Swansea University
Prifysgol Abertawe

SWANSEA UNIVERSITY

REPORT SERIES

Updating Objective-C

by

David Chisnall

Report # CSR 14-2008

Updating Objective-C

David Chisnall

July 21, 2008

Abstract

The Objective-C language is over two decades old. Since its creation, numerous advances in the area of object oriented programming have been made. This report presents implementations of several of these ideas in Objective-C, including traits, mixins, futures and prototype-based object orientation. This serves to highlights how original design decisions in the language have allowed new concepts to be added and how others have limited the flexibility of the language.

1 Background

Objective-C [3] was created as a set of Smalltalk-like extensions to the C language to provide support for component-based development. It began life as the Object Oriented Pre-Compiler [4]. At the time of its creation, it aimed to provide most of the flexibility of Smalltalk without sacrificing the speed of C. The largest semantic difference between Objective-C and Smalltalk is the fact that Objective-C uses control structures from C, while Smalltalk implements them in terms of message passing. This made certain classes of optimisation easier in Objective-C than Smalltalk, since flow control could be determined more easily at compile time at the cost of some flexibility. This, in turn, allowed Objective-C to be used on machines less powerful than the workstations required for a Smalltalk environment.

As with Smalltalk, the designers of Objective-C attempted to move as many features as possible into the library and out of the language. This allows users of the language to extend it in ways which were not necessarily obvious at the time the original design decisions were made. This report describes a number of improvements which have recently been made to Objective-C, demonstrating the flexibility and limitations in the original design decisions. All of the additions described here are implemented solely as library routines and the source code is available.

2 Mixins and Traits

Objective-C uses a Smalltalk-like object model based on classes. Methods can be added to classes in two ways, either in the class definition or via a *category*. Categories are simple collections of methods that can be loaded at runtime and composed with those already defined.

The composition of categories with classes is very similar to that of traits [7]. As with traits, an category may not declare any state (instance variables). Unlike traits, it may access them directly as long as they are declared in the class. This is due to the fact that categories may only be applied to a single class, while traits may be inserted at different points as an alternative to multiple inheritance.

We add typesafe mixins and traits to the language. Both are implemented the same way but use a different composition mechanism. Both mixins and traits are declared in the source code as classes, however they are intended to be combined with other classes rather than used directly.

We relax the constraint that traits may not access state directly and replace it with the following two constraints on classes being used in this way:

- Any instance variables declared in the composed

class must be present at the same offset and with the same type (although not necessarily the same name) in the class with which it is being composed.

- Any methods declared in the composed class must either not exist in the class with which it is being composed and if they exist in a superclass must have the same type signature.

Allowing direct access to instance variables is more important in Objective-C than other dynamic languages, since it is statically compiled in current implementations. Although method calls are performed via dynamic dispatch, instance variable accesses are performed using a static lookup. The layout of an object is static and direct accesses to instance variables is performed simply by adding a precalculated offset to the object's address. This is significantly faster than going via a dynamic mechanism, at the expense of flexibility. A language like Smalltalk, which is dynamically compiled at runtime, does not have this limitation; accessor methods can be inlined trivially by the runtime.

The mixin and traits support consists of two components. The first performs run-time type checking, ensuring that instance variable and method types match. This is done by introspecting the runtime structures representing the class. The second manipulates these structures and performs the method composition.

In the case of mixins, the composition is performed via the standard inheritance mechanism. To accomplish this, a new class structure is inserted into the class's inheritance chain. This is effectively a shallow copy of the mixed-in class. The method list from the two classes are then exchanged and the dispatch tables updated. This ensures that methods in the mixed-in class will replace those in the receiver.

For traits, the method list in the receiving class is modified to include the methods from the composed class.

For completeness, a `flattenMixin:` method is also provided. This uses traits-style composition but with mixin composition rules, allowing methods to be replaced in the receiving class. Listing 8 shows the `NSObject` category providing support for mixins. This depends on a number of helper functions which wrap the runtime library manipulations and can be found in Listing **??** in the appendix.

The implementation is inspired by the work on Strongtalk [1] where mixins were the fundamental unit of composition and classes were treated as a special case of mixins. Since Objective-C has a concept of classes both as a language construct and an implementation construct already, this idea is reversed and mixins (and traits) are treated as a special case of classes.

