Polymorphic Identifiers: Uniform Resource Access in Objective-Smalltalk

Marcel Weiher
Hasso Plattner Institute, University of Potsdam
marcel@metaobject.com

Robert Hirschfeld
Hasso Plattner Institute, University of Potsdam
hirschfeld@acm.org

Abstract

In object-oriented programming, polymorphic dispatch of operations decouples clients from specific providers of services and allows implementations to be modified or substituted without affecting clients.

The Uniform Access Principle (UAP) tries to extend these qualities to resource access by demanding that access to state be indistinguishable from access to operations. Despite language features supporting the UAP, the overall goal of substitutability has not been achieved for either alternative resources such as keyed storage, files or web pages, or for alternate access mechanisms: specific kinds of resources are bound to specific access mechanisms and vice versa. Changing storage or access patterns either requires changes to both clients and service providers and trying to maintain the UAP imposes significant penalties in terms of code-duplication and/or performance overhead.

We propose introducing first class identifiers as polymorphic names for storage locations to solve these problems. With these *Polymorphic Identifiers*, we show that we can provide uniform access to a wide variety of resource types as well as storage and access mechanisms, whether parametrized or direct, without affecting client code, without causing code duplication or significant performance penalties.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Input/output; Polymorphism; Dynamic storage management

Keywords Identifiers;REST;Uniform Access Principle;Extensibility

1. Introduction

Imperative programming languages feature the concept of a store with locations. Locations or *L-Values* are characterized by the essential features of having content (an associated R-Value) and 2 operations to access and update this value from the location, the *Load-Update Pair* or *LUP*. Strachey decouples the definition of a store from the notion of memory or addressability [31].

Programming languages such as Pascal or C implement the store as main memory, with identifiers mediating access. We can compose these identifiers, use them on the LHS and RHS and partially use them for indirect access. Computational abstraction is handled separately.

Object-oriented programming languages such as Smalltalk, Java and Objective-C mostly maintain the distinction between computational and storage abstraction, with polymorphism and most other abstraction mechanisms only applying to computational abstraction. Access to storage, when not mediated via a method, defeats encapsulation and makes code brittle against changes or subclassing, which is why convention moved toward having all access to state (instance variables) mediated by methods.

The Uniform Access Principle (UAP) [22] was coined by Meyer to formalize this convention: access to state should be indistinguishable from computation. The purpose of the UAP is to isolate clients from changes to the storage strategy of an object.

Different languages implement the UAP using different mechanisms: Java [14] and Smalltalk [13] mostly by convention and manually written accessor methods. Eiffel and Scala [24] make access to instance variables and simple methods indistinguishable for the client. Self [32] and Newspeak [4] hide state completely behind slot accessor. Objective-C [20], C# [1] and Python [34] introduce *properties* that automatically generate accessor methods for instance variables and also allow clients to use classical L-Value assignment syntax rather than procedural invocation to set a property (ECMA-standard Eiffel [7] uses aliases [21] to achieve a similar effect).

Problem Statement: However, the current implementations of the UAP are limited by only applying this uniformity to a single store, state stored in instance variables of objects. Only then do we have generic Load-Update Pairs provided automatically by the language. For state modeled using means other than instance variables, only procedural abstraction is available, which means providing distinct Load-Update Pairs for every individual L-Value to be stored, rather than one LUP applied generically/polymorphically to all such L-Values.

This problem is especially apparent when dealing with external resources, which can only be accessed procedurally despite the fact that filesystems and more generally REST [9] URIs perfectly fit Strachey's model of a store with L-Values and a polymorphic Load-Update Pair (HTTP [8] GET and PUT methods, respectively).

```
1 name;
2 [person name];
3 [person objectForKey:@"name"];
4 [NSStrings stringWithContentsOfFile:@"name"];
5 [[NSData dataWithContentsOfURL:[NSURL URLWithString:
6     @"http://www.person.com/name"]] stringValue];
```

Listing 1. Non uniform access by resource type.

Without going into too much detail, Listing 1 briefly illustrates the problem using just a few of the variations that are implied by

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DLS '13, October 28, 2013, Indianapolis, Indiana, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2433-5/13/10. . . $15.00.
http://dx.doi.org/10.1145/2508168.2508169

	C#	Objective-C
read property	`c = a.b;`	`c = a.b;`
write property	`a.b = c;`	`a.b = c;`
send message	`a.b(c);`	[a b:c];
define property autogenerated	`public int b {get; set; }`	`@property int b; @synthesize b;`

Table 1. C# and Objective-C syntax

different storage mechanisms: retrieving a **name** when it is a local variable or instance variable of the local object, by message, when stored in a dictionary, a file or on a server.

Furthermore, the syntactic sugar that allows clients to treat procedurally implemented storage like L-Values is only superficial, being mapped directly to method invocations. This means that only simple, direct access patterns can treat such a value like an L-Value, indirect and parametrized access has to deal with the procedural abstraction directly. This procedural abstraction both separates the Load-Access Pair for a single L-Value, and at the same time combines the identifier for a specific L-Value with the operation to be performed on it.

We propose that these problems can be solved by putting abstraction over state (stores and L-Values) on an equal footing with procedural abstraction. Instead of having to map directly to per-attribute access methods for non-instance-variable stores, developers should be able to define custom stores with generic, per-store access methods that can then be applied generically. *Polymorphic Identifiers* adapt both the syntax and semantic model of URIs [2] for the definition and use of custom stores within a programming language. With this mechanism, we are able to overcome the limitations of the UAP as currently implemented and naturally integrate custom stores and external resources into our programs.

We first explore the limitations of current approaches to the UAP in section 2, demonstrating that trying to extend the UAP to custom stores or access methods results in code-bloat, performance problems or both. Section 3 introduces Polymorphic Identifiers, which we evaluate in terms of the problems shown earlier in section 4.

2. Non-uniform access

This section provides three resource access scenarios where uniform access is difficult to achieve with standard mechanisms, leading to compromises and non-uniform access patterns in practical code. Please note that this section is not about exhaustively enumerating all possible solutions or conclusively proving that no better solutions exist. Instead it highlights that for common and simple situations, achieving the goals of the UAP is not necessarily simple and obvious, and non-uniform solutions are often viable and commonly used alternatives.

We use Objective-C in our examples because it has both properties and a "message not understood" hook for handling messages without implementing them. Table 1 shows basic Objective-C syntax compared to C#.

2.1 User-defined storage implementations

Although language features supporting the UAP make it easy to substitute a single instance variable with a pair of methods or vice versa, substituting custom stores, and doing so wholesale, is more difficult.

