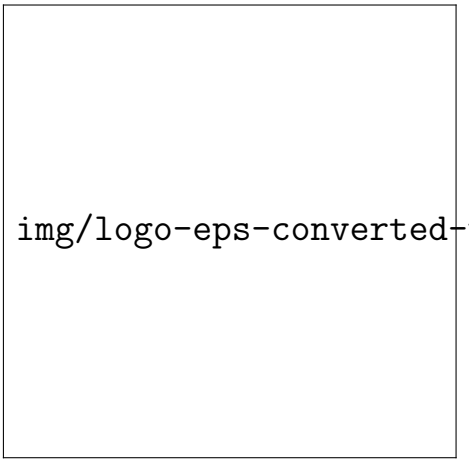
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bc. Kryštof Váša

Modular Objective-C Run-Time

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Martin Děcký
Study programme: programme
Specialization: specialization

Prague 2012-13

Dedication.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Název práce

Autor: Jméno a příjmení autora

Katedra: Název katedry či ústavu, kde byla práce oficiálně zadána

Vedoucí diplomové práce: Jméno a příjmení s tituly, pracoviště

Abstrakt:

Klíčová slova:

Title:

Author: Jméno a příjmení autora

Department: Název katedry či ústavu, kde byla práce oficiálně zadána

Supervisor: Jméno a příjmení s tituly, pracoviště

Abstract:

Keywords:

Contents

Introduction **2**

1 Objective-C **3**
1.1 Foreword . . . 3
1.2 Brief history of Objective-C . . . 3
1.3 Compilation of Objective-C . . . 4
1.3.1 Calling methods . . . 5
1.3.2 Creating Classes Programmatically . . . 7
1.3.3 Translating Methods to Functions . . . 8

2 Apple's Implementation **9**
2.1 Portability . . . 9
2.1.1 Example 1 - malloc . . . 9
2.1.2 Example 2 - issetguid . . . 10
2.2 Limitations . . . 10
2.2.1 16B Object Minimum Size . . . 10
2.2.2 Dynamic Loader Support . . . 11
2.2.3 C++ Influences . . . 11
2.3 Summary . . . 11

3 GCC Implementation **12**
3.1 Differences from Apple's Implementation . . . 12
3.1.1 Message Sending . . . 12
3.1.2 Module Loading . . . 12
3.1.3 Typed Selectors . . . 12
3.2 Portability and Limitations . . . 13

4 Étoilé / GNUstep **14**
4.1 Étoilé . . . 14
4.1.1 Slots . . . 14
4.1.2 Inline Caching . . . 15
4.1.3 Message Sending . . . 16
4.1.4 Tags . . . 17
4.1.5 Metaclasses . . . 17

Conclusion **18**

Bibliography **19**

List of Tables **20**

List of Abbreviations **21**

Attachments **22**

Introduction

This thesis will analyze source codes of existing versions of Objective-C run-time, their limitations or requirements for compilation. Result of this work will be a prototype of a modular Objective-C run-time, which will allow easy configuration of the run-time environment both at the compilation and run time. For example, for a single-threaded application, you can turn off the locking of internal structures without affecting stability, yet performance can be improved (with each message sent[1], a lock can be potentially locked when the method implementation isn't cached and the class hierarchy has to be searched) - this may save quite a few syscalls.

There are four available Objective-C run-time implementations (to my knowledge) - one is provided by *Apple* and is used in its OS X and iOS systems - there are slight differences between the iOS and OS X versions of the run-time (e.g. iOS doesn't support garbage collection and only the new 2.0 run-time is available). Within this thesis, when talking about Apple's implementation of the run-time, the OS X version will be the one talked about. Then there's a run-time provided with *GCC* and a more experimental one called *Étoilé* which has been used to improve the *GNUStep* run-time[2].

Even though I will mention a few words about the garbage collection and ARC[3], not much attention will be paid to them as garbage collection is being deprecated in OS X 10.8 (TODO - ELABORATE IN APPLE SECTION - and has severe dependencies on Mac OS X itself) and ARC is relatively new and uses a lot of compiler-dependent features as well as auto-zeroing weak references, etc.; which is beyond the scope of this work. Instead, the focus will be put on the core functionality of the run-time, analysis of the current implementations and designing the modular run-time itself.

[1] In Objective-C method calls are called messages being sent to objects, just like in Smalltalk.