Listing 1: Implementation of traits and mixins.

```
+ (void) mixInClass:(Class)aClass
{
  Class class = (Class)self;
  checkSafeComposition(class, aClass);
  Class newSuper = calloc(1,sizeof(struct
       objc_class));
  /* Move ivar and method definitions to the new
       superclass */
  newSuper->ivars = class->ivars; class->ivars =
       NULL;
  newSuper->methods = class->methods;
       class->methods = aClass->methods;
  newSuper->instance_size = class->instance_size;
  /* Insert into the class hierarchy */
  newSuper->super_class = class->super_class;
  class->super_class = newSuper;
  newSuper->dtable = sarray_new (200, 0);
  __objc_update_dispatch_table_for_class(newSuper);
  __objc_update_dispatch_table_for_class(class);
}
+ (void) applyTraitsFromClass:(Class)aClass
{
  Class class = (Class)self;
  checkSafeComposition(class, aClass);
  struct objc_method_list * methods =
       aClass->methods;
  while(methods != NULL)
  {
    // Check that the method doesn't exist in this
         class
    for(unsigned int i=0 ; i<methods->method_count
         ; i++)
    {
      Method_t method = &methods->method_list[i];
      if(findMethod((char*)sel_get_name(method->method_name),
           class, NO) != NULL)
      {
        [NSException
            raise:@"TraitMethodExistsException"
              format:@"Methods class %@ redefined
                   in %@.", self, aClass];
      }
    }
    methods = methods->method_next;
  }
  methods = aClass->methods;
  /* Add all of the methods from the class to the
```

```
        mixin */
    while(methods != NULL)
    {
      addMethods(class, methods);
      methods = methods->method_next;
    }
    __objc_update_dispatch_table_for_class(class);
}
+ (void) flattenedMixinFromClass:(Class)aClass
{
    Class class = (Class)self;
    checkSafeComposition(class, aClass);
    struct objc_method_list * methods =
        aClass->methods;
    /* Add all of the methods from the class to the
        mixin */
    while(methods != NULL)
    {
      addMethods(class, methods);
      methods = methods->method_next;
    }
    /* Update the dispatch table so the runtime knows
         about these methods */
    __objc_update_dispatch_table_for_class(class);
}
@end
```

3 Prototypes

Prototypes remove the special status of classes from an object oriented language and allow methods and instance variables to be added to any object, rather than to just classes. It has been proposed that prototypes are beneficial for user interface code ([8]) and this claim is supported by the fact that the JavaScript language used for client-side code in web applications uses a prototype-based object model.

Implementing prototypes in Objective-C represents an interesting problem. A full implementation of prototypes requires the ability to add both methods and instance variables to an object at runtime.

Adding methods presents a practical problem. Adding instance variables presents a theoretical problem due to the way in which they are implemented in the language. Objective-C objects are simply C structures with their first field pointing to a structure representing the class. As such, they have a fixed layout.

When bridging Io with Objective-C, the solution was to 'hide' a pointer dictionary object in front of the Objective-C object and expose this through some special accessors. In particular, the idea of Key-Value Coding is a common idiom in Objective-C. This specifies a single accessor method (`valueForKey:`) that will either return the contents of an instance variable or the result of calling a specific accessor method depending on the implementation. Forcing instance variables to go via this mechanism allows the corresponding set method to insert values into the hidden dictionary if they do not correspond to an existing instance variable.

This mechanism was implemented. The KVC mechanism as implemented in the NSObject base class provides a second-chance mechanism for both setting and getting values for undefined keys. This was used to provide a transparent mechanism for adding and removing instance variables.

Adding methods is conceptually simpler, since the address of method implementations is looked up at runtime. Unfortunately, the runtime function which performs this lookup does so by inspecting the class structure directly and does not provide a good mechanism for overriding this behaviour.

As with other dynamic languages, Objective-C does provide a second-chance dispatch mechanism. If no method is found then a second-chance method will be called. The default implementation of this then queries the object for the type signature of the method and wraps up the arguments and selector up in an invocation object.

It is possible to implement message dispatch atop this mechanism, but it is undesirable. The overhead of dispatching messages in this way is approximately 300 times that of the standard mechanism. Another problem is that it is impossible to use this for methods that are already implemented. An object that inherits a method implementation from its prototype or class can not make use of this mechanism. Doing so would require every prototype object to be wrapped in a proxy implementing no methods other than those related to forwarding, adding yet more overhead.

