Ownership You Can Count On: A Hybrid Approach to Safe Explicit Memory Management

Adam Dingle
Google
1600 Amphitheatre Parkway
Mountain View, CA 94043
+1 (415) 425-6891
adamdingle@gmail.com

David F. Bacon
IBM T.J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532
+1 (914) 784-7811
dfb@watson.ibm.com

ABSTRACT

While many forms of memory management have been proposed, the only ones to achieve widespread adoption have been explicit deallocation and garbage collection. This leaves programmers requiring explicit control of memory behavior unable to obtain the software engineering and security benefits of programming in a safe language. We present a new approach to memory management called *alias counting* which combines type-based object ownership and run-time reference counting of references by non-owning (aliasing) pointers. When an owning pointer is destroyed, if the target object's reference count is zero, the object is destroyed; otherwise a run-time error occurs. We have implemented alias counting in a safe C#-like language called Gel. The Gel compiler includes an effective reference count elimination optimization that takes advantage of ownership semantics. Our anecdotal experience in writing the Gel compiler and some smaller benchmarks in Gel is that the annotation requirements and extra programming effort are small, and the type system catches most ownership errors at compile time. Quantitatively, we compare microbenchmarks written in Gel, Java, C#, and C++, and versions of the Gel compiler written in both Gel and C#. In our benchmarks, Gel programs require at most 40% more memory than the corresponding C++ programs, while C# and Java generally require at least twice as much memory and often considerably more. Run-time performance varies considerably across all languages, with no clear winner, but Gel outperforms either C++ or Java/C# in all but one benchmark. Our reference count optimization typically eliminates 90% of all reference counting operations, speeding Gel programs by 20-40%.

1. INTRODUCTION

Today, most system programming is done in languages which are *unsafe*: program errors may lead to arbitrary or undefined behavior. C and C++ are unsafe languages commonly used for system programming. Unsafe languages have certain fundamental disadvantages over *safe* languages such as Java, C# and ML. Certain classes of bugs such as buffer overruns or reads from deallocated memory can occur only in unsafe languages. Many such bugs are subtle or non-deterministic, and hence expensive in software engineering. In addition, trusted and untrusted code may safely share a single address space only in a safe language.

There would thus be many advantages to using safe languages for system programming. But there are challenges involved in making safe languages as efficient as unsafe languages in their run-time resource usage. In particular, any safe language must have some sort of automatic memory management mechanism: explicit memory allocation and deallocation as found in C and C++ is inherently unsafe, as there is nothing to prevent a program from writing to deallocated memory. The most widely used automatic memory management mechanisms include classical reference counting and garbage collection [18]. Reference counting may incur substantial run-time overhead and cannot easily free cyclic data structures. Garbage collected-systems often use substantially more memory than their manually managed counterparts. An additional disadvantage of garbage collection is that it frees memory non-deterministically. Deterministic destruction is useful in system programming, especially in languages which allow arbitrary user code to run when objects are destroyed; such code may be used to release system resources when no longer needed.

In this paper we propose *alias counting*, an automatic memory management mechanism based on ownership and run-time reference counts. The mechanism can be used to implement a safe language which frees objects deterministically. To show that alias counting is practical, we've constructed a simple object-oriented language called Gel which is essentially a subset of C# [14] extended with ownership-based memory management. We've implemented an interpreter and compiler for Gel, written in the Gel language itself. We've ported a number of common benchmark programs to Gel and have found that the Gel implementations often perform similarly to their C++ counterparts and use significantly less memory than their Java and C# equivalents.

Unlike languages with simple manual memory management schemes (such as malloc/free or new/delete), Gel is *safe*: a bug in a Gel program can never corrupt memory arbitrarily. Unlike garbage-collected languages, memory management in Gel is *deterministic*: objects are freed as soon as they are no longer needed. Unlike traditional reference-counting schemes, Gel can free data structures including *cycles* of pointers as described below.

In this paper, we describe only Gel's ownership extensions to the C# language. A reader familiar with either C# or Java should be able to follow with no trouble; differences between the core C# and Java languages are relatively minor and are not relevant to the discussion here. For a complete description of Gel, see the language reference [24]. The Gel project site [23] contains the Gel interpreter/compiler, released as open source under the MIT License. Our decision to extend C# rather than, say, Java was arbitrary, and it would be straightforward to construct a similar language extending Java or another object-oriented language. The Gel language is large enough to write useful programs, such as the Gel interpreter/compiler itself.

Our proposed memory management mechanism has the following fundamental characteristics:

- Objects are allocated individually using a heap allocator. Deallocation is automatic; there is no `delete` statement.
- The type system distinguishes *owning* and *non-owning* pointers to objects. This is a compile-time distinction only: at run time, all pointers are represented as ordinary memory addresses.
- The type system ensures that at run time there is always exactly one owning pointer to every object.
- Ownership of an object may *transfer* as a program runs. Any owning pointer which loses ownership of an object becomes either null (if it is a field) or inaccessible (if it is a local variable or parameter).
- At run time, the language implementation keeps a reference count of the number of non-owning pointers to each object.
- When a non-null owning pointer is destroyed, the object it points to is destroyed as well. If a destroyed object has a non-zero reference count, a run-time error occurs and the program is terminated; it is the programmer's responsibility to avoid this condition.

Our goal in this work is to show that a safe language with such a type system and memory management mechanism can be implemented, and can be practical in the following senses:

- It is easy to write real programs so that they pass the type checker; implementing common data structures is straightforward.
- It is easy to write programs so that they never yield run-time errors due to non-zero reference counts on destroyed objects.
- Programs run efficiently, ideally consuming roughly as much memory and CPU time as their manually managed counterparts.