[2] http://www.jot.fm/issues/issue_2009_01/article4/index.html

[3] ARC - automatic reference counting, a feature introduced in Xcode 4.2 (Xcode is Apple's IDE) that uses compiler's static analysis combined with special keywords to automatically insert retain/release calls so that the developer doesn't need to manually manage reference counts on objects.

1. Objective-C

1.1 Foreword

This thesis will analyze source codes of existing versions of Objective-C run-time, their limitations or requirements for compilation. Result of this work will be a prototype of a modular Objective-C run-time, which will allow easy configuration of the run-time environment both at the compilation and run time. For example, for a single-threaded application, you can turn off the locking of internal structures without affecting stability, yet performance can be improved (with each message sent[1], a lock can be potentially locked when the method implementation isn't cached and the class hierarchy has to be searched) - this may save quite a few syscalls.

There are four available Objective-C run-time implementations (to my knowledge) - one is provided by *Apple* and is used in its OS X and iOS systems - there are slight differences between the iOS and OS X versions of the run-time (e.g. iOS doesn't support garbage collection and only the new 2.0 run-time is available). Within this thesis, when talking about Apple's implementation of the run-time, the OS X version will be the one talked about. Then there's a run-time provided with *GCC* and a more experimental one called *Étoilé* which has been used to improve the *GNUStep* run-time[2].

Even though I will mention a few words about the garbage collection and ARC[3], not much attention will be paid to them as garbage collection is being deprecated in OS X 10.8 (TODO - ELABORATE IN APPLE SECTION - and has severe dependencies on Mac OS X itself) and ARC is relatively new and uses a lot of compiler-dependent features as well as auto-zeroing weak references, etc.; which is beyond the scope of this work. Instead, the focus will be put on the core functionality of the run-time, analysis of the current implementations and designing the modular run-time itself.

1.2 Brief history of Objective-C

In the early 1980s, Brad Cox and Tom Love decided to bring the object-oriented concept to the world of C while maintaining full backward compatibility, strongly inspired by Smalltalk

In 1988, NeXT has licensed Objective-C from Stepstone (the company Cox and Love owned), added Objective-C support to the GCC compiler and decided to use it in its OpenStep and NeXTStep operating systems.

After Apple had acquired NeXT in 1996, Objective-C stayed alive in Rhapsody[4] and later on in Mac OS X, where it's the preferred programming language

[1] In Objective-C method calls are called messages being sent to objects, just like in Smalltalk.

[2] http://www.jot.fm/issues/issue_2009_01/article4/index.html

[3] ARC - automatic reference counting, a feature introduced in Xcode 4.2 (Xcode is Apple's IDE) that uses compiler's static analysis combined with special keywords to automatically insert retain/release calls so that the developer doesn't need to manually manage reference counts on objects.

[4] http://en.wikipedia.org/wiki/Rhapsody_(operating_system)

to the date.

For this whole time, the Objective-C language stayed almost the same without any significant changes. In 2006, Apple announced Objective-C 2.0 (which was released in Mac OS X 10.5 in 2007), which introduced garbage collection (which has been deprecated in 10.8 in favor of more efficient ARC - automatic reference counting[5]), properties (object variables with automatically generated getters and/or setters with specified memory management), fast enumeration (enumeration over collections in a foreach-style), and some other minor improvements.

Lately, more improvements have been made to Objective-C, most importantly the aforementioned ARC (automatic reference counting). Apple's run-time a hardcoded set of selectors (method names) that handle the memory management, -autorelease, -retain, -release (together called ARR), in particular. ARC automatically inserts these method calls and automatically generates a -dealloc method (which is called when the object is being deallocated) - which requires compiler support, though.

This, however, presents a problem - none of the ARR calls must be called directly in the code - hence you need to convert all of your code to ARC. One disadvantage which results in a big advantage - compatibility with all libraries (Apple calls Objective-C libraries frameworks) - this was a big disadvantage of garbage collection:

You could keep the code as it was as the run-time itself redirected the ARR methods to a no-op function on the fly, however, all linked libraries/frameworks/plugins needed to be recompiled with garbage collection support turned on. This caused two things: mess in the code as if you migrated your code to garbage-collection-enabled environment, it was riddled with ARR calls, however, newly written code typically omitted those calls, making the code inconsistent; and some libraries never got GC support anyway, so you couldn't use them in GC-enabled applications.