To avoid this, the runtime library was modified. Since the runtime library interfaces are not specified by the language, and the changes did not break the application binary interface (ABI), we do not regard this as a change to the language. Each class structure contains a flags field. One of the unused flags was used to indicate a secondary lookup mechanism. Setting this flag will cause the runtime to call the

+messageLookupForObject:selector: method before performing the standard lookup. If this returns a non-nil value then it will be treated as the implementation address.

The ETPrototype class uses both of these mechanisms. It includes two instance variables, a map table[1] for extra instance variables and another for methods.

The code for this class is shown in Listing 2. The `ETPrototype_t` type is the prototype object represented as a C structure. This is used to allow modification of instance variables that are otherwise inaccessible.

Prototypes implemented in this way also allow a form of delegation orthogonal to that of class-based inheritance. This is implemented by having an instance variable called `prototype` which is used for recursive lookups. This can be used in the same way as `super` for calling superclass methods.

We implement differential inheritance by keeping a reference to the prototype object (preventing it from being destroyed) and using it for delegated lookups. We can not use arbitrary objects as prototypes with this implementation. Another approach would be to store the dispatch tables in an external map, however this would require two extra map lookups for every single message send. The described implementation allows prototypes to be used only in classes where they make sense, avoiding the extra overhead in the general case.

Listing 2: Prototype object implementation.

```
#import "ETPrototype.h"
#include <objc/objc-api.h>
#include <objc/objc.h>

typedef struct { @defs(ETPrototype) }*
    ETPrototype_t;
@implementation ETPrototype
- (id) init
{
  if(nil == (self = [super init]))
  {
    return nil;
  }
  dtable = NSCreateMapTable(
      NSNonOwnedPointerMapKeyCallBacks,
      NSNonOwnedPointerMapValueCallBacks,
      5);
  return self;
}
/**
 * Set this class as having a custom method lookup
     mechanism.
 */
+ (void) initialize
{
  CLS_SETOBJECTMESSAGEDISPATCH((Class)self);
  [super initialize];
}
/**
 * Message lookup function.  Looks for method in
     the dtable ivar.
 */
+ (IMP) messageLookupForObject:(id)anObject
    selector:(SEL)aSelector
{
  ETPrototype_t ivars = (ETPrototype_t) anObject;
  /* Avoid map lookup if there are no methods added
       yet. */
  if(ivars->isPrototype)
  {
    IMP method = (IMP)NSMapGet(ivars->dtable,
        sel_get_name(aSelector));
    if(method == NULL && (anObject->prototype) !=
        NULL)
    {
      return [self
          messageLookupForObject:(anObject->prototype)
          selector:aSelector];
    }
    return method;
  }
  return NULL;
}
/**
 * Add a new method.
 */
- (void) setMethod:(IMP)aMethod
    forSelector:(SEL)aSelector
{
  isPrototype = YES;
  NSMapInsert(dtable, sel_get_name(aSelector),
      (void*)aMethod);
}
- (id)copyWithZone:(NSZone *)zone
{
  ETPrototype * copy = (ETPrototype*)
      NSCopyObject(self, 0, zone);
  ((ETPrototype_t)copy)->dtable =
      NSCopyMapTableWithZone(dtable, zone);
  ((ETPrototype_t)copy)->otherIvars =
      NSCopyMapTableWithZone(otherIvars, zone);
  return copy;
}
- (id) clone
{
  id obj = [[[self class] alloc] init];
  ETPrototype_t ivars = (ETPrototype_t) obj;
  ivars->prototype = self;
  return obj;
}
- (void)setValue:(id)value
    forUndefinedKey:(NSString *)key
{
  if(otherIvars == NULL)
  {
    otherIvars = NSCreateMapTable(
        NSObjectMapKeyCallBacks,
```

[1]For speed, a C implementation of an associative array is used rather than a dictionary object.

```
        NSObjectMapValueCallBacks,
        5);
  }
  NSMapInsert(otherIvars, key, value);
}
- (id)valueForUndefinedKey:(NSString *)key
{
  ETPrototype_t ivars = (ETPrototype_t) anObject;
  if(otherIvars == NULL)
  {
    return NULL;
  }
  id val = NSMapGet(otherIvars, key);
  if(val == NULL && (anObject->prototype) != NULL)
  {
    return [(anObject->prototype)
        valueForUndefinedKey:aSelector];
  }
  return NULL;
}
- (void) dealloc
{
  NSFreeMapTable(dtable);
  NSFreeMapTable(otherIvars);
  [super dealloc];
}
@end
```

4 Better Exception Handling

Exceptions are not part of the Objective-C language. They are traditionally implemented using a combination of the `setjmp` and `longjmp` calls, a custom class and some C preprocessor macros.

