Reducing the Cost of Object Boxing

Tim Owen and Des Watson

Department of Informatics, University of Sussex, Brighton, UK

Abstract. Language implementations that use a uniform pointer representation for generic datatypes typically apply boxing operations to convert non-uniform objects into the required form. The cost of this boxing process impacts upon the performance of programs that make heavy use of genericity with non-uniform data such as integers and other primitive value types. We show that the overhead of boxing objects into heap storage can be significantly reduced by taking a lazy approach: allowing pointers to stack-allocated objects that are only copied to the heap when necessary. Delaying the boxing of objects avoids unnecessary heap allocation, and results in speedups of around 25% for a range of test programs.

1 Introduction

The implementation of languages supporting genericity typically use a uniform representation for generic data: a pointer to the actual object. This approach works well for objects that are of *reference type*, since they are already represented as a pointer to a heap allocated object. Objects of *value type*, however, are handled directly, can be allocated on the stack and passed by value. For example, integers and real numbers are value types in most languages. Since no pointer indirection is used, value type objects are not in the uniform representation required for use by generic code.

A common solution to this mismatch is for the compiler to insert a *boxing* coercion operation into the generated code that copies a value type object from the stack onto the heap. A pointer to this copy is then used by generic code. For example, when using a generic List to store integers, each integer will be boxed when passed to the code that implements the List data structure. A similar approach has also been applied in the C# language[1] to allow code expecting reference type data to be used for integers and other value types, by automatically boxing and unboxing objects where necessary.

Whilst the use of boxing solves the problem of achieving a uniform representation, it incurs an overhead in terms of extra memory used to store the boxed copy, and a performance reduction due to the task of copying objects from stack to heap. In addition, the extra use of heap storage increases the load on the garbage collector, which may cause further slowdowns. Therefore, it is desirable to either reduce the cost of boxing or even eliminate it altogether where possible.

In this paper, we introduce an optimisation called *lazy boxing* in which value type objects (including primitive type objects such as integers) are not immediately copied to the heap when passed to a generic context. Instead, generic

E. Duesterwald (Ed.): CC 2004, LNCS 2985, pp. 202–216, 2004.

code uses a pointer to the object stored on the stack. A boxing operation is only performed later if the stack object is about to be deallocated. We discuss the implementation details of this technique, and show experimental results that demonstrate its effectiveness.

2 Related Work

Languages with genericity have different approaches to the support of value types in a generic context. Earlier proposals for extending Java with genericity, such as GJ[2] and NextGen[3], do not allow Java's value types (e.g. primitive types such as `int`) to be used as type arguments — only reference types are permitted. For example, GJ allows `List<Integer>` but not `List<int>` which simplifies their compiler implementation, but effectively requires the programmer to manually insert boxing and unboxing code. In our work, we allow generic types to work with all types of data, including value types such as integers.

While Java has been extended with support for genericity, first through research projects such as Pizza[4] and GJ and now officially in the 1.5 release, the language still does not support user-defined value types. The C# language allows programmers to define `struct` objects that are handled by value, and specifies that automatic boxing takes place when these objects are handled by reference.

A proposal for extending the Microsoft Common Language Runtime (CLR) with genericity[5] enables integrated support for value classes and genericity, by using a virtual machine infrastructure that builds type-specific information and generates specialised method code at class load-time. This avoids the need for boxing, but at the expense of increased code size and a more complex run-time support system. While their implementation follows one set of design choices and trade-offs, our work examines a different point in the design space. We take a fully-compiled approach, supporting value types and genericity by using boxing, but without need for sophisticated run-time code generation or specialisation.

Templates in C++ provide genericity[6], and may be used with value or reference types. Like the generic CLR, this approach completely avoids the need for object boxing, but with the disadvantage that it involves compile-time (or possibly load-time) expansion of generic definitions, which can result in undesirable code duplication. Besides C++, Eiffel also provides genericity and value types, although implementation details are hard to come by. The design of the SmallEiffel compiler[7] suggests that separate code is produced for different parameterisations of a generic class, with a common instantiation for all pointer-based types. Again, this avoids the need for boxing at the expense of duplicated code.

