JWJ01] for a link to a postscript file describing the gbeta bytecodes) combined with the use of origins, multiple part objects, and the semantics of virtuals (see pages 267–279 in [Ern99a]). The `MERGE_ptn` bytecode does the actual linearization of lists of mixins using the linearization algorithm described in [Ern99b] and elaborated on in the next section.

In Figure 2.1 we show a very simple example of how class-like patterns can be propagated in gbeta. The pattern `mammal` is the super-pattern of the patterns `meatEater` and `catSpecies`, respectively. Each of these two patterns further-binds the virtual-method-like pattern `printString` defined in the `mammal` pattern.

It is possible to combine the behaviours of the two `printString` method-like patterns further-bound in `meatEater` and `catSpecies`. This is done in line 26 indirectly by the propagation of the patterns `meatEater` and `catSpecies`. A new pattern is created which is a list of the mixins of the two class-like patterns. If we instantiate this pattern and invoke its `printString` attribute (implementing the contributions from the `printSting` attributes of the two sub-patterns), we see that the behaviours of the super-pattern and both of the sub-patterns are combined. This program displays the string `I am a mammal called Simba which likes meat, and I also like to purr`.

The order in which the strings are displayed depends both on the use of `INNER` statements, the use of virtual attributes, the actual linearization of the patterns which creates the merged pattern, and which pattern is on the left side of the pattern combination operator `&`.

The `INNER` statements are necessary to call upon the behaviours of sub-patterns. Like in BETA, the `printString` method of the super-pattern is invoked first, and then the `printString` attribute of the sub-pattern is invoked (the `INNER` statement of the `printString` attribute in the `mammal` pattern invokes the `printString` attribute in the `meatEater` pattern) [MPN93].

Virtual patterns are necessary for propagation because method-like patterns must be further-bound. For example, if we had chosen not to let `meatEater` and `catSpecies` inherit from `mammal` and let the `printString` methods be non-virtuals, trying to propagate these two classes would have resulted in a warning and no propagation. The reason for this warning is that the resulting pattern would have two unrelated method-like patterns called `printString`. In fact, an

instance of the resulting pattern would be an instance of both arguments to `&` merged together, but its most-specific `printString` method-like pattern would be the one obtained from the right hand argument to `&` (the other `printString` may be used by executing the code `lion(:meatEater:).printString` instead of `lion.printString`[1]).

It is also not unimportant which of the `meatEater` and `catSpecies` patterns is on the left hand side of the `&` pattern combination operator in line 26 – if the two were reversed, the program would display `I am a mammal called Simba, and I also like to purr which likes meat`.

In CLOS, the use of methods qualified with either `:before`, `:after`, or `:around` combined with the use of the `call-next-method` method which calls the method with the same message selector in the super-class allows expressiveness similar to propagation [Dal01]. Unlike gbeta, however, propagation can be done at one level only.

A class which uses something that necessitates a post-hoc addition of a "super-class" is called a mixin in CLOS – so using mixins is a "coding convention" [BC90] in CLOS. Mixins in CLOS are hence used as abstract sub-classes to specialize super-classes and their methods.

Method combination can also be simulated in a multiple-inheritance, non-mixin-based language such as C++. Multiple inheritance makes it possible to create a new, more complex class which inherits sets of non-abstract routines from different classes. However, it is necessary to use the scope resolution operator `::` to handle ambiguities in the new class [Joy01] and manually perform the composition which is inconvenient.

2.1.1 Performance Impact of Differences

The major difference between the gbeta language and the BETA language is how mixins are used in inheritance and merging of virtual attributes. Both BETA and gbeta have mixins and virtual attributes and use `INNER` to call methods in sub-classes. In this respect, the languages are identical.

To explain why gbeta is a generalization of BETA, it makes sense to discuss mixins in BETA. In BETA, inheritance is only possible by prefixing definitions of patterns [BC90]. Mixins are, therefore, "not liberated" [Ern99b] from their super-class and are just class increments. This means that when the most specific mixin (see [JWJ00] for a definition) is known, a pattern is fully determined. This is not the case in gbeta because it is possible to merge patterns.

An implication of the existence of a general facility for merging patterns is that the most specific mixin does not determine a pattern fully. The mixins preceding the most specific mixin in a merged pattern are not generally known.

[1] The gbeta `(::)` type cast operator resembles the `()` operator in Java and C++. In the example above, it effectively tells the gbeta type system that the first operand of `&` (the pattern combination operator) is `p` but `p` about which is known only that it is the `object` pattern.

```
1  -- betaenv:descriptor --
2
3  (#
4   mammal:
5    (#
6      name : @String;
7      printString :< (#
8      do 'I am a mammal called '+ name -> stdio; INNER
9    #);
10  #);
11
12  meatEater : mammal
13   (#
14     printString ::< (#
15     do ' which eats meat' -> stdio; INNER
16   #);
17  #);
18
19  catSpecies : mammal
20   (#
21    printString ::< (#
22    do ', and I also like to purr.' -> stdio; INNER
23   #);
24  #);
25
26 lion : @meatEater & catSpecies;
27
28 do
29
30 'Simba' -> lion.name;
31 lion.printString;
32
33 #)
```

Figure 2.1: A propagation example

Because the most specific mixin of a pattern does not generally determine it fully and patterns may be created at run-time, the same efficiency as in a BETA implementation is not easily achieved in an implementation of gbeta. Because we do not generally know the number or order of the mixins of a pattern at run-time in gbeta, part objects must be linked together as opposed to in BETA where they are in-lined in the enclosing object (see pages 402–403 in [KLM93]). In effect, we cannot access every attribute of an object with an offset calculation (like in BETA), but have to access the attributes of linked part objects instead.

This is more expensive; e.g., if we do not know the number or order of the mixins of a pattern statically, we cannot combine a number of up-steps or down-steps to traverse part objects in just one step looking for attributes (see pages 34–37 in [JWJ00] for a definition of and an elaboration on up-steps and down-steps, respectively). During traversals of up-steps or down-steps, we have to compare the main part ids of increasingly more specific or general part objects [JWJ00]. In fact, the increased number of objects in gbeta also means that garbage collection is more expensive.

There are also cases in BETA where it is impossible to determine the exact pattern denoted by an attribute at compile-time. The difference is that the prefix of a list of mixins is always determined. If this list has the elements `[M1, M2, M3]` at run-time, we know that it has the elements `[]`, `[M1]`, `[M1, M2]`, or `[M1, M2, M3]` at compile-time. If we know, e.g., that this list has the elements `[M1, M2]` at compile-time, then `M1` will be the most-general mixin (see [JWJ00] for a definition) and `M2` will be the second-most-general mixin (in gbeta, if we know that a list of mixins has the elements `[M1, M2]` at compile-time, the list may, e.g., have the elements `[M5, M1, M3, M4, M2]` at run-time). This means that we do not have to compare the main part ids of increasingly more specific or general part objects during traversals of up-steps or down-steps, like in gbeta.

Linearization of lists of mixins (merging of patterns) also imposes a performance penalty compared to BETA where it is not used. This is because it takes place at run-time. The linearization algorithm implemented in gVM takes two lists of mixins and creates a new list. What basically happens in a step of the algorithm is that an element from one of the two argument lists is chosen and added to the result list. If an element is in both of the argument lists, it is only added once (see pages 12–13 of [Ern99b] and pages 65–69 of [Ern99a] for details about linearization as implemented in gbeta). The complexity of the implemented linearization algorithm is $O(N \times M)$ where N and M are the number of mixins in the first and second argument list, respectively.

In fact, since virtuals are also linearized in gbeta as opposed to in BETA, this is a significant performance inhibitor. In gbeta, virtuals may be further-bound with any pattern that the linearization algorithm can handle. This is not the case in BETA which allows a virtual to be further-bound to a sub-pattern of itself only. This is shown in the BETA code in Figure 2.2, which results in an error if the `v` pattern in line 2 is not further-bound to a sub-pattern of `x`.

```
1 a: (# v:< x #);
2 b: a(# v::< y #);
```

Figure 2.2: An example showing the dependency of the `y` pattern on the `x` pattern

The reason for this BETA restriction on the patterns with which a virtual can be further-bound has to do with its single-inheritance property. If we allowed a

virtual to be further-bound with any pattern, we would have to use a linearization algorithm when merging the virtual contributions or opt for inhertance from virtuals to get the same effect.

Linearization of virtuals is not expensive in the rather common case where virtual contributions are singleton patterns. In this case, we may simply collect them in order from the enclosing object [Ern99b].

Linearization is also necessary in CLOS to make sure that a mixin extends the behaviour of the correct super-class. This is accomplished by making sure that the mixin is placed before the super-class at the source-code level. Extending the behaviour of the correct super-class can be done by using the `call-next-method` method which simply calls the method with the same message selector in the super-class or by overriding a method in the super-class.

There is also another reason as to why BETA performance may be difficult to reach. In gbeta, it is possible to get a reference to an instance of a primitive pattern, to get a reference to a primitive pattern, or inherit from a primitive pattern.

As a result, the value of, e.g., an integer object cannot be saved in the bit-pattern of its reference without implementing a boxing/unboxing scheme. A boxing/unboxing scheme would use the bit-pattern representation of a reference to an instance of a primitive pattern to store its value and only create the instance on-demand (i.e., when someone asks for a reference to it). Unfortunately, our choice of Java as our implementation language prohibits implementing such a scheme, as noted in 4.1.1.

There are many obstacles in reaching performance which comes close to BETA, as described above. Some of these obstacles are so restrictive that it seems impossible to reach BETA performance, e.g., the need to linearize virtuals in gbeta as opposed to in BETA. We did, however, examine some ideas in more detail with the intent of improving the performance of gVM. Some of these were implemented and some were not. The ones which were not implemented are described in the next section. The ones which were implemented are described in the next chapter.

2.2 Alternative Extensions to gVM

A number of ideas for improving the performance of the existing gbeta virtual machine were considered. These ideas were focused on the following.

