Newspeak Programming Language Draft Specification Version 0.101

Gilad Bracha

June 6, 2021

Contents

1 Introduction

Caveat Emptor

This document is a **draft** *specification. It is incomplete, and everything is subject to change. Please ensure you are reading the latest draft. Certain language features described herein have not been implemented; in other cases, the implementation deviates from this specification. When such features are discussed, we will endeavor to note this clearly in the text.*

Notation and Conventions: Sections that provide design rationale or examples appear in *italics*.

When we refer to a class with a specific name n, we mean the class n defined by the modules of the underlying Newspeak platform (8.3).

Newspeak is a programming language in the Smalltalk [GR83] tradition. Newspeak is:

- *Network-serviced.* Newspeak applications can be updated over the internet *while running;* the language supports *orthogonal synchronization,* making it straightforward to synchronize persistent data with a remote server, supporting backup, sharing and collaboration [Bra]. *The synchronization features are in their early design stages, and only partially implemented. See section 6.3.3*

- *Message-based.* All computation - even an object's own access to its internal structure - is performed by sending messages to objects. Hence, everything in Newspeak is an object, from elementary data such as numbers and booleans up to functions, classes and modules.

- *Secure.* Newspeak objects encapsulate their representation, and Newspeak programs have no static state, providing a sound basis for an object-capability security model [Mil06].

- *Reflective.* Newspeak programs are causally connected to their executable representation via a reflective API. Reflection in Newspeak is mirror based [BU04], with mirrors acting as capabilities. Given access to the appropriate mirrors (and only given such access), a running program and can both introspect and modify itself.

- *Modular.* Newspeak module definitions are independent, immutable, self-contained parametric namespaces. They can be instantiated into modules which may be stateful and mutually recursive. These modules are inherently re-entrant, because there is no static state in Newspeak. All inter-module dependencies are explicit. Modules and their definitions are first class objects that can be manipulated at run time.

- *Concurrent.* Concurrency in Newspeak is based on *actors.* Actors are objects with their own thread of control. They share no state with other actors; they communicate exclusively via asynchronous message passing. Actors are non-blocking, race-and-deadlock free, and scalable. *Note that the FFI (8.5) can undermine actor isolation as C can take state passed from one actor, store it globally, and return it to another actor. Non-blockingness also requires care, as a callback passed in by one actor can be invoked when C is called by another. Must ensure that said call back acts as a future, or fails (the former, to allow event processing). In an ideal world, one would only communicate with foreign languages running in a distinct actor. This would be more secure, and require less special handling; this was part of the original vision of Smalltalk. Newspeak is pragmatic in this regard; it remains idealistic, but only to an extent.*

- *Optionally typed.* Newspeak supports pluggable types [Bra04], allowing the language to be extended with arbitrary type systems. These type systems are necessarily optional, and never affect run-time semantics. They utilize Newspeak's metadata facility (4.3), which allows annotations to be attached to any node in a program's abstract syntax tree. *Unimplemented.*

2 Overview

In this section, we provide a quick introduction to those properties of Newspeak that are unusual, in order to provide intuition when reading the specification proper. The normative part of this specification begins with section 3.

2.1 Terminology

We follow Smalltalk/Self/Objective-C terminology in using the term *message* to refer to both synchronous and asynchronous messages. A synchronous message *send* is like a virtual method call. A message must be sent to a *receiver*. Some readers may prefer to think of the receiver as the target of a method invocation.

Our use of the term *message* may be slightly non-standard, but it conveys a valuable intuition about loose coupling. The term *method* does not; common usage allows for methods that are static, final/non-virtual etc.

2.2 Syntax

The syntax of Newspeak is close to those of Smalltalk and Self [US87]. We expect some details to change over time, to make the language somewhat more familiar to most programmers. However, we intend to retain key features of the Smalltalk syntax where they confer a real advantage. Here we highlight the differences between Newspeak and other languages briefly.

2.2.1 Object Member Selection

In most languages, a notation such as `o.m` or `o.m()` is used to denote a member `m` of an object `o`. The object member selection operator is dot, and if the member is a method the member name is followed by a parenthesized parameter list.

In Newspeak, as in Smalltalk, the only operation on objects is member selection, and so the dot conveys no information. On the contrary, it is redundant, and makes it harder to embed domain specific languages within Newspeak. Therefore, member selection is implicit: we write `o m` for object member selection. Member selection always means sending a message (invoking a method), and there is no need for parentheses to distinguish field access from method access. Hence, references to methods without arguments are not distinguished from references to slots (aka fields) or to nested classes.

2.2.2 Parameter Lists

As noted above, if there are no parameters, no parameter list needs to be written in Newspeak. If there are parameters, we follow the Smalltalk lead of distinguishing between methods that denote binary operators (aka binary methods) and other methods. Binary methods are written in the traditional infix notation: `5 + 4`. However, all binary operators have the same precedence and are evaluated from left to right, so `5+4*2` evaluates to 18 rather than 13.

This is controversial, as it may surprise most programmers. An alternative is to have the most common operators follow conventional precedence, while others follow the usual left to right precedence. Scala [OSV08] takes this route. However, beyond addition/subtraction and multiplication/division, it's not clear what to do. Should one treat all C operators specially? All Java operators? All Python operators?

Another issue is that while the common precedence rules may make sense for general purpose programming, they may not be good choices in domain specific languages. Good Newspeak practice is to embed domain specific languages within Newspeak as much as possible.

Other methods that take parameters are known as *keyword methods*. We follow Smalltalk's rules (not Self's!). Parameters are interspersed with the method name in a mixfix notation. Places where a parameter is expected are denoted by a colon, which is then followed by the parameter, and then the rest of the method name (if further parameters are needed).

Example: anArray at: 5 + 1 put: 0+ 6 factorial would probably be written as anArray.atPut(5+1, 0 + 6.factorial()) in a more traditional syntax, and, assuming that the receiver is indeed an array, places the value 720 into the receiver's 6th element.

This notation makes it impossible to have an arity error when calling a method. In a dynamically typed language, this is a huge advantage.

I am keenly aware that this syntax is unfamiliar to most programmers, and is a potential barrier to adoption. However, it improves usability massively.

2.2.3 Closures and other Literals

An n-ary closure is written as in Smalltalk: [:p1 ... :pN | body].

This notation is much more concise than those used in functional programming. Closures are still rare enough that most programmers are not wedded to a particular syntax. However, we will likely change the syntax to {:p1 ... :pN | body} as we adopt curly braces for class and method delimiters.