For our examples, we will be using a trivial `Person` class shown in Listing 2 using Objective-C syntax with property definitions for attributes `name` and `age`. We wish to replace the instance variable storage of our attributes with a custom store, in this case a dictionary.

```
1 @interface Person : NSObject
2 {}
3 @property NSString *name;
4 @property NSNumber *age;
5 ...@end
6 @implementation Person
7 @synthesize name,age, ...;
8 @end
```

Listing 2. Person class.

Dictionaries provide a generic interface to access their contents, so they have one method for retrieving objects by key and another method for storing objects by key, in our example `objectForKey:` and `setObject:forKey:` whereas object properties can be accessed using dot notation and the property name. Listing 3 summarizes the differences, it is clear that implementations cannot simply be substituted as is.

```
1 person.name;
2 person.name = newName;
3 person[@"name"];
4 person[@"name"] = newName;
```

Listing 3. Dictionary vs. Property surface syntax.

It should be noted that similar to C# and Eiffel, the syntax in Listing 3 is surface syntax only that is mapped by the compiler to the message sends shown in Listing 4.

```
1 [person name];
2 [person setName:newName];
3 [person objectForKey:@"name"];
4 [person setObject:newName forKey:@"name"];
```

Listing 4. Dictionary vs. Property messages.

There are three basic options for bridging the gap and making access uniform between the dictionary and the object:

1. Listing 5 adapts the object to the dictionary's protocol by implementing `objectForKey:` and `setObjectForKey:` and mapping the keys to properties.
2. Listing 6 adapts the dictionary to the object's protocol by implementing cover methods for every potential attribute that is to be stored.
3. Listing 7 also adapts the dictionary to the object's protocol, but this time by implementing the "forwardInvocation:" error handler that is invoked when a message is not understood by an object.

None of these options is particularly appealing, as they all involve either performance penalties, repetitive code or both.

Adopting the dictionary's protocol as in Listing 5 as the uniform interface between clients and their service providers means that the language's built in notations for access will no longer be used for interfacing between objects, rather `objectForKey:` and `setObject:forKey:` will now be used universally. In addition to bloating client code and making it virtually unreadable, this also means having to implement the matching method pair `objectForKey:`/`setObject:forKey:` for every object. These methods are not just boilerplate that duplicates the already-existing property definitions, they are also more than an order of magnitude slower than the built-in access, and slower even than dictionary access. This performance overhead is imposed by the API, it cannot easily be removed by clever implementations.

```
-objectForKey:aKey
{
  if ( [aKey isEqual:@"name] ) {
    return self.name;
  } else if ( [aKey isEqual:@"age] ) {
    return self.age;
  } else if ...
  }
}
-setObject:newObject forKey:aKey
  if ( [aKey isEqual:@"name] ) {
    return self.name=newObject;
  } else if ( [aKey isEqual:@"age] ) {
    return self.age=newObject;
  } else if ...
  }
}
```

Listing 5. Making an object dictionary-compatible.

In combination, these aspects of this solution strongly discourage object-modeling, as just using dictionaries everywhere is not just less code, but also faster than introducing objects.

```
-(NSString*)name {
    return [self objectForKey:@"name];
}
-(void)setName:(NSString*)newName {
  [self setObject:newName forKey:@"name];
}
-(NSNumber*)age {
    return [self objectForKey:@"age"];
}
-(void)setAge:(NSNumber*)newAge  {
  [self setObject:newName forKey:@"age"];
}
...
```

Listing 6. Making a dictionary object-compatible.

The second option is to keep property access as the universal interface, mapping to dictionaries inside the object using cover methods as shown in Listing 6. The advantages of this approach are that we do no abandon the language-defined interface facilities and there is no performance overhead imposed by the API. The disadvantage is obvious from Listing 6: introduction of duplicated boilerplate code on a massive scale.

Although independent identifiable concerns are supposed to be isolated in a single module [25], and object-orientation is supposed to help us with this goal, this "storage concern" is now spread all over the code-base. If we want to change storage from instance variables to dictionaries or back for a variety of objects, we must modify all the objects involved, introducing or removing these cover methods as needed.

In dynamic languages, we have a third option, using the error handler invoked when a method is not found to simulate the cover methods without having to implement them, as shown in Listing 7. Although this approach does reduce the need to write cover methods for each individual attribute, it still needs to be implemented for every keyed access class or injected if injection facilities are available. The fact that it must convert distinct message names to keys via string processing not only makes it fragile, but also quite slow, at around three orders of magnitude slower than a plain message send accessing an attribute.

Although we have shown three possible ways of honoring the UAP when dealing with custom attribute storage in Objective-C, all of them come with serious drawbacks, and so the fourth, not previously listed option is usually chosen: simply live with non-uniform access for custom stores.

```
-(void)forwardInvocation:(NSInvocation*)inv
{
  SEL sel=[inv selector];
  NSString *msg=NSStringFromSelector( sel );
  if ( [msg hasPrefix:@"set"] ) {
    NSRange  r      = NSMakeRange(3,1);
    NSString *first = [msg substringWithRange:r];
    NSString *rest  = [msg substringFromIndex:4];
    NSString *key   = nil;
    id arg=nil;
    [inv getArgument:&arg atIndex:2];
    first=[first lowercaseString];
    key=[first stringByAppendingString:rest];
    [self setObject:arg forKey:key];
  } else {
    id result=[self objectForKey:msg];
    [inv setResult:&result];
  }
}
...
```

Listing 7. Compatibility via forwardInvocation.

Other object-oriented languages such as C#, Scala, Python, Ruby, Eiffel, Java and Smalltalk, present the developer with the same tradeoffs, though details vary slightly and the dynamic message resolution technique from Listing 7 is only available in the dynamic languages.

The problems presented apply to any custom store, not just dictionaries, because any custom store has to identify the storage locations (*L-Values*) to reference, so it has to translate from messages to *L-Values* and back somehow. Furthermore, they are transitive: they apply just as much if we hide a custom store inside of an object as they do when we substitute the custom store directly for the object.

The special machinery available in languages to translate instance variable storage to messages is of no help here.

2.2 Parametrized access

Another source of non-uniformity is the difference between direct access, where the attribute to be accessed is known at compile time and present in the source code, and parametrized access, where the exact attribute is passed as a parameter at run time. Parametrized access is frequently used in UI toolkits, serialization and persistence mechanisms in order to generically access attributes of an object. For example, a UI text field is parametrized in order to be able to read and write the `name` attribute of an object.

Table 2 summarizes how read and write access to an attribute differs between direct and parametrized access in Objective-C. Property access is the simplest, as there simply is no parametrized variant available. Keyed access can use the same message format for indirect and direct access, and more crucially, the same key for both read and write access. Message-based access, on the other hand, must not only switch from the direct messaging syntax to indirect `perform:` messages, but also must provide two different message selectors, the read and the write selector (e.g. `name` and `setName:`).