- the attribute initialization scheme
- run-time path traversals
- object creation

We started by focusing on these aspects of gVM, because they influence the execution speeds of nearly all gbeta programs – only the most simplistic gbeta programs have no attributes that need initialization, no run-time paths to traverse, and create no objects (see [JWJ01] for at link to a directory containing most of the gbeta programs ever written if you need convincing).

2.2.1 An Alternative Attribute Initialization Scheme

In gbeta, the initialization of one attribute can depend on the prior initialization of another [JWJ00] and, in turn, another and so on. Consequently, our existing implementation of gVM performs on-demand initialization of attributes. This means that a check is made to determine whether or not an attribute has been initialized before it is accessed and if not, it is initialized. The existing scheme is similar to that handling "G-code" which is used to initialize attributes in BETA [Sch01], since this code can also do "everything". There are differences, however. For example, G-code that initializes aliases (different names for the same instance of a pattern) will never be generated in BETA.

If we could come up with a scheme that determined an order for attribute initialization in which only attributes guaranteed to depend only on attributes already initialized are ever accessed, then we could avoid the run-time checks of the existing scheme.

A scheme to determine a "safe" way to initialize attributes is difficult to implement, at best. This is because determination of all dependencies between attributes is an undecidable problem, so no scheme works in all cases. The reason that it is an undecidable problem is the presence of arbitrary code.

We explain this by the use of a simple example. Determination of a safe way to initialize attributes is similar to the way the verification algorithm in Java determines whether class-files are safe. The verification algorithm accepts all class files that fulfil certain criteria and rejects all that do not.

In Figures 2.3 and 2.4 we show a situation in which it is possible to find a safe sequence of initializations and a situation in which it is impossible, respectively.

In Figure 2.3 it is possible to find a safe order in which to initialize the attributes `x` and `y`. If we initialize the attribute of the enclosing part object denoted by the `ADD-mainpart ... origin ...` bytecode of `y` first, initialize `y` second, and finally initialize the `x` attribute, we have a safe order.

The situation in Figure 2.4, however, is "unsafe". The reason is the call to evaluate the `doit` attribute of the enclosing part object in line 10. As a result, arbitrary code may be executed and there is no way to guarantee that it is safe to delay the initialization of `x` until the initialization of `y` has finished.

Another attribute initialization scheme would be to reduce our demands, and accept fewer gbeta bytecode files as input. We could then have gVM output a warning in unsafe situations automatically (or resort to run-time checks, or

```
1  MainPart("‘101-225"
2
3   "x/0": (
4     PUSH-ptn {"y/1"}
5     NEW,_ptn->obj
6     INSTALL-obj 0
7   )
8
9   "y/1": (
10    PUSH_ptn {<-2,"integer/4"}
11    ADD_mainpart ‘109-148 origin {}
12    INSTALL-ptn 1
13  )
14
15  |
16
17     ...
18
19  )
```

Figure 2.3: A safe situation

both). It is then the responsibility of the programmer to avoid situations which will result in gVM outputting warnings or to create ad hoc proofs af the safety.

The use of virtual attributes also complicates things because a virtual attribute is initialized by collecting contributions from every part object of the current object (this is implemented in gVM by the use of the `GATHER_virt` bytecode). This means that if there is a virtually declared attribute in a main part, all the attributes containing bytecodes that add to the initialization of this attribute in other main parts must be initialized in advance.

Given all these difficulties with an alternative attribute initialization scheme, we have decided not to expend effort on this possibility for optimization.

Problems with initializations, however, are by no means unique to gbeta. In Java, the initialization scheme employed imposes a constraint on the order of variable initialization. Consider Figure 2.5 which shows an impossible way to initialize the variables `one` and `two` in Java. It is impossible because the initializations of the numbers are performed in textual order, starting with the first variable of the class and ending with the last. The problem occurs in line 3 where `one` is undefined.

A scheme similar to the one employed in Java could also be employed in gVM, but this would mean a restriction on the way programmers are allowed to initialize attributes.

```
MainPart("‘51-112"

 "x/0": (
   PUSH-ptn {"y/1"}
   NEW,_ptn->obj
   INSTALL-obj 1
 )

 "y/1": (
   CALL {<-1,"doit/1"}
   INSTALL-ptn 1
 )

|

   ...

)
```

Figure 2.4: An unsafe situation

```
public class Numbers  {

  int two = one + 1;

  int one = 1;

}
```

Figure 2.5: An impossible way to initialize variables in Java

2.2.2 Alternative Run-time Path Traversals

Traversals of run-time paths in gbeta are more expensive than the similar traversals of the object graph in BETA (see Section 5 for results which document this).

In BETA, when the most specific mixin of a pattern is known, it is statically determined as opposed to in gbeta. This means that attributes can be looked up more easily in BETA than in gbeta. In BETA, there is no need to compare the main part ids of the mixins of increasingly more general part objects during traversals of up-steps or the main part ids of the mixins of increasingly more specific part objects during traversals of down-steps. Traversals of out-steps are performed in the same way in BETA and in gbeta which means that their costs

are similar (see Section 5 for details).

We examined various ideas for increasing the performance of run-time path traversals. These ideas focused on how to decrease the number of steps to traverse in a path at run-time. We were forced not to change the way traversals of up-steps and down-steps are implemented because the expressiveness of gbeta would be affected (Section 2.1 explains why). The ideas we examined were mostly influenced by the implementation of BETA [KLM93], and we wanted to determine if they could be applied to gbeta. The ideas were:

- to add the attributes of the part objects of a statically determined object (which is an instance of a statically determined pattern) to the origin part object, and then discard the old part objects
- to add the attributes of the part objects of a statically determined object to the most specific part object of this object and then in-line the `INNER` calls between the part objects. All the part objects of this object except the most specific would then be discarded.

The benefit of the first idea is avoiding an out-step by discarding the current part object. Pages 38–44 of [JWJ00] contains an elaboration on the different steps (out-steps, amongst others).

Unfortunately, this idea has a serious flaw. If the attributes of the part objects of a statically determined object were in-lined in the origin part object and the part objects of the statically determined object discarded, then it would no longer be possible to access this object; i.e., it would no longer be possible to obtain a reference to it. The reason is that the attributes of the part objects of this object are in-lined in a part object which has another `thisObject` reference (which is a reference to its containing object that every part object has, see pages 23–27 of [JWJ00] for details).

Even if we solved the issue of determining the containing object, we would still have to analyze the input program for occurrences of references to it. Every run-time path traversing the discarded object would have to be located and changed because the in-lined attributes may have different indices in the origin part object.

In fact, a possible solution may be difficult to implement in Java because Java does not permit manipulations of memory at the bit-level. This is needed if we are to access one instance of `UserDefinedPartObject` as another instance to resolve the issue of the incorrect use of the `thisObject` reference when the attributes of part objects are in-lined in their origin part object). A `UserDefinedPartObject` is a part object created by the user as the result of writing arbitrary code. In contrast, a predefined part object (e.g., a `PredefinedIntegerPartObject`) is predefined by us like the name suggests and accessible to user code (see pages 21–27 of [JWJ00] for details on predefined types).

Also, there would be a problem if the part objects to be in-lined did not have the same origin part object. If they were in-lined in their origin part objects, they would no longer have the same `thisObject` reference.

The second idea (in-lining `INNER` calls between the part objects of instances of statically determined patterns) also posed a problem. We wanted to avoid traversals of up-steps in run-time paths because these occur frequently (unlike down-steps) due to the rules for global lookup of attributes in gbeta [Ern99a].

We examined the possibility of adding the attributes of a part object associated with a more-general mixin of a statically determined pattern to a part object associated with a more-specific mixin of the same pattern, and then discard the more-general part object.

For this idea to work we would have to in-line the `INNER` calls in each of the do-parts of the main parts associated with the mixins. To explain how in-lining of `INNER` calls would work, we refer to Figure 2.6. In this figure, `a` represents the main part of the mixin associated with the most-general part object, `b` represents the main part of the mixin associated with a more-specific part object, and `c` represents the main part of the mixin associated with the most-specific part object. Of course, all of these mixins are mixins of a statically determined pattern. The `B`s represent various bytecodes (except for `INNER` bytecodes). The result of in-lining the `INNER` calls would be that the do-part of `c` would contain the bytecodes `B1 B2 B3 B6 B7 B8 B9 B4 B5` .

```
1 a.dopart = ( B1 B2 B3 INNER B4 B5 )
2 b.dopart = ( B6 B7 INNER )
3 c.dopart = ( INNER B8 B9 )
```

Figure 2.6: In-lining INNER calls

The benefits of in-lining `INNER` calls are:

- one or more up-steps of a run-time path are eliminated
- the overhead of executing `INNER` calls themselves is eliminated

However, in-lining `INNER` calls also has drawbacks. We would have to locate and change all run-time paths which involve part objects separated by `INNER` calls. Also, since a more-general part object need not have the same origin part object as a more-specific, some extra administration would be required to discern them.

In addition, we have to distinguish between in-lined objects and other objects of the same type during traversals of run-time paths which means that we have to determine of which pattern the object is an instance.

The need for this determination has to do with the indices of attributes collected

in the most-specific part object of an object and is explained by the use of an example.

The intent of this example is to describe a piece code of code that will force us to determine the pattern of an object during traversal of a run-time path if this idea is implemented. In Figure 2.7, `x` is statically determined and consists of three part objects. The attributes of these three part objects are collected in the most-specific part object and the other two part objects discarded, as described.

In Figure 2.7 `x` is statically determined and the attributes of instances of `q` and `r`, respectively, can be added to `x` as intended. If this idea were implemented, then the first attribute of the one part object of `x` would be the attribute of the part object of `q` and the second attribute of the part object of `x` would be the attribute of the part object of `r`.[2].

In essence, the problem is that there can be different instances of `r`. There can be an `x` instance or an instance of some sub-pattern of `r` which is not `x` and not in-lined here, and when we collect the attributes from the part objects of the instances of `q` and `r`, respectively, in `x`, an attribute like `attribute1` may get another index in `x` than, e.g., in an instance of some other sub-pattern of `r`. Consequently, when we search for `attribute1`, we do not know which index it has unless we know which instance it is.