The Pizza project (an early extension of Java with generics) permits use of value types with generic types, and demonstrates a choice of implementation: either the heterogeneous approach of generating separate code for each use of a generic type, or the homogenous technique that shares a single code but introduces boxing and unboxing operations as required.

In the functional language community, the problem of supporting different sized data in a generic context is known as unboxed polymorphism. The aim is to avoid using the uniform representation (i.e. a pointer to a heap-allocated object) as much as possible. Coercion-based solutions[8,9] apply boxing and unboxing operations at run-time, whenever value type objects are passed to and from a generic context. This is the conventional eager boxing strategy. One limitation of the straightforward coercion approach is that the boxing operation for recursive data structures such as linked lists and trees can involve large amounts of expensive and redundant boxing. Passing type information at run-time can alleviate this problem[10], and has been applied to the FLINT[11] and TIL[12] compilers for ML.

Compile-time techniques have been developed that can be used to eliminate redundant boxing by analysis of the program code. Most notably, escape analysis[13,14] determines the extent of references to an object. This information can then be used to safely stack-allocate some objects, when the analysis is able to determine that no references to the object will outlive the object itself. Our work is complementary to this, in that we are interested in handling the boxing of those objects for which escape analysis *cannot* prove is redundant. Since escape analysis is conservative, it will leave a boxing operation in place when references to an object may escape. We are interested in optimising these remaining cases, by delaying the boxing until it is actually necessary. The descriptions of systems that we have reviewed suggest that when analysis cannot remove the need to box an object then an eager boxing operation is then used.

In summary, existing approaches to implementing the combination of generics and value types typically take one of two paths:

- Generate separate code for each different use of a generic type, thereby avoiding boxing completely — C++, Eiffel, Generic C#/CLR and Pizza (heterogeneous implementation) take this approach.
- Share a common code implementation, but insert boxing coercions as required to convert between the two representations — this is done by the ML compilers cited above and in Pizza (homogenous implementation).

However, the nature of the boxing in this second method of implementation is that it is applied eagerly. Our work refines this approach by delaying the boxing where possible.

3 The Pluto Language and Implementation

In order to examine the costs associated with supporting genericity and value type objects, we devised a prototype object-oriented language called Pluto and implemented a compiler for it[15]. This language supports the definition of value types and provides genericity in the form of parameterised types. The compiler implements the lazy boxing optimisation (described in Sect. 4) which can be switched on or off. This enables us to examine the effectiveness of the technique, and to measure its effect empirically.

The Pluto language is essentially similar to a generic version of Java, such as GJ[2] or NextGen[3], but with the addition of support for user-defined value types. The example code in Fig. 1 shows a value class called `point` and a generic `Cell` class. All of the code in the figure is programmer-level class definitions, rather than pre-defined. Therefore, a value class like `point` is similar to a struct in C#.

3.1 Value Types

A value class is akin to Eiffel's expanded classes[16] or a struct in C#[1], so its instances are handled directly and copied by value. In Pluto, value classes are part of the class hierarchy and can be used in generic situations just like ordinary reference-based classes. To provide uniformity, there are no primitive types in the language syntax — simple types such as integers and booleans are represented at the language level as value classes, although for efficiency reasons their implementation is optimised by the compiler.

In keeping with conventional value semantics, all value classes are *immutable* so instances of a value class cannot change their internal state. This property is critical to ensure that the compiler can make boxed copies of value class objects without concerns over aliasing and visible state changes. While the objects of a value class are immutable, this does not imply that we disallow updating of variables. Indeed, the Pluto language is still imperative in the sense that a variable of value type can be updated to hold a new value object — but we do not allow the internal state of the objects themselves to be modified. For example, the `point` class in Fig. 1 is a value type, so its internal fields `x` and `y` cannot be updated. But the local variables `p` and `q` in the `Main.run` method can be updated to hold a different `point` object.

Apart from the restriction of immutability, value classes can be used in much the same way as reference classes: they can define methods, have subclasses and implement interfaces. Most specifically for the work discussed here, value classes can be used as types for generic type parameters, e.g. `List[point]` or `Cell[point]`.