In order to have parametrized read/write access to an attribute, we again seem to have effectively three options:

1. Use dictionaries or another keyed store exclusively when parametrized access is required
2. Map a keyed API to message-sends dynamically (option 2 of section 2.1).
3. Implement an "L-Value Reference" object that either captures or dynamically derives the two message required to access the specified attribute relative to an object.

	direct	parametrized
property read	`person.name;`	-
property write	`person.name=newName;`	-
message read	`[person name];`	`[person perform:readAttrSel];`
message write	`[person setName:newName];`	`[person perform:writeAttrSel with:newName];`
keyed read	`person[@"name"];`	`person[anAttribute];`
keyed write	`person[@"name"]=newName;`	`person[anAttribute]=newName;`

Table 2. Direct vs. parametrized access

Option 1 leads to either non-uniform access or standardizing on dictionaries and foregoing object-oriented modeling. It also means that even direct access, with all values known at compile-time, must use the same run time access path, with the order-of-magnitude performance penalty.

Option 2 is feasible, but has the negative performance consequences we looked at in section 2.1. Option 3, performing the computation of the selectors in an object and reusing that object has the potential of being much faster.

```
@interface ValueAccessor : MPWObject
{
  id target;
  NSString *attributeName;
  SEL         getSelector,putSelector;
}
@property (strong) NSString *attributeName;
-initWithTarget:aTarget attribute:(NSString*)attrName;
-value;
-(void)setValue:
@end
@implementation ValueAccessor
@synthesize attributeName;
-initWithTarget:aTarget attribute:(NSString*)attrName
{
  self=[super init];
  getSelector=NSSelectorFromString( attrName );
  NSString *setName=[attrName capitalizedString];
  setName=[setName stringByAppendingString:@":"];
  setName=[@"set" stringByAppendingString:setName];
  putSelector=NSSelectorFromString( setName );
  return self;
}
-value
{
  return [self perform:getSelector];
}
-(void)setValue:newValue
{
  [self perform:setSelector with:newValue];
}
```

Listing 8. Class encapsulating message-based attribute access.

A `ValueAccessor` class implementing this idea is shown in Listing 8. However, as demonstrated in Listing 9, using this `ValueAccessor` class is significantly more verbose and less convenient than just using the direct keyed access method of option 2. This means that it is unlikely to be used in a direct-use setting, so transformation from direct to parametrized use implies a large change in client code.

In addition, we have to take into account the findings from section 2.1, which concluded that we are likely to not have uniform access based on messages only, which option 3 relies on.

Again, we find that although we have several solutions that offer different trade-offs, no obvious method presents itself for achieving uniformity of access when parametrized access is involved. In fact the problems of not having uniform direct access compound the problems of parametrized access.

```
id valueAccessor=[[ValueAccessor alloc]
           initWithTarget:person
           attributeName:@"name"];
a=[valueAccessor value];
```

Listing 9. Keyed access vs. using a ValueAccessor

2.3 External resources

Interaction with resources outside the application's memory space keeps increasing in importance, for example Apple's Pages DTP application has 3.9MB of compiled code, but 250MB of non-code resources in 18792 files (not counting frameworks), and mobile applications are often just thin wrappers around web-resources accessible via HTTP [8]. However, API and language support for abstracting over such resources is limited. In a way that is similar to the custom stores we looked at in section 2.1, languages make it difficult to create APIs that separate concerns and respect the UAP.

Listing 10 shows loading an image file from a file using a convenience API. An instance of the `NSBitmapImageRep` class is initialized by reading the file from the absolute file-system path `/Users/john/button.png`.

```
image=[NSBitmapImageRep imageRepWithContentsOfFile:
                            @"/Users/jo/button.png"];
```

Listing 10. Accessing an image from the file system

Rather than hiding implementation details and allowing them to be varied independently, the API used in Listing 10 requires the caller to expose and determine virtually all implementation details right at the call site:

1. The fact that we are loading data from a filesystem
2. The name of the image and the full access path to the image, encoded as a literal string, so the API assumes that strings are filesystem paths
3. The fact that the image is encoded in the PNG file format
4. The class that will be used to represent the image in code

If we wish to separate the resource name from its access path, we need to do string processing to combine the two back into a full path. Using relative paths is dangerous at best because those are relative to the current working directory, which is global to the process and not necessarily predictable.

If we wish to load the image from the web rather than from the local filesystem, we need to change APIs, as shown in Listing 11. Being URI-based, the API in this example can also be used to load images from the filesystem, but its additional complexity and verboseness make the previous convenience API preferable as long as it is sufficient.

Similarly, if we decide to use a vector file format such as PDF, we need to change the class to `NSPDFImageRep` and also need to change the file name. Finally, if we decide to create the image

```
uristring=@"http://example.com/button.png"
uri=[NSURL URLWithString:uristring];
image=[NSBitmapImageRep imageRepWithContentsOfURL:uri];
```

Listing 11. Loading image from web

programmatically rather than loading it from a file, we must change to a message send `[imageProvider button];`.

We can improve the situation a little by introducing a `Resourcer` class like the one shown in Listing 12, which hides differences between different access methods and base paths, exposing just the resource name itself.

```
@interface Resourcer : NSObject
{
        NSURL *baseURL;
}
-initWithBaseURLString:(NSString*)str;
-objectForKeyedSubscript:key;
-(void)setObject:newValue forKeyedSubscript:key;
@end

@implementation Resourcer

-initWithBaseURLString:(NSString*)str
{
    self=[super init];
    baseURL=[NSURL URLWithString:str];
    return self;
}
-(NSURL*)urlForRelativePath:(NSString*)path
{
    return [NSURL URLWithString:path relativeToURL:baseURL];
}
-objectForKeyedSubscript:key
{
    NSURL *loc=[self urlForRelativePath:key];
    return [NSData dataWithContentsOfURL:loc];
}
-(void)setObject:newValue forKeyedSubscript:key
{
    NSURL *loc=[self urlForRelativePath:key];
    [newValue writeToURL:loc atomically:YES];
}
@end
```

Listing 12. Resourcer implementation

Once we have a `Resourcer` instance, access to a specific resource is simple: `button=resources[@"button.png"];` With an additional `Mapper` (not shown) we can map specific resource types to classes, so retrieving a resource using the syntax above doesn't just return raw bytes but initialized instances.