An assignment like that in line 12 means that an object reference like `aQ` denotes `x`. The implication of this assignment is that when we search for `attribute1`, we must search at the index it has in `x`. If `aQ` denoted, e.g., an instance of a sub-pattern of `q`, we would have to search for `attribute1` at the index it has in `q`.

The implication of this is that code like that in line 14 must be executed differently depending on whether `aQ` denotes the `x` instance or another instance. When executing the code in line 14, we have to traverse a direct-lookup-step (see page 37 of [JWJ00] for an elaboration), and determine the pattern of the instance to access the correct attribute.

In fact, implementing this idea means that we have to check every instance in the code for the direct-lookup-step and only the `x` instance will traverse the up-steps faster. We would also have to change the code in places throughout the entire input program which means that separate compilation of the modules of the program cannot be achieved.

Because of the complexity associated with improving performance of run-time path traversals and employing a new attribute initialization scheme, we decided not to pursue it any further. Our primary focus then became the elimination of the extra layer of interpretation needed in our previous virtual machine implementation and exploiting information about statically determined patterns (see Section 3.1.3 for details).

[2] This is only one way to add the attributes to `x`. We could also choose to add the attributes in another order.

```
1     ...
2  (#
3
4   p: (# do INNER #);
5   q: p(# attribute1: @integer do INNER #);
6   r: q(# attribute2: ^string do INNER #);
7   x: @r;
8   aQ: ^q;
9
10 do
11
12  x[]->aQ[];
13    ...
14  3->aQ.attribute1;
15
16 #)
17    ...
```

Figure 2.7: An example demonstrating ways to access a pattern

2.3 The Prototype Paradigm for Object Creation

Although the efficiency of *construction* of statically determined patterns[3] has been improved in the current implementation of gVM (as noted in Section 3.2 and detailed in Section 4.2.2), the efficiency of their *instantiation* has not.

We considered using a caching technique to effectively implement the prototype paradigm for object creation [US91] for instances of static patterns. A caching technique would potentially allow us to create instances of static pattern without executing any attribute initialization code and hence avoid its expensive run-time path traversals (see Section 5).

The prototype paradigm for object creation would be used whenever a new instance of a static pattern was required. New instances of dynamic patterns would (still) be created through instantiation, as described on page 15–16 in [JWJ00].

In essence, using the prototype paradigm for object creation, an instance of a static pattern is created by simply *copying* an existing (prototype) instance in the cache. In copying an object, its state (including its object graph) is copied as well.

We found out that the principal problem with such an object-copying scheme is dealing with the attributes of the copies. For correct initialization, an ob-

[3]Henceforth, we shall use just the terms "static pattern" (or "compile-time pattern") and "dynamic pattern" (or "run-time pattern") to refer to statically determined patterns and dynamically determined patterns, respectively.

ject attribute may depend on the run-time environment. These dependencies manifest themselves in run-time paths present in the initialization code for the object attribute. As noted in [JWJ00], the result of a run-time path traversal is highly context-dependent. For example, two traversals of the same run-time path initiated in different part objects can lead to different entities. Likewise, traversals of two different run-time paths starting in the same part object can lead to the same entity.

Consequently, any scheme to create an object by simply copying an existing one will fail (in the general case) in that each attribute of the new object will have been initialized in a context (i.e., a part object) of the old one. As an example, consider Figure 2.8.

```
1  MainPart("‘26-63"
2     "p/0": (
3        PUSH-ptn_"object"
4        ADD-mainpart ‘35-60 origin {}
5        INSTALL-ptn 0
6     )
7  |
8  )
9
10 MainPart("‘35-60"
11 |
12 )
```

Figure 2.8: Context-dependent attribute initialization code

This figure shows the initialization code for an attribute named `p` which occupies attribute index `0` in the *betaEnv part object* (as shown in Figure 5.1 on page 24 in [JWJ00])[4].

The initialization code pushes the "empty" pattern (i.e., the pattern whose list of mixins is empty) onto the expression stack (`PUSH-ptn_"object"`). It then adds the mixin which is associated with the main part with id `‘35-60` and has origin in the part object obtained by traversing the run-time path {}, i.e., the betaEnv part object (which is the "current" part object), to it (`ADD-mainpart ‘35-60 origin {}`). Finally, `INSTALL-ptn 0` assigns the resulting pattern to attribute index `0` in the part object whose attribute is currently being initialized, i.e., the betaEnv part object.

If two instances of the *betaEnv pattern* (i.e., two *betaEnv objects*[5]) were required and instance number two was created by copying the betaEnv object whose `p`

[4]The code in Figure 2.8 is the result of compiling the "002.gb" program included in version 0.81.13 of the gbeta distribution (with no optimizations turned on).

[5]For an illustration of a betaEnv object including its immediate run-time environment we refer to Figure 5.6 on page 35 in [JWJ00].

attribute is initialized using code shown in Figure 2.8 (including its single part object and its `p` attribute), then the `p` attribute in the copy would have been initialized in the context of the original betaEnv object and not the copy.

In other words, the mixin of the `p` pattern attribute in the new betaEnv object, though correctly associated with the main part with id `'58`, will have origin in the wrong betaEnv part object (i.e., the betaEnv part object of the *old* betaEnv object) because traversal of the run-time path {} relative to the betaEnv part object of the *new* betaEnv object in the initialization code for `p` is omitted.

The fact that an origin is wrong is obviously bad because it violates the semantics of the source language. Even worse is the fact that no inexpensive scheme to correct it appears to exist. Firstly, *without* knowledge of a run-time path leading to the origin part object, it is impossible. Secondly, *with* knowledge of such a run-time path, it is impossible in general without actually traversing the run-time path, in which case nothing is gained in terms of performance. Figure 2.3 shows our best, general algorithm to correct the origins of the mixins of the part objects of an object created by copying an existing object.

```
1  procedure correct_origin(object)
2   for each part object
3    for each attribute
4      if is_object(attribute) then
5        correct_origin(attribute)
6      else if is_objectref(attribute) then
7        correct_qualification(attribute)
8      else
9        for each mixin
10         if length(run-time path to origin part object) > 0 then
11           traverse run-time path in context of current part object
12           set origin to resulting part object
13         else
14           set origin to current part object
15     end if
16       end for
17     end if
18  end for
19 end procedure
```

Figure 2.9: Algorithm to correct the origins of the mixins of the part objects of an object created by copying an existing object

Notice that this algorithm avoids traversals of run-time paths only when the run-time path is the empty one. Since traversals of the empty run-time path are already practically free, and any potential saving due to their avoidance

would be greatly outweighed by the extra costs associated with creating and using a cache of prototype objects for object creation, we have not implemented this algorithm in gVM. For this reason, the `correct_qualification` procedure referenced in Figure 2.3 is not shown.

In conclusion, using the prototype paradigm for object creation is generally *not feasible* because object creation depends on executing attribute initialization code for correctness – it is an integral part of the gbeta language semantics.

Chapter 3

Extensions to gVM
or Faster is Better

In this chapter we discuss extensions to improve the performance of gVM, our implementation of a virtual machine for gbeta. Whereas Section 2.2 discussed alternative ideas and on what grounds we ruled them out, the following sections present the ideas which we opted to implement. Section 4 details the implementation of these extensions to gVM.

3.1 From gbeta Bytecode to Java Source Code

In order to improve the performance of gVM, we extended it in four major ways:

- by in-lining method calls,
- by replacing time-inefficient data structures,
- by compiling to Java source code,
- by exploiting information about statically determined patterns.

In what follows, each of these four extensions to gVM is presented in its own subsection.

3.1.1 In-lining Method Calls

In-lining (non-recursive) procedure calls is one of the traditional approaches to improving performance in procedural languages, and combined with techniques such as customized compilation and message splitting [USCH92], performance improvements can be achieved in object-oriented languages as well.

In the source code generated by gVM, only method calls that occur at the Java level are in-lined. In particular, all method calls that pertain to traversing run-time paths are in-lined. This is possible because run-time paths themselves are statically determined [JWJ00] (pages 38–41) – unlike the entities they eventually lead to which, in general, cannot be determined statically (e.g., a run-time path such as 'foo/1' leads to an entity which must be looked up dynamically). We make no attempt to in-line gbc-level `INNER` statements or `CALL` statements. Section 2.2 discusses our reasons for making this decision.

3.1.2 Replacing Time-inefficient Data Structures

In our previous development effort we made the decision to opt for flexible data structures that were easy to identify and understand and hence to modify or replace [JWJ00] (page 4). Because use of these data structures is no longer subject to frequent changes, it is sensible to replace them with less flexible, but more efficient versions in the source code generated by gVM (e.g., we replace general lists traversed using iterators with arrays traversed using loops and array accesses[1]).

Another argument for trading off readability for performance in the generated source code is the fact that it is not intended for perusal by humans. It is only intended to be machine-readable.

3.1.3 Compiling to Java Source Code

Unlike some previous efforts such as e.g. the SOAR project [SUH86] or the SELF system [USCH92], in gVM the object language for compiled bytecodes is not low-level, native machine code, but rather high-level, platform-independent source code; namely Java. In this respect, our approach bears more of a resemblance to that taken in ISE Eiffel [Eif01] which uses C as the object language for its intermediate bytecodes. Similarities to any one of several translators from other languages to Java (e.g., Smalltalk [Boy01]) or even Java bytecodes (see [Tol01]) are also apparent.

However, compilation in gVM is not restricted to bytecodes. In fact, compilation in gVM produces Java source code that is self-contained and customized to running just one gbc-format program. In effect, compilation removes the layer of interpretation embodied in our previous virtual machine implementation, thereby reducing overhead.

Our approach has the following distinct advantages over the approaches taken in [SUH86] and [USCH92]:

[1] Though the effect of any one such replacement is confined to relatively few lines of generated Java source code (i.e., a peep-hole), the Pareto principle [Cal01] (also known as the 80/20 rule) and the fact that replacements occur for *all* bytecodes with a run-time path as argument ensures that the performance gain achieved is potentially substantial (see Chapter 5 for performance results).