The down side of using `jongjmp` is that you need to save the CPU state at the point where the exception handler is defined, which is a relatively expensive operation. Newer implementations use the host ABI's zero-cost exception system where throwing an exception is expensive but unused exception handlers have a near-zero cost.

Objective-C exceptions, as defined by OpenStep, are very similar to Java exceptions. A block of code is wrapped in macros as shown in Listing 3. If an exception is raised then the stack is unwound to this point and the handler run.

Listing 3: OpenStep exception macros.

```
NS_DURING
  /* Code that may thrown a exception */
NS_HANDLER
  /* Code run if an exception is raised */
NS_ENDHANDLER
```

To improve this mechanism, we are inspired by both Lisp exceptions and CPU traps. CPU traps are divided into three broad categories. *Faults* must be handled before the current instruction can complete, *traps* must be handled before the next instruction can complete and *aborts* can can typically not be handled at all.

We use this as a model for defining a new exception handling regimen. First, we allow handlers to be registered for any named exception, rather than a single handler for all exceptions. These handlers may, at runtime, define one of three behaviours. The classic exception handling behaviour (stack unwinding and jumping to an error handler) can be invoked, in a way analogous to an abort. The stack can be unwound and the block which caused the exception can be retried, in a manner similar to a fault. Finally, the code causing the exception can be resumed, as with a trap.

Each of these can be useful in different situations. An exception handler may be able to fix the problem which caused the exception as it runs, in which case the correct behaviour is to resume. Alternatively, it may alter global conditions in such a way that the exception would not have been raised, in which case the correct behaviour is to restart the block in which the exception was raised.

Objective-C lacks closures as a language construct, however a similar mechanism is available in the form of nested functions as a GCC extension. Unlike true blocks, nested functions that reference local variables should never be used outside the scope in which they are defined. Passing a pointer to a nested function up the stack has undefined behaviour.

To avoid this happening, a macro is provided that uses another GCC extension, the cleanup attribute on a local variable is used. As soon as the scope in which the `SET_HANDLER` macro is used becomes invalid, the cleanup routine will be called and the extra code will

Listing 4: Using the new exception handling mechanism.

```
SET_HANDLER(@"RESUMABLE_EXCEPTION", resume);
SET_HANDLER(@"NORMAL_EXCEPTION", ex_abort);
int pass = 0;

ETExceptionType retry_nested(NSException *
    exception)
{
```

```
  pass++;
  return retry(exception);
}
SET_HANDLER(@"RESTARTABLE_EXCEPTION",
    retry_nested);
NS_RESTARTABLE_DURING
  NSLog(@"Entering block.");
  [[NSException
      exceptionWithName:@"RESUMABLE_EXCEPTION"
                         reason:nil
                       userInfo:nil] raise];
  NSLog(@"Exception point passed.");
  if(pass == 0)
  {
    [[NSException
        exceptionWithName:@"RESTARTABLE_EXCEPTION"
                           reason:nil
                         userInfo:nil] raise];
  }
  [[NSException
      exceptionWithName:@"NORMAL_EXCEPTION"
                         reason:nil
                       userInfo:nil] raise];
  NSLog(@"Never reached");
NS_RESTARTABLE_HANDLER
NS_ENDHANDLER
```

Listing 4 shows an example of how the new code is used. Each of the `resume`, `ex_abort` and `retry` functions simply log the exception they are passed and return the correct value. When run, this produces the following output:

```
Entering block.
Resuming exception: RESUMABLE_EXCEPTION
Exception point passed.
Retrying exception: RESTARTABLE_EXCEPTION
Entering block.
Resuming exception: RESUMABLE_EXCEPTION
Exception point passed.
Aborting exception: NORMAL_EXCEPTION
```

Note how the resumable exception does not cause any interruption in program flow, and so is called twice. The restartable exception causes the flow of execution to jump to the start of the exception block, but since the handler alters some of the function's state preventing it being raised a second time, it is only called once. Finally, the normal exception causes the standard exception handling code to be invoked.