The contributions of this work are:

- A unique approach to safe memory management combining static type-based ownership with run-time reference counts;
- An implementation of the approach in a dialect of C# we call Gel;
- Implementation of the Gel compiler and a number of benchmarks in Gel;
- Intra- and inter-procedural optimizations that take advantage of ownership properties and make the reference counting cost acceptable by eliminating typically 90% of all counting operations;
- An experimental evaluation comparing the performance of one large Gel program (the Gel compiler) to the same program written in C#, and of several microbenchmarks written in Gel, C++, Java, and C#.

Currently, the major limitation of Gel is that its reference counting optimizations can not easily be applied in the presence of aliasing arising under multi-threading. As a result, Gel currently only supports single-threaded programs.

2. THE GEL LANGUAGE

We proceed as follows. We first give an informal overview of ownership in Gel, illustrated with a number of code examples. We then present detailed rules for type checking and an argument that the single-owner property holds in Gel.

Gel, like C#, includes both *value types* and *reference types*. Value types in Gel are primitive types including `bool`, `int` and `float`. Reference types include `object`, `string` and user-defined subclasses of `object`. Gel extends C#'s type system with *owning types*. The type T ^ (where T is a reference type) represents an owning pointer to an object of type `T`. A reference type written without a ^ character indicates a non-owning pointer. Like a non-owning pointer, an owning pointer may hold either an object reference or `null`.

In Gel, as in C#, all value and reference types are ultimately derived from the top-level `object` type. Strictly speaking, inheritance applies only to value and reference types; owning types are not themselves considered members of the class inheritance graph.

Local variables, method parameters, method return values and object fields may all have owning or non-owning types.

In Gel there is always exactly one owning pointer to every object. As a consequence of this fact, all objects in a running Gel program belong to a set of ownership trees; tree roots include static fields of every class and local variables in every stack frame. (During the course of expression evaluation, temporary values of owning type may form additional tree roots.)

Ownership of an object can *transfer* from an owning pointer P to an owning pointer Q. In this case we say that P *loses ownership* of the object and that Q *takes ownership* of the object.

At run time, the Gel implementation keeps a count of the number of non-owning pointers to each object; this count may be zero or an arbitrary positive value. The count is used only for ensuring that no non-owning pointers remain to a destroyed object as described in a following section.

Here's a minimal Gel program:

```
class Foo {
 public static void Main() {
  Foo ^f = new Foo();
 }
}
```

The `new` operator returns an owning pointer. In this program, the local variable `f` receives ownership of the newly allocated object. When `Main()` exits and `f` goes out of scope, Gel destroys the object automatically.

If an owning pointer variable is updated with a new value, any previously pointed-to object is destroyed immediately:

```
void Fun() {
 Foo ^f = new Foo(); // allocates a new Foo
  f = new Foo(); // the first Foo allocated
is destroyed here
  f = null; // the second Foo allocated is
destroyed here
 ...
}
```

An assignment to an owning variable causes an ownership transfer:

```
void Fun() {
 Foo ^f = new Foo();
 Foo ^g = f; // ownership transfers to g
}
```

It is a compile-time error to read from any local variable which may have lost ownership:

```
void Fun() {
 Foo ^f = new Foo();
 Foo ^g = f; // ownership transfers to g
 Foo h = f; // error: f has lost ownership
}
```

The Gel type checker enforces this restriction by performing control flow analysis on each method and ensuring that no owning variable loses ownership on any code path leading to a point where the variable is read. This is a straightforward extension of the mechanism used to ensure that every variable is initialized before use.

Gel could, alternatively, cause an owning local variable to become null if it loses ownership. We've decided against this behavior since this side effect might not be obviously apparent to someone reading the code. More fundamentally, this would destroy the language's *erasure* property, described in a following section.

2.1 Fields

As mentioned above, object fields (including both instance fields and static fields) may have either owning or non-owning types. Here is a singly linked list class which holds an integer in each node:

```
class Node {
 public Node ^next;
 public int i;
 public Node(int j) { i = j; }
}
```

Gel includes an operator `take` which takes ownership of a value from an owning field, and has the side effect of setting the field's value to `null`. An expression which reads an owning field without using `take` has a non-owning pointer type:

```
void main() {
 Node ^n = new Node(5);
 n.next = new Node(4);
 Node ^p = n.next; // compile-time error: no
conversion from Node to Node ^
 Node ^p = take n.next; // okay: ownership
transfers; now n.next is null
}
```

Gel could conceivably take ownership from and null out owning fields automatically whenever they're read in an owning context; then the `take` operator would be unnecessary. But then the side effect of nulling out a field would be much less apparent to programmers; Gel includes `take` so that this side effect is explicit (and to preserve the erasure property described below).

2.2 Methods

A method parameter with owning type takes ownership of the value passed to that parameter when the method is invoked. If a method's return type is owning, then the method's caller takes ownership of the return value when the method returns.

Here is a doubly linked list class with an integer in each node; its constructor takes arguments with owning and non-owning pointer types:

```
class Node {
 int i;
 public Node ^next;
 public Node prev;

 public Node(int j, Node ^n, Node p) {
  i = j; next = n; prev = p;
 }
}
```

As a larger example, the following method takes an integer `k` and returns a doubly linked list containing the integers 1 through `k`:

```
Node ^IntList(int k) {
 Node ^n = null;
 for (int i = 1 ; i <= k ; ++i) {
  // keep pointer to previous head of list
  Node o = n;

  n = new Node(i, n, null);
  if (o != null)
   o.prev = n; // create backlink
 }
 return n;
}
```

In the line which calls the `new Node()` constructor, the owning variable `n` first loses ownership of the object it points to, since it passes that value to a constructor parameter with owning type: the constructor takes ownership of the object passed. After the constructor returns, `n` receives ownership of the newly allocated object.