- optimizations can occur at a higher level,
- optimizations are platform-independent,
- the translated program is platform-independent
- the generated code is guaranteed to be type-safe, and
- memory reclamation in the translated program is automatic.

The fact that we do optimizations at a level higher than the bits-and-bytes level offered by conventional CPU instruction sets means that we are fundamentally better off to do top-level, global optimizations.

Our optimizations and the translated program are platform-independent in that we compile to *Java* source code, adhering to Sun's "Pure Java" rules available at [PJC01][2]. If we had chosen e.g. (ANSI) C as our object language instead, "a little care" [KR88] would be required to generate portable code. In actuality, a considerable effort may be have to expended to generate portable C (see [Col01] for a partial list of rules to follow). Also, since our virtual machine for gbeta is written in Java already, much of the source code that it must now emit exists already and can be reused.

The type-safety guarantees offered by the Java compiler reduce development time and effort. Had we chosen native code or a weakly-typed language such as C as our object language, more time and effort would have to be devoted to debugging.

Because code generated by gVM is eventually to be executed on an underlying Java virtual machine, memory reclamation (i.e., garbage collection) is automatic. As a consequence, no explicit code to reclaim objects eligible for collection needs to be emitted by our virtual machine.

Of course, our approach also has disadvantages over the approaches taken in [SUH86] and [USCH92]:

- the generated code must be compiled before it can be executed,
- certain optimizations are impossible, and
- Java programs are potentially slower than native programs.

The extra compilation cycle potentially increases pause times between invocation of gVM and start of execution of the gbc-format program. However, the generated code needs only be compiled once. Subsequent invocations of gVM with the *same* gbc-format program (as determined by modification times) will execute without a compilation cycle. In addition, since compilation is not dynamic, a user will not experience any distracting pauses due to compilation

[2]These rules forbid use of things such as hard coded file names, line separators, and user-defined native methods. The rules are summarized in [Fla99].

during execution of a program. As a result, we do not need to consider issues such as adaptive recompilation, like in [USCH92]. Also, since we intend the compiled version of the code generated for a gbc-format program to be used primarily in a final distribution, the time penalty incurred by the extra compilation cycle becomes moot.

Since the generated code is compiled from within gVM and the translated program executed either from within gVM or manually by the user invoking the Java virtual machine on it directly, using the compilation feature does not *require* any extra action by the user. This is contrary to most efforts aiming to extend the Java language through preprocessing, e.g., with features such as operator overloading and conditional compilation[3]. These preprocessors typically require the user to invoke the preprocessor, then the Java compiler, and finally the Java VM to execute a program.

Because Java, as a strongly-typed language, does not support the notion of variables whose allocated storage areas contain values of different types or sizes at different times (i.e., there is no "union"-like structure [KR88] in Java), we cannot optimize accesses to instances of "basic" patterns [Ern01b] by representing their values as immediate descriptors [Shi01] and "box" only on the rare occasion that a reference is actually required. Indeed, some (again, [Shi01]) suggest that the presence of a boxing/unboxing capability in an implementation of a dynamically-typed language (e.g., Smalltalk, SELF, Dylan, or Scheme) is imperative in order to achieve acceptable performance. However, the fact that gbeta is not dynamically typed and development is said to be a factor of two to four times faster [Rij01] in Java compared to C/C++ (our main alternative(s) to Java) makes a strong case for Java.

Though Java programs are generally considered slower than native programs[4], some studies, e.g., [Rij01], suggest that the performance gap between Java programs and optimized C programs is non-existent, at least for some types of applications.

The generated Java source code is our foundation for building higher-performance gbc-format programs. The extensions described in the following three subsections all require and build on this foundation. Some of these extensions (in particular, replacing time-inefficient data structures) could (also) have been used to improve the performance of our existing interpreter-based implementation of gVM. However, except for adding support for exploiting information about statically determined patterns (see the following section), we decided against this option because of the inherent performance limitations of purely interpreter-based implementations of virtual machines (e.g., see [Har01]).

[3] A partial list of these Java preprocessors is available at [Tol01]. This page also contains links to several other languages running on the Java VM.

[4] This common conception may be rooted in the fact that the Java VM implementation shipped with Java version 1.0 was purely interpreter-based.

3.2 Exploiting Information about Statically Determined Patterns

In the current implementation of gVM, no longer are all patterns created equal[5], i.e., in increments, by means of the `ADD-mainpart ... origin ...` and `MERGE-ptn` instructions.

In the previous implementation of gVM, all patterns had to be created incrementally, by adding single mixins with `ADD-mainpart ... origin ...` and merging existing patterns with `MERGE-ptn` because no information to identify statically determined patterns was available in the gbc-format. The only exceptions to this rule were certain "predefined" or "basic" patterns [Ern01b]. However, this information is now available, through the work of Erik Ernst who designed the gbeta language and implemented a compiler for it, and this means that most of the expensive (see Chapter 5) merge operations can now be avoided.

Also, information about statically determined patterns allows us to cache them. Our particular pattern caching scheme ensures that a static pattern is (almost always) created only once, as explained in Section 4.2.4.

[5] Pun intended, see [OWB96].

Chapter 4

Implementing the Extensions to gVM

or How Things Really Work

This chapter discusses the implementation of the extensions we made to gVM. It contains two major parts. Whereas the first of these elaborates on more traditional optimizations implemented in gVM, the second part goes into details about its use of information about statically determined patterns.

4.1 From gbeta Bytecodes to Java Source Code

In this section we discuss how gbeta bytecodes are compiled to Java source code. Focus is on an understanding of the compilation process. In Section 4.1.1 we go into detail about the compilation process, but we start with a discription of how the compiler transalates the gbeta bytecodes into Java source code:

Compiler Overview

Our compiler translates a gbc-format file into several Java-format files (one for each main part, see below) and then runs an appropriate Java compiler to compile these files into class files. The following steps are performed:

1. the input gbc-format program is parsed, creating a list of main parts.
2. each main part in this list is compiled. This generates a Java source code file for each main part.
3. a Java program file containing Java source code to initialize the run-time system and execute the first main part is generated.
4. each generated Java source code file is compiled into one or more[1] class

[1] A Java source code file is compiled into one class file for each class defined in the source code file.

files by calling a Java compiler. The code emitted by the compiler can now be executed by executing the compiled version of the source code file generated in step 3.

In the compilation process of the main parts making up the gbc-format program and when generating the Java program to initialize the run-time system we make use of in-lining:

In-lining
In-lining is a technique used to speed up the execution of programs by expanding non-recursive method calls. In-lining is used in, e.g., the SELF system described in [US91] and in the Java HotSpot VM [hot01].

We in-line compiled gbeta code as follows:

- instructions and attributes are in-lined in main parts
- run-time paths are in-lined in bytecodes, i.e., the code of run-time steps to be traversed is in-lined in a main part for each bytecode with a run-time path as argument

In-lining in gVM removes additional overhead by eliminating list accesses to instances of classes representing instructions and run-time steps, respectively. In particular list accesses to traverse the steps of run-time paths were frequent in our previous gbeta virtual machine.

Other compilers are integrated in run-time systems that detect methods which are candidates for in-lining (e.g., the Java HotSpot VM [hot01]). A method must be called frequently and fulfil certain other requirements[2] to be in-lined.

In contrast, the gVM run-time system does not provide any feed-back to the compiler. A program is compiled once and no run-time optimizations are attempted.Because we do not know, and do not try to predict, which methods will be called frequently, we in-line everything that passes through the compiler that can be in-lined.

4.1.1 Compiling to Java Source Code

Main parts are compiled into separate classes. A compiled main part class may contain a number of attributes. Each attribute is implemented as an inner class in the main part class and has a method to initialize itself, i.e., to execute the bytecodes associated with the attribute. These bytecodes are used to install a statically or non-statically determined object, object reference, pattern, or pattern reference.

A compiled main part also contains the bytecodes making up its do-part. These can be bytecodes to, e.g., push a statically determined pattern or a dynamic

[2] In Java, methods that are declared `final`, `private`, or `static` are candidates for in-lining. In some cases, however, they must also not have any local variables [Eck98].

pattern onto the expression stack or to deal with entities regardless of whether or not they are statically determined.

The code to initialize the run-time environment must also be generated. This run-time environment does not differ from that of the previous virtual machine. In essence, code to create the predefined part object is generated, and so is code to execute the first user-defined part object (see pages 23–27 in [JWJ00] for an elaboration on how the run-time environment is created). This code is written in a separate file. To run the compiled program we execute the class file generated from this file.

Main Parts

A main part is executed by executing the compiled bytecodes making up its do-part. Calls to these bytecodes can be in-lined because the bytecodes are known statically and the current part object stays the same during their execution.

Attributes

In our previous virtual machine, initialization of an attribute occurs by alternatingly fetching and executing the bytecodes from a list in its containing main part.

This approach can be improved when compiling to Java source code. We have chosen to represent each attribute of a main part as an inner class in the main part. Each inner class is a subclass of the class `Attribute` [JWJ00]. This is possible because the number of attributes and the bytecodes of which they consist are known statically.

Figure 4.1 shows the attribute `ir`[3]. Compiling the `ir` attribute outputs the Java source code shown in Figure 4.2. Notice how the `PUSH-ptn_"object"` bytecode has been translated into Java code with the same semantics.

Because attributes can be initialized out-of-order in context of arbitrary part objects, they each have an execute method, accepting the current part object as a parameter, among others.

```
1 "ir/0": (
2     PUSH-ptn_"object"
3     ...
4 )
```

Figure 4.1: gbeta bytecodes to initialize the attribute `ir`

[3] We only show one of its constituent bytecodes.

```
1 private class Attribute_0 extends Attribute {
2   public void execute(VM vm, PartObject context,
3                       int frameLevel) {
4     vm.allPurposeStack.pushPattern(new Pattern());
5     ...
6   }
7 }
```

Figure 4.2: Attribute `ir` compiled

Virtual Attributes

As with ordinary attributes, virtual attributes are compiled to Java source code, and are represented as inner classes inside the main part they belong to.

Virtual attributes have been described in detail in [JWJ00].