However, as `resource[@"button.png"]` is still syntactically different from `resource.button`, our solution does not conform the UAP, so if we decide to compute the button image instead of using a pre-rendered version, we still need to change client code. At this point we have the same three options discussed in section 2.1 for bridging the gap between storage and computation: (1) standardize on dictionary syntax, (2) create individual cover methods or (3) use a message error handler.

In addition to the problems with these solutions already discussed, using messaging to access external resources by name has the additional issue that resource names frequently don't conform to language messaging syntax, so either some resources are unreachable or some mapping must be created and maintained.

Furthermore, paths with multiple components are difficult to map to messaging without running into stratification issues: either the messages are immediately resolved to a resource, in which case paths with multiple components cannot be expressed, or the messages just build a path, which must then be resolved to an actual resource separately, preferably using a message name that doesn't conflict with a potential path component.

This section highlighted a small cross-section of the problems developers encounter when dealing with external resources. There are many more and different APIs available, the very abundance of these APIs suggesting that the problem is not solved.

2.4 Problem Statement Summary

Although we managed to conform fully or at least partially to the UAP in most of these examples in the end, the results have been highly unsatisfactory. Why? First, the significant issues identified with these solutions make it unclear if conforming to the UAP is worth the cost. Secondly, the very diversity of the solutions, with little to clearly mark one as favorite, works against the uniformity that the UAP is trying to achieve.

As we saw in Sections 2.1 and 2.2, the difficulty we had with custom stores and references was having to map between a message that performs an operation on a specific L-Value (`setName:`) and the combination of the L-Value itself (`name`) and the operation (`set:`).

Language support for the UAP automates this mapping, but only in a few special cases: when the store being mapped to is instance variable storage and when the external access is direct. In all other cases, some of which we have presented here, existing language support is insufficient and the UAP unravels.

3. Polymorphic Identifiers

Polymorphic Identifiers extend the equivalence between storage and computation promised by the UAP from built-in instance variable storage to user-defined stores including the filesystem and remote storages such as the web.

Conceptually, Polymorphic Identifiers add multiple, user-defined stores each with their own kind of L-Values and associated identifiers. The built-in stores such as instance variables fit within this framework but do not have special status. Each store defines its own kind of Load-Update Pair (LUP) [31], rather than having such pairs defined for each individual L-Value.

The Polymorphic Identifier approach consists of three basic parts, which were implemented as part of Objective-Smalltalk [35], a Smalltalk implementation based on the Objective-C runtime:

1. *Polymorphic Identifiers* (PI) themselves are URIs[1], with the scheme-part of the URI designating the actual scheme handler.
2. *References* are created and used transparently by the run-time and compiler to mediate access to a specific L-Value identified by a *Polymorphic Identifier*. They correspond to the LUP in Strachey's model.
3. *Scheme-handlers* each manage a single store, a set of L-Values. Scheme-handlers translate from identifiers to references.

Programming languages already support the notion of multiple stores. Even in C, the identifier `a` can mean a local variable of a procedure, which could be in registers, or a global variable stored in memory at a specific address. With OO languages, `a.b` can be direct access to the instance variable `b` of object `a`, or it can mean calling a method aliased to that identifier. Polymorphic Identifiers regularize and generalize these ad-hoc mechanisms.

3.1 URI syntax

The PI conceptual model maps almost perfectly to URI syntax and semantics, with the *scheme* part of the URI designating a store and the URI representing an L-Value with polymorphic read and update operations (`GET` and `PUT` in HTTP). PIs bring URIs into

[1] Other syntax variants such as dot notation should also work.

the language, with compiler and runtime support. In addition to basic URI syntax, we also support URI templates [15] in order to support partially varying identifiers with dynamically evaluated components.

For a programming language, URI syntax is a somewhat unusual as it uses the forward slash ('/') character as a path delimiter, whereas most programming languages prefer the dot ('.'). However, it makes it easy to integrate both external and internal resources with a common syntax, as shown in Listing 13.

```
person
name
var:person/name
var:person/{attribute}
file://tmp/button.png
http://www.example.com/button.png
file:{env:HOME}/rfcs/{rfcName}
```

Listing 13. Valid Polymorphic Identifiers

URI syntax is very general, for example XPath [5] query language for XML is expressible as URIs, making it possible to express queries statically as identifiers rather than operationally.

3.2 References

References are the expression of Strachey's Load-Update Pair within the Polymorphic Identifier system, they mediate access to a specific L-Value. Similar to the `ValueAccessor` class presented in section 2.2, they hide variations in access mechanics from clients, but due to compiler support without negative impacts in usability or performance. References can also be exposed to the user for generic indirect/parametrized access, with the potential for store-specific behavior.

Although References can in theory be arbitrary, whatever is needed to access an L-Value, we have identified three common cases: indexed references that are offsets to their context, for example for local or instance variables within an object, messaging references that send L-Value-specific messages, and finally keyed messages that send generic access messages along with a key specifying the L-Value in question.

Listing 14 shows conceptually how the actual `Reference` implementation that is used for most in-memory references differs from the `ValueAccessor` class shown if Listing 8, with the `value` method combing the three modes of access: first the integer offset, then messaging, and finally keyed access. Which specific access method is chosen is determined at binding time, with cooperation from the scheme-handler, the target object, and the source of the reference.

The key to achieving good performance despite the extra indirection introduced by the References is that binding can be separated from evaluation and performed earlier, for example the first time an object of a specific class is referenced. The effect is similar to JIT compilers for late-bound messaging inlining accessors, but without the overhead of compiling during code execution or the security risks of adding code at run time. When the target object and its client agree, binding can occur completely at compile time, compiling away any overhead.

The generic interface to arbitrary L-Values that are References can also be exposed to user programs to fill the need for parametrized references we discussed in section 2.2. Listing 15 shows example use of a reference that is the equivalent of Listing 9.

References have generic behavior in addition to their basic Load and Update (GET/PUT, value/setValue:) operations: assignment operations are not handled by the language, but passed on to the references in questions so copy operations can be optimized. Examples include streaming data from a web-resource to a file instead

```
@interface Reference : MPWObject
{
  id target;
  int offset;
  NSString *attributeName;
  SEL         getSelector,putSelector;
}
...
@end
@implementation Reference
...
-value
{
  if ( offset >= 0 ) {
     return ((id*)target)[offset];
  } else if ( getSelector ) {
      return [target perform:getSelector];
  } else {
     return [target objectForKey:attributeName];
  }
}
...
```

Listing 14. Reference class extract

```
valueAccessor := ref:person/name.
name := valueAccessor value.
```

Listing 15. Using a first class reference

of loading it fully into memory first or performing a copy operation on a remote server without first downloading the contents and then uploading it again. References are also capable of listing their child references, making generic browsing and export to HTTP and file systems possible.