Tuples are written {e1 . .... . eN }.

If we change the closure syntax, we would use square braces for tuples, which is closer to widely recognized javascript notation. The remaining difference would be the use of dots rather than commas as separators. We see real value in using comma as an operator, and do not plan to conform exactly to javascript notation.

The current implementation uses Smalltalk syntax for strings and symbols. We anticipate moving to a more mainstream notation. Indeed, we will likely make all string literals act as symbols, and represent characters as a special case of these.

2.3 Class Declarations

Like most object oriented languages, but unlike Smalltalk, Newspeak supports a syntax for class declarations. The details of the syntax are likely to evolve.

Newspeak classes contain slots, methods and nested classes. Slots are like fields/instance variables, but accessed exclusively via messages. Methods are always virtual (i.e., subject to override) and may be inherited via mixin-based inheritance. Nested classes are discussed later in this section.

Here are some examples to convey an intuition:

class Empty = ()()

A class named Empty. As minimal a class declaration as possible. No superclass is specified, so it inherits from Object. Parentheses are used as delimiters (this will change). What might be surprising is that there are two sets of parentheses. This will be explained shortly.

class JustAsEmpty = Object ()()

Same as above, with the superclass listed explicitly.

class Box = (| contents |)()

This class has a single slot named contents. Slots are declared in between vertical bars, much like Smalltalk local variables. The first set of parentheses

in the above declaration delimits the *instance initializer* of the class. Such an initializer always exists, though, as we've seen, it may be empty. It contains all slot declarations for the class. Slots may be followed by initialization expressions

```
class BoxWithNonNullContents = ( | contents =  {1. 2. 3}. | )()
```

The above sets the slot contents to a *tuple* (an array) with 3 elements - 1, 2 and 3. Slot declarations may be followed by further initialization code, as in

```
class AnotherBox = (
 |  public contents =  Array new: 3. |
 1 to: 3 do:[:i | contents at: i put: i].
 )()
```

The instance initializer is followed by the class body (delimited by the last set of parentheses in our examples), which may contain nested classes and methods.

Below, class WorkingBox has a method named doAction: that takes a single parameter, *a*. The method body is also delimited by parentheses.

```
class WorkingBox = (
 | contents |
 )(
     public doAction: a = ( a value: contents )
 )
```

Class declarations create a class factory object that provides the means of producing instances of the class. The factory object supports at least one message that produces new instances. This is known as the *primary factory method.* By default, it is called new. So the following code produces an instance of class AnotherBox and queries it for its contents, evaluating to an array with the three elements 1,2 and 3.

```
AnotherBox new contents
```

Finally, here is a class with a single nested class:

```
class Outer = () (
  public class Inner = ()()
)
```

Nested classes deserve discussion. As in Beta [MMPN93], and unlike Java [GJSB05], every instance of an enclosing class has its own distinct set of nested classes. These nested classes may be accessed by sending messages to the instance. For example, the code above defines a class Outer with a nested class Inner. We find that if we create two distinct instances of Outer and query each for its nested class, the two nested classes are different. In other words

```
Outer new Inner = Outer new Inner.
```

evaluates to false.

The situation is depicted in figure 1. The class Outer is shown with two instances, anOuter 1 and anOuter 2. Each such instance has its own class Inner. Each Inner class can have its own instances - in this case anInner1 and anInner 2 respectively.

The relationship between an instance o of an enclosing class EC and its nested classes is bidirectional. Each such nested class NC is tied to o, which is known as its *enclosing object.*

The enclosing object relationship is also shown in figure 1.

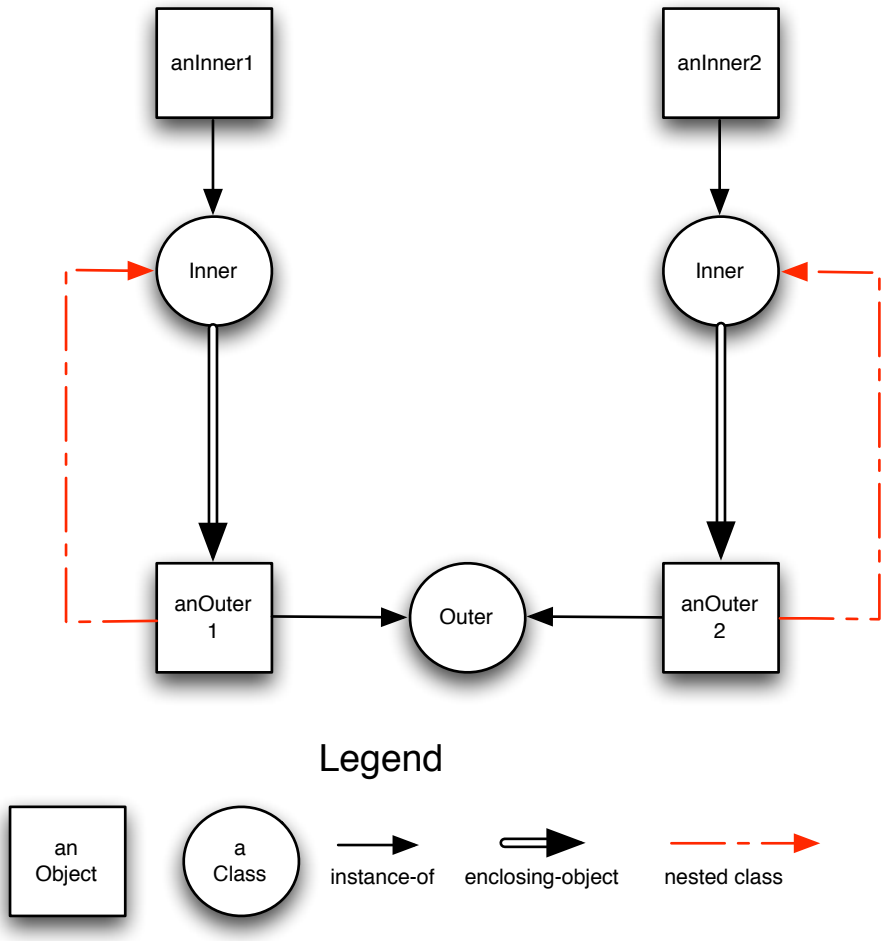

Figure 1: Enclosing classes, nested classes and their instances

The enclosing object is a property of a class, and so it is common to all the class' instances. Hence we may also speak of the enclosing object of an object i, which is the enclosing object of i's class.