Instructions

Compiling a bytecode involves executing a method which outputs Java source code for this bytecode. This method is called `compileExecute`. As an example, the `compileExecute` method for the `PUSH-ptn` bytecode is shown in Figure 4.4. The steps required to generate customized code for the instance of this bytecode shown in Figure 4.3 are:

1. generate customized code to traverse the run-time path `{<-1,"integer/3"}`
2. generate the rest of the code for this bytecode

The first step is carried out in line 6 in the call to the `compileTraverse` method. Line 8 carries out step two and may reuse the name `aPattern` of the pattern variable as elaborated on in Section 4.1.1.

Run-time Paths

In the previous virtual machine (see [JWJ00]), a run-time path was implemented using a class whose instances represented particular run time paths. Also, a separate class for each type of run-time step was used. During compilation of a run-time path into Java source code, we simply in-line each of the calls to traverse a step of this particular run-time path, thereby generating code customized to traverse this run-time path.

As an example, the run-time path of a `PUSH-ptn` bytecode is shown in Figure 4.3. The run-time path consists of an out-step and a direct-lookup-step (see page 37 in [JWJ00] for an elaboration). The result of compiling a call to a traverse

method of the run-time path instance corresponding to `{<-1,"integer/3"}` is shown in Figure 4.5. It contains the code to traverse the non-final step `<-1` and the final step `"integer/3"`, respectively (we refer to Table 5.1 on page 37 in [JWJ00] for an explanation of the difference between non-final and final steps).

The generated code is customized and reuses variables. This is elaborated on in the next section.

```
1 PUSH-ptn {<-1,"integer/3"}
```

Figure 4.3: A bytecode with a run-time path as argument

```
1  public String compileExecute(String baseIndent, String
2  indent, int level) throws Exception {
3
4 ...
5
6  runtimePath.compileTraverse(baseIndent, indent, level)
7  ...
8  "vm.allPurposeStack.pushPattern(" + aPattern + ");"
9  ...
10 }
```

Figure 4.4: Java method to compile the PUSH-ptn bytecode

```
1  PartObject partObject_0_2 = context;
2  // compileTraverse in class OutRuntimeStep
3  for (int int_0_3 = 0; int_0_3 < 1; int_0_3++) {
4    partObject_0_2 = partObject_0_2.mixin.origin;
5  }
6  // compileTraverseLast in class LookupDirectRuntimeStep
7
8   Here code to traverse a final lookup direct step
9   is generated
10
```

Figure 4.5: A compiled run-time path

Variable System

Compiling main parts, attributes, instructions, and run-time paths leads to usage of a lot of temporary variables. A method to reuse the variables is needed.

The issue of reusing as many variables as possible when compiling bytecodes can be compared to the issue of assigning variables to machine registers [App98] in that we want to reuse variables which are no longer needed, just as machine registers are assigned to new variables when their current ones are no longer "alive".

A naive solution to the problem is to not reuse any temporary variables at all. This solution is very simple, but it wastes memory by allocating space for new variables when existing ones could be reused.

The scheme we employ considers the (lexical) scoping rules of Java. In the generated Java code, an arbitrary number of nested scopes may exist.

Variables declared in an outer scope are visible in subsequent inner scopes, but not the other way around. The Java scoping rules dictate this. To use these scoping rules correctly, we have developed a system in which variables are "checked out" and "checked in".

A variable is "checked out" the first time it is needed in the `compileExecute` method of a bytecode and "checked in" when it is no longer needed in that method. A check out specifies both the type of a variable (e.g., `int`), and the Java scoping level at which it will be used.

If a check out of a particular variable type at a particular Java scope level is attempted and no variable of that type is declared at a scope level less-then or equal to the one specified, then a new variable is generated, added to the pool of declared, checked out variables, and used. Otherwise a variable is moved from a pool of declared, checked in variables to the pool of declared, checked out variables and used.

A check in of a variable simply returns it to the pool of declared checked in variables. It is the responsibility of each bytecode to ensure that variables potentially declared in a scope introduced in that bytecode (e.g., a `SimpleIf`) are purged from the pool of declared, checked in variables when that scope level ends.

4.2 Static Patterns

Though gbeta allows for the type-safe creation of patterns that are not statically determined (as opposed to BETA, [Ern99a]), their use in real programs is relatively infrequent compared to the use of patterns that *are* statically determined (see the programs included in the gbeta distribution for examples). Consequently, it makes sense to optimize for the case in which one or more of the patterns created in a gbeta program are statically determined. Section 3.2 elaborates on our effort to speed up creation of such statically determined patterns. The intricacies associated with creating *instances* of statically determined patterns efficiently was discussed in Section 2.3.

4.2.1 Static Patterns versus Dynamic Patterns

In a more conventional, strongly-typed object-oriented languages such as Java, classes are monolithic entities that are always determined in their entireties at compile-time and loaded at run-time[4].

In contrast, patterns in gbeta are lists of mixins determined *either* at compile-time *or* at run-time. However, patterns are associated with one or more enclosing objects [Ern01b], meaning that they exist only at run-time, just like the `Class` objects of Java.

As noted above, gbeta allows for the type-safe creation of two kinds of patterns: compile-time patterns and run-time patterns. Consider the minimalistic gbeta program shown in Figure 4.6.

```
1 -- betaenv:descriptor --
2 (# p: (# #);
3    q: integer&p(:object:)
4 #)
```

Figure 4.6: A statically determined pattern and a non-statically determined pattern in gbeta

This program declares two pattern attributes, `p` and `q`. The `p` attribute is known statically to be the `(# #)` pattern whereas `q` is known statically only to be the *pattern combination* [Ern01a] of the `integer` basic pattern and the `p` pattern type cast to the `object` pattern, i.e., the pattern whose list of mixins is the empty list.

Figure 4.7 shows the result of compiling the code shown in Figure 4.6 to the gbc-format using the gbeta compiler in version 0.81.13 and with all optimizations turned on.

The creation of dynamic patterns such as `q` in Figure 4.7 in gVM was elaborated on in [JWJ00]. The following section discusses creating static patterns in gVM, such as `p` in Figure 4.7.

4.2.2 Creating Static Patterns

Because attribute `p` in Figure 4.6 is known statically, its initialization code contains a single bytecode which is a member of the `INSTALL-static` family. These bytecodes include `INSTALL-static-ptn`, `INSTALL-static-obj`, `INSTALL-qua-static-ptn`, and `INSTALL-qua-static-obj`.

[4] In Java, access to representations of loaded classes (or `Class` objects) are what give programs the capability of run-time introspection [Fla99] (or self-reflection); a Java program can look at any class and determine its superclass, what methods it defines, and so on.

```
1  MainPart("‘28-70"
2    "p/0": (
3      INSTALL-static-ptn 0 1
4    )
5    "q/1": (
6      PUSH-ptn_"integer"
7      PUSH-ptn "p/0"
8      MERGE-ptn
9      INSTALL-ptn 1
10   )
11 |
12 )
13
14 MainPart("‘34-38"
15 |
16 )
17
18 Pattern(1
19   composite static slice = ‘34-38 with origin at
20 )
```

Figure 4.7: Statically determined and non-statically determined patterns in the gbc-format

The `INSTALL-static` bytecodes, like the `INSTALL` bytecodes described in [JWJ00] page 30–31, assign entities (i.e., patterns, objects, and references to patterns and objects, respectively) to attribute arrays in part objects.

However, whereas the operand pattern of an `INSTALL` bytecode results from the execution of any number of bytecodes occurring before the `INSTALL` bytecode in the initialization code for an attribute (in particular, `ADD-mainpart ... origin ...` and `MERGE-ptn` bytecodes), the operand pattern of an `INSTALL-static` bytecode is created entirely from information available in a `Pattern` entry in the same gbc-format file as the `INSTALL-static` bytecode itself. The same holds true for the operand patterns of the rest of the `static` bytecodes generated by version 0.81.13 of the gbeta compiler with all optimizations turned on, namely `NEW,_static-ptn->obj`, `NEW,_static-ptn->tmp`, and `PUSH-static-ptn`. Once created, the operand patterns of the `static` bytecodes are cached, as elaborated on in Section 4.2.4.

The `NEW,_static-ptn->obj` bytecode instantiates its operand pattern and pushes the resulting instance onto the expression stack. The `NEW,_static-ptn->tmp` bytecode instantiates its operand pattern and pushes the resulting instance onto the stack of temporaries. The `PUSH-static-ptn` pushes its operand pattern onto the expression stack. In comparison to their dynamic counterparts, each of the

`static` bytecodes has an additional argument that maps it to a `Pattern` entry. For example, in Figure 4.7, the `INSTALL-static-ptn 0 1` bytecode maps to the `Pattern` entry with index `1`. There is a many-to-one relation between `static` bytecodes and `Pattern` entries, meaning that no `Pattern` entry is redundant.

Figure 4.8 shows another example of a `Pattern` entry.

```
1  Pattern(3
2    composite static slice = '32-36 with origin at {}
3    integer static slice with origin at {<-2}
4  )
```

Figure 4.8: A pattern entry in a gbc-format file

Like the index of the `Pattern` entry (`1`) in line 18 of Figure 4.7, the index of the `Pattern` entry (`3`) in line 1 of Figure 4.8 associates the `Pattern` entry with one or more of the `static` bytecodes. The remainder of the `Pattern` entry in Figure 4.8 describes the mixins of a statically known pattern in order from most-general mixin to most-specific mixin. This pattern has two mixins.

Line 2 indicates a mixin whose main part has id `'32-36` and whose origin is in the part object obtained by traversing the run-time path `{}`. A mixin designated `composite` is *user-defined*. Line 3 indicates a mixin whose origin is in the part object obtained by traversing the run-time path {<-2}. A mixin designated `integer` is *predefined* and of the `integer` variety. Several varieties of predefined mixins are possible in a `Pattern` entry: `integer`, `real`, `char`, and others. A predefined mixin is not associated with a main part in the input gbc-format program and has always origin in the *predefined part object* [JWJ00]. Consequently, the run-time path information of a predefined mixin can be (and is) ignored.