2.3 Type Casts

Type casts in Gel never affect whether a type is owning; a cast is always from an owning type to an owning type or from an non-owning type to an non-owning type. Syntactically, a cast is always written without the "^", even if the type it affects is owning. Thus:

```
void Fun() {
 Foo ^f = new Foo();
 Bar ^b = (Bar) f; // f loses ownership to b
}
```

Gel, like C#, includes operators `is` and as which perform run-time type checks. The expression `E is T` evaluates E to a value v, and returns true if v is of type T and false otherwise. The expression `E as T` evaluates E to a value v, and returns v if it is of type T and null otherwise. Like type casts, the `is` and `as` operators are always written with non-owning types, but may operate on either owning or non-owning types:

```
void Fun() {
 Foo ^f = new Foo();
 Bar ^b = f as Bar;
}
```

2.4 this

In Gel `this` is always a non-owning pointer:

```
class Foo {
 static Foo ^f;
 ...
 void Fun() {
  f = this; // compile-time error:
            //can't convert from Foo to Foo ^
 }
}
```

2.5 Strings

Strings in Gel are not owned; the type `string ^` does not exist. In a compiled Gel program, strings are internally reference counted; a string is freed when its reference count reaches zero.

We considered making strings owned in Gel, but strings are so common in real-world programs that we thought it might burdensome for programmers to have to worry about string ownership. There's no problem with using classical reference counting for strings: they don't point to other objects so they can never be involved in a pointer cycle.

A `string` may be converted either to an `object ^` or to an `object`:

```
void Fun() {
 object ^o = "a"; // ok
 object p = "b"; // ok
}
```

2.6 Arrays

In Gel an array may hold either owning or non-owning pointers:

```
void Fun() {
 Foo [] ^a = new Foo[5]; // an owned array
of non-owning pointers to Foo objects
 Foo ^[] ^b = new Foo^[5]; // an owned array
of owning pointers to Foo objects
 Foo ^[] c = new Foo^[5]; // a non-owned
array of owning pointers to Foo objects
 b[0] = new Foo();
 a[0] = b[0];
}
```

The `take` operator may operate on array elements:

```
void Fun() {
 Foo ^[] ^a = new Foo^[5];
 a[0] = new Foo();
 Foo ^f = take a[0]; // a[0] is null after
the take
}
```

2.7 Autoboxing

Gel (like Java 5 and C#) provides *autoboxing*: a simple value such an `int` or `bool` may be stored in a variable holding an object. In Gel, a boxed value always has the owning type `object ^`. For example:

```
void Fun() {
 object ^o = 7; // ok
 object p = false; // compile-time error:
can't convert from bool to object
}
```

In an owning type `T ^`, `T` must not be a value type; the types `int ^` and `bool ^` do not exist in Gel, for example. An unboxing conversion must be explicit, and yields a value type:

```
void Fun() {
 object ^o = 7;
 int i = (int) o;
}
```

2.8 Object Destruction

Whenever an owning pointer to an object o is destroyed without transferring ownership elsewhere, then Gel *destroys* o as follows. First Gel destroys all fields in o (in reverse lexical order). If any of these fields are owning pointers to other objects then those objects are destroyed recursively; if any fields are non-owning pointers the reference count of their referent is decremented. Finally Gel checks o's non-owning reference count. If the reference count is non-zero, Gel issues a run-time error and terminates the program; otherwise Gel frees the memory used to store o and continues execution.

This simple destruction algorithm can destroy linked data structures automatically. If each node of a singly linked list contains an owning pointer to the next node in the list, then when a pointer to the list head goes out of scope Gel will destroy the entire list.

Because an object's subobjects are destroyed before the object's reference count is checked, the algorithm can destroy many data structures containing pointer cycles. As a simple example, the following method uses the `IntList` method presented earlier to create a doubly-linked list containing 2 nodes:

```
void MakeList() {
 Node ^n = IntList(2);
}
```

When the method exits, the variable n goes out of scope and the list is destroyed automatically. Initially, the first list node has a non-owning reference count of 1 since the second list node points to it. Before checking the first node's reference count, Gel destroys the second list node, which destroys the non-owning pointer to the first node and hence decrements the first node's reference count to 0.

In general, then, Gel can automatically destroy any data structure in which every non-owning pointer from an object o points to an object which is an ancestor of o in the object ownership tree.

Gel's object destruction algorithm has some significant limitations. Most fundamentally, it cannot destroy arbitrary graphs of non-owning pointers; if a data structure contains non-owning pointers to non-ancestor nodes, then the programmer must write code to null out those pointers before the structure is destroyed. Another problem is that the algorithm may use a large amount of stack space when destroying objects due to its recursive nature. In the Extensions section, below, we propose a better algorithm for destroying objects in Gel.

2.9 Type Conversions

In the sections above we've described how ownership interacts with various Gel language features, and presented various examples where Gel allows or disallows conversions between owning and non-owning types. In this section we give a more complete description of Gel's rules for type conversions.

Gel classifies each expression in a program as one of the following *kinds*:

- A *variable*: either
  - a local variable or method parameter variable
  - `(type) E`, where E is classified as a variable
  - `E as type`, where E is classified as a variable
  - `E ? E1 : E2`, where E1 and E2 are classified as variables
- A *field*: either
  - an access to a instance field or static field
  - an access to an array element
- A *value*: any other expression

Gel, like C# and Java, includes both *implicit* and *explicit* type conversions. A number of language constructs implicitly convert a

value to a destination type. Explicit type conversions are performed by the type cast operator and by a small number of other language constructs.

Every implicit conversion in Gel takes place in one of the following conversion contexts:

- Assignment context: in an assignment V = E, when implicitly converting from the type of E to the type of V
- Argument conversion context: in a method call, when implicitly converting a method argument to its corresponding parameter type
- Return context: when converting the expression in a return statement to the method's return type
- Other context: used for a small number of additional conversions

We now present Gel's implicit and explicit conversion rules. (We omit rules relating to conversions between value types, such as between int and char; these rules are detailed in the Gel reference manual and are similar to value conversion rules in C#.)

2.9.1 Implicit conversions

Every type T is implicitly convertible to itself.

If a type T is derived from a type U, then T is implicitly convertible to U.

The null type is implicitly convertible to any reference type or owning type.