In addition, references can exhibit additional behavior that is specific to a particular store. For example, storing a reference to file in another file results in the creation of a symbolic link, and file references can retrieve and manipulate meta-data about the file in question.

3.3 Schemes and scheme-handlers

Schemes in Polymorphic Identifiers serve almost the same role as schemes in URIs in general: they visibly partition the identifier namespace and allow specific identifiers to be associated with specific scheme-handlers. Scheme-handlers manage a specific store, examples include object instance variables, local variables, dictionaries, files and web resources. In addition, composite scheme-handlers can be used to create new stores by modifying access to an underlying store, by transforming identifiers or data coming in and out of the store. These mechanisms provide a rich toolset for abstracting from details of particular store and providing interoperable abstractions on storage.

Unlike Uniform Resource Locators (URLs) [3], URIs do not specify an access path to a specific location, but rather specify a name that can then be mapped to a location. Schemes in this context are also not protocol specifications and do not specify an actual store, even though they may appear to do so. Instead, scheme names provide user-visible partitions of the name space that are mapped dynamically to specific scheme handlers via the special `scheme:scheme` that contains scheme-handlers.

The expression `scheme:http := URLSchemeResolver new` binds the `http:` scheme to an instance of the `URLSchemeResolver` class. With this definition in place an identifier such as the previously introduced `http://www.example.com/button.png` will actually be resolved by that URLSchemeResolver instance. Although users can just as easily bind the `http` scheme to a different

scheme-handler, possibly one using completely different storage or transport or one performing additional actions before sending an HTTP-request, we will use default scheme-bindings to refer to scheme-handlers for the remainder to ease the exposition.

Listing 16 shows how a web-resource can be downloaded to a file or updated from a string literal using plain assignment syntax. This type of compact syntax for dealing with files or web resources is usually reserved for scripting languages such as the Bourne shell. In fact, Objective-Smalltalk is used with minor modifications as a Unix shell and scripting language `stsh`. Expressions such as `file:{env:HOME}/rfcs/{rfcName}`, which resolves to an RFC specified in the variable `rfcName` located in the `rfcs` subdirectory of the user's home directory, are very close to the expressiveness (and obscurity) of the equivalent Bourne shell expression: `$HOME/rfcs/$rfcName`.

```
1 file:rfc1738:=http://datatracker.ietf.org/doc/rfc1738.
2 http://datatracker.ietf.org/doc/rfc1738:='New RFC'.
```

Listing 16. Downloading an RFC to a file.

Polymorphic Identifiers bring compact, shell-like expressiveness for file and web access to a general purpose programming language not as special-cases, but rather as part of a general framework for making resource access more uniform.

3.3.1 In-memory stores

Although adding file and web references as integrated, first class objects to a programming language is useful, it solves only parts of the problems from section 2. In addition, we also wish to integrate classical variable and attribute access.

In-memory stores such as the local execution context (often the stack), object instance variables, thread-local and global heap variables are also managed by scheme-handlers and made available using Polymorphic Identifiers. Listing 17 shows several examples, `ivar:` for instance variable access, `local:` for the current method context ("stack"), and `thread:` for thread local variables.

```
1 local:tempAnswer := ivar:myAnswer.
2 thread:deepThought/privateAnswer := local:tempAnswer.
```

Listing 17. Different memory variables.

These schemes allow path-based access that is mediated by the objects in question. They return references of the type shown in Listing 14 that can be used for keyed or message-based access or even for accessing directly via an offset into the object.

There is also a `global:` scheme for globals and a `class:` scheme that separates the namespace for instances from that for classes. Having separate schemes for these different storage classes is useful when wanting to disambiguate, but can become cumbersome in itself. To alleviate this, the composite `var:` scheme combines these schemes into a single namespace with prioritized lookup similar to conventional languages.

The `var` scheme is also usually bound as the resolver for the `default` scheme, which is assigned to identifiers that do not specify their scheme. Combining the sequential lookup of the `var:` scheme and the `default` scheme allows us to implement the usually hard-coded identifier lookup rules for plain identifiers in Smalltalk and other languages using our toolbox of scheme-handlers.

In addition to the schemes that recreate existing variable lookup, a number of other useful scheme-handlers have been implemented so far. We've already seen the `env:` scheme, which provides access to Unix environment variables.

A further scheme-handler that is not pre-loaded but available for defining custom schemes is the `SiteMap`. A `SiteMap` stores arbitrary objects in memory in a tree structure that acts a bit like a memory filesystem.

3.3.2 Composite scheme-handlers

In addition to the flexible mapping of scheme names to scheme-handlers discussed earlier, composite schemes provide another means of abstracting from the specifics of storage and providing a uniform interface that hides implementation details such as location, representation or access patterns. Composite schemes never provide their own storage, but rather mediate access to one or more underlying *base schemes*.

A *relative scheme* modifies access to its base scheme by making access relative to a base URI, almost exactly like the `Resourcer` class in Listing 12. Listing 18 shows the definition of an `rfc:` scheme that points to the IETF web-site. With this scheme mapping in place, programs can refer to individual RFCs using the identifier `rfc:rfc2396`.

```
1 base := ref:http://datatracker.ietf.org/doc .
2 scheme:rfc:=RelativeScheme schemeWithBase: base.
```

Listing 18. Defining a custom rfc: scheme.

Relative schemes are the basis for decoupling application-level intent from specific resource implementations and most schemes in use in a program will be user-defined schemes that at some point resolve to a base scheme via a relative scheme.

Rather than modifying the identifier, a *filter scheme* modifies the data coming from or going to its base scheme, similar to stacked filesystem that provide transparent encryption or compression services. The `Mapper` class mentioned earlier that deserializes raw bytes into application-specific objects based on their MIME type and a user-defined table mapping MIME types to classes is a prime candidate for such a filter.

Sequential schemes were already introduced in the previous section when defining the `var:` scheme, they just search a number of base schemes for the L-Value in question. A *caching scheme* also scans two base schemes sequentially, but deposits any values found in the scheme scanned first. Fig. 1 shows how multiple caching schemes can be combined with file and memory stores and an actual `HTTP` protocol handler handlers to construct an `http:` scheme-handler that caches retrieved resources both on disk and in memory.

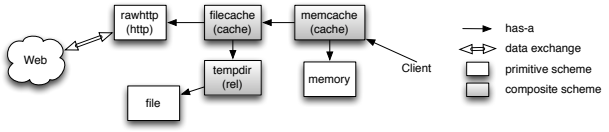

Figure 1. `http:` scheme-handler with caching, composed from simpler schemes

Composite scheme-handlers make it possible to abstract from many details of storage such as location, optimized access mechanisms and data formats while at the same time enabling expressiveness very similar to specialized scripting languages in a general purpose programming language.