As noted above, a `Pattern` entry describes the structure of a statically known pattern. In effect, its main purpose is the same as the *prototype data structure* [BS01] in the Mjølner implementation of the BETA system[5].

In the Mjølner implementation, exactly one prototype data structure is created for each object descriptor in the source code and each object is directly associated with its prototype data structure. A prototype data structure describes the structure of a pattern (and hence the layout of its instances) for the purpose of garbage collection and contains a v-table used in the allocation of virtuals.

In contrast to its Mjølner implementation counterpart, a `Pattern` entry (or rather, a run-time representation of it) is not directly associated with any objects, it describes the structure of a pattern for the purpose of creating a run-time representation of it, and contains no v-table. Because not all patterns in gbeta are known statically, allocating virtuals via static v-tables will not work, in gen-

[5]The *prototype* term as used in this section is unrelated to the same term as used in Section 2.3 where we discussed the prototype paradigm for object creation.

eral. Also, since garbage collection in gVM is handled by an underlying Java virtual machine, it is not an incentive for run-time representations of `Pattern` entries to describe the structures of patterns. As a consequence of these two factors (no use for v-tables and no need for garbage collection in gVM), run-time representations of `Pattern` entries are not associated with objects.

4.2.3 Anonymous Patterns

An anonymous pattern is a pattern with no name. It is used to specialize the behaviour of an instance of a pattern. Figure 4.9 shows how object behaviour is specialized with an anonymous pattern. In line 5, the execution of the `p` pattern coerced into an object outputs:

```
This is normal behaviour.
```

In line 6, however, the execution of the `p` pattern coerced into an object and specialized with the anonymous pattern

```
(# do 'This is specialized behaviour.\n' -> stdio #)
```

outputs:

```
This is normal behaviour. This is specialized behaviour.
```

```
1 -- betaenv:descriptor --
2 (#
3   p: (# do 'This is normal behaviour.\n' -> stdio; INNER #)
4 do
5 p;
6   p(# do 'This is specialized behaviour.\n' -> stdio #)
7 #)
```

Figure 4.9: Using anonymous patterns to specialize object behaviour

The anonymous patterns of gbeta (and hence BETA) are similar to the anonymous inner classes of Java. An anonymous inner class in Java combines the syntax for class definition with the syntax for class instantiation as does an anonymous pattern in gbeta when used in an object context; e.g., as an imperative[6]. In effect, if an object is used only once, using anonymous syntax will help improve readability since definition of the class (in Java) or pattern (in gbeta and BETA) and use of the object occur in exactly the same place.

Anonymous patterns are determined statically. Whereas the gbeta compiler used in [JWJ00] and in version 0.81.13 without any optimizations turned on generates a pair of `PUSH-ptn` and `ADD-mainpart ... origin ...` bytecodes for each use of an anonymous pattern, in version 0.81.13 with all optimizations

[6] An imperative in BETA or gbeta is an object context, meaning that whatever category of entity occurs in an imperative (or "statement") is implicitly coerced into an object [Ern99a].

turned on it generates a single `PUSH-static-ptn` bytecode instead. Consequently, anonymous patterns can be created faster than dynamically determined patterns (see Section 5 for details) and then cached, as elaborated on in the following section.

4.2.4 Caching Static Patterns

In gVM, every static pattern created is cached. Our pattern caching scheme ensures that only in the unlikely event that the *same* pattern is declared multiple times will it be created more than once.

A static pattern is cached in one of two ways.

If it is created as the result of the execution of one of the `INSTALL-static` bytecodes (as explained in Section 3.2), it is cached in the attributes array [JWJ00] in the part object whose attribute it is initializing. Static patterns cached in this way are indistinguishable from dynamic patterns in terms of how they are cached and how they are accessed. Both kinds of patterns are looked up by traversing a run-time path. The `INSTALL-static` bytecodes occur only in attribute initialization code.

On the other hand, if a static pattern is created as the result of the execution of a `NEW,_static-ptn->obj` bytecode, a `NEW,_static-ptn->tmp` bytecode, or a `PUSH-static-ptn` bytecode, it is cached in an array in the current part object dedicated to caching static patterns. Every part object whose mixin is associated with a main part whose do-part contains one or more occurrences of these three bytecodes has such an array and each such occurrence is statically assigned a unique position in the array for caching of its operand static pattern. Static patterns cached in this way are accessed without traversing a run-time path because the appropriate pattern cache is always located in the current part object. Hence, looking up a static pattern in a dedicated static pattern cache requires accessing an array at a statically known index only.

For best performance, lookup by accessing an array at a statically known index is preferable to lookup by traversing a run-time path. However, a static pattern created as the result of the execution of one of the `INSTALL-static` bytecodes cannot be *guaranteed* to be looked up only from within its own part object without inspecting every step of every run-time path in the entire input program. Although such an inspection is feasible and would also identify those attributes that *require* initialization (i.e., those attributes referenced in lookup steps), we consider this an area of future work. Consequently, a static pattern created as the result of the execution of one of the `INSTALL-static` bytecodes must be looked up by traversing a run-time path.

The first execution of a `NEW,_static-ptn->obj` bytecode, a `NEW,_static-ptn->tmp` bytecode, or a `PUSH-static-ptn` bytecode always creates the operand pattern and caches it whereas subsequent executions always fetch it from the cache.

In effect, the pattern caching scheme in gVM does not dynamically adapt to

the actual usage patterns (no pun intended) of patterns – like the optimizing compiler in the SELF system [USCH92] recompiles (and optimizes) code or the Java HotSpot VM [hot01] in-lines method calls based on the actual flow of execution. In one sense, our caching scheme is wasteful of memory. In another, however, its simplicity keeps the run-time system size to a minimum, one of the traditional virtues of compiled languages like C.

Chapter 5

Performance Results
or Why gVM is the Execution Engine of Choice

In this chapter we document that gVM in compilation mode is indeed faster than gVM in interpretation mode and that the generality of gbeta comes at a cost. We present results that show that the goal of creating a virtual machine which is faster than the one employed by Erik Ernst [Ern99a] has been achieved. These results provide empirical evidence for the conclusions made in Section 2. Results are presented in Sections 5.2 and 5.3.

We start with elaborating on our benchmark programs and the methodology used to provide the results presented in Sections 5.2 and 5.3, respectively.

5.1 Methodology

The essential ingredient in our methodology is executing a suite of very simplistic benchmark programs using gbeta, gVM, and BETA binaries, respectively.

The BETA binaries are included in order to establish a context for our benchmark results. They are the results of compiling the benchmark programs with the Mjølner BETA system compiler in version 5.2.1.

Each benchmark program evaluates the same expressions and executes the same statements across all execution engines to ensure that it provides comparable performance results. The benchmark programs are presented in Appendix A. The gbeta and BETA programs are identical except for the fact that the latter have the header

instead of only `-betaenv:descriptor-`.

Each benchmark program in the suite is designed to exercise either a high-level BETA (and hence gbeta) language construct or a low level-level implementation of a run-time path traversal (or equivalent for BETA). This enables us to

```
ORIGIN '~beta/basiclib/betaenv'
-- program:descriptor --
```

Figure 5.1: header

compare the performance of the different execution engines.

The language constructs exercised in the benchmark programs are:

- assignment of an object coerced into an object reference to an object reference (`objectReference`)
- assignment to an object (`assignment`)
- addition of two integers (`arithmetic`)
- assignment of a pattern coerced into a pattern reference to a pattern reference (`patternReference`)

The low-level implementation of run-time path traversals exercised in the benchmark programs are:

- traversal of a run-time path with one direct-lookup-step (`runtimePath1`)
- traversal of a run-time path with one indirect-lookup-step (`runtimePath2`)
- traversal of a run-time path with one up-step and one direct-lookup-step (`runtimePath3`)
- traversal of a run-time path with one out-step and one direct-lookup-step (`runtimePath4`)

We have chosen these particular benchmark programs because they emphasize both high-level and low-level aspects. Also, we believe that they will reveal the main differences between gbeta and BETA and thus support the conclusions in Section 2.

gVM in interpretation mode and gVM in compilation mode both depend on the output from the gbeta compiler. Consequently, if gbeta scores particularly low in a benchmark, we expect gVM in interpretation mode and gVM in compilation mode to do so also. This means that we need only compare gbeta and BETA. The results of these comparisons can be extrapolated to gVM in interpretation mode versus BETA and gVM in compilation mode versus BETA, respectively.

It may seem odd to include both a benchmark that assigns one (pattern coerced into a) pattern reference to another pattern reference and a benchmark that assigns one (object coerced into an) object reference to another object reference in our benchmark suite. One could argue that there is no need for both of them

because pattern reference assignment and object reference assignment is very similar in gbeta. In gbeta, assignment to a pattern reference involves looking up an attribute (i.e., checking its qualification) and storing a value. In BETA, however, assignment to a pattern reference involves looking up an attribute, creating a "struct" object [KLM93], and then storing a value.

To get accurate average execution times, each benchmark program is executed a number of times and the total time spent on execution is divided by the number of executions. The number of executions of a particular benchmark program is determined by the differences in execution times. If the execution times of a benchmark program do not differ by more than 15 per cent when the benchmark program is executed 1, 2, 3, 4, 5, and 6 times, respectively, we consider the results accurate enough to be valid.

The limit at 15 per cent distinctly orders the execution engines in terms of performance when executing each of the benchmark programs. It is determined by calculating the differences between the values in each pair of cells in a row for all rows of Table 5.1 and remembering the minimum difference for each row. In order to get a distinct ordering of the execution engines in terms of performance, it is only necessary to remember the minimum difference for each row. However, we use the limit at 15 per cent as a global value. This value is less than the difference between the values in the cells (`arithmetic`, `gVMi`) and (`arithmetic`, `gVMc`) in Table 5.1 and this difference is the global, minimum difference.

We document that average execution times do not differ by more than 15 per cent by presenting histograms of the differences in Appendix B.

Each benchmark program is executed using each of the four different execution engines: BETA, gVM in interpretation mode, gVM in compilation mode, and gbeta. To compare only times spent on executing the benchmark programs, times spent on parsing (for both gVM in compilation mode and gVM in interpretation mode) and parsing and static analysis (for gbeta) are not included in execution times.