Although all of these exceptions are raised in the same function (or method) as the handling block, they could be raised anywhere further down the stack without altering the behaviour.

The resumable exception defined in this are very similar in nature to those proposed by Brad Cox [2].

5 Futures

The final feature added to the Objective-C language was an implementation of futures. It is often useful to perform a large amount of work asynchronously.

This is commonly done by spawning a worker thread. The cross-platform mechanism for doing this is using POSIX threads, which have a procedural interface. These can often be wrapped up in an object-oriented interface.

The object oriented model, that of simple computers communicating via message passing, fits well with concurrency. Passing messages is a good way of communicating between threads. In Smalltalk-like languages, including Objective-C, it is common for methods (the code executed as the result of a message passing) to return a value synchronously.

To address this, we borrow an idea from the functional programming world, that of futures. In functional programming, futures are tied to the concept of lazy evaluation. When a function is called, it returns immediately in this model. Only when the result is used is the function actually executed.

In object oriented programming, this can not be used directly since message passing often has side-effects. Instead, we place the object in a separate thread and pass messages asynchronously. There are three different cases for returns in Objective-C that must be treated differently:

- Void returns.
- Intrinsic (non-object) returns.
- Object returns.

The first case is trivial. Nothing is returned so no handling is required. The second case is the hardest, and this is handled simply by blocking the caller until it returns. In a system which supported Mondrian memory management, it would be possible to mark the area of the stack to which the return value would be written as read-only and block the caller when it attempted to access it. Currently, however, no platforms with an OpenStep implementation support this fine-grained memory control.

The final case is the most interesting since we can properly implement the futures in this case. Rather than blocking when the message is sent, we block when the result is used. When a message is sent to an object in another thread, it goes via a proxy. The proxy returns another proxy object, the future, immediately. This proxy simply blocks when any message is sent to it until the real return value is available.

Passing messages between the threads needs to be handled efficiently. The simplest way of doing this would be via mutexes and condition variables, however this involves a large number of system calls which have a significant overhead. Instead, we use a hybrid ring buffer approach based on the work of Keir Fraser. This switches between a locked and lockless mode depending on the volume of messages.

When the message queue is empty, the code switches to using the mutex/condition variables to wait. When there are messages in the queue then it uses the very low-overhead lockless model.

The lockless ring buffer uses free-running counters as producer and consumer indexes. This allows a simple test of $producer - consumer$ to give the space in the ring buffer. After the sender has inserted a request into the ring, it tests whether it is only message. If it is, then there might have been a transition from empty to full and so it signals the condition variable. If not, then it skips the signal. Similarly, the object's thread only waits on the condition variable if the buffer is empty.

A form of futures was presented in [11], however this made extensive use of high-overhead locking and only allowed a single message to be queued. The implementation presented here allows multiple messages to be queued awaiting execution and has very low overheads for transferring messages between threads.

Testing has shown that the cost of using the second-chance message dispatch mechanism, which creates an `NSInvocation` object from the invocation, is approximately three hundred times that of performing a direct dispatch. For this reason, futures are only useful in cases where the overhead of sending the message is a very small fraction of the total cost of execution. It is ideal as a way of interacting with long-running worker threads.

Listing 5 shows how futures can be used. The new constructor, `+threadedNew`, spawns a new thread containing an object and returns a proxy used to insert objects into its message queue and return a proxy representing the future.

Listing 5: Using futures.

```
id proxy = [ThreadTest threadedNew];
[proxy log:@"1) Logging in another thread"];
NSString * foo = [proxy getFoo];
NSLog(@"2) [proxy getFoo] called.  Attempting to
    capitalize the return...");
NSLog(@"3) [proxy getFoo] is capitalized as %@",
    [foo capitalizedString]);
```

The `ThreadTest` object in this example implements a two methods, `log:`, which sleeps for two seconds and then logs the argument and `getFoo` which returns a constant string "foo". The output from this will be:

```
2) [proxy getFoo] called.  Attempting
to capitalize the return...
1) Logging in another thread
3) [proxy getFoo] is capitalized as FOO
```

When `-log:` is called, the method returns immediately. The message is added to the object's queue, begins executing immediately, and then sleeps for two seconds in the second thread. Meanwhile, the main thread logs message 1 and then sends a `capitalisedString` message to the return value of `getFoo`. Because this return value is a future, sending messages to it will block until the method has executed. Since methods in the second thread are queued, and dispatched in-order, the `log:` method will always complete before the return value from `getFoo` is available.