4. Evaluation

Having introduced Polymorphic Identifiers, we must now show whether they solve the problems we had in section 2 implementing

the Uniform Access Principle, particularly if we can provide uniform access to custom stores without requiring code-bloat at either the call site or the called object. In addition, it should be possible to integrate references to external files, abstracting from specific location and media types. Finally, all this should be accomplished without major impacts to performance.

First, let's look at substituting a dictionary with an object or vice versa. Listing 19 shows how to access the `name` property of an object or a dictionary using Polymorphic Identifiers. The access syntax is the same, because the `var:` scheme handler uses the `Reference` class we showed earlier that uses messaging or keyed access transparently.

```
1 name := var:person/name
2 var:person/name := newName.
```

Listing 19. Accessing a dictionary or object via PI.

We have already shown how to provide a reference to an object via the `ref:` scheme, so getting a reference to the `person` object is as simple as `valueRef := ref:person`, which is significantly less code to write than the code in Listing 9. As before, this uses the `Reference` class to perform the access, so it works with both keyed access and messaging.

External resources are slightly more challenging. We could use `button := http://www.example.com/button.png`, but that actually still encodes path information. Better to create a `image:` relative scheme pointing to `http://www.example.com`. With that, we refer to the button as `image:button.png`, which is much easier to point to another location, for example by initializing the `image:` scheme with a file path. Furthermore, the image: scheme should also probably include a filter that converts PNG image data to image objects, so the object retrieved with `image:button.png` is directly usable. Finally, we can add another filter that automatically adds the file extension, leaving the identifier `image:button`. With these few simple steps, we have not only made the original image reference more compact and expressive, we have also abstracted from the data source, format and location sufficiently that we can replace the file retrieval with a computation if we want to do so.

Using Polymorphic Identifiers to access resources makes conformance to the UAP automatic for in-memory stores such as objects and dictionaries, independent of whether access is direct or parametrized, and without trade-offs in terms of duplicated boilerplate code. Although not quite as automatic as the in-memory stores, PIs also make external resources such as files accessible to the UAP with very little effort.

4.1 Performance

In addition to duplicated boilerplate code, one of the problems with the methods for achieving uniform access in section 2 was performance. Table 3 compares the performance of the different methods for compensating the differences between a dictionary as a custom store and an object using native instance variables and messages to access them.

Each column of table 3 represents one (uniform) access method, with the rows being the underlying stores that are being accessed. Times are reported in nanoseconds per access. Tests were run on a 2012 MacBook Pro 13" with 2.9 GHz Core i7 processor, the code was self-timing using the `getrusage()` call and an automatically adjusted iteration count that ensures each test runs for at least 1/10th of a second. Three runs were averaged.

The *Message* column shows the times for using messages or properties to access, so `person.name` for the name property of the person objects. This access methods is native for the object but requires cover methods for the dictionary. With object based storage, it is the second fastest access method overall, with dictionary based storage the third fastest.

The *Keyed* column shows the times for keyed access, which is native for the dictionary but requires a cover method mapping keys to messages for the object. As noted earlier, keyed access for object-based storage is actually 2x slower than pure dictionary access and 20x slower than native object access, so using keyed access is not tenable from a performance point of view.

The times for Polymorphic Identifiers are shown in two columns. The first, labeled *Polymorphic Identifier* shows access times when the `Reference` object negotiates message-based and keyed access. These are each slightly slower than the access underlying access mechanism, 25% for object based storage (third fastest) and 7.5% for dictionary based storage (second fastest for dictionaries).

The second column, labeled *Polymorphic Identifier/Offset*, shows the access time when the `Reference` object is able to negotiate direct access. This is 12% faster than using a message, and fastest overall, despite the fact that it retains even more flexibility than a message send. Direct access to a public instance variable would of course be faster still, but would not be provide uniform access.

The other alternative for unifying access via messaging, using the `forwardInvoction` takes over 2 microseconds for the case of underlying dictionary storage so isn't a viable option. It is also left out of Fig. 2 comparing the results visually, because it wouldn't fit. In fact, 2 microseconds is sufficiently slow that it is the only access method that has a noticeable impact on small file operations, adding more than 25% to the time for a cached read of a 36KB file.

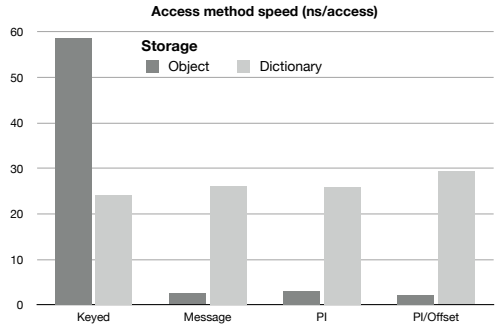

Figure 2. Performance

In performance terms, Polymorphic Identifiers are faster than or at least competitive with the fastest of the methods for achieving uniform resource access that we looked at. However, those comparisons are with methods that were deemed unacceptable due to the code bloat introduced. When comparing with methods that have a similar code footprint, performance of Polymorphic Identifiers is much better, ranging from 2x to 80x better.

4.2 Applications

In addition to scripting tasks using `stsh`, Polymorphic Identifiers have been used in a number of applications, ranging from an iPad language learning game, the site generator that produces the `objective.st` web-site to a live remote IDE that uses HTTP to upload code and download debug information from running processes. Fig. 3 has three IDE windows that show, respectively, the setup code for the web-site, the part of the IDE code itself that uploads new method definitions, and finally a window with a live

Storage	Keyed	Message	Polymorphic Identifier	Polymorphic Identifier/Offset	forwardInvocation
Object	58.7	2.50	3.11	2.22	2.48
Dictionary	24.2	26.2	25.9	29.3	2051