T ^ is implicitly convertible to U ^ if T is implicitly convertible to U.

In an argument conversion or local assignment context, T ^ is implicitly convertible to U if T is implicitly convertible to U, and either

- the conversion is in an argument conversion context, or
- the conversion is in an assignment context, and the source expression is classified as a variable.

Any value type is implicitly convertible to `object ^`. In an argument conversion context, any value type is also implicitly convertible to object. These conversions are *boxing conversions*. In a compiled Gel program, a boxing conversion allocates a boxed object instance.

`string` is implicitly convertible to `object ^` and to `object`.

2.9.2 Explicit conversions

If T is implicitly convertible to U, then T is explicitly convertible to U.

If type T is derived from a type U, then U is explicitly convertible to T. This conversion will fail if the source value is not an instance of T.

T ^ is explicitly convertible to U ^ if T is explicitly convertible to U.

`object` and `object ^` are explicitly convertible to any value type T; this is an *unboxing conversion*. This conversion will fail if the source value is not actually an instance of T.

`object` and `object ^` are explicitly convertible to `string`. This conversion will fail if the source value is not actually a string.

2.9.3 Discussion

Gel never allows a type conversion to a non-owning to an owning pointer type. A conversion from an owning to a non-owning pointer is allowed only in certain contexts as per the rules above; when such a conversion occurs, at run time the source pointer retains ownership and a new non-owning pointer is created. The conversion rules disallow conversions which would inevitably lead to object destruction errors at run time. As an example, consider the following method:

```
Foo Fun() {
 return new Foo();
}
```

The type conversion from `Foo ^` to `Foo` occurs in a return context, and is hence disallowed. Gel could theoretically allow this type conversion and generate code for the method. Then at method exit the newly allocated Foo object would be destroyed (since it is owned by a temporary owning pointer, and ownership hasn't transferred elsewhere). But the destroyed object's non-owning pointer count would be 1, representing the non-owning pointer returned from the method, and so the object destruction would yield a run-time error.

2.10 Erasure

Note that Gel adds only two syntactic elements to C#: the type constructor ^ and the operator `take`. Suppose that we apply the following *erasure* transformation to a Gel program P:

- We remove every instance of the ^ type constructor: we replace T ^ with T everywhere.
- For each instance `take E` of the `take` operator, let T ^ be the compile-time type of E. Then we replace `take E` with the function call `Take(ref E)`, where `Take` is defined as follows:

```
T Take(ref T p) {
 T v = p;
 p = null;
 return v;
}
```

(Informally, this is a function which works like the `take` operator but acts on a non-owning type).

Then the resulting program P' is a valid C# program. If P terminates without error, then P' will also terminate without error and will yield the same value as P.

Note that the converse is not true: it is possible that P' may terminate without error, but that P will yield a run-time error. To see this, consider the following Gel function:

```
int Fun() {
 Foo ^f = new Foo();
 Foo g = f;
 f = null;
 return 4;
}
```

This function will yield a run-time error in Gel, since at the moment f is destroyed there is still a non-owning pointer g to the same object. But if we apply the erasure transformation to this function, then the resulting C# function will execute without error.

3. PROGRAMMING IN GEL

We have described the Gel language. We must now address two questions: how easy is it to use Gel's type system for writing real programs, and how well will Gel programs perform? We discuss usability in this section and performance in following sections.

3.1 Data structures

We believe that many programmers writing in languages such as C++ write code which frees objects based on ownership relationships between objects. For example, most C++ programmers have written destructors which free objects pointed to by the destroyed object. The popular scoped_ptr template class from the Boost ++ library implements the ownership relationship; when a scoped_ptr is destroyed the object it points to is freed automatically. In some sense, then, Gel takes the existing informal notion of object ownership and encodes it into the type system.

But how easy is it to find a single owner for every object in a program as Gel's type system requires? Fortunately many common data structures such as lists and trees have a natural hierarchical structure. Many programs contain hierarchies of objects linked in parent/child relationships; in these programs, pointers from parents to children can be owning, and backpointers from children to parents can be non-owning.

Some data structures are not naturally hierarchical and may in fact involve arbitrary pointer graphs. To represent an arbitrary graph, a Gel program can allocate all graph nodes from an array whose only purpose is to hold an owning pointer to each node; we call such an array a *pool*. The program can then use non-owning pointers between graph nodes arbitrarily. When the owning pool is destroyed, all graph nodes will be destroyed as well. (The Gel implementation, in fact, includes a built-in type `pool` which can be used for allocations of this sort; for details, see the Gel language reference.) Note that a C++ program implementing an arbitrary graph must also generally keep all graph nodes in a list in order to be able to free each node exactly once.

3.2 Circular data structures

It's possible for a Gel program to create cycles of owning pointers. The following program creates two `Bar` objects which point to each other:

```
class Bar {
 Bar ^next;
 static void MakeACycle() {
  Bar ^n = new Bar();
  Bar ^o = new Bar();
  Bar p = o;
  n.next = o;
  p.next = n;
 }
}
```

These objects will never be destroyed. Gel only destroys structures containing no owning pointer cycles; this is a weaker guarantee than that provided by a garbage collector, which can destroy structures containing arbitrary pointer cycles.

But this hardly matters in practical programming. It's hard to create an owning pointer cycle by mistake in Gel, because the type system guarantees that a programmer can't create an owning pointer cycle and return an owning pointer to it. For example, if we modify the MakeACycle method above to return a `Bar^` and add the line `return n;` at the end of the method, we receive a compile-time error: we can't return `n` because ownership has already transferred away from it (in the assignment to `p.next`).

All common data structures can be coded without using owning pointer cycles. An inherently circular structure such as a circular linked list is a special case of an arbitrary graph; as described above, the Gel programmer can allocate all list nodes using a separate owner object and then create the circular links using non-owning pointers. Alternatively, a programmer can intentionally create a cycle of owning pointers as in the `MakeACycle` method above, then break the the cycle manually when deciding to destroy the list.