Listing 6: Forwarding messages between threads.

```
- (void)forwardInvocation:(NSInvocation
    *)anInvocation
{
  BOOL concreteType = NO;
  int rc = [anInvocation retainCount];
  if(![anInvocation argumentsRetained])
  {
    [anInvocation retainArguments];
  }
  ETThreadProxyReturn * retVal = nil;
  char returnType = [[anInvocation methodSignature]
       methodReturnType][0];
  if(returnType == '@')
  {
    retVal = [[[ETThreadProxyReturn alloc] init]
        autorelease];
```

```
    proxy = retVal;
    SEL selector = [anInvocation selector];
    [anInvocation
         setSelector:@selector(returnProxy)];
    [anInvocation invokeWithTarget:self];
    [anInvocation setSelector:selector];
  }
  //Non-void, non-object, return
  else if(returnType != 'v')
  {
    concreteType = YES;
  }
  [anInvocation retain];
  INSERT(anInvocation, retVal);
  if(concreteType)
  {
    while ([anInvocation retainCount] > rc)
    {
      // do nothing... just poll...
      sched_yield();
    }
  }
}
```

Listing 6 shows how a message is forwarded from one thread to another. The type signature is a string containing an encoded type. Pointers to objects are represented by '@' and void returns by 'v'. This allows us to have three cases, for object returns, void (no value) returns, and everything else. After inserting the message into the ring buffer, the caller spins if the return type is something that can not be handled asynchronously.

The insertion is handled by the macro shown in Listing 7. This inserts the invocation and a pointer to where the return value should be stored in to the ring buffer.

Listing 7: Inserting a value into the ring buffer.

```
#define INSERT(x,r) do {\
  /* Wait for space in the buffer */\
  while(ISFULL)\
  {\
    sched_yield();\
  }\
  invocations[MASK(producer)] = x;\
  invocations[MASK(producer+1)] = r;\
  __sync_fetch_and_add(&producer, 2);\
  if(producer - consumer == 2)\
  {\
    pthread_mutex_lock(&mutex);\
    pthread_cond_signal(&conditionVariable);\
    pthread_mutex_unlock(&mutex);\
  }\
}while(0);
```

6 Related Work

Objective-C has been extended in other ways. *Higher Order Messaging* [11] describes a mechanism where proxy objects are used to extend the notion of currying to messaging semantics and allow operations like map and fold to be applied to collections. This was also used to implement a simpler form of futures to that presented in this report, which only allowed a single message to be queued.

Concurrent Object Oriented 'C' (cooC) [9] is a language which is a pure superset of Objective-C (i.e. any Objective-C object is a cooC object) which adds additional concurrency semantics. Unlike the work presented in this report, cooC provided a number of additions to the language, including keywords for designating which methods in an interface interact in a way that is not concurrency-safe.

This report aims to demonstrate that the flexibility of the Objective-C metaobject protocol to adapt to existing techniques and so all of the presented techniques presented here have been implemented in other languages. Lieberman prototypes [6] were the foundation of the Self [10] programming language and more recently languages such as Io [5] and JavaScript.

Traits [7] were first implemented in Smalltalk, and have since been used in other languages.

7 Availability

The code presented in this report is available under a three-clause BSD license from the Étoilé project subversion repository: http://svn.gna.org/svn/etoile/trunk.

Some of the listings presented in this report are slightly simplified versions, for space reasons.

8 Conclusions

Although Objective-C is over two decades old, this report has shown that the language is flexible enough to adapt to many new ideas in object-oriented programming. Implementations of new forms of composition of methods, such as mixins and traits, are possible

due to the fact that the metaobject protocol is exposed to the developer.