Because all references to nested classes are message sends, nested classes can be overridden.

```
class ChildOfOuter = Outer () (
      public foo = (^Inner new) (* Note that the caret  acts like  return *)
)
class GrandChildOfOuter = ChildOfOuter( | public Inner = String. |) (
)
GrandChildOfOuter new foo. (* returns a String, not an Inner *)
```

Here we have overridden a nested class declaration with a slot. It is also possible to override a superclass of a nested class; consider this variation:

```
class ChildOfOuter = Outer () (
   public class Nested = Inner () ()
   public foo = (^Inner new)
)
class GrandChildOfOuter = ChildOfOuter( | public Inner ::= String. |) (
)
GrandChildOfOuter new Nested. (* returns a subclass of String, not Inner *)
(GrandChildOfOuter new Inner: Array) Nested. (* returns a subclass of Array *)
```

The code above demonstrates that one cannot, in general, expect two instances to have the same nested classes. Nested classes are created lazily, and cached thereafter. Hence Nested is created after inner has been set to Array. This also shows that all nested classes must be compiled as mixins, as the superclass cannot be reliably known at compilation time.

Here are a few more illustrative examples. If g is an instance of GrandChildOfOuter, then

```
g Inner. (* returns String *)
g Nested. (* returns a subclass of String *)
(g Inner: Array) Nested. (* returns the same subclass of String, even though Inner has changed *)
g Inner (* returns Array *)
```

2.3.1 Implicit Receivers and Scope

In most object-oriented programming languages, if the receiver is **self** (aka **this**) it can be omitted. It is often said that we are referring to a method name that is *in scope*.

In the presence of class (or object literal) nesting, if the receiver is implicit (i.e., omitted), it may be either **self** or an enclosing object of **self**. In statically typed languages, the lexical level of the receiver is determined at compile time. In dynamically typed languages, it is usually done at run time as part of the method lookup process. Typically, one starts the lookup with the class of

self (or **self** itself in a prototype based language) and proceeds up its inheritance chain; if no method is found, one jumps to the enclosing lexical level and recurses.

Newspeak differs in that lookup proceeds up the lexical scope chain (starting with the lexically deepest activation record) and only if no lexically visible matching method is found, do we proceed up the inheritance chain of **self**. After that, no further lookup is done. See figure 4 for an illustration. For a discussion of the rationale for this decision see [Bra07b]. An extensive treatment of Newspeak's nested classes and their impact on modularity is given in [BvdAB+10].

3 Concepts

3.1 Objects

An *object* is an entity that can perform computation in response to a *message* (3.4). Only objects perform computation, and they do so only in response to messages sent to them. Every object is an instance of some *class* (6). The class determines the set of *slots* (6.3.2), *methods* (3.5), and *nested classes* (6) that are associated with the object. Objects are the only entities that exist during the execution of a Newspeak program.

3.1.1 Values: Deeply Immutable Objects

Some objects cannot change after they are created. They are *deeply immutable* and known as *value objects.* A value object is globally unique, in the sense that no other object is equal to it. An object o is a value object iff one or both of the following conditions hold:

- o is either a module definition (6.4) or an instance of String, Symbol, Character, Boolean or Number.
- Under the assumption that o is a value object, it can be shown that:
  - All its slots are immutable, have been initialized and contain value objects; and
  - Its enclosing objects (3.3) are all value objects; and
  - Its class inherits from class Value and does not define an identity method (==).

If the class Value declares an identity method, that method must return the same results as Value's equality method (=).

The implications for actor concurrency (3.7) are that values can be passed among actors. Values can be copied freely across actor boundaries, or shared, as desired - but only if their initialization is complete.

Examples of such objects are numbers (5.1.1), booleans (5.1.2), characters (5.1.4), literal strings (5.1.5), symbols (5.1.6) and module definitions (6.4).

At the moment, Newspeak still relies on Squeak Smalltalk for some of its libraries. There is no class Value yet.

Values are only immutable at the base level. If the code that defines a value changes, the behavior of the value is necessarily different. For example, if the code of the Integer class changes, the actual behavior of integers might differ. As a more typical example, module definitions can be mutated via reflection.

There are two scenarios regarding the impact of reflective changes across actors.

One applies to actors that co-exist in the same address space. Values may be shared among such actors (rather than copied). Reflective changes to such shared values will be seen across actor boundaries.

The second scenario is when an actor resides in a distinct address space. Here, values are necessarily copied, and changes in an actor in one address space will not impact an actor in another address space.

To make sense of this, some construct that incorporates the idea of an address space is needed. My current thinking is that a VM (as reified by a VM mirror (8.3.1)) is responsible for a set of actors, and reflective operations will impact values seen by all these actors. Actors that are managed by another VM will not.

3.1.2 Eventual References

Eventual references are objects that are handles for objects that are not available in the heap of the currently executing actor (3.7). There are two kinds of eventual references: promises and far references. *Promises* represent the result of an asynchronous message send. *Far references* are proxies for objects of a different actor (3.7).

3.2 Classes, Mixins and Inheritance

A *class* is an object that defines a family of objects, known as its *instances*. All instances of a class respond to the same set of messages. A class is either the empty class Top or the application of a *mixin* to another class known as its *superclass*. Only the class Object may have Top as its superclass. See figure 2.

The purpose of Top is to allow for a uniform definition in which all code resides in mixins - including the code in Object. Top doesn't exist yet.

A class S is *a proper superclass of* a class C iff S is either the superclass of C or a proper superclass of the superclass of C. A class S is *a superclass of* a class C iff S is either a proper superclass of C or $S = C$. A class S is *a proper subclass of* C iff C is a proper superclass of S. A class S is *a subclass of* C iff C is a superclass of S.