3.3 Experience writing the compiler in Gel

We've only written one large program in Gel, namely the Gel compiler/interpreter itself. In writing the Gel compiler we found that most objects had natural owners; ownership works particularly well for trees of objects such as the abstract syntax tree generated by parsing. Certain objects' lifetimes were indefinite, and we generally chose to allocate such objects from pools as described above; we used non-owning pointers to such objects throughout the program.

The Gel implementation includes both a compiler and an interpreter. In implementing the interpreter, we chose to represent owning pointers in the interpreted language using owning pointers in the interpreter itself. This means that the interpreter has no code to free objects in the interpreted program explicitly; instead, objects in the interpreted program are freed automatically when no longer needed. (This is similar to implementing an interpreter for a garbage-collected language in a garbage-collected language, in which case the interpreter itself need not concern itself with garbage collection, which "flows through" the interpreter to the interpreted program.)

The interpreter contains numerous code paths which must manipulate values in the interpreted language which might either be owned or unowned pointers. To handle this situation, the interpreter code has a class `RValue` with two subclasses: `GValue`, which holds a value directly, and `Reference`, which holds a non-owning pointer to a `GValue`. Code which manipulates values in the interpreted language generally represents them using a variable of type `RValue ^`. If this underlying value is owning, the variable holds the underlying value directly; if it is not, the variable holds a `Reference` pointing to the value. This was slightly awkward. We hope that this particular coding difficulty was an artifact of implementing a Gel interpreter in Gel itself, and that most programs in Gel will not need to manipulate values whose ownership status is unknown at compile time.

4. REFERENCE COUNT OPTIMIZATION

As described above, Gel keeps a count of non-owning pointers to every object. Gel's reference counts are significantly cheaper than those in a traditional reference-counted system, for three reasons:

1. Gel doesn't need to keep reference counts for owning pointers; an object's reference count doesn't change as it is passed from owner to owner. In a traditional reference-counting system, a reference count is kept for every pointer to an object.
2. In Gel, a reference count decrement is simply an integer decrement, which may well be a single instruction. A traditional system must check whether a reference count is zero after each decrement; in Gel this is unnecessary because ownership, not the reference count, determines when it's time to free an object.
3. The Gel compiler can use ownership information to optimize away many reference counts at compile time in a single-threaded Gel program.

This last point deserves more explanation. The Gel compiler implements a reference count elimination optimization based on the following idea. Suppose that, for a type T, a given block of code never mutates any pointer of type U ^, where U is T, any

superclass of T, or any subclass of T. Then the code cannot possibly destroy any object which can be held in a variable of type T, and therefore any non-owning variable or temporary of type T whose lifetime is contained in the block of code need not hold a reference count.

As an example, consider the following loop, which iterates down a linked list, adding the integer values which are found in each node of the list:

```
Node ^first = construct();
int sum = 0;
for (Node n = first ; n != null ; n = n.next)
 sum += n.i;
```

A traditional reference-counted implementation would increment and then decrement the reference count of each node in the list as it is pointed to by the loop variable n. The Gel compiler can optimize these reference counts away: the variable n is live only during the loop execution, and no code in the loop mutates a owning pointer of type T ^, where T is `Node` or any superclass or subclass of it (indeed, the code mutates no owning pointers at all.) And so no `Node` object can possibly be destroyed during the loop.

The compiler implements this optimization interprocedurally. Suppose that we modify the loop body above to call `n.Get()`, where the `Get()` method merely returns the instance variable i. Then the optimization still holds, since the compiler recognizes that the `Get()` method also mutates no owning pointers of type `Node`.

To implement this optimization, the compiler generates a call graph of all methods in the program being compiled. Using this graph, the compiler computes, for each method, the set of owning types which may be mutated during a call to M. When compiling each method, the compiler generates a graph of control flow in the method. For each non-owning local variable (or temporary value in a expression), the compiler examines the control flow nodes where the variable is live and generates the set of owning types which may be mutated during the variable's lifetime, including types which may be mutated during method calls invoked from those nodes. If none of these types are a supertype or subtype of the variable's own type, the compiler does not bother to count references from the variable to any object.

5. BENCHMARK PERFORMANCE

To compare Gel's performance with that of languages using manual memory management and languages using garbage collection, we've implemented several small benchmark programs in C++, Java, C# and Gel. The source code to these benchmarks is available at the Gel project site. The benchmarks include the following:

- `sort`: on each of a series of iterations, generates a list of 1,000,000 randomly generated integers and sorts them using a recursive mergesort algorithm.
- `sortstring`: like the `sort` benchmark, but uses a list of 400,000 randomly generated strings.
- `binarytrees`: allocates and frees a large number of binary trees of various sizes. This program is from the Computer Language Shootout Benchmarks at http://shootout.alioth.debian.org/.

We have implemented the Gel compiler both in C# and in Gel, so we use the compiler itself as an additional, larger benchmark:

- `self-compile`: compiles a 73,000-line Gel program (consisting of 10 copies of the Gel compiler concatenated together, with classes renamed in each copy to avoid duplicate class definitions)

We've measured the performance of these benchmarks on a computer with a 1.83 Ghz Intel Core 2 Duo processor and 2 Gb of RAM, running Ubuntu Linux 6.10. The C++ benchmarks (and the C++ code output by the Gel compiler) were compiled using gcc 4.1.2 using the -O2 optimization level. We used the Sun JDK 1.5.0_08 to compile and run the Java benchmarks; we used the "server" (not "client") VM from the JDK. We compiled and ran the C# benchmarks using version 1.1.17.1 of Mono [22], an open-source implementation of C#. Note that Mono uses the Boehm garbage collector [21] .