In the newest release of OS X 10.8, several new features have been included - default synthesis of getters (in prior versions, you had to declare `@property` in the header file and use `@sythesize` or `@dynamic` in the implementation file - see Syntax of Objective-C), type-safe enums, literals for NSArray, NSDictionary and NSNumber (classes declared in Apple's Foundation framework), etc.

1.3 Compilation of Objective-C

Objective-C is an object-oriented programming language that is a strict superset of C. Any C code can be used within Objective-C source code[6]. Its run-time is written in C as well, some parts in assembly language (mostly performance optimizations) or more recently in C++ (more about that later on). This thesis assumes that you have some brief knowledge of both C and Objective-C, at least syntax-wise.

[5] http://cocoaheads.tumblr.com/post/17719985728/10-8-objective-c-enhancements

[6] The examples below relate to Apple's version of the run-time. Other versions of the run-time may call different function, and even use different structures. These examples are here to simply illustrate the mechanism of translation to C functions.

All Objective-C code can be translated to calls of C run-time functions[7] - for example, sending a message to an object isn't anything else than calling a run-time function `objc_msgSend`.

This section will cover compilation of Objective-C code and how it's translated into calling run-time functions.

1.3.1 Calling methods

This is a sample code that sends two messages - each to a different object, though[8]:

```
SomeClass *myObj = [[SomeClass alloc] init];
```

This will be translated to:

```
SomeClass *myObj = ((id (*)(id, SEL, ...))(void *)objc_msgSend)
             ((id)((id (*)(id, SEL, ...))(void *)objc_msgSend)
                                (objc_getClass("SomeClass"),
                                sel_registerName("alloc")),
                                sel_registerName("init"));
```

Which after removing the casting and adding a little formatting is:

```
SomeClass *myObj =
  objc_msgSend(
    objc_msgSend(
      objc_getClass("SomeClass"),
      sel_registerName("alloc")),
    sel_registerName("init"));
```

So it's two nested `objc_msgSend` calls[9]. `objc_msgSend` is a method that can be said to be the core of Objective-C run-time. It's the most used function of the run-time. Every method call in Objective-C gets translated into this variadic function call, which takes `self` as the first argument (i.e. the object that the method is called on, or the message is sent to[10]), the second argument is a selector (generally the method's name) and can be followed by arguments.

The run-time then looks up the object's class, finds a function that implements this method (so called `IMP`[11]) and calls it. There's a several things to point out:

• method *names* are used. `sel_registerName` is a function that makes sure that for that particular method name only one selector pointer is kept. • each of the calls goes to a different object. The first call gets to something returned by `objc_getClass` which returns an instance of a meta class (which is an object as

[7] There's a LLVM Clang compiler option -rewrite-objc which will convert all the Objective-C syntax into calls of pure C methods - the run-time methods. When run 'clang -rewrite-objc test.m', where test.m contains the Objective-C code, a new test.cpp is created, containing the translated code.

[8] As will be explained later on, each class actually consists of two classes - the meta class, which has the +alloc method and the regular class, which has the -init method.

[9] There are actually specific functions for methods that return floating point numbers or structures, as these may require special ABI treatment on some architectures.

[10] Here can be seen the Smalltalk influence.

[11] IMP is defined as a pointer to a function: id (*IMP)(id, SEL, ...).

well) • every class consists of two classes - a class pair - one regular of which you create objects and one meta - which typically (unless you manually craft another one) has only one instance: a receiver for class methods (static methods).

If you care to investigate this hierarchy, here's a small example:

`NSObject` class is part of the Foundation framework Apple supplies. Even though its often assumed to be the one and only root class in Objective-C, this is quite incorrect: there can be as many root classes in Objective-C as you wish - `NSProxy` is an example and you can easily create your own.

```
@interface ClassWithoutSuperclass
@end
```

This will declare a new root class. It has absolutely no functionality - no memory management `retain` and `release`, no `+alloc` method is declared either - you wouldn't be able to even create a new instance of this class without the run-time function `class_createInstance`[12].

Each object is a pointer to a structure. Let's use this simplified class structure (it's more complex in real life):

```
typedef struct class_t {
  struct class_t *isa;
  struct class_t *superclass;
} objc_class_t;
```

Every object is a pointer to a structure like this one. The first field, so-called `isa`, is a pointer to the class structure (to the meta class instance). Second field points to the superclass. It might be confusing at first that it's a class structure, but remember that even the object's class is actually an object - an instance of the meta class[13].