Some features, such as the implementation of prototypes, are only possibly my modifying the metaobject protocol as implemented by the runtime library. While the fact that these interfaces are not specified as part of the language can sometimes be frustrating to developers, it has proved useful in this situation since it allows the interfaces to be extended without breaking existing code.

The strengths of Objective-C are also its weaknesses. The fact that it is a pure superset of C means that a lot of low-level functionality is exposed to users of the language. Much of the runtime system can be modified in this way. The fact that it must remain compatible with the C ABI also presents some limitations. Languages like Smalltalk implement autoboxing by using the low bits in pointer values to indicate primitive (non-object) types. This is not possible with Objective-C, since C functions can not be required to perform the requisite conversions.

Another issue is related to garbage collection. A number of schemes have been proposed for integrating conservative garbage collection into C, and these can be used by Objective-C. Accurate garbage collection is much harder, since we must work around the fact that C allows pointers to objects to be stored in non-pointer variables. Current Objective-C code uses simple reference counting. For future work we intend to combine this with accurate garbage collection. When a garbage collector becomes aware of an object, it will increment its reference count. When it believes the object is no longer referenced, it will decrement it.

This approach will allow multiple concurrent garbage collectors to track a single object. Objects will not be freed until their reference count reaches zero, which means that no garbage collectors have references to it and neither does any non-garbage-collected code. This also permits the implementation of a copying garbage collector. By inspecting the reference count of an object, a garbage collector can determine if there are any references to the object which the garbage collector is not tracking. If there are not, then it can be safely copied without breaking C or Objective-C language semantics. This approach is not new—bridges to language which support accurate garbage collection, such as Smalltalk, have used this approach in the past.

References

[1] Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold, and Urs Hölzle. Mixins in strongtalk. In *ECOOP Workshop on Inheritance*, June 2002.

[2] Brad Cox. Taskmaster position paper. In *ECOOP'91 Workshop On Exception Handling and OOPLsFoundation*, 1991.

[3] Brad J. Cox and Andrew J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.

[4] Brad L. Cox. The object oriented pre-compiler: programming smalltalk 80 methods in c language. *SIGPLAN Not.*, 18(1):15–22, 1983.

[5] Steve Dekorte. Io: a small programming language. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 166–167, New York, NY, USA, 2005. ACM.

[6] Henry Lieberman. Using prototypical objects to implement shared behavior in object- oriented systems. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 214–223, New York, NY, USA, 1986. ACM.

[7] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behavior. Number IAM-02-005, Universität Bern, Switzerland, November 2002. Also available as Technical Report CSE-02-014, OGI School of Science & Engineering, Beaverton, Oregon, USA.

[8] Walter R. Smith. Using a prototype-based language for user interface: the newton project's experience. In *OOPSLA '95: Proceedings of the*

tenth annual conference on Object-oriented programming systems, languages, and applications, pages 61–72, New York, NY, USA, 1995. ACM.

[9] Rajiv Trehan, Nobuyuki Sawashima, Akira Morishita, Ichiro Tomoda, Toru Imai, and Ken-Ichi Maeda. Concurrent object oriented 'C' (cooC). *SIGPLAN Not.*, 28(2):45–52, 1993.

[10] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA*, pages 227–242, 1987.

[11] Marcel Weiher and Stéphane Ducasse. Higher order messaging. In *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*, pages 23–34, New York, NY, USA, 2005. ACM.