For each benchmark run, we collected the following statistics:

- CPU time (including both user and system time) in seconds.
- Maximum virtual memory. This includes all pages mapped into the process's address space, including those not resident in physical memory, and including both private and shared pages.
- Maximum resident memory. This includes only pages resident in physical memory. (Both private and shared pages are included.)
- Maximum private writeable virtual memory. This includes only pages which are private to the process and are writeable. (Both resident and non-resident pages are included.)

We collected the virtual memory statistics using `ptime`, a program we wrote specifically for Linux; its source code is available on the project site.

Here are the benchmark results; a discussion follows below.

Benchmark: sort 5

	CPU (sec)	virtual (Mb)	resident (Mb)	private (Mb)
C++	7.1	18.3	16.5	15.9
Java	5.2	709.5	74.7	651.4
C#	3.3	42.3	27.4	30.4
Gel	4.8	18.3	16.6	15.9

Benchmark: sortstring 5

	CPU (sec)	virtual (Mb)	resident (Mb)	private (Mb)
C++	4.4	25.4	23.6	23.0
Java	6.5	708.0	127.8	650.8
C#	6.8	50.8	33.5	38.9
Gel	5.1	34.8	33.0	32.4

Benchmark: binarytrees 18

	CPU (sec)	virtual (Mb)	resident (Mb)	private (Mb)
C++	11.4	27.2	25.4	24.9
Java	3.6	707.4	92.6	650.2
C#	17.6	76.2	65.9	64.3
Gel	17.8	27.2	25.5	24.9

Benchmark: self-compile

	CPU (sec)	virtual (Mb)	resident (Mb)	private (Mb)
C#	5.3	80.2	67.4	66.5
Gel	5.6	68.7	66.8	65.6

Memory usage was remarkably consistent across the benchmarks. For every memory statistic in every benchmark, Java consumed the most memory, followed by C#, followed by Gel, followed by C++. In every benchmark except for `sortstring`, Gel's memory consumption was only slightly greater than C++'s. In the `sortstring` benchmark Gel used about 40% memory than C++; this is presumably because Gel represents a string using a structure containing a reference count and a character pointer, whereas the C++ version of the benchmark represents a strings using a character pointer alone. In this benchmark Gel nevertheless used significantly less virtual memory than C# and significantly less physical memory than Java.

CPU usage was not nearly so consistent across benchmarks. In two benchmarks C++ was faster than both Java and C#; in one benchmark both Java and C# were faster; and in one benchmark (`binary-trees`) Java was, surprisingly, over twice as fast as C++, but C# was much slower. Gel's performance relative to C++ also varied from benchmark to benchmark. In sortstring Gel was about 15% slower; in `binary-trees` Gel was about 70% slower. In the `sort` benchmark Gel was, oddly, faster than C++; this is surprising because the Gel compiler generates C++ code. Perhaps the code generated by the Gel compiler triggers some optimization which doesn't occur when the C++ version is compiled; we hope to have an explanation for this by the time the final version of this paper is published.

5.1.1 Effectiveness of reference count elimination

To measure the effect of Gel's reference count optimizer, we instrumented our benchmark programs to report the number of reference count increments which occur in each benchmark run. The benchmark results follow:

Effect of reference count optimization

benchmark	with optimization		without optimization	
	cpu (sec)	incr ops	cpu (sec)	incr ops
`sort`	4.8	93.4M	5.9	868.2M
`sortstring`	5.1	34.7M	5.7	326.6M
`binarytrees`	17.8	0	20.2	471.1M
`self-compile`	5.6	13.0M	6.9	225.7M

In these benchmarks the optimizer eliminates 89% to 100% of reference count increments, speeding execution by 10% to 35%. Interestingly, in the `binarytrees` benchmark the optimizer is able to eliminate every reference count in the program.

6. PROPOSED EXTENSIONS

In this section we sketch a number of possible extensions to the Gel language. We have not implemented any of these.

6.1 Object Destruction

Gel's object destruction mechanism as outlined above has a number of limitations:

- Gel does not let the programmer define a *destructor* for a class, i.e. a method which will run when an instance of the class is destroyed.
- Gel can automatically destroy structures in which each internal non-owning pointer points to an ancestor in the ownership tree, but cannot automatically destroy structures containing arbitrary non-owning pointers to internal nodes.
- Object destruction in Gel may consume a large amount of stack space since it is inherently recursive.

To overcome these limitations, we propose adding destructors to Gel, and propose a multi-phase object destruction mechanism. Gel could destroy an object graph rooted at a pointer P in the following phases.

1. Recursively descend all objects in the ownership tree below P, running each object's destructor.
2. Recursively descend again; for each non-owning pointer in every object, decrement the reference count of the object pointed to.
3. Recursively descend again; for each object, check that its reference count is zero and free the object.

In many cases the compiler should be able to optimize away some or all of the work to be performed in the first and/or second phases. In particular, using static type analysis it will often be possible to determine that all objects in the graph below an object of type T will never have any destructors or non-owning pointers; in that case the first or second phases need not run at all below such objects. As a simple example, consider a singly linked list in which each node contains only an integer and an owning pointer to the next node; then the first and second phases can be optimized away entirely in destroying a list of this type.

Note also that in many structures each node contains only a single owning pointer; in such structures, the compiler can save stack space by eliminating tail recursion in generating the code to perform the recursive descent in each phase. With these optimizations in place, we believe that this generalized destruction algorithm might outperform the algorithm in the existing Gel implementation in many cases.

User-written destructors might potentially mutate the graph of objects being destroyed. If a destructor inserts objects into the graph, then these objects might be destroyed in phases 2 and 3 even though their destructors have never run. Similarly, if a destructor removes objects whose destructors have already run, then those objects' destructors might run again when they are destroyed at a later time. This possibility does not compromise the safety of the language, but does mean that each object's destructor is not guaranteed to run exactly once. It might be possible to detect destructor-induced mutations in some way and report an

error if they occur; we will not discuss this possibility further here.