Table 3. Time per access in nanoseconds for different access methods and stores

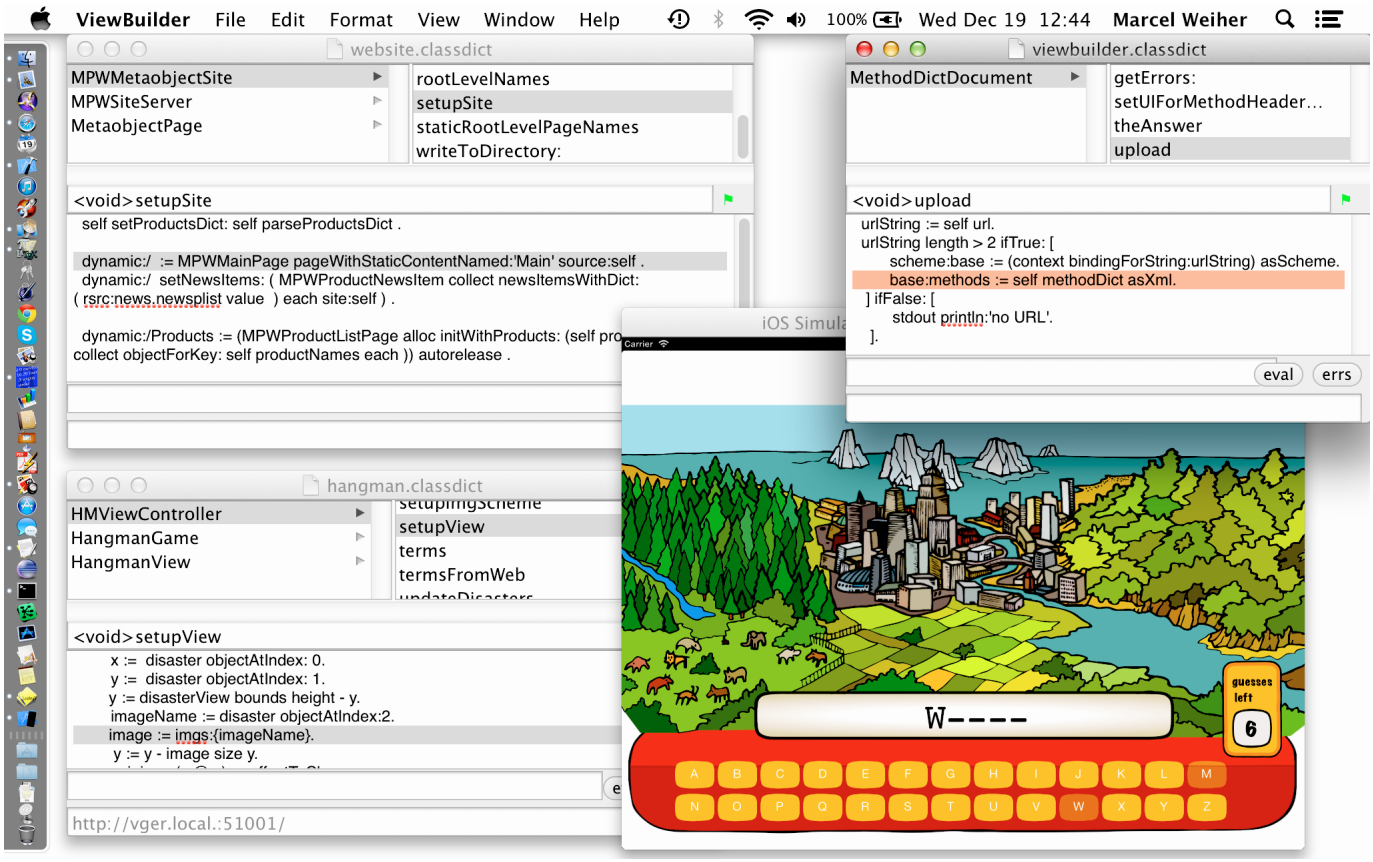

Figure 3. Polymorphic Identifiers in use

connection to the language learning game running on the iOS simulator in the background.

The game has a large number of graphical assets that could be used much more simply in code by using special schemes for accessing app resources, though the overall impact was not very high due to large amounts of layout logic. The IDE was initially completely written in Objective-C, but dealing with remote resources was so cumbersome that those parts were rewritten using Objective-Smalltalk with Polymorphic Identifiers.

The application that gained most from Polymorphic Identifiers was the site generator: the asset pipeline consisting of static content, templating mechanism, dynamic content and a page cache could be directly expressed as a series of connected scheme-handlers. The object that dynamically generates the site's root is placed there using the simple assignment: `dynamic:/ := MPWMainPage ...`.

5. Related Work

Self [32] and Newspeak [4] take the messaging approach to its logical conclusion by making all identifiers message-sends and therefore late-bound, eliminating all nouns and replacing them with verbs[2]. Although this elimination of L-Values as a simplification is conceptually elegant, it leads to the problems we have shown, because whereas an identifier associated with an L-Value has a Load-Update Pair, an identifier associated with a message can only be executed, so Load and Update must be separate identifiers. The fact that these separate identifiers are only related by convention and must be derived from each other at times "left a complexity that bothers us to this day" [33]. Handling the assignment part of resource access leads to ad-hoc rules and mechanisms to tie together the slot accessor methods that Ungar and Smith say "troubled us a bit" [33].

Polymorphic Identifiers are a result of the same basic premise that identifiers should always be late-bound. However, it doesn't follow the apparent implicit assumption that only messaging can be late-bound and therefore identifiers must be messages in order to be late-bound, and the related assumption that interfaces must therefore be message-based.

This assumption certainly doesn't apply to the World Wide Web and the REST architectural style [9], where it is (universal) resource identifiers that are the interfaces, and messaging is the hidden implementation detail. This approach is taken to its logical

[2] Literals are the exception

extreme by by Resource Oriented Computing [12], which encodes all computation into identifiers, with an `action:` scheme that is "a functional programming language encoded as a URI".

Although scheme-handlers can be exported via http, Polymorphic Identifiers leave the decision of whether to present a messaging or a resource-based interface to the developers, similar to properties in C# [1] and Objective-C 2.0 [20], which closely match the proposal in [29]. Properties are syntactic sugar for a pair of accessor methods and allow clients to use plain identifiers syntax to access values via message-sends, including assignment for setting the value. However, properties are just syntactic sugar for one type of resource access, they are not user-extensible like Polymorphic Identifiers and don't integrate access to external resources or first class references.

Common LISP [30] has `symbol-macrolet`, which was explicitly limited to lexical scoping due to fears that having simple identifiers with overloaded meaning could be confusing [11], echoing similar concerns by the creators of Smalltalk-72 of syntax that was too flexible [19]. Having clear syntactic markers in the form of scheme-prefixes and directly associating each prefix with one scheme-handler in Polymorphic Identifiers avoids confusion as to the meaning of specific identifiers.

The E language [23] supports URI-Expressions as a direct language feature using angle brackets for access to resources such as files: ⟨`file:/home/marcs/myFile.txt`⟩, and to the underlying Java classes: ⟨`unsafe:java.util.makeCalendar>.getYEAR()`⟩.

E even allows custom schemes to be defined, but only for read-access, separate from other identifiers and without the ability to extract references.

A different approach to resource access comes from the operating system community: Plan9 integrates a wide variety of local and remote [27] resources and services into a single directory tree [26] that is made available on a per-process basis, but mediated by the kernel and accessed only indirectly via system calls and string-based identifiers. User level filesystems like FUSE [28] or the BSD Pass-to-User-Space [18] system bring some of these ideas to commercial operating systems, but make these filesystems visible globally to all processes on a machines.