3.2 Implementation Overview

Our compiler generates C code, using the traditional representation of objects as a C `struct` containing the instance fields. Instance methods are translated into functions with an extra parameter for the `this` object. Instances of reference type classes are heap-allocated and handled via a pointer, whereas value class instances are handled directly: C structs and the C primitive types such as `int` that are used directly for Pluto's integer types, are passed around and copied by value.

For example, the code generated for the `Main.run` method in Fig. 1 will declare the two `point` objects `p` and `q` as local variables with a struct type, so they will be stored in the function's stack frame. Since the `point` class is declared final, then the C structure for objects of this class contains just two

```
interface Comparable [E] {
  abstract method ==(other:E) : boolean;
}


final value class point(x:int, y:int) implements Comparable[point] {
  field x:int = x;
  field y:int = y;

  method ==(other:point) : boolean {
    return (this.x == other.x) && (this.y == other.y);
  }
}


class Cell [T <: Comparable[T]] (initialData:T) {
  field element:T = initialData;

  method get() : T {
    return this.element;
  }

  method set(newData:T) {
    var this.element = newData;
  }

  method elementEquals(otherData:T) : boolean {
    return (this.element == otherData);
  }
}


class Main() {
  static method run() : int {
    let p:point = new point(3,4);
    let c:Cell[point] = new Cell[point](p);
    let q:point = new point(3,4);
    if (c.elementEquals(q)) {
      String.println("c is holding a point equal to q");
    }
    return 0;
  }
}
```

Fig. 1. Example Pluto code

integer members. The `Cell` class is a reference class, so object `c` in the example code will be allocated on the heap and only a pointer to it will be stored in the stack frame.

We are assuming a traditional C implementation of memory management, with local variables stored in stack frames and heap allocation through the `malloc` library function. Our generated code uses an off-the-shelf conservative garbage collector library[17] to manage the heap. Since we are generating C code, the issue of managing storage for value type objects is largely down to the C compiler used, which may choose to store small value objects in registers rather than stack frames. Naturally this reliance on C and a conservative garbage collector means that we give up some control on the handling of object storage. In future work we could consider generating assembly code directly, or using a lower-level intermediate representation such as `C--`[18].

3.3 Generic Types

A generic class is implemented using a single copy of the code at run-time — it is not expanded or specialised for each use like C++ templates. Generic data is handled with a uniform representation, i.e. a pointer to the actual object. For example, the `element` field of the `Cell` class will be implemented as a pointer. The type parameter of a generic class, such as `T` in `Cell`, can abstract over all subtypes of its bound, including value classes. When a generic class is used to store value class data, the value type objects are boxed when passed into a generic context and unboxed when passed back. For example, a boxing operation will be used to implement the code fragment where the `Cell[point]` variable `c` is initialised with the value class object `p`.

To implement the lazy boxing process, the implementation of a generic type may need to consult information about the actual type of data that a type parameter has been instantiated with. To support this, we add a single member to the object instance C structure for each type parameter in the generic type. This field is akin to a vptr, but links to information about the type parameter rather than the generic type itself. Like normal vptrs, these fields are set once when the generic object is constructed.

3.4 Final Value Classes and the Vptr

The implementation of the Pluto language uses the traditional object model where an object's representation at run-time is a structure containing all the instance fields, headed with a pointer (the vptr) to some class-specific data. One important difference in the handling of value classes in Pluto is that a value class marked `final` is represented at run-time without a vptr. The rationale for this choice is that objects of a final class do not require polymorphic method calls, because the exact type of object is known at compile-time. Hence the vptr field will not be consulted and can be omitted. Since value class objects are copied by value, omitting the vptr saves space and reduces the amount of data to be copied.

Because generic code may be used to handle objects of any type, it must account for the possibility that some objects it manipulates will have a vptr and some will not. This explains the need for the extra vptr fields described above for the implementation of generic types. A type parameter vptr field is used to obtain information about the generic object, unless it indicates that objects with their own vptrs are being handled, in which case those are consulted instead. Dealing with datatypes such as integers or other final value classes that do not have vptr fields has implications for the lazy boxing optimisation — we return to this topic in Sect. 5.2.