6.2 Polymorphism

The Gel type system does not support any sort of polymorphism between owning and non-owning types. This means that it's not possible to write a method which can receive either an owning or non-owning pointer as an input parameter. It's also not possible to implement a single data structure (such as a tree or hash table) which can hold either owning or non-owning pointer. This limitation is reflected in the Gel class library, which contains separate `ArrayList` and `NonOwningArrayList` classes.

It would be useful to extend Gel with generic types as found in Java and C#, allowing a generic type to represent either an owning or non-owning type. As a simple example, suppose that we'd like to implement a class `Pair<T>` which holds two objects of type T. We'd like to be able to instantiate the class using either an owning or non-owning pointer type. Our generic type system might allow this as follows:

```
class Pair<T> {
 T first, second;
 Pair(T first, T second) {
  this.first = first;
  this.second = second;
 }
 T% First() { return first; }
 T% Second() { return second; }
}
```

The system includes a type operator % which can be applied only to a generic type, and which removes ownership from a type. In particular, if T is Foo ^, then T% is Foo; if T is Foo, then T% is also Foo.

A generic type system for Gel should also support *covariance*: it should be possible to write a single function which can take either a `Pair<Foo>` or a `Pair<Foo ^>` as a parameter, and similarly it should be possible to write a single function which can take either a `Foo[]` or a `Foo^[]` as a parameter.

We believe that it should not be difficult to design a complete generics system for Gel including covariance, but we will not explore the idea further in this work.

6.3 Multithreading

As described above, the Gel compiler's reference count elimination optimization will work only in single-threaded programs; this is a significant limitation.

It would be useful to extend Gel to be able to execute multithreaded programs safely and efficiently. We can imagine two different possible approaches here. First,it might be possible to extend Gel's optimizer to be able to infer that many data structures are accessed only from a single thread; then reference counts in such structures could be optimized away using the existing algorithm. Such inference seems difficult and we are relatively pessimistic about such an approach.

As another approach, we could possibly extend Gel so that objects are grouped into *apartments* which may be accessed from only a single thread at once. With this approach, the existing optimization algorithm could be used to eliminate reference count accesses in code which executes inside a single apartment. It might be possible to design system which guarantees that only pointers of a certain kind may cross apartment boundaries and which allows groups of objects to move efficiently between apartments; we leave this to further work.

7. RELATED WORK

Our work represents one of many attempts to find a "sweet spot" in the trade-off space between fully manual and fully automatic memory management (that is, garbage collection). There are many inter-related issues, chief among them being memory footprint, run-time cost, determinism, and ease of programming. We compare our work to that of others across these various axes.

Region-based memory management [15] provides bulk-freeing of objects, often based on a stack discipline. Regions generally provide low cost and good determinism, but if provided in a safe manner either require either burdensome annotation [19] or an analysis which is frequently over-conservative [2]. Region systems that follow a stack discipline often suffer from large numbers of object being assigned (by analysis or by the programmer) to the global region which is never collected (or must be collected by a supplementary garbage collector).

The Real-time Specification for Java [3] provides a stack-oriented region-based memory management based on *scopes*. Scopes have neither annotations nor automatic inference; instead it is the responsibility of the programmer to allocate objects in the correct scope. Storing a pointer that violates the stack discipline of scopes causes a run-time exception. Scopes are thus type-safe but subject to run-time exceptions that are difficult to prevent and are highly dependent on interactions between different components. In addition, it is often either convenient or necessary to place objects in the outermost scope (called Immortal memory), in particular when it is necessary to share objects between scope-managed and garbage-collected memory.

Gay and Aiken [12] suggest the use of reference counted regions that need not follow a stack discipline. Safety is ensured because explicit freeing is not allowed to succeed unless the region's reference count has dropped to zero. They include annotations for specifying that pointers are in the same region or are pointers to parent regions; updates of such pointers do not change reference counts. Although the granularity and abstraction level are different from ours, these annotations also allow them to optimize away a large fraction of reference counting operations.

The goals of the Cyclone language [17] are probably most similar to those of Gel; its goal is to provide a safe alternative to C for systems programming. Cyclone provides several forms of safe memory management: regions, unique pointers, and reference counting. Cyclone's regions require explicit annotations in the source code, and can be both lexically scoped (in the style of [Tofte 1]) or dynamically scoped. In the latter case, if a region is freed dynamically subsequent attempts to access objects in that region will generate a run-time exception.

Cyclone's unique pointers are based on linear types [13] and are essentially like Gel's owning pointers, but without the capability to reference count non-owning references. Some polymorphism across memory management styles is provided by allowing sets of unique pointers to be temporarily treated as though they belong to a region. Cyclone also provides explicit (but safe) reference counting; reference counted objects are treated as belonging to a special region called RC.

The experience of the language designers in writing the Cyclone compiler is instructive: first coarse-grained regions were used, which reduced the annotation burden. However, coarse regions did not free memory soon enough and led to excessive space consumption. The compiler was then rewritten using fine-grained regions, at which point the annotations became overly cumbersome. Eventually the compiler was rewritten to use garbage collection [Greg Morrisett, personal communication].

Ownership types have received considerable attention in recent years. Unlike Gel's concept of ownership, these approaches rely entirely on the type system [5] [7] [9]. Ownership is used for a variety of purposes: modularity [8], avoidance of run-time errors, improved alias analysis and optimization [9], prevention of data races and deadlocks [4], and real-time memory management [6] [20]. Generally speaking, these approaches have a high annotation overhead and/or require significant redundant code due to the coupling of types to ownership contexts. Clarke and Wrigstad [10] also point out that ownership types often require external interface changes in response to purely internal changes in data structures; they attempted to alleviate this problem using a combination of unique pointers and ownership types, but the resulting language still suffers a high annotation burden.

Fahndrich and DeLine [11] suggest adoption and focus as constructs for incorporating linear types with an abiity to temporarily create and relinquish aliases, and successfully applied it to the writing of low-level systems code. This provides very strong static information, but requires significant and not always intuitive annotation.