The class' mixin specifies how the class differs from the superclass. The class *inherits* all the properties of its superclass that are not explicitly *overridden* (i.e., specified to be different) by its mixin. Mixins are associated with class declarations (6). A class' mixin may be used in a mixin application expression (6.2)

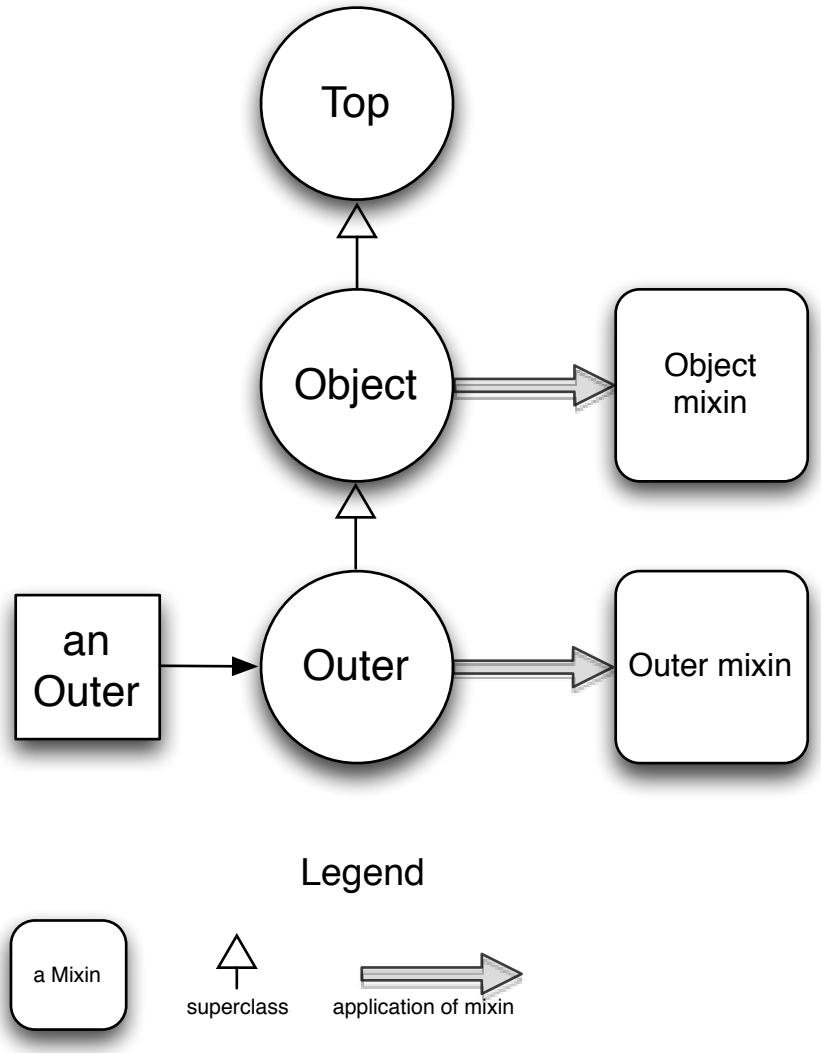

Figure 2: Mixins, classes and instances

to derive additional classes that share the same mixin but may have different superclasses.

Except for Top all classes are subclasses of class Object, which provides a small number of methods available to all classes. These include equality, identity, the corresponding hashes etc. They also include accessors for predefined classes and objects, such as the classes of the built-in literals (5.1) e.g., String. In addition, the methods Array and ByteArray provide predefined factory objects for arrays.

Currently, we use the Array and ByteArray classes provided by Squeak as these factories. However, the intent is to use a factory which is not a class. Newspeak, like most languages (and unlike Smalltalk), does not support the definition of variable sized classes like arrays. Of course, the arrays created have such a class, but access to it is restricted by the mirror system.

Each class is an instance of a unique *metaclass*. All metaclasses are instances of class Metaclass, and are direct subclasses of Class. Metaclasses may not be subclassed.

This differs from Smalltalk, where the metaclass would be a subclass of the metaclass of the class' superclass. The metaclass hierarchy in Newspeak is deliberately kept very flat.

3.3 Enclosing Objects

Every class has an *enclosing object* defined as follows:

- The enclosing object of a class definition expression (6) is its surrounding activation, if any, and **nil** (5.1.3) otherwise, as specified in section 6.
- The enclosing object of an object literal's (5.1.10) class is the literal's surrounding activation, if any, and **nil** otherwise, as specified in section 5.1.10.
- The enclosing object of a class created by a mixin application expression (6.2) is the enclosing object of the class used to derive the mixin, as specified in section 6.2.
- Otherwise, the class is necessarily a member class; the enclosing object of a member class C is the object of which C is a member - the object that received the message that created the class object C, as specified in section 6.
- The enclosing object of a class and the enclosing object of its metaclass are always the same.

*The above rules imply that the enclosing object of the result of evaluating a top level expression (3.8) is always **nil** and in particular, the enclosing object of a module definition (6.4) is **nil**.*

Method and closure activations (3.6) also have enclosing objects. The enclosing object of a method activation a is a's current instance. The enclosing

object of an activation of a closure c is the (method or closure) activation that instantiated c.

Let o be an instance of class C and let S be a superclass of C. The *enclosing object of o with respect to S* is the enclosing object of S.

Given an object o:

The 0th *enclosing object of o with respect to* a class C is o if C is a superclass of the class of o and undefined otherwise. The 1st *enclosing object of o with respect to* a class C is the enclosing object of C. For $n > 1$, an object o_n is the nth *enclosing object of o with respect to* a class C if o_n is the enclosing object of S_{n-1} where:

1. o_{n-1} is the $n-1$st enclosing object of o with respect to C.
2. C_{n-1} is the class of o_{n-1}.
3. S_{n-1} is a superclass of C_{n-1}.
4. S_{n-1} is an application of the mixin associated with the $n-1$st lexically enclosing class declaration of C (6).
5. There is no class S_k such that
   - S_k is a superclass of C_{n-1}.
   - S_k is a proper subclass of S_{n-1}.
   - S_k is an application of the mixin associated with the $n-1$st lexically enclosing class declaration of C.

The definition above may seem rather esoteric and unmotivated. However, it is useful in order to to define the precise meaning of method lookup. The concept of the kth enclosing object with respect to a class is used in section 5.9, which in turn is referenced when defining self sends and implicit receiver sends in general.

To establish intuition, it helps to visualize some of the relationships, as shown in figure 3.

The figure shows the nesting structure of a class S_3, which contains a nested class S_2, in which is nested the class S_1, which contains the class $C = S_0$. It also shows an instance $o = o_0$ of some subclass of C, and the chain of kth enclosing objects of o_0 with respect to C, for k between 0 and 3. If a method of S_0 is invoked upon o_0, and this method refers to a message declared in one of the lexically enclosing classes S_k, the receiver for that message will be o_k, the kth enclosing object of o_0 with respect to C.