5.2 gVM and BETA Performance Results

In Table 5.1 we show relative execution times for each of the benchmark programs executed 6 times on a lightly-loaded Solaris 8 machine with two 450 MHz processors and 2 GB of memory.

For each benchmark the average execution time for BETA is set to 1 and the average execution times for the rest of the execution engines are relative to this value.

Table 5.1 shows the distinct ordering of the execution engines in terms of performance. BETA is cheapest, gVM in compilation mode is second cheapest, gVM in interpretation mode is third cheapest, and gbeta is most expensive. To get the results in seconds, we refer to Table C.1 in Appendix C.

program	BETA	gVMc	gVMi	gbeta
objectReference	1	17,8	54,2	724,2
assignment	1	22,25	61	683,5
arithmetic	1	90,25	113	1358,5
patternReference	1	17,22	38,56	401
runtimePath1	1	21,33	71,67	1003,67
runtimePath2	1	23	71	985,33
runtimePath3	1	31,33	79,33	1296,67
runtimePath4	1	35,67	79,33	1559,33

Table 5.1: Relative average execution times for each of the benchmark programs

In Table 5.1, notice the gbeta (relative) execution times for the `runtimePath1` and `runtimePath3` benchmark programs. These values clearly show that traversals of longer run-time paths (with an up-step) are more expensive than traversals of shorter ones (without an up-step) in gbeta, and thus that traversal of a single run-time up-step is more expensive in gbeta than in BETA.

The reason is that traversals of an up-step (or a down-step) and a direct-lookup-step (or an indirect-lookup-step) are combined in BETA and involve only an inexpensive access at a statically known offset within an object (because part objects are in-lined in their containing object). In gbeta, however, traversals of an up-step and a direct-lookup-step require following a reference from a more specific part object to a more general one and an access at a statically known offset within the more general part object.

Adding integers in gbeta is more expensive than in BETA. Table 5.1 shows this. Every integer value in BETA is represented as an immediate descriptor [Shi01] and boxed only when a reference is required. In contrast, integers in gbeta are genuine objects. As a result, accessing the value of an integer in gbeta requires an indirection (or dereference).

Table 5.1 shows that it is more expensive to traverse an out-step than it is to traverse an up-step in gbeta. It also shows that traversing an indirect-lookup-step is less expensive than traversing a direct-lookup-step in gbeta. These results are unexpected and we attribute them to inaccuracies of measurements. BETA and gbeta use a similar global lookup strategy to perform out-steps. Also, BETA performs a lookup-direct-step by accessing the *self* (or *this*) object at a statically known offset whereas gbeta does it by accessing the current part object at a statically known offset. For both execution engines, an indirect-lookup-step requires an additional indirection over a direct-lookup-step.

Another conclusion which can be drawn from Table 5.1 is that pattern reference assignments are cheaper than object reference assignments in gbeta. As explained above, a pattern reference assignment in BETA requires the creation of a struct object because there are no pattern attributes in part objects.

program	gbeta [s]	gbeta [d]	gVMc [s]	gVMc [d]
merge2	1	1.33	1	2,45
merge5	1	2.78	1	3.74

Table 5.2: Relative average execution times for `merge2` and `merge5` on gbeta and gVM in compilation mode, respectively

5.3 Performance Impact of Use of Information about Static Patterns

In Table 5.2 we show (relative) average execution times for gbeta and gVM in compilation mode, respectively.

Two benchmark programs have been executed on each execution engine. These benchmark programs are shown in Appendix A and do the following.

- merge 2 patterns (`merge2`)
- merge 5 patterns (`merge5`)

Each benchmark program is executed with and without using information about statically determined patterns and iterates pattern creation a large number of times. Each iteration creates a pattern using static information about it (in the `[s]` column of Table 5.2) or by merging existing patterns (in the `[d]` column of Table 5.2).

The purpose of these benchmarks is to determine the number of pattern merges that makes it cheaper to create a pattern using static information about it (if available).

We used Erik Ernst's integrated gbeta compiler and interpreter in version 0.81.13 with the arguments `-fp -fb -fi -fq -fl` and `-g` to generate the gbc-format file used for `gVMc [s]` and `-g` only to generate the gbc-format file used for `gVMc [d]`.

The `-fp -fb -fi -fq -fl` arguments instruct the compiler to make use of knowledge about statically determined patterns, and the `-g` argument tells it to generate gbc-format files.

For `gbeta [s]` we used the aforementioned integrated gbeta compiler and interpreter with the `-f*` and `-r` arguments. For `gbeta [d]` we used it with the `-r` argument only.

The `-f*` argument instructs the compiler and interpreter to use all possible optimizations, and the `-r` argument tells it to output execution time in clock ticks[1].

[1] We convert clock ticks to seconds by dividing by the number of clock ticks per second.

The `gbeta [s]` results presented in Table C.2 in Appendix C are set to 1 in Table 5.2 and are used to calculate the execution times for `gbeta [d]` relative to `gbeta [s]`.

Likewise, the results for `gVMc [s]` in Table C.2 are set to 1 in Table 5.2 and used to calculate the execution times for `gVMc [d]` relative to `gVMc [s]`.

We used the same method to get accurate (relative) average execution times in Table 5.2 as for Table 5.1. The limit is different because the global, minimum difference depends on the values in Table 5.2 and not Table 5.1.

Table 5.2 indicates that it is always more expensive to create patterns by merging existing ones than it is to create them from static information about them. However, the patterns merged in the benchmark programs are special because each has only a single mixin and it is always associated with the `(# #)` main part. Whether these results are valid in the case of merging more general patterns remains to be determined.

Chapter 6

Conclusion

The gbeta virtual machine (gVM) and its theoretical foundation were presented. gbeta is an extension of the BETA language with support for new features such as *dynamic specialization* and *propagating pattern combination*. gVM is a stand-alone Java implementation of a virtual machine for executing gbc-format programs generated by the gbeta compiler and interpreter [Ern99a].

gVM was created as a platform for

- experimenting with different schemes to improve performance over that of the existing gbeta interpreter [Ern99a] as well as over that of our own gbc-format program interpreter [JWJ00]
- running benchmarks to obtain performance results for comparisons with a (Mjølner) BETA implementation and the two existing interpreters in order to determine the performance penalty incurred by a gbeta execution engine for supporting the generality of gbeta

Early on, we realized that inherent differences between gbeta and BETA would prevent us from reaching BETA implementation performance in gVM. The most significant of these differences in terms of impact on performance is the general support for pattern combination in gbeta.

In order to narrow the gap to the performance of the BETA implementation, several ideas from the known literature and the BETA implementation were considered and adaptations to a gbeta context were attempted.

A scheme to initialize the attributes of objects in a safe order was considered, but ultimately rejected. Such a scheme would allow us to omit certain run-time checks. However, it would also restrict the gbeta programmer in the kinds of programs he or she could write because of the undecidable nature of the problem. A number of ideas dealt with statically determined patterns or instances of statically determined patterns. Representing attributes of part objects of

instances of statically determined patterns in origin part objects to avoid out-steps or in the most-specific part object to avoid up-steps and `INNER` calls were two. Another idea was creating instances of statically determined patterns by copying prototype objects in order to avoid executing attribute initialization code more than once. All three of these ideas were rejected on grounds of no apparent, generally applicable implementations, Java related implementation difficulties, and/or no performance benefits.

The extensive (and exclusive) use of statically determined patterns in the BETA implementation was an incentive to optimize for the case in which one or more patterns created in a gbeta program are statically determined. Information available in every gbc-format file allows us to create these patterns efficiently and a simple caching scheme provides fast access to them once created.

A couple of more traditional optimizations were also implemented in gVM. Compiling to Java source code and the class file format instead of executing an input gbc-format program directly eliminates a layer of interpretation while producing platform-independent code suitable for distribution. In-lining method calls trades of size for speed by physically replacing method calls with method bodies.

The generality of gbeta does not come free. The benchmarks show that certain language constructs in gbeta are relatively more expensive than others than in BETA. However, the language features exercised in most of our benchmarks are not inherently gbeta-only. In fact, we believe that continued development of the gbeta compiler will ensure that the cost of the generality of gbeta will be paid only by those who use it. One particular crude way to achieve this is to integrate a BETA compiler with the gbeta compiler and use the BETA part whenever an input program contains BETA syntax. A more sophisticated approach would use finer granularity than an entire program to determine which compiler to use, but would possibly require auxiliary syntax provided by the programmer as well.

Part II

Appendices

Appendix A

Benchmarks Programs

```
(* assignment of an object coerced into an object
reference to an object reference*)
-- betaenv:descriptor --

(#
   simplePattern: (# #);
   simpleObject: @simplePattern;
   reftoSimpleObject: ^simplePattern;

    do

     (for 750000 repeat
       simpleObject[] -> reftoSimpleObject[];
     for)

#)
```

Figure A.1: objectReference.gb

```
(* assignment to an object *)
--betaenv:descriptor--

(#
  i: @integer;
do

  (for 750000
    repeat
     3 -> i;
  for)
#)
```

Figure A.2: assignment.gb

```
(* addition of two integers *)
--betaenv:descriptor--

(#
  i: @integer;
 do

  1->i;
  (for 750000
    repeat
     (i+i) -> i;
  for)
#)
```

Figure A.3: arithmetic.gb

```
(* assignment of a pattern coerced into a pattern
reference to a pattern reference*)

-- betaenv:descriptor --

(#
  simplePattern: (# #);
  refToSimplePattern: ##simplePattern;

  do

   (for 750000 repeat
     simplePattern## -> refToSimplePattern##;
   for)
#)
```

Figure A.4: patternReference.gb

```
(* traversal of a run-time path with one
lookupDirectStep *)
--betaenv:descriptor--

(#
  anObject: @(# #)

do

  (for 750000
    repeat
    (* PUSH-obj,DISCARD *)
     anObject[];
  for)
#)
```