The omission of vptrs is particularly important for performance since integers and real numbers are supported in Pluto as final value classes. Although at the Pluto language level types such as integers are declared as final value classes, these types are implemented directly using the appropriate C types rather than using a struct. For example, an integer object in Pluto is just an `int` in the generated C code, because a vptr field is unnecessary.

4 Lazy Boxing Optimisation

The brief description of the Pluto language implementation given in the previous section indicates that boxing immediately occurs whenever a value type object is passed into a generic context. This is the conventional implementation of boxing, and could be described as an *eager* strategy since it creates a heap-allocated box for the object as soon as it is passed into a generic context. However, the requirement of a uniform representation of generic values is merely that we supply a pointer to an object — the actual location of the object does not necessarily have to be on the heap. Based on this observation, we could supply a pointer to an object stored on the stack and the generic code is just as able to handle the object. The benefit of using pointers to stack-allocated objects is that we do not incur the boxing overhead of obtaining a piece of heap memory and copying the object into it.

4.1 Object Lifetimes

One difficulty with using pointers to stack-allocated objects is due to the different nature of heap and stack storage. With a heap managed by a garbage collector, an object stored on the heap exists for as long as there is a pointer to the object. With a stack, an object stored in a stack frame only exists as long as the frame exists. If generic code retains a pointer to an object that no longer exists (a so-called dangling pointer) then dereferencing that pointer could have disastrous results. To ensure the safe use of pointers we can insert a boxing operation just at the point when a stack-allocated object that is still in use is about to be destroyed. This leads to a *lazy* strategy for boxing where pointers to objects in a stack frame are used until the frame is about to be removed, whereupon any objects still required are boxed. The potential benefit of this approach is in situations where a stack-allocated object exists for at least the time that it

is used by generic code — in this case, boxing is avoided entirely. Section 5.1 describes the lazy boxing analysis used by the Pluto compiler to determine when it is safe to use pointers to stack-allocated objects.

4.2 Dynamic Boxing

In situations where a stack-allocated object does need to be boxed at a late stage, because it is still required, the boxing operation does not benefit from compile-time information about the object. In the eager approach, boxing code is inserted in the non-generic context where the type of object to be boxed is known at compile-time. A consequence of the lazy approach is that the boxing code is now in a generic context, where it only has a pointer to some stack-allocated object and needs to copy this object to the heap. The type and size of the object being pointed to is unknown at compile-time, so run-time type information must be available to guide the boxing process. We call this operation *dynamic boxing* because it works without the benefit of compile-time information. Section 5.2 explains in more detail how this is achieved in the Pluto implementation.

4.3 Example of Redundant Boxing

Naturally, the effectiveness of using the lazy strategy in general depends on the extent to which typical uses of generic code with value type objects can actually avoid boxing. One area in which redundant boxing can be found is the passing of generic data as arguments to methods. For example, the code in Fig. 1 contains a method call `c.elementEquals(q)` where the value type object `q` is allocated on the stack. Since the method parameter expects generic data in the uniform representation, a pointer to an object is required. Using the eager boxing strategy, the `q` object would be immediately copied to the heap and the address of this copy used as the method argument.

However, the boxing is actually redundant in this case because a pointer to `q` on the stack could have been safely used instead. We can simply pass the address of `q` (i.e. a pointer to the stack) to the `elementEquals` method, which in turn passes the pointer on as an argument to the `==` method of the generic `element` field. Since the `Cell` object is being used to contain a `point` element type, the `==` method that is actually called is that defined by the `point` class. Here, the required `point` argument is obtained by dereferencing the pointer that has been passed in from the call site. This pointer refers to the `q` object on the stack, which is still in existence so the dereferencing is safe. Hence, this simple example shows one case in which use of eager boxing would result in unnecessary heap allocation and copying.

5 Implementation Details

This section describes the two main aspects to our implementation of lazy boxing: analysing when the optimisation can be made, and generating the necessary code to do dynamic boxing.