If, after reading this and section 5.9, you are still baffled, you may find it helpful to consult the literature on nested classes, in particular [Mad99] , [SD03].

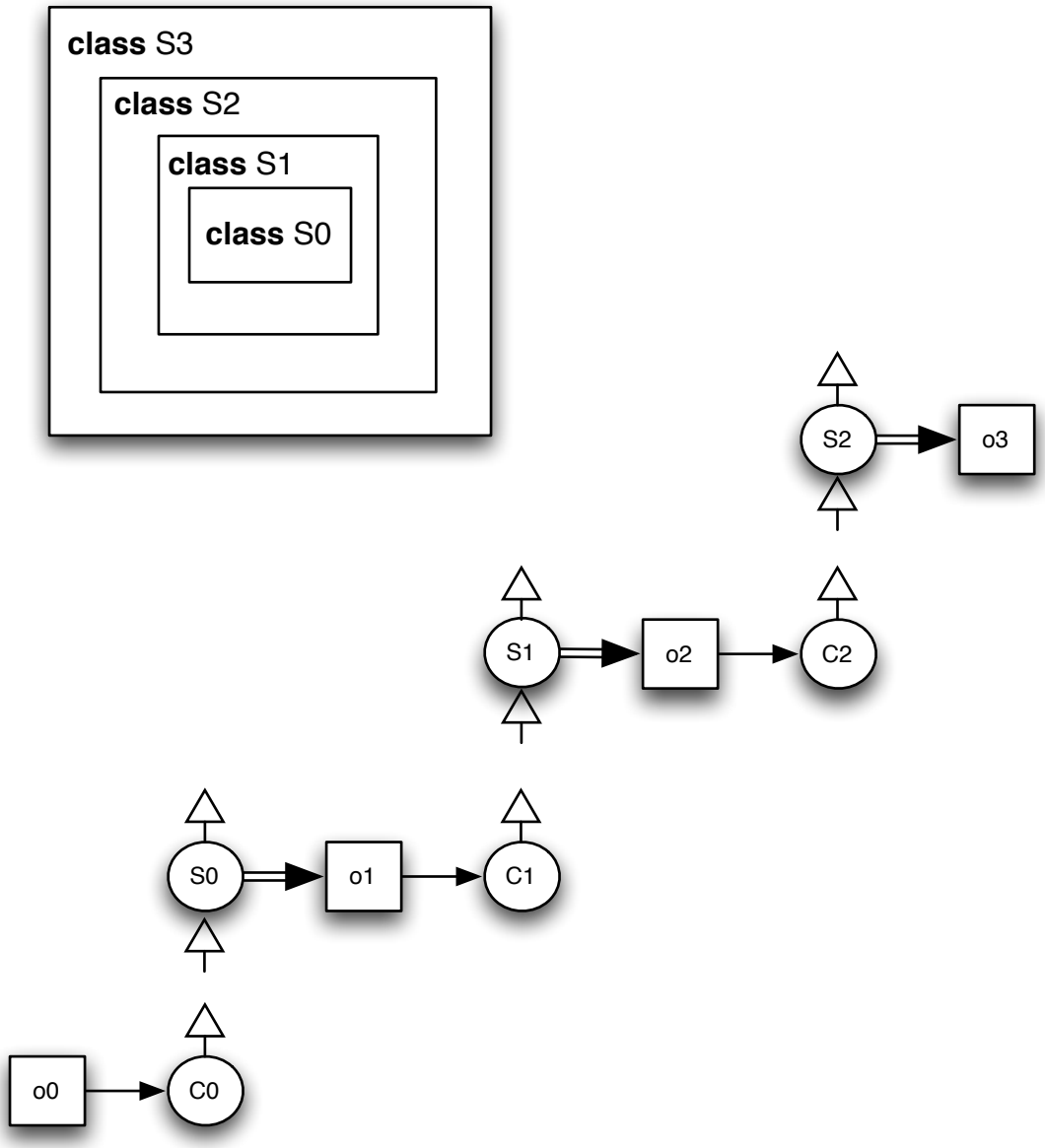

Figure 3: The kth enclosing object of $o = o_0$ with respect to $C = S_0, k \in 0..3$

3.4 Messages

A message consists of a distinguished object known as its *selector,* and a list of argument objects. Computation takes place when an object receives a message. Messages are usually inaccessible to an executing Newspeak program, but they may be *reified* by the implementation by means of a *message mirror.* A message mirror on a message μ is an object obeying the protocol of class MessageMirror that provides access to μ's selector and arguments.

In the Squeak implementation, class Message is used to represent messages.

We say that a message selector s is *defined on* an object o if a method with selector s is defined for (3.5) the class C of o.

3.5 Methods

A method defines an action be executed when it is *invoked on* an object *in response to* a message. Methods are declared within class declarations and have an associated selector (3.4).

We say that a method f is *defined by* the mixin (6) of the class declaration in which f is declared. A method f is *defined for* a class C under the following conditions:

- If f is defined by the class' mixin.
- Otherwise, if f is defined for the class' superclass.

Let C be a class and s be a selector; the *defining class of s with respect to C* (written $definingClass(s, C)$) is

- undefined, if $C =$ Top
- C, if a method f with selector s is defined by the mixin of C
- $definingClass(s, superclass(C))$ otherwise.

When a user defined method f is invoked on an object o in response to a message μ, an activation a (3.6) derived from f in response to μ is instantiated. The current instance of the activation is set to o.

The code in the method is then executed (7) in the context of a. If execution of the method completes, control is passed back to a's continuation object (3.6) and a's continuation object becomes **nil**. If no explicit return statement (7.2) is executed, the value returned is o (the current binding of **self**).

Severing the connection between an activation and its continuation object upon method termination means that closures are not full continuations by default. Continuations should be created very explicitly via mirrors on activations.

The syntax of methods is described in section 6.3.4.

3.6 Activations

An *activation* is an object that is created in order to process a message (3.4). An activation is always derived from either a method definition (3.5) or a closure (5.1.8) or is a *top level activation*. If an activation is derived from a method, it is known as a *method activation*. If its derived from a closure it is called a *closure activation*. Closure activations differ from method activations in how they support the **return** statement (7.2).