Assume the following code:

```
@interface Rootclass
@end
@implementation Rootclass
@end

@interface Subclass : Rootclass
@end
@implementation Subclass
@end
```

This declares two classes - `Rootclass` and `Subclass`. The `Rootclass` is a new root class with no superclass. As neither of these classes declares any methods, calling anything on either class would result in a run-time exception, even the usual object creation via `[[Rootclass alloc] init]` isn't available as `Rootclass` doesn't declare the `+alloc` method - it's declared on the `NSObject` class, which

[12]This is basically why it is recommended to inherit all classes from NSObject (or any other root class) which implements some basic communication with the run-time as well as some basic memory management, etc.

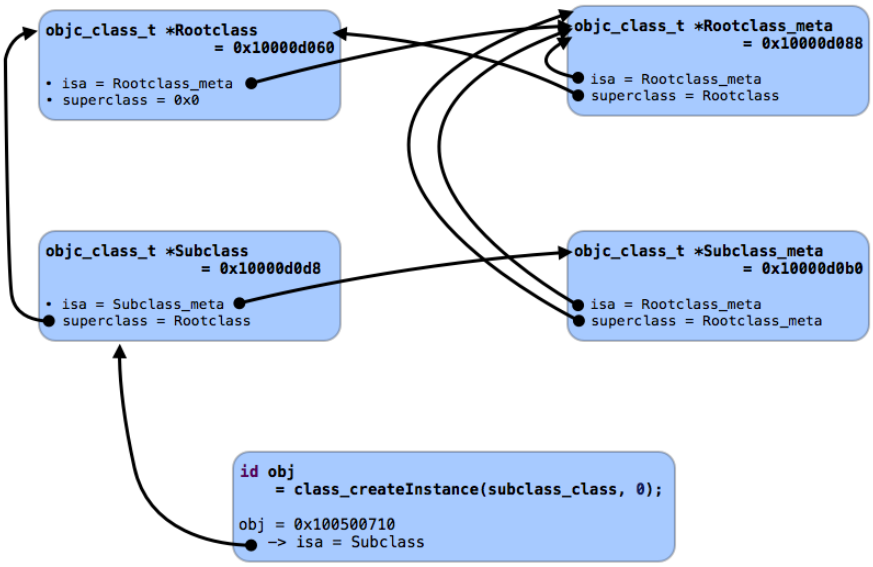
[13]The root meta class's isa pointer points to the structure itself - i.e. there's a pointer cycle.

is why you can create instances of the "regular" classes inheriting from `NSObject` this way.

Hence we need to use the run-time `class_createInstance` function to create an instance of the class:

```
id obj = class_createInstance(objc_getClass("Subclass"), 0);
```

The `objc_getClass` function returns a pointer to the class called `Subclass`, the `objc_getClass` function creates an instance of the `Subclass` class, with `0` extra bytes[14]. Here's a class and meta-class diagram of this situation.



1.3.2 Creating Classes Programmatically

There is a function called `objc_allocateClassPair` which creates a brand new class and its meta class (together a class pair) on the run. All you need to specify is the superclass, new class name and extra bytes. Using functions such as `class_addMethod`, `class_addIvar`, `class_addProtocol` and `class_addProperty` can be used to add methods, ivars, protocols and properties to a class.

Using these methods, you can easily substitute the Objective-C compiler, creating all classes at the beginning of the application run.

In reality, declaring a class doesn't cause the compiler to generate function calls, however, instead, the compiler creates static class structures which are later on copied by the linker into the `__OBJC` section of the Mach-O binary (on OS X), which is copied on the launch time to memory and the classes just get registered to the run-time[15], which is much faster that dynamically create classes one by one, connecting all methods. We will, however, focus on the run-time methods, ignoring linker dependencies.

[14]TODO: explain extra bytes.

[15]Creating the class using the objc_allocateClassPair function isn't enough in order to create an instance of this class, you need to register the class pair with the run-time as well using objc_registerClassPair. This is simply to avoid creating an instance of the class before it gets fully initialized, e.g. from a different thread.