5.1 Safe Use of Stack Pointers

As briefly explained in Sect. 4.1, using pointers to objects on the stack can be problematic if the object no longer exists when a pointer to it is dereferenced. To prevent this situation from occurring, we must only generate C code that uses stack pointers in a safe manner. Our compiler adheres to the following two safety rules for the C code it generates:

1. A pointer to a stack-allocated object cannot be stored as an instance field in any object. Therefore, all instance fields that are represented with a C pointer will point to a heap-allocated object.
2. A pointer to a stack-allocated object cannot be stored in a stack frame that is older than the one in which the object is stored.

These rules ensure that a pointer is always safe, meaning it does not outlive the object that it points to. It should be noted that these are not the only possible rules for safe stack pointer usage — indeed, they can be overly conservative in some cases. Similar restrictions are made and formalised in [19], which explains how a core fragment of the Microsoft IL is type checked to prevent unsafe stack pointers.

In order to abide by these rules, the compiler must ensure that a pointer to a stack-allocated object is not used in those places where a violation could occur. Its task is to intercept stack pointers and replace them with pointers to the heap. There are two areas of the generated code that require attention:

1. Initialising or updating the value of an instance field of generic type.
2. Returning a value from a method whose declared return type is generic.

The actual replacement operation required in these situations depends on the type of value being stored or returned. If it is a value type object, then the object is eagerly boxed, which provides us with a heap pointer. However, if the value is already generic (i.e. a pointer) then the replacement task depends on whether the pointer could refer to an object on the stack or one already on the heap. If the object is on the stack then dynamic boxing is required, which is considered further in Sect. 5.2.

This section describes the situations in which unsafe uses of stack pointers could occur and shows how they are avoided. However, we are able to use pointers to stack-allocated objects in any other situation, e.g. when passing value type objects as generic value arguments. For example, Sect. 4.3 showed a case where no kind of boxing was needed. The C code generated for that fragment will just pass the address of a stack-allocated object as an argument, with no interception or replacement required.

5.2 Dynamic Boxing Using Run-Time Type Information

The main aim of lazy boxing is to use pointers to stack-allocated objects for as long as it is safe to do so. In situations where the object exists on the stack for

at least as long as any pointers to it are in use, then we avoid boxing entirely and save the cost of heap allocation and copying. However, there are cases where generic code may need to refer to an object for longer than it exists on the stack.

As explained in the previous section, if a generic pointer value is about to be stored in an object field or returned from a method then we must ensure that it points to an object on the heap. Since there is no reliable and portable way to determine in C whether a pointer actually refers to stack or heap storage[20] our compiler takes a conservative approach. We examine the actual run-time type of the object being pointed to, and if it is an instance of a reference type then we know it must already be allocated on the heap and hence no boxing is required. However, if the run-time type information indicates that the pointer refers to a value class object then our assumption is that it might be stack-allocated, so a dynamic boxing operation is needed to ensure safety. Some implications of this conservative assumption can be seen in the results shown in Sect. 6.2, which are discussed in Sect. 7.

The difference between ordinary eager boxing operations and the dynamic boxing described in this section is that the latter works in a generic context. This means that the compiler cannot determine statically the size of the object that is to be copied to the heap. The only information available in the generic context is a pointer to the object, and access to run-time type information. Consequently, a dynamic boxing operation is implemented by reading the size of the object from the run-time type information and creating a copy of the object on the heap.

From the description of the dynamic boxing process detailed here, two pieces of information about each type are required at run-time: whether the type is a value class or not, and the size of an object of that type. In our implementation, we store this information in the type-specific structure (the vtable) accessible via the vptr field in objects. One complication here is that objects of a final value class (such as integers or real numbers) do not have a vptr field, as explained in Sect. 3. In order for generic code to access run-time type information when a generic value may point to an object without a vptr, we maintain a pointer to the type-specific information in the generic object itself. For example, an object of the generic `Cell` class from Fig. 1 will contain a pointer to information about the particular type that it is being used for — such as `point` in the example code. This link allows the dynamic boxing code to determine the necessary information to copy an object to the heap.

6 Experimental Results

To examine the effectiveness of the lazy boxing optimisation in practice, we wrote some small test programs and built executables both with and without the boxing optimisation. Since our compiler can be configured to use either the eager or lazy boxing approaches, we were able to measure the differences in total (heap plus stack) memory usage and execution time for the test programs under the two techniques.