An activation a derived from a method definition or closure p in response to message μ has the following properties :

- Zero or more *parameter slots*, one for each formal parameter of p. Parameter slots are immutable (6.3.2). The nth parameter slot is initialized with the value of the nth argument of μ.
- Zero or more *local slots*, one for each slot declaration in the body of p. Local slots are defined by their corresponding declarations (6.3.2). They are initialized in the context of a, according to their declarations.
- The current instance (**self**). For method activations, the current instance is determined when the method is invoked. For closure activations, the current instance is the current instance of the closure p.
- The current class. In the case of method activations, the current class is $definingClass(s, C)$ where s is the selector of p and C is the class of the current instance, unless explicitly specified otherwise (*the only such cases are super sends (5.10)*). For closure activations, the current class is the current class of p.
- The current *continuation object*. Let *sender* be the activation that sent the message that caused this invocation. Then *sender* determines the continuation object as follows:
  - If *sender* has further computation that it needs to perform after this invocation, then the continuation object is *sender*.
  - Otherwise, the continuation object is the continuation object of *sender*.

This specifies tail call elimination, to the extent that it is observable. Unimplemented.

It is a compile-time error to define a method or closure that has a parameter and a local with the same name.

Activations may be marked *uncontinuable*. When an activation a is marked uncontinuable, it's continuation object is set to **nil**. Furthermore, any activation x whose continuation object is a is marked uncontinuable.

Newspeak programming environments must always allow developers to retain activations during debugging so that accurate and complete stack traces can be maintained.

Eliminating tail calls can be detrimental to the development experience, because such elimination may throw away activations representing valuable information regarding the history of a computation being debugged. This need not be the case, and runs counter to Newspeak philosophy (Goodthink).

In general, the problem of keeping the history of a computation is not limited to the stack. Back-in-time debuggers should be able to keep garbage collected parts of the heap available, for example. We do not provide such functionality, and do not mandate it - yet.

Top level activations have no slots, and their current instance, their current class and their continuation object are **nil**. An activation a responds to messages that access its slots.

3.7 Actors

Actors are units of explicit concurrency. Actors communicate only via *asynchronous messages*. Asynchronous messages immediately return a *promise* as a result (3.1.2). When the receiving actor has processed the message, the promise is *resolved*. If processing the message produced a result that is made available to the sender, the promise is *fulfilled.* If processing the message raised an exception to transmitted to the sender, the promise is *broken*. A promise may also be broken due to a failure of the communication system between actors.

The use of asynchronous messages between actors means that a sending actor cannot be blocked waiting for another actor to reply.

An actor is associated with a *heap*, a memory that is distinct from that of other actors.

If computation of a promise produces an object o in the heap of the current actor, the result is o. Otherwise, the result of promise resolution is the remote representation of the object produced by computation of the promise. The *remote representation* of an object o is o, if o is a value; otherwise the remote representation of o is a far reference to o.

Hence an actor never has a far reference into its own heap.

When an object is used as an argument to an asynchronous send to another actor, its remote representation is incorporated into the message delivered to the other actor.

But again, if the receiver is in fact in the same heap, the object is passed directly.

Consequently, actors are isolated from each other and share no mutable state with other actors.

Modulo reflection, as discussed in section 3.1.1 above.

An actor has a *mailbox*, where messages sent to it arrive. It processes these messages in the order they arrive in the mailbox.

There is no notion of a pattern matching construct by means of which an actor can choose which messages to receive. Indeed, there is no construct for receiving messages explicitly. This means that an actor cannot block while waiting for a message of some particular form. Since an actor cannot block on sending or on receiving, deadlock cannot occur.

Furthermore, since an actor never blocks in the middle of execution, one can implement a non-preemptive scheduler that swaps actors out only when they have completed the processing of an asynchronous message (a turn, in E parlance). This means that the system need not maintain a stack per actor, which makes it easy to scale to large numbers of actors.

More sophisticated strategies would allow preemption as long as free threads are available, or serialize preempted actors to prevent starvation/denial of service.

The order in which messages are delivered is the E-Order defined by Miller in [Mil06].

Actors are created from top-level mixins.

We use mixins to instantiate actors because one cannot, in general, construct an actor from an object; if the object is not deeply immutable, this would result in shared state across actors. We can only create an object from a value. To allow mutable state within an actor, we must create the actor from a deeply immutable mixin object, from which we derive a class that can be instantiated into a (possibly mutable) instance.

We restrict ourselves to top level mixins because nested mixins should create nested classes, which in turn require non-nil enclosing objects. A fresh actor has no access to an appropriate enclosing object for the class it is creating. For top level classes (created from top level mixins) ***nil*** *is an appropriate enclosing object. For nested classes this might lead to failure, depending on whether the mixin accesses its surrounding lexical scope. Depending on compilation strategy, the failure might be extremely hard to explain to users.*

Instead, a suitable top level mixin leads to the creation of a new actor A, from which one can then extract a suitable far reference to any nested class within A as desired.

Given a mixin M, an actor is created by applying M to Object, producing a fresh class in the new actor's heap. A far reference to the class is returned to the creating actor. Typically, the fresh class' factory is then invoked asynchronously to produce a (far-reference to an) instance of the actor.

How do we efficiently enforce these requirements? Using a bit that marks values (a deeply immutable bit)?

3.8 Programs

A *top level expression* is one of:

- An object literal (5.1.10) that is not lexically enclosed inside a class declaration (6) or an object literal.
- A class declaration that is not lexically enclosed inside a class declaration or an object literal.
- A literal expression whose meaning is not dependent on an implicit message send, and is not lexically enclosed inside a class declaration or an object literal.

- An ordinary message send (5.5) whose receiver and arguments are top level expressions.
- A top level expression that is parenthesized (5.3).

A top level expressions is executed in the context of a top level activation. A Newspeak *source program* is a top level expression. *This also means that a top level expression is not evaluated in the context of an enclosing object; this precludes implicit receiver sends (5.7). Literal expressions (5.1) that depend on an implicit send to the enclosing object for their meaning are inexpressible at the top level as well. Eventually, it may be that all literals are defined this way.*

Likewise, by definition there is no surrounding class at the top level, so ***self*** *(5.2), self sends (5.8), outer sends (5.9) and super sends (5.10) are excluded as well.*

What can be expressed are module declarations (6.4), and sends that can be used to send these declarations to a module that can instantiate them and link them together.

At this time, object literals have not been implemented.

4 Lexical Conventions

4.1 Reserved Words

The following are reserved words: **self**, **super**, **outer**, **true**, **false**, **nil**.