Figure A.5: runtimePath1.gb

```
(* traveral of a run-time path with one lookup
indirect step *)
--betaenv:descriptor--

(#
 anObject: @(# #);
 reftoanObject: ^object;
do

 anObject[] -> reftoanObject[];
 (for 750000
  repeat
  (* PUSH-obj,DISCARD *)
   reftoanObject[];
  for)

#)
```

Figure A.6: runtimePath2.gb

```
(* traversal of run-time path with one up step
and one lookupDirect step *)
--betaenv:descriptor--

(#
 a:(#
  anObject: @(# #);
 do
  INNER;
   #);

 b:@a(#
 do
  (for 750000
   repeat
    anObject[];
  for)

 #);

do
 b;
#)
```

Figure A.7: runtimePath3.gb

```
(* traversal of a run-time path with one out step
and one lookup direct step*)
--betaenv:descriptor--

(#
anObject: @(# #);
  a:@(#
  do
   (for 750000
    repeat
     anObject[];
   for)

     #);

do

   a;
#)
```

Figure A.8: runtimePath4.gb

```
(* Merging 2 patterns *)

--betaenv:descriptor--
(#
  a: (# #);
  b: a(# #)
do

  (for 750000
    repeat
      a & b;
  for)
#)
```

Figure A.9: merge2.gb

```
(* Merging 5 patterns *)

--betaenv:descriptor--
(#
  a: (# #);
  b: a(# #);
  c: b(# #);
  d: c(# #);
  e: d(# #)
do

  (for 750000
    repeat
      a & b & c & d & e;
  for)
#)
```

Figure A.10: merge5.gb

Appendix B

Histograms

In this appendix we document the accuracy of the average execution times of our benchmark programs. We do this by depicting histograms.

We executed our benchmark programs 1, 2, 3, 4, 5, and 6 times, respectively. The histograms have a block representing the average execution time for each of these times.

As an example, the histograms depicted in Figure B.1 show the differences in average execution times relative to 1 for the `assignment` benchmark program when executed from 1 to 6 times, and for each of the four execution engines. We chose to increase the number of iterations of the loop in the `assignment` benchmark program for the BETA binary to get comparable results for the differences in execution times. This alteration does not affect the correctness of the numbers in 5.1. All differences in execution times relative to BETA are still correct.

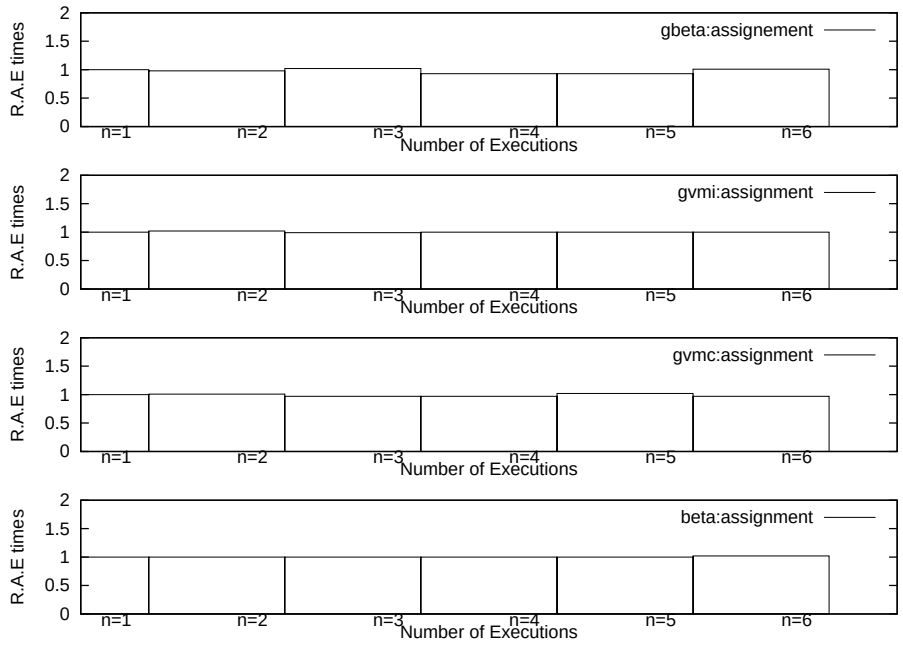

Figure B.1: Differences in average execution times for the `assignment` benchmark program

Figure B.1 shows that there is no variation in average execution time greater than 15 per cent for any of the execution engines. The same goes for any combination of benchmark program and execution engine so our conclusions about an ordering of the execution engines in terms of performance (in Section 5.1) is correct. For more histograms we refer to [JWJ01].

Appendix C

Performance Results

Benchmark Program	BETA	gVMc	gVMi	gbeta
objectReference	0,05	0,89	2,71	36,21
assignment	0,04	0,89	2,44	27,34
arithmetic	0,04	3,61	4,52	54,34
patternReference	0,09	1,55	3,47	36,09
runtimePath1	0,03	0,64	2,15	30,11
runtimePath2	0,03	0,69	2,13	29,56
runtimePath3	0,03	0,94	2,38	38,9
runtimePath4	0,03	0,66	2,27	46,77

Table C.1: Average executions times for each of the benchmark programs (in seconds)

Benchmark Program	gbeta [s]	gbeta [d]	gVMc [s]	gVMc [d]
merge2	63,53	84,53	4,57	11,20
merge5	98,57	273,55	8,63	32,24

Table C.2: Average execution times for each of the `merge2` and `merge5` benchmark programs (in seconds)

Bibliography

[App98] Andrew Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[BC90] Gilad Braha and William Cook. Mixin-based Inheritance. In *Proceedings ECOOP'90*, pages 303 - 311. ACM Press, 1990.

[Boy01] Nik Boyd. Toward Smalltalk and Java Language Integration. Internet URL `http://www.jps.net/nikboyd/papers/sttojava`, june 2001.

[BS01] Michael Larsen BETA Support. Email from BETA support. Internet URL `www.gbeta.dk/emailfromsupport.phtml`, june 2001.

[Cal01] John D. Callos. The Pareto Principle (a.k.a. The 80:20 Rule). Internet URL `http://www.4hb.com/08jcparetoprinciple.html`, june 2001.

[Col01] John Robert Collins. Portable C Programming. `http://users.erols.com/johnrobertcollins/portabc.html`, March 2001.

[Dal01] Jeff Dalton. A Brief Guide to CLOS. Internet URL `http://www.aiai.ed.ac.uk/~jeff/clos-guide.html`, March 2001.

[Eck98] Bruce Eckel. *Thinking in Java.* Prentice Hall PTR, first edition, 1998.

[Eif01] ISE Eiffel. Internet URL `http://www.eiffel.com`, june 2001.

[Ern99a] Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance.* PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[Ern99b] Erik Ernst. Propagating Class and Method Combination. In *Proceedings ECOOP'99*, LNCS 1628, pages 67 - 91, Lisboa, Portugal, June 1999. Springer-Verlag.

[Ern01a] Erik Ernst. Combination of Patterns Everywhere. Internet URL `http://www.daimi.au.dk/~eernst/gbeta/advanced_index7.html`, june 2001.

[Ern01b] Erik Ernst. Tutorial on gbeta. Internet URL `http://www.daimi.au.dk/~eernst/gbeta`, March 2001.

[Fla99] D. Flanagan. *Java in a Nutshell.* O'Reilly & Associates, Inc., third edition, 1999.

[Har01] Jonathan Hardwick. Java Microbenchmarks. Internet URL `http://www.cs.cmu.edu/~jch/java/benchmarks.html`, june 2001.

[hot01] The java hotspot virtual machine. Internet URL `http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Fi%nal_4_30_01.html`, March 2001.

[Joy01] Ian Joyner. The C++ Critique. Internet URL `http://www.elj.com/eiffel/ij/inheritance/mi`, March 2001.

[JWJ00] Ricki Jensen, Michael Wojciechowski, and Christian Jørgensen. A Virtual Machine Used to Execute Byte Codes. internal report, Department of Computer Science, Institute for Electronic Systems, Aalborg University, Fredrik Bajers vej 7a1 9220 Aalborg Øst, Denmark, 2000.

[JWJ01] Ricki Jensen, Michael Wojciechowski, and Christian Jørgensen. The gbeta Homepage. Internet URL `http://www.gbeta.dk`, March 2001.

[KLM93] Jørgen Lindskov Knudsen, Mats Løfgren, and Ole Lehrmann Madsen. *Object-Oriented Environments: The Mjølner Approach*, chapter 26, pages 389–408. Prentice-Hall, 1993.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[MPN93] Ole L. Madsen, Birger M. Pedersen, and Kirsten Nygaard. *Object-oriented Programming in the BETA Programming Language.* Addison – Wesley Publishing Company, 1 edition, 1993.

[OWB96] G. Orwell, C. M. Woodhouse, and R. Baker. *Animal Farm.* Mass Market Paperback, 1996.

[PJC01] 100% Pure Java. Internet URL `http://java.sun.com/100percent/100PercentPureJavaCookbook-4_1_1.pdf`, March 2001.

[Rij01] Chris Rijk. Binaries Vs Byte-Codes. Internet URL `http://www.aceshardware.com/Spades/read.php?article_id=153`, March 2001.

[Sch01] Rene W. Schmidt. MetaBETA. Internet URL `http://www.daimi.au.dk/PB/506/PB-506.pdf`, may 2001.

[Shi01] Olin Shivers. Supporting dynamic languages on the Java virtual machine. Internet URL `http://www.ai.mit.edu/people/shivers/javaScheme.html`, june 2001.

[SUH86] A. D. Samples, D. Ungar, and P. Hilfinger. SOAR: Smalltalk without Bytecodes. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 107–118, New York, NY, 1986. ACM Press.

[Tol01] Robert Tolksdorf. Programming Languages for the Java Virtual Machine. Internet URL `http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html`, March 2001.

[US91] D. Ungar and R. B. Smith. SELF: The Power of Simplicity. *Lisp and Symbolic Computation*, 4(3), June 1991.

[USCH92] D. Ungar, R. B. Smith, C. Chambers, and U. Hölzle. Object, Message, and Performance: How They Coexist in Self. *IEEE Computer*, 25(10):53–64, Oct 1992.