Table 1. Comparing reference and final value classes

Test Program	Memory Usage (Kilobytes)		Running Time (seconds)		Relative Speedup
	Reference	*Final Value*	*Reference*	*Final Value*	
matrix_mult	3472	1708	7.07	0.85	88%
gen_comptr_tree	704	704	3.14	2.92	7%
word_comptr_tree	704	616	3.11	2.83	9%
gen_compare	180	180	1.59	1.18	26%
word_tree	820	660	6.30	5.81	8%
gen_tree	816	1028	6.47	7.61	-18%

Since the Pluto language is an experimental one, we had no large test suites with which to measure performance. Therefore, the test programs simply exercise generic data structures such as binary search trees and matrices, where the element type is a value class. These programs are admittedly only micro-benchmarks, that perform large amounts of boxing: value type objects are stored in, and retrieved from, generic container structures. The tests were deliberately chosen to isolate the boxing activity that we are interested in optimising, so that any change in performance can be attributed to the optimisation effect. Naturally, we would need to implement the optimisation in an existing language compiler to measure its effects on larger real-world programs.

The test programs used for this work were built and run on a P3-500 PC with 384MB RAM, running Redhat Linux 6.1. We used gcc version 2.95.2 to compile the C code generated by our compiler, and statically linked the code with version 5.3 of the Boehm garbage collection library.

6.1 Measuring the Benefits of Value Types

Although our primary interest is in measuring the effect of lazy boxing, the first set of test results (shown in Table 1) just compares the performance of test programs where the element type used in a generic data structure is either a reference type or a value type. The purpose of these results is to show whether the language distinction between reference types and value types provides any benefits at all. Since the only need for any kind of boxing is due to the use of value types with genericity, it is important to establish whether a program using value type objects that are boxed as required, performs better than an equivalent program which uses reference type objects instead. If not, we could just use the uniform pointer representation for objects of all types at all times — no boxing would ever be needed.

The programs listed in Table 1 were tested in two configurations: the data structure element type was declared as either a reference class (i.e. a pointer-based representation such as that used for Java classes) or a final value class.

Table 2. Comparing eager and lazy boxing

Test Program	Memory Usage (Kilobytes)		Running Time (seconds)		Relative Speedup
	Eager Boxing	*Lazy Boxing*	*Eager Boxing*	*Lazy Boxing*	
matrix_mult	2036	1708	0.89	0.85	5%
gen_comptr_tree	988	704	3.58	2.92	18%
gen_compare	180	180	2.45	1.18	52%
word_tree	660	660	8.24	5.81	30%
gen_tree	816	1028	7.41	7.61	-3%
gen_stack	308	392	5.23	2.23	57%

All of the tests shown in this table were conducted with the lazy boxing optimisation enabled. The results show that, for all the programs except gen_tree, the amount of memory used and the running time of the final value class version was an improvement on the reference class version. Clearly, programs such as matrix_mult, which generates large numbers of temporary objects during its calculations, perform much better when using stack-allocated objects. However, the figures for the gen_tree program show that not all programs benefit from using value class objects. This particular program highlights an inefficiency in our implementation of boxing, and is examined further in the context of the boxing tests below.

The results of this set of tests show that the combination of value type objects and boxing can provide benefits over using reference type objects. While this outcome is unsurprising, and certainly not new, it provides a foundation for the following set of tests which measure the difference between different approaches to boxing.

6.2 Comparing Eager and Lazy Boxing

In the previous set of tests, we compared the difference between reference and value element types while keeping the boxing approach fixed. In the next set of tests (results shown in Table 2) the element type is fixed as a final value class, and we compare different approaches to boxing. Most of the programs used here are the same as those from the previous test set. The purpose of this experiment is to examine whether the lazy boxing optimisation leads to actual improvements in program speed and memory usage. The figures in Table 2 show that lazy boxing provides a speedup compared to eager boxing in all cases except one (the gen_tree program again) with a geometric mean speedup of 25% for this set of programs.