Polymorphic Identifiers are similar to Embedded Domain Specific Languages [17] in that they allow domain-specific language elements to be added to a language, rather than having to create a completely new language with an external DSL or attempt to achieve the desired effect with an internal DSL[10].

Like polymorphic embedding of DSLs [16], Polymorphic Identifiers allow a single syntax to be used with multiple, pluggable semantic interpretations permitting composition of functionality [6]. However, Polymorphic Identifiers are applicable to general purpose programming languages, not just DSLs, while at the same time restricting their focus to the identifiers used.

6. Summary and Outlook

We have introduced Polymorphic Identifiers, a mechanism for abstracting over storage in a way similar to the way that messaging abstracts over computation. Polymorphic Identifiers solve problems encountered when attempting to extend the Uniform Access Principle to custom stores, parametrized access and external resources.

The uniformity of the syntax and the use of common interfaces for scheme-handlers enables polymorphic behavior for resource access, permitting different scheme-handlers to be substituted without affecting client code. Application-specific schemes make it easy to hide storage details such as access mechanisms, protocols or paths. Custom scheme-handlers can be implemented as normal objects and added to the language, or even composed from pre-existing handlers and combinators.

With the user-extensible identifier architecture of Polymorphic Identifiers, it becomes possible to add abstraction and information-hiding capabilities to identifiers and expand the use of REST-style programming beyond network and Web-environments.

Acknowledgements

This paper is based upon work supported in part by the Japan Society for the Promotion of Science (JSPS) Invitation Fellowship Program for Research in Japan.

References

[1] T. Archer. *Inside C#*. Microsoft Press, Redmond, WA, USA, 2nd edition, 2002. ISBN 0-7356-1648-5.

[2] T. Berners-Lee. RFC 2396: Uniform Resource Identifiers (URI). Technical report, MIT, 1998. URL `http://www.rfc-archive.org/getrfc.php?rfc=2396`.

[3] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). RFC 1738 (Proposed Standard), Dec. 1994. URL `http://www.ietf.org/rfc/rfc1738.txt`. Obsoleted by RFCs 4248, 4266, updated by RFCs 1808, 2368, 2396, 3986, 6196, 6270.

[4] G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as objects in Newspeak. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 405–428, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14106-4, 978-3-642-14106-5. URL `http://dl.acm.org/citation.cfm?id=1883978.1884007`.

[5] J. Clark and S. DeRose. XML path language (XPath) version 1.0. W3C recommendation, W3C, Nov. 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[6] T. Dinkelaker, M. Eichberg, and M. Mezini. An architecture for composing embedded domain-specific languages. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 49–60, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-958-9. . URL `http://doi.acm.org/10.1145/1739230.1739237`.

[7] ECMA. *ECMA-367: Eiffel analysis, design and programming language*. ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA:adr, June 2005. URL `http://www.ecma-international.org/publications/standards/Ecma-367.htm;http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-367.pdf`.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616, hypertext transfer protocol – http/1.1, 1999. URL `http://www.rfc.net/rfc2616.html`.

[9] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. AAI9980887.

[10] M. Fowler. *Domain Specific Languages*. Addison-Wesley, September 2010.

[11] R. P. Gabriel. Concerns on identifier confusion in Common LISP. Personal communication, 12 2012.

[12] T. Geudens. *Resource-Oriented Computing with NetKernel*. O'Reilly Media, May 2012.

[13] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison Wesley, 1983.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005. ISBN 0321246780.

[15] J. Gregorio, R. Fielding, M. Hadley, M. Nottingham, and D. Orchard. URI Template. RFC 6570 (Proposed Standard), Mar. 2012. URL `http://www.ietf.org/rfc/rfc6570.txt`.

[16] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of dsls. In *Proceedings of the 7th international conference on Generative programming and component engineering*, GPCE '08, pages 137–148, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-267-2. . URL `http://doi.acm.org/10.1145/1449913.1449935`.

[17] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28, December 1996. ISSN 0360-0300. . URL `http://doi.acm.org/10.1145/242224.242477`.

[18] A. Kantee. puffs - Pass-to-Userspace Framework File System. In *AsiaBSDCon*, 2007. URL `http://2007.asiabsdcon.org/papers/P04-paper.pdf`.

[19] A. C. Kay. History of programming languages—II. chapter The early history of Smalltalk, pages 511–598. ACM, New York, NY, USA, 1996. ISBN 0-201-89502-1. . URL `http://doi.acm.org/10.1145/234286.1057828`.

[20] S. Kochan. *Programming in Objective-C 2.0*. Addison-Wesley Professional, 2nd edition, 2009. ISBN 0321566157, 9780321566157.

[21] B. Meyer. EiffelWorld Column. URL `http://www.eiffel.com/general/column/2005/Sept_October.html`.

[22] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988. ISBN 0136290493.

[23] M. S. Miller. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Baltimore, MD, USA, 2006. AAI3245526.

[24] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008. ISBN 0981531601, 9780981531601.

[25] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972. ISSN 0001-0782. . URL `http://doi.acm.org/10.1145/361598.361623`.

[26] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name space in plan 9. *Operating Systems Review*, 27(2):72–76, April 1993. URL `http://plan9.bell-labs.com/sys/doc/names.pdf`.

[27] D. Presotto and P. Winterbottom. The organization of networks in plan 9. In *USENIX Conference*, pages 271–280, 1993.

[28] A. Rajgarhia and A. Gehani. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 206–213, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-639-7. . URL `http://doi.acm.org/10.1145/1774088.1774130`.

[29] D. Spinellis. A modest proposal for curing the public field phobia. *SIGPLAN Not.*, 37(4):54–56, Apr. 2002. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/510857.510868`.

[30] G. L. Steele, Jr. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990. ISBN 1-55558-041-6.

[31] C. Strachey. Fundamental concepts in programming languages. *Higher Order Symbol. Comput.*, 13:11–49, 1967,2000. ISSN 1388-3690. . URL `http://portal.acm.org/citation.cfm?id=609150.609208`.

[32] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA*, pages 227–242, 1987.

[33] D. Ungar and R. B. Smith. Self. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 9–1–9–50, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7. . URL `http://doi.acm.org/10.1145/1238844.1238853`.

[34] G. van Rossum. Python reference manual. Report CS-R9525, Apr. 1995. URL `http://www.python.org/doc/ref/ref-1.html`.

[35] M. Weiher. Objective-Smalltalk: `http://objective.st`, 11 2010. URL `http://objective.st`.