One can debate whether ***true****,* ***false*** *and* ***nil*** *should be reserved words or just message sends.*

We may also change the syntax for returning from a method from `^e` *to* `return:: e`*. This would make* **return** *a reserved word (or at least effectively preclude its use for mutable slots, or keyword methods).*

4.2 Lexical Rules

Here is the lexical grammar for Newspeak. It and all subsequent syntax are written in Newspeak, using the parser combinator library described in [Bra07a].

```
colon = tokenFromChar: ":".
comma = tokenFromChar: ",".
dollar = tokenFromChar: "$".
dot = tokenFromChar: ".".
equalSign = tokenFromChar: "=".
hat = tokenFromChar: "^".
lbracket = tokenFromChar: "[".
lcurly = tokenFromChar: "{".
lparen = tokenFromChar: "(".
langleBracket = tokenFromChar: "<".
mixinOperator = tokenFromSymbol: '<".
pound = tokenFromChar: "#".
```

```
rangleBracket = tokenFromChar: ">".
rbracket = tokenFromChar: "]".
rcurly = tokenFromChar: "}".
rparen = tokenFromChar: ")".
semicolon = tokenFromChar: ";".
slash = tokenFromChar: "/".
vbar = tokenFromChar: "|".
letter = (charBetween: "a" and: "z") | (charBetween: "A" and: "Z").
specialCharacter = (char: "+") | (char: "/") | (char: "\") | (char: "*") | (char: "~" ] |
                   (char: "<") | (char: ">") | (char: "=") | (char: "@") | (char: "%") |
                   (char: "|") | (char: "&") | (char: "?") | (char: "!") | (char: ",".
underscore = char: "_".
id = (letter | underscore), (letter | digit | underscore) star.
identifier = tokenFor: id.
kw = id, (char: ":").
kws = kw plus.
keyword = tokenFor: kw.
setterKw = kw, (char: ":").
setterKeyword = tokenFor: setterKw.
beginComment = (char: "("), (char: "*").
endComment = (char: "*"), (char: ")").
metadataTag = (char: ":"), id, (char: ":").
comment = beginComment, metadataTag opt, ((endComment not, any) | comment) star, endComment.
binSel = (specialCharacter | (char: "-"), (specialCharacter star).
binarySelector = tokenFor: binSel.
```

4.3 Metadata

Metadata appears in nestable comments of the form (*:id: xyz *) where id is the name of a particular metadata interpreter that is intended to parse and process the text xyz. The identifier given between the colons is the comment's *tag*, and the text following the second colon is the comment's *payload*.

A metadata comment can appear between any two expressions or declarations.

A metadata comment mc is considered to appear immediately before an expression, formal parameter or slot declaration x if nothing but whitespace and comments appears after mc and before x.

A metadata comment mc is considered to appear immediately after an expression, formal parameter or slot declaration x if nothing but whitespace and comments appears after x and before mc.

A metadata comment that appears immediately prior to an expression is associated with that expression. A metadata comment that appears immediately prior to a slot or parameter declaration is associated with that slot declaration. A metadata comment that appears immediately after a method or class factory's message pattern is associated with that method or class factory. A metadata

comment that appears immediately after a classes' header is associated with that class.

The above implies that multiple metadata comments can be associated with an expression or declaration; that is, metadata comments can appear in sequence with no intervening expressions or declarations.

Metadata comments are available at runtime via mirrors. *This means that they are available at runtime iff the platform supports mirrors. Reflective applications have full access to all metadata. Non-reflective applications are not burdened by it at runtime.*

Ordinary comments are given as metadata that does not specify a tag That way, they are attached to a known place in the program and are not thrown away, so they can be preserved during refactoring.

We expect to use the metadata mechanism for a pluggable type system. In the example code below, tags such as tag *and* return-type *are used to provide type information.*

We currently support a built-in optional type annotation syntax based on the Strongtalk type system. This is an ad hoc measure, and serves only for documentation at the moment - no typechecker has been implemented. Newspeak is significantly more dynamic, and harder to typecheck, than Smalltalk, so it is not possible to directly carry over the Strongtalk system to Newspeak.

Another possible use for metadata is to faciliate liveness in an IDE. We can annotate methods and factories with metadata describing a sample call to the method/factory. The IDE can use this information to provide exemplar instances of the class (in case of a factory) or live activations of a method, so that users have access to live data for parameters and can evaluate expressions freely. The example code below uses the tag exemplar *for this purpose.*

Here is an example showing how metadata is distributed in a class declaration:

```
class Foo bar: x (*:exemplar: Foo bar: #baz *) = (* :superfactory: Bla send meta-
data *) Bla bar = (
    | (:type: Integer *) x = 3. |
   (* :baz-metadata: *) baz:  (* :x=meta: *) x.
) (* :class-meta: *) (
   baz: (* :type: Integer *) t (* :return-type: ^Integer *) (* :exemplar: baz: 42*) = (
       (*:setter-meta: *) y:: (*:send-meta: *) x.
       (*:return-meta: *) ^ (*:send-meta: *) x
    )
)
```

The slot and expression metadata precedes the AST (so send metadata is unambiguous). Method metadata immediately succeeds the last parameter (or the method name if there are no parameters). This means that the return type is given as is method-wide metadata. The same rules work for class factories and super factory calls. Class wide metadata is provided after class header and before class body.

5 Expressions

Expressions are either *literals* (5.1) or *message sends* (5.4). Expressions are evaluated in the context of an activation (3.6) a, known as the *current activation.*

```
expression = setterKeyword opt, cascadedMessageExpression.
```

5.1 Literals

```
literal =  pattern | number | symbolConstant | characterConstant | string | tuple.
```

There are two basic approaches to specifying the meaning of literals in Newspeak.

One is to define them with respect to fixed classes in the underlying platform, or as reserved words (in the case of booleans and ***nil****). This is what the existing implementation does. It is the path of least resistance; it is easier to implement and easier to make efficient.*

The alternative is to specify literals as the results of implicit receiver message sends (5.7). This means that they can be overridden by user code, effectively changing the meaning of literals. This is closer to the late bound spirit of the language, and allows for some neat usage patterns when embedding domain specific languages in Newspeak.

We intend to investigate this option in the future. It does raise serious issues. It seems very important that literals should be value objects. The compiler should be able to rely on that. Can we enforce this on user code? Not in general, though we coud give warnings for obvious cases. So if the user is foolish enough to replace a value type with a mutable one, should they expect chaos? It seems reasonable to me (and poetic justice) but is it too error prone?