The results for the gen_tree program show a small slowdown and increased memory consumption for the lazy boxing version compared to eager boxing. This is counter to the trend seen in the other test programs. We investigated the cause of this unexpected result and determined that it is due to *reboxing*, where

a boxed copy of an object is made even though the original object is already on the heap. This accounts for the extra memory consumption and extra time required to make the redundant copy. We discuss the source of this inefficiency, and consider a solution, in Sect. 7 below.

7 Discussion

The initial motivation for using a lazy approach to boxing value type objects was that in some cases a boxing operation would never be needed at all. Naturally, the effectiveness of the technique depends on whether the extra complexity and possible cost of doing dynamic boxing operations at a later stage is outweighed by actual savings due to boxing being avoided completely in some cases. The results presented in the previous section certainly show that this optimisation is effective, and provides performance benefits for all but one of the test programs. As previously discussed in the review of other relevant work, there exist compile-time analysis techniques that can remove boxing operations. These could work in concert with our optimisation by removing boxing operations that are definitely redundant, leaving the remaining cases where the analysis was inconclusive to be handled using the lazy boxing approach.

Another factor in measuring the benefit of using lazy boxing is determining the range of situations in which it is most applicable and therefore effective. Since we have only tested the optimisation with small benchmark programs, it is admittedly not clear how the technique would fare with larger programs. Most of the benchmark programs involved creating value type objects and passing them into a generic context, such as code for a generic tree data structure. As stated in Sect. 4.3 there are certain patterns of code in which lazy boxing can likely be applied. For example, when passing a value type object as a generic argument to a method call, it is often the case that the pointer argument is only used to examine the object rather than store a copy of it. So the frequency of such code patterns in real programs really determines how often the lazy boxing optimisation can be exploited. A next step would be to instrument an existing language compiler that currently performs eager boxing, and measure how often lazy boxing *could* have been performed for a suite of large programs. This would tell us how widely applicable the technique is, beyond micro-benchmarks.

One design decision taken by our work (also by the C# and Eiffel languages) is that programmers should annotate classes as either reference or value type. We chose to take this approach because programmers typically have a good intuition about which data types are immutable value-like and which are shared reference-like. Adding the `value` keyword to particular class definitions is then a trivial task, so comparing program performance in the value and reference scenarios is easy to do. The alternative would be for the compiler to perform escape analysis and attempt to decide automatically which objects should be allocated on the stack and which on the heap. This is a finer-grained approach, since the allocation can be chosen per-object rather than per-class. While we did not pursue such a design, it would be useful to compare whether a programmer's

choice of value class annotation actually results in better performance than an analysis-based allocator. We believe that current language designs tend towards enabling the programmer to explicitly make the decision about which types should be value-based, hence our work focuses on implementing such languages.

The results for one of the test programs indicate that lazy boxing can perform worse than eager boxing in certain circumstances. The source of this inefficient handling can be found in the implementation of dynamic boxing as described in Sect. 5.2. Recall that the dynamic boxing process cannot determine whether a pointer actually points to the heap or stack. When the pointer refers to a value class object, it might be stack-allocated so the compiler takes a conservative approach and makes a boxed copy, to be safe. If the object had in fact already been boxed then it will be boxed again, unnecessarily. In contrast, the eager approach boxes objects exactly once. This shows a limitation of the analysis used in our implementation, which could be improved to distinguish pointers that definitely refer to objects on the heap from those that may not. Our aim is to implement the lazy boxing technique such that it never performs worse than the eager approach. Implementing this extra analysis is future work.

Although our work focuses on the boxing of value type objects for use with generic code (i.e. parametric polymorphism) the general technique of boxing may also be used to implement languages that provide polymorphism through subtyping. For example, the C# language does not currently support genericity, but it allows value type objects such as integers to be used by code that expects reference type objects. In effect, value types are considered part of the class hierarchy. The language specifies that boxing and unboxing will occur on the boundary between value and reference type contexts. Similar boxing rules appear in the Eiffel language specification. Therefore, our approach of using pointers to objects on the stack could also be applied in C# or Eiffel implementations.