1.3.3 Translating Methods to Functions

As noted several times before, all Objective-C code can be rewritten in pure C code. This brings us to a question, how are the methods translated to C constructs - obviously into a function, so-called `IMP`. Let's use this class to demonstrate:

```
@implementation SomeClass
+(void)doSomethingStatic{
  // ...
}
-(void)someMethod:(void*)arg1 secondArgument:(void*)arg2{
  // ...
}
@end
```

This gets translated into two functions:

```
typedef struct objc_object SomeClass;

void _C_SomeClass_doSomethingStatic(Class self, SEL _cmd){
  // ...
}

void _I_SomeClass_someMethod_secondArgument(SomeClass *self,
                              SEL _cmd, void *arg1, void *arg2){
  //...
}
```

As you can see, each method gets translated into a function of at least two parameters. The first argument is `self` - a pointer to the object the message is being sent to. In the first case a `Class` object[16], in the second case a pointer to the `SomeClass` object. The second argument, `_cmd`, is the selector (`SEL`). Selector is a structure that consists of just the method's name, so theoretically, it's possible to simply retype `char*` to `SEL`, however, shouldn't be done as the run-time requires equally named selectors to point to the same structure (that's why the `sel_registerName` should be used to convert a `char*` to `SEL` which unifies the equally named selectors).

The function names get slightly obfuscated - `_X_ClassName_method_name_` - where `X` is either `I` for instance methods or `C` for class methods. As Objective-C method names can have multiple parts, each followed by a semi-colon (e.g. `someMethod:secondArgument:`), each part gets concatenated using an underscore.

[16]The Class is defined as typedef struct objc_class *Class.

2. Apple's Implementation

First thing that comes across mind when studying Apple's source codes is history - there's a lot of historical code there. You can find, for example, some code from the NeXT era and a port to Windows - NeXT had a set of APIs called OpenStep (which is the predecessor of today's Cocoa on OS X), which was written in Objective-C and was aiming to run on virtually any reasonable system[1]. Also, some of Apple's own software for Windows was written in Objective-C, hence the run-time needed to be compilable under Windows as well[2].

While the legacy code is understandable as it is required to maintain binary compatibility of all existing binaries and the portability is generally an objective of this work, neither is done in a very clean fashion.

2.1 Portability

The portability is ensured by rather large `\#ifdef - \#else - #endif` statements that use macros and static inline functions to define aliases to some OS X-specific functions on Windows, such as `malloc_zone_malloc` or even POSIX-specific functions, like `issetguid`. I will elaborate on these two examples.

2.1.1 Example 1 - malloc

On OS X, the malloc function has an extension to support memory zones[3] - this way you can create multiple heaps, which can get destroyed entirely at once. It has a very limited usage nowadays, but back in the day when computers had only a little memory, it was useful, to allocate temporary objects (e.g. during a specific calculation) in its own zone, freeing it as a whole when you are done with these objects. For example, the NSMenu, which is a class representing a menu on OS X, has a class method +menuZone which returns a zone that is used for menu allocations - as a menu is an element that gets displayed on the screen for only a short period of time, all memory used to represent it is stored in a separate zone, freeing all the memory when the menu is dismissed and hence preventing memory fragmentation.

As Windows supports only the regular `malloc` function, this had to be solved - and it has been solved rather radically, by defining the `malloc_*` methods as static inline methods that call the Windows API functions[4]:

```
static inline void *malloc_zone_malloc(malloc_zone_t z,
                                       size_t size) {
    return malloc(size);
}
```

[1] http://en.wikipedia.org/wiki/OpenStep

[2] http://jongampark.wordpress.com/2009/02/24/safari-4-beta-for-windows/

[3] http://developer.apple.com/library/Mac/#documentation/Darwin/Reference/ManPages/man3/malloc_zone_malloc.3.html

[4] objc-os.h

While it's functional, if Microsoft ever decided to implement zones, it would break the code. Moreover, there is another function defined in the source code[5]:

```
void *_malloc_internal(size_t size) {
    return malloc_zone_malloc(_objc_internal_zone(), size);
}
```

It would, of course, make much more sense to simply declare these internal functions that would pass to the actual function using if-else.

// TODO - point to my implementation using function pointers

2.1.2 Example 2 - issetguid

The other example is how is the `issetguid` function transferred to the Windows environment:

```
#define issetugid() 0
```

2.2 Limitations

There are several limitations that can be found in the code.

2.2.1 16B Object Minimum Size

Apple supplies several large libraries (or as they call it frameworks), that a vast majority of applications builds upon. In particular, the CoreFoundation and Foundation.

CoreFoundation, although it implements many Objective-C classes[6], they are used privately and it provides only C exports and headers.