Real-time garbage collection [1] [16] provides overall timing determinism and ease of programming, but suffers from the memory overhead issues associated with garbage collection in general, and does not provide determinism in terms of the times at which individual objects are freed. Also, during garbage collection some fraction of the CPU time (on the order of 30%) is unavailable to real-time tasks.

8. ACKNOWLEDGEMENTS

We'd like to thank Linus Upson for his invaluable feedback and encouragement. Discussions with Paul Haahr were also useful. Feng Qian ported the Gel compiler from Windows to Linux. Many thanks are due to Google for allowing us to release the Gel compiler as open source.

9. REFERENCES

[1] Bacon, D. F., Cheng, P., and Rajan, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA, January 15 - 17, 2003). ACM Press, 285-298.

[2] Birkedal, L., Tofte, M., and Vejlstrup, M. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, United States, January 21 - 24, 1996). ACM Press, 171-183.

[3] Bollella, G., Gosling, J., Brosgol, B. M., Dibble, P., Furr, S., Hardin, D., and Turnbull, M. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[4] Boyapati, C., Lee, R., and Rinard, M. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Seattle, Washington, USA, November 04 - 08, 2002). ACM Press, 211-230.

[5] Boyapati, C., Liskov, B., and Shrira, L. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA, January 15 - 17, 2003). ACM Press, 213-223.

[6] Boyapati, C., Salcianu, A., Beebee, W., and Rinard, M. Ownership types for safe region-based memory management in real-time Java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA, June 09 - 11, 2003). ACM Press, 324-337.

[7] Boyland, J. Alias burying: unique variables without destructive reads. *Softw. Pract. Exper.* 31, 6 (May. 2001), 533-553.

[8] Clarke, D. G., Noble, J., and Potter, J. Simple Ownership Types for Object Containment. In *Proceedings of the 15th European Conference on Object-Oriented Programming* (June 18 - 22, 2001). Springer-Verlag, 53-76.

[9] Clarke, D. G., Potter, J. M., and Noble, J. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Canada, October 18 - 22, 1998). ACM Press, 48-64.

[10] Clarke, D., and Wrigstad, T. External uniqueness is unique enough. In *European Conference on Object-Oriented Programming* (ECOOP). , 176–200.

[11] Fahndrich, M. and DeLine, R. Adoption and focus: practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany, June 17 - 19, 2002). ACM Press, 13-24.

[12] Gay, D. and Aiken, A. Language support for regions. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, United States). ACM Press, 70-80.

[13] Girard, J. Linear logic. *Theoretical Computer Science* 50:1 (1987), 1-102.

[14] Hejlsberg, A., Wiltamuth, S., and Golde, P. *The C# Programming Language*. Addison-Wesley, 2004.

[15] Henglein, F., Makholm, H., and Niss, H. Effect types and region-based memory management. In Pierce, B. ed. *Advanced Topics in Types and Programming Languages*, MIT Press, 2005, 87-136.

[16] Henriksson, R. *Scheduling Garbage Collection in Embedded Systems*. Ph.D. Thesis, Department of Computer Science, Lund University, September 1998.

[17] Hicks, M., Morrisett, G., Grossman, D., and Jim, T. Experience with safe manual memory-management in Cyclone. In *Proceedings of the 4th international Symposium on Memory Management* (Vancouver, BC, Canada, October 24 - 25, 2004). ACM Press, 73-84.

[18] Jones, R., and Lins, R. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[19] Tofte, M. and Talpin, J. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, United States, January 16 - 19, 1994). ACM Press, 188-201.

[20] Zhao, T., Noble, J., and Vitek, J. Scoped Types for Real-Time Java. In *Proceedings of the 25th IEEE international Real-Time Systems Symposium (Rtss'04) - Volume 00* (December 05 - 08, 2004). IEEE Computer Society, 241-251.

[21] http://www.hpl.hp.com/personal/Hans_Boehm/gc/

[22] http://www.mono-project.com/

[23] http://code.google.com/p/gel2/source

[24] http://gel2.googlecode.com/svn/trunk/doc/gel2_language.html

10. APPENDIX: SAMPLE PROGRAM

The following Gel program generates a list of 1,000,000 random integers, sorts them using a recursive mergesort and then verifies that the resulting list is in sorted order.

```
class Random {
 static int r = 1;
 public static int Next() {
  r = r * 69069;
  return r;
 }
}

class Node {
 public readonly int i;
 public Node ^next;
 public Node(int i, Node ^next) {
  this.i = i;
  this.next = next;
 }
}

class Sort {
 static Node ^RandomList(int count) {
  Node ^first = null;
  for (int x = 0 ; x < count ; ++x)
   first = new Node(Random.Next(), first);
  return first;
 }

 static Node ^Merge(Node ^a, Node ^b) {
  Node ^head = null;
  Node tail = null;
  while (a != null && b != null) {
   Node ^top;
   if (a.i < b.i) {
    top = a;
    a = take top.next;
   } else {
    top = b;
    b = take top.next;
   }
  if (head == null) {
   head = top;
   tail = head;
  } else {
   tail.next = top;
   tail = tail.next;
  }
 }
  Node ^rest = (a == null) ? b : a;
  if (tail == null)
   return rest;
  tail.next = rest;
  return head;
 }

 static Node ^MergeSort(Node ^a) {
  if (a == null || a.next == null) return a;
  Node c = a;
  Node b = c.next;
  while (b != null && b.next != null) {
   c = c.next;
   b = b.next.next;
  }
  Node ^d = take c.next;
  return Merge(MergeSort(a), MergeSort(d));
 }

 public static void Main(String[] args) {
  Node ^n = RandomList(1000000);
  n = MergeSort(n);
  while (n != null) {
   Node next = n.next;
   if (next != null && n.i > next.i) {
    Console.WriteLine("failed");
    return;
   }
   n = take n.next;
  }
  Console.WriteLine("succeeded");
 }
}
```