8 Conclusions

We have presented an optimisation scheme for object boxing, called lazy boxing, whereby pointers to a value type object in stack storage can be used by generic code. Only when a stack-allocated object is about to be deallocated is a boxing operation necessary. This technique reduces the cost of object boxing, compared to the conventional eager approach, by avoiding boxing operations that are unnecessary. We have implemented this optimisation in a compiler for a language with value types and genericity, and showed experimental test results that demonstrate its effectiveness, achieving 25% program speedups on average. The lazy boxing optimisation scheme is applicable to language implementations that employ boxing of value type data for use in a context requiring a uniform pointer representation.

Acknowledgements. This work was financially supported by an EPSRC studentship.

References

1. Hejlsberg, A., Wiltamuth, S.: C# language reference. Microsoft Developer Network Library (2000)
2. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: Adding genericity to the Java programming language. In: Proceedings of OOPSLA'98. Volume 33(10) of SIGPLAN Notices., ACM (1998) 183–200
3. Cartwright, R., Steele, G.: Compatible genericity with run-time types for the Java programming language. In: Proceedings of OOPSLA'98. Volume 33(10) of SIGPLAN Notices., ACM (1998) 201–215
4. Odersky, M., Wadler, P.: Pizza into Java: Translating theory into practice. In: Proceedings of POPL'97, Paris, France, ACM Symposium (1997) 146–159
5. Kennedy, A., Syme, D.: Design and implementation of generics for the .NET Common Language Runtime. In: Proceedings of PLDI'01. Volume 36(5) of SIGPLAN Notices., ACM Conference (2001) 1–12
6. Stroustrup, B.: The Design and Evolution of C++. Addison-Wesley (1994)
7. Collin, S., Colnet, D., Zendra, O.: Type inference for late binding: The SmallEiffel compiler. In: Joint Modular Languages Conference. Number 1204 in Lecture Notes in Computer Science. Springer-Verlag (1997) 67–81
8. Leroy, X.: Unboxed objects and polymorphic typing. In: Proceedings of POPL'92, Albequerque, New Mexico, ACM Symposium (1992) 177–188
9. Shao, Z., Appel, A.W.: A type-based compiler for Standard ML. In: Proceedings of PLDI'95. Volume 30(6) of SIGPLAN Notices., ACM Conference (1995) 116–129
10. Harper, R., Morrisett, G.: Compiling polymorphism using intensional type analysis. In: Proceedings of POPL'95, San Francisco, California, ACM Symposium (1995) 130–141
11. Shao, Z.: An overview of the FLINT/ML compiler. In: Workshop on Types in Compilation, TIC'97 (1997)
12. Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., Lee, P.: TIL: A type-directed optimizing compiler for ML. In: Proceedings of PLDI'96. Volume 31(5) of SIGPLAN Notices., ACM Conference (1996) 181–192
13. Choi, J.D., Gupta, M., Serrano, M., Sreedhar, V.C., Midkiff, S.: Escape analysis for Java. ACM SIGPLAN Notices **34** (1999) 1–19
14. Gay, D., Steensgaard, B.: Fast escape analysis and stack allocation for object-based programs. In: International Conference on Compiler Construction (CC'2000). Volume 1781., Springer-Verlag (2000)
15. Owen, T.: A Type-Passing Implementation of Value Types and Genericity. PhD thesis, University of Sussex, Brighton, UK (2002) Available at `http://www.cogs.susx.ac.uk/users/timo/thesis.pdf`.
16. Meyer, B.: Eiffel: The Language. Prentice-Hall (1992)
17. Boehm, H.J.: Space efficient conservative garbage collection. ACM SIGPLAN Notices **28** (1993) 197–206
18. Peyton Jones, S.L., Nordin, T., Oliva, D.: `C--`: A portable assembly language. In: Proceedings of the 1997 Workshop on Implementing Functional Languages. Lecture Notes in Computer Science. Springer-Verlag (1998)
19. Gordon, A., Syme, D.: Typing a multi-language intermediate code. In: Proceedings of POPL 2001, London, England, ACM Symposium (2001) 248–260
20. Meyers, S.: More Effective C++. Addison-Wesley (1996)