On the other hand, Foundation, its counterpart, has Objective-C exports.

For example, `CFStringRef` is a pointer to a structure used by the CoreFoundation to represent a string. Foundation has a `NSString` class, which does generally the same. In order to prevent unnecessary code duplication as well as data conversions when using CoreFoundation functions (which accept `CFStringRef`) from Objective-C code, toll-free bridging has been introduced.

There is an intricate mechanism behind it[7], but means that instances of classes that support toll-free bridging, can be simply casted to their CoreFoundation counterpart and vice versa. Using the `CFStringRef`/`NSString` couple, this code if fully valid:

```
NSString *myString = @"Hello World!";
CFStringRef duplicatedString = CFStringCreateCopy(NULL,
                                    (CFStringRef)myString);
NSString *duplicatedString2 = (NSString*)duplicatedString;
```

[5] objc-class.m

[6] This can be easily verified using the class-dump tool - http://www.codethecode.com/projects/class-dump/

[7] You can read more about it here: http://www.mikeash.com/pyblog/friday-qa-2010-01-22-toll-free-bridging-internals.html

While this is very convenient to every Apple developer, it poses an unexpected limitation: all object instances need to have the same minimum size as CoreFoundation "objects" - 16 bytes.[8]

2.2.2 Dynamic Loader Support

As it's faster to simply copy the data from the binary[9] than to construct the classes using the run-time functions, Apple's implementation contains a set of functions that are called by the dynamic loader (`dyld`) to load classes from the binary image, link them to their superclasses and to register them.

This, however, adds `dyld` to the list of library dependencies, which is transitively required by `libSystem`[10] anyway, but it adds dependencies to the code itself.

Also, loading a binary image of the classes introduces and issue with binary compatibility - any change in the internal representation of a class will cause the binary not to launch.

Creating classes manually using the run-time methods ensures binary compatibility, while poses a question where should be the class information stored. One option is to generate a function with a specific name that's called by the dynamic loader, or calling it manually before anything else in the `main` of your program.

2.2.3 C++ Influences

The newest parts of the Objective-C run-time use many C++ features, such as methods on structures, some C++ classes, e.g. `vector` and tries to unify Objective-C and C++ exceptions.

While this may help to clean up the code a little, it adds additional dependencies on C++ libraries.

2.3 Summary

Apple's implementation is riddled with a lot of obscure code and other very specific details[11]. Moreover, the documentation is very brief, not every function has a description of what it exactly does, or only has a very short note that it is used from a different function somewhere else in the code. As will be described below, the GCC version of the run-time is much better documented in this matter.

[8] Can be also found in a comment in objc-class.m file within the _class_createInstancesFromZone function.

[9] In particular from the __OBJC section.

[10] Basic system library on OS X that every application needs to be linked to.

[11] For example, when the class images are stored in the __OBJC binary section, the superclass field is pointing to a string containing the name of that superclass. When the run-time is connecting the images, it's casting the superclasses pointer to char*, to read the superclass name, which is not a very clean solution.

3. GCC Implementation

In comparison to Apple's source codes, GCC's code is much cleaner and very well documented - even every `#include` is commented why and which functions from that file are used. Aside from this, there are multiple differences between the Apple and GCC implementations of the run-time.

3.1 Differences from Apple's Implementation

3.1.1 Message Sending

Apple's run-time uses the `objc_msgSend` function to send messages to objects. This function needs to handle finding the correct `IMP` function for the selector, execute it and return the return value of the function. This, unfortunately, has a slight disadvantage - on some architectures, some values (`double` and `struct` values, for example) get returned a different way - using a different register, which needs to be taken into account. Hence Apple's implementation contains several other functions, such as `objc_msgSend_stret` for structures and `objc_msgSend_fpret` for float values[1].

The GCC implementation takes another approach, which requires no specialized functions. A `[receiver method]` gets compiled to the following two lines:

```
IMP function = objc_msg_lookup(receiver, @selector(method));
id result = function(receiver, @selector(method));
```

While this solution has a disadvantage that several calls to Objective-C objects cannot be chained as in the example in Chapter 1, it is not a crippling disadvantage as this code is very rarely written by the developer and chaining function calls require the C compiler to place the value into a temporary variable anyway, so there isn't any performance cost - if any, it may one instruction of fetching an extra variable - the function, but this is outweighed by the message lookup mechanism anyways.