A Full listing of Mixin Implementation

Listing 8: Implementation of traits and mixins.

```
#import "NSObject+Mixins.h"
#include <objc/objc.h>
#include <objc/objc-api.h>

static inline BOOL validateMethodTypes(Method_t
    method1, Method_t method2)
{
  return strcmp(method1->method_types,
      method2->method_types) == 0;
}

static inline Method_t findMethod(char* methodName,
     Class aClass, BOOL searchSuper)
{
  while(aClass != Nil)
  {
    struct objc_method_list * methods =
        aClass->methods;
    while(methods != NULL)
    {
      for(unsigned int i=0 ;
          i<methods->method_count ; i++)
      {
        Method_t method = &methods->method_list[i];
        if(strcmp(methodName,
            sel_get_name(method->method_name)) ==
            0)
        {
          return method;
        }
      }
      methods = methods->method_next;
    }
    if(searchSuper)
    {
      aClass = aClass->super_class;
    }
    else
    {
      aClass = Nil;
    }
  }
  return NULL;
}

static inline BOOL methodTypesMatch(Class aClass,
    Class aMixin)
{
  struct objc_method_list * methods =
      aMixin->methods;
  while(methods != NULL)
  {
    for(unsigned int i=0 ; i<methods->method_count
        ; i++)
    {
      Method_t method = &methods->method_list[i];
      Method_t oldMethod =
          findMethod((char*)sel_get_name(method->method_name),
           aClass, YES);
      /* If there is an existing method with this
          name, check the types match */
      if(oldMethod != NULL
        &&
        strcmp(method->method_types,
            oldMethod->method_types) != 0)
      {
        return NO;
      }
    }
    methods = methods->method_next;
  }
  return YES;
}

static inline BOOL iVarTypesMatch(Class aClass,
    Class aMixin)
{
  struct objc_ivar_list * mixinIVars =
      aMixin->ivars;
  struct objc_ivar_list * classIVars =
      aMixin->ivars;
  if(mixinIVars != NULL)
  {
    /* If the mixin has more ivars than the class
        */
    if(classIVars == NULL
      ||
      classIVars->ivar_count <
          mixinIVars->ivar_count)
    {
      return NO;
    }
    /* Look at each ivar in the mixin */
    for(unsigned int i=0 ; i<mixinIVars->ivar_count
        ; i++)
    {
      /* If the mixin has ivars of a different type
          to the class*/
      if(strcmp(mixinIVars->ivar_list[i].ivar_type,
          classIVars->ivar_list[i].ivar_type) !=
          0)
      {
```

```
        return NO;
      }
    }
  }
  return YES;
}
id test(id self, SEL cmd)
{
  return nil;
}
//Objective-C runtime library private function
void __objc_update_dispatch_table_for_class(Class);

static inline void addMethods(Class aClass, struct
    objc_method_list * methods)
{
  struct objc_method_list * newMethods = malloc(
      sizeof(struct objc_method_list)
      +
      (methods->method_count * sizeof(struct
          objc_method)));
  int usedMethods = 0;
  /* We need to copy the entire method list,
      because otherwise replacing
   * methods in it will cause us problems later. */
  for(unsigned int i=0 ; i<methods->method_count ;
      i++)
  {
    Method_t mixinMethod =
        &methods->method_list[i];
    Method_t oldMethod =
        findMethod((char*)sel_get_name(mixinMethod->method_name),
         aClass, NO);
    if(oldMethod != NULL)
    {
      /* Update the old IMP to point to the new
          method */
      oldMethod->method_imp =
          mixinMethod->method_imp;
    }
    else
    {
      /* Add the new method to the list */
      memcpy(&newMethods->method_list[usedMethods++],
          mixinMethod,
          sizeof(struct objc_method));
    }
  }
  /* Free up any bonus memory we allocated */
  if(usedMethods > 0
    &&
    methods->method_count < usedMethods)
  {
    newMethods = realloc(newMethods,
      sizeof(struct objc_method_list)
      +
      (usedMethods * sizeof(struct objc_method)));
  }
  /* Add the new method list to the class */
  if(usedMethods > 0)
  {
    newMethods->method_count = usedMethods;
    newMethods->method_next = aClass->methods;
    aClass->methods = newMethods;
  }
  else
  {
    /* Sometimes, all of our methods will be
        replacing existing ones */
    free(newMethods);
  }
}

static void checkSafeComposition(Class class, Class
    aClass)
{
  /* Check that the mixin will never try to access
      ivars from after the end of the
   * object */
  if(class->instance_size < aClass->instance_size)
  {
    [NSException raise:@"MixinTooBigException"
                format:@"Class %@ is smaller than
                    composed class %@.  Instance
                    variables access from mixin is
                     unsafe.", class, aClass];
  }
  if(!iVarTypesMatch(class, aClass))
  {
    [NSException
        raise:@"MixinIVarTypeMismatchException"
                format:@"Instance variables of
                    class %@ do not match those of
                     composed class %@.  Instance
                    variables access from composed
                     class is unsafe.", class,
                    aClass];
  }
  if(!methodTypesMatch(class, aClass))
  {
    [NSException
        raise:@"MixinMethodTypeMismatchException"
         format:@"Method types of class %@ do not
             match those of mixin %@.", class,
             aClass];
  }
}
```