3.1.2 Module Loading

TODO - investigate

3.1.3 Typed Selectors

In Apple's implementation, selectors (`SEL`) are pointers to a structure with just one field - a `char*` which includes the selector's name. Whenever you want to send a message to an object, you need to retrieve a selector for name (using the `sel_registerName` function). As the run-time needs the message sending to be as fast as possible, it hashes the selector in order to find the `IMP` for that

[1] On i386 computers, the 'fpret' is used for double values, on x86-64, just for long double values. The 'stret' function has, unlike other objc_msgSend functions has a pointer to the structure address as a first argument and returns void. Other arguments follow the structure pointer.

particular object. Thanks to registering the selectors, each selector is unique and there can't be two selectors with the same name in the run-time. This allows the message lookup mechanism to simply create a hash from the pointer and find a method by a simple pointer comparison, without actually reading the string.

GCC's implementation extends the selectors into typed selectors - the selector structure has a second field which also contains `char*`, but this time, there are encoded types the method takes. This means that `-(void)hello:(int)anInt;` and `-(void)hello:(id)anObject;` have different selectors, while they yield in the same selectors in Apple's implementation[2].

While it is a nice idea to bring a little more type-safety into the Objective-C world, it just brings mess into the run-time, in my opinion. The GCC run-time tries keep ABI compatibility with Apple's run-time, which doesn't have typed selectors. So, suddenly, there is a mix of typed and untyped selectors.

3.2 Portability and Limitations

The portability of the run-time is its only limitation and is defined simply: it requires a POSIX layer[3]. Most of the files import at least one POSIX file, usually `<string.h>` for `memcpy` function and its relatives.

Another issue is that it relies on the `gthread` library instead of`pthread`. All threading support in this run-time is just a wrapper around `gthread` that are part of GCC. While this allows some more efficient threading support on systems that natively do not use `pthread` structures, it ties the run-time to GCC. Also, the run-time uses thread-local storage using `__thread` keyword, which isn't supported on all systems as it requires support from the linker, dynamic loader and system libraries[4].

[2]This, of course, requires an introduction of new run-time functions, such as 'sel_registerTypedName'.

[3]On non-POSIX systems, it requires an additional POSIX layer, for example, on Windows, it requires Cygwin or MinGW.

[4]http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/Thread_002dLocal.html

4. Étoilé / GNUstep

4.1 Étoilé

Étoilé is an experimental run-time written by David Chisnall, a research assistant in computer science at Swansea University. It has been written as a part of his paper called *A Modern Objective-C Runtime*[1]. Even though it is not a real-world run-time (i.e. no compiler supports it), it has introduced quite a few interesting ideas how to speed up the method lookup time as well as make the run-time more generic.

Unlike the previously described run-times, the Étoilé run-time tackles the task of providing a run-time from a totally different angle. While the other implementations simply aim to create a traditional Objective-C run-time, where Apple chooses to keep almost all of the original API for its Objective-C 2.0, Étoilé run-time tries to create a very generic run-time that could be used with many other languages as well, which would result in a very easy language bridging[2].

This task, however, required to start from scratch and leave all compatibility behind. As the author notes, the run-time itself was never standardized, unlike the language, so a person shouldn't rely that much on it. Hence all the `objc_msgSend`, `objc_getClass`, `sel_registerName` are not available in this run-time. The data structures are modified, or completely missing - for example `SEL` is defined as a `uint32_t`, which is a hash for an internal representation of the selector, which is a `objc_selector` structure, which contains name and a type string, like in GCC's run-time.

The source code of this run-time is much shorter than the other implementations[3], however, its extensive use of macros makes it less readable, at least for me.

4.1.1 Slots

Instead of defining methods, the more generic approach is to define slots. A slot is the basic type for message lookup, a structure containing five fields: `int offset`, `IMP method`, `char *types`, `void *context` and `uint32_t version`. This allows the run-time to store both properties and methods using a single structure: as the properties, introduced in Objective-C 2.0 run-time, are just wrappers around automatically generated getter and setter methods, this approach allows to define a property simply using a slot that has a defined `offset` field, which servers as a number of bytes after the beginning of the object is that variable stored. By setting the `method` field of the slot, the slot functions like a regular method.

As I have mentioned, the run-time was built to meet needs of other languages as well, for example JavaScript (and other prototype-based languages), where you

[1] http://www.jot.fm/issues/issue_2009_01/article4/index.html

[2] A mechanism, where objects from one language can be interacted with from another language.

[3] According to the author the run-time is just 15% of the size of GCC's implementation, however, this run-time doesn't currently provide some of the modern features of run-times.

can dynamically add variables to an object. This can be done by adding a slot, which will then hold the value as well - in the `context` field of the structure.

Whenever the structure gets updated (`IMP` is changed, etc.), the version is increased. This is important later on for caching.

4.1.2 Inline Caching

Using monomorphic or polymorphic inline caching, the author of Etoilé run-time was able to achieve impressive speeds, reducing the method call time to only twice the time of a pure C function call, even faster than the C++ method calls are.

With every dynamic object-oriented language, a question arises, how to fetch the function that implements a method. This lookup function is usually the critical part of the run-time's performance. As this lookup is expensive, all of the run-times described here use some caching mechanisms.

Imagine your class `FCButton`, which is a subclass of `NSButton`, which is a subclass of `NSControl`, `NSView`, `NSResponder`, `NSObject`. If the dispatch had to lookup each method, for example the very commonly used `retain` and `release`, which are implemented most likely just in the `NSObject` root class, the lookup function would have to climb the whole class hierarchy, until it found the method implementation - the `IMP` pointer.

For this, cache has been introduced. Both Apple's and GCC's run-times have a cache, often called dispatch table - when the user wants to call a method, the run-time needs to look up the function pointer. First, it looks into the cache (which usually is fast), returns it, if it has been found, otherwise looks up the function pointer in the class hierarchy and saves it into the cache for further use.

An issue here is that when the implementation of a method in some class is changed, all cache entries with the original function pointer need to be removed. The same applies when a new module has been loaded with a class category - a category may replace already existing method of that class.

But even this lookup costs something - there's at least one C function call to fetch the data from the cache, plus the actual fetching from the cache. The approach of Etoilé run-time is to generate inline caches.

Every time a method is supposed to be called, the following lookup macro is applied[4]:

```
#define SLOT_LOOKUP_MIC(obj, sel_name, sel_types, sender, action)\
do\
{\
    static SEL selector = 0;\
    struct objc_slot * slot;\
    if(selector == 0)\
    {\
        selector = lookup_typed_selector(sel_name, sel_types);\
```

[4]This example uses the monomorphic cache. Unless you assume the same message will sent to objects of different classes, it is well sufficient. The polymorphic cache may be useful for class clusters (for example, NSString - which can be of multiple different classes, depending whether it's a constant string, created by run-time, etc.), or if you are expecting subclasses of the variable type class to be passed as well.

```
    }\
    static __thread struct inline_cache_line cache;\
    if(cache.slot != NULL \
        &&\
        cache.type == (id)obj->isa\
        &&\
        cache.version == cache.slot->version)\
    {\
        slot = cache.slot;\
    }\
    else\
    {\
        id object = (id)obj;\
        slot = slot_lookup(&obj, selector, sender);\
        if(obj == object)\
        {\
            cache.version = slot->version;\
            cache.slot = slot;\
            cache.type = obj->isa;\
        }\
    }\
    struct objc_call call = { slot, selector, sender };\
    action\
} while(0)
```

Inline caches are static `__thread`[5] structures, so that each thread has its own cache. This static cache is created for every place in the code, where an method is called. The selector itself is cached, so no lookup is needed for it either. Then the cache is checked if it is filled with this object's class[6] and the version matches the slot's version - if the slot has been modified since, the cache is invalidated.

4.1.3 Message Sending

The traditional message sending, where the method gets translated into a function with the first parameter `self` and the second one `_cmd` has been abandoned for a slightly more complicated, yet more flexible call:

```
typedef struct objc_call {
  SLOT slot;
  SEL selector;
  id sender;
} *CALL;
#define _cmd (_call->selector)
id method(id self, CALL _call, ...);
```

[5] As noted above, this is a GCC extension, that needs support from the OS.

[6] I.e. this place in the code has been visited before, but the object has been of another class before - a polymorphic cache should be used here instead.

This extends the original simple `_cmd` of type `SEL`: it adds a more context to the call, which can be used to implement a private inner class, for example, which determines with each call that the `sender` is indeed its parent class.

4.1.4 Tags

4.1.5 Metaclasses

Thanks to the slot-based approach, it was easy to 0

Conclusion

Bibliography

List of Tables

List of Abbreviations

Attachments