5.1.1 Numeric Literals

Numeric literals are value objects. Their form is given by the following grammar:

```
digit = charBetween: '0' and: '9'.
digits =  digit plus.
uppercaseLetter = charBetween: 'A' and: 'Z'.
extendedDigits = (digit | uppercaseLetter) plus.
radix = (digits, char: "r").
fraction =  dot, digits.
extendedFraction =  dot, extendedDigits.
exponent = (char: "e"), (char: "-" ) opt, digits.
decimalNum = (char: "-")  opt, digits, fraction opt, exponent opt.
radixNum = radix,
           (char: "-")  opt,
           extendedDigits,
           extendedFraction opt,
           exponent opt.
num =   radixNum | decimalNum.
number =  tokenFor: num.
```

A number that includes neither a fraction nor an exponent denotes an object that obeys the protocol of class Integer. That instance denotes an integer whose value is $sgn \cdot \Sigma r^i \cdot d_i$, where sgn is -1 if a leading minus was present and 1 otherwise, $d_i, i \in 0..n$ are the digits of the number (given by digits) from right to left, and r is the radix given by radix (or 10 if no radix is given). The radix itself is interpreted similarly, except that its radix is always 10.

The radix is always with respect to base 10 since no meta-radix can be specified.

Note that integers are not limited in size.

Non-integral numbers denote an object that obeys the protocol of class FixedPointNumber.

Actually, we don't have a class FixedPointNumber at the moment, and non-integral numbers are currently represented as instances of the Squeak class Fraction. This is a bug.

Most users of Newspeak are not very performance sensitive, and are better off with numbers that they understand. Financial applications in particular need accurate decimal numbers. Machine supported floating point numbers can behave in counterintuitive ways and require a great deal of expertise to manage their subtleties.

On the other hand, there are certainly applications where floating point performance is critical. Recent experience shows that using arbitrary precision rationals in numeric computations doesn't always work very well. The precision computed my greatly exceed what is required, becoming very expensive for no benefit.

One possibility is to allow users to choose a numeric implementation. The crude way to do this would force those users who really want machine floats/doubles to specify their literals in an awkward way. This is the opposite default from most languages, which make it hard for users to get accurate numbers. A better option would be to define fractional literals as message sends (as suggested above for all literals), and allow people to choose.

5.1.2 Boolean Literals

The reserved word **true** denotes the unique instance of class True. The reserved word **false** denotes the unique instance of class False. They are both value objects, and correspond to the boolean values true and false, respectively.

5.1.3 nil

The reserved word **nil** denotes the unique instance of class UndefinedObject. It is a value object. **nil** responds to almost all messages with a messageNotUnderstood error. The exceptions are isNil, which answers **true** and methods inherited from class Value.

5.1.4 Character Literals

We plan on doing away with character literals, and having people use string literals instead.

Character literals have the form "c", where c is a printable character in the ASCII character set. The character representing the double quote is represented by "" (doubling the nested quote). The value of a character literal "c" is equivalent to the expression C codePoint: j, where C is the class Character and j is an integer corresponding to to the unicode code point for c. A character is a value object.

The syntax for character literals is given below:

```
character = digit | letter | specialCharacter | (char: "[") | (char: "]") |
            (char: "{") | (char: "}") | (char: "(" ) | (char: ")") |
            (char: "^") | (char: ";") | (char: "$") | (char: "#") |
            (char: ":") | (char: ".") | (char: "-") | underscore | (char: "'").
twoDblQuotes = (char: "'"), (char: "'").
aChar = (char: "'"", (character | twoDblQuotes | (char: "'") | (char:' ')), (char: "'"").
characterConstant = tokenFor: aChar.
```

5.1.5 String Literals

String literals have the form "σ"or 'σ' where σ is a sequence of zero or more Unicode extended graphemes. The value of a string literal is equivalent to the result of the expression S fromCollection: $\{c_1. \ldots . c_k\}$ where S is the class CanonicalString and $c_i, 1 \leq i \leq k$ are integers representing the encoding of the string in Unicode normal form C.

The actual source program may be encoded in various ways. The implementation may choose to represent strings in different ways as well. The String, Symbol *and* Character *classes provide an API that encapsulates this decision (up to performance differences). The requirement above ensures that the semantic effect is as if the source was converted to Unicode NFC, and that strings were stored using that encoding.*

String literals are value objects. Such objects will be canonicalized by the default implementation.

Currently, we only support the syntax 'abc' *but not* "abc"

The current , temporary, syntax of string literals is given below:

```
twoQuotes = (char: "'"), (char: "'").
stringBody = (character | aWhitespaceChar | (char: "'") | twoQuotes) star.
str = (char: "'" ), stringBody, (char: "'").
string = tokenFor: str.
```

Should we allow more escapes, so that expressions can be embedded within? The plan is to allow some form of string interpolation.

5.1.6 Symbol Literals

These may go away (sniff). I will miss the concise syntax, but it is non-standard. And string literals will behave as symbols anyway. We will likely keep an im-

mutable string class that is not canonicalized, but it will not be used for literals. A mutable string class (aka StringBuffer) is also worth having, but would only be used by people working with very large strings, and not for literals.

Symbol literals have the form #s, where s is either a string literal (5.1.5), an identifier, a binary selector (5.4.3) or a sequence of keywords (5.4.3). Symbols are value objects that are instances of class Symbol. Their syntax is given below:

sym = str | kws | binSel | id.
symbol = tokenFor: sym.
symbolConstant = pound, symbol

5.1.7 Tuple Literals

A tuple literal is either the *empty tuple*, {}, or has the form $\{e_1 \ldots e_n\}$ for $n \geq 1$. The syntax of tuple literals is given below:

tuple = lcurly, (expression, (dot, expression) star, dot opt) opt, rcurly.

The empty tuple denotes

(*R* new: 0)

and $\{e_1 \ldots e_n\}$ is equivalent to the following

R fromArray: ((Array new: n) at: 1 put: e_1; ...; at: n put: e_n; yourself)

where R is the class ReadOnlyTuple. Tools such as debuggers may treat the expression as an atomic composition of the results of $e_i, i \in 1 \rightarrow n$.

In other words, a debugger need not step through the calls to fromArray:, new:, at:put: *and* yourself, *but they cannot ignore the* e_i.

Tuple literals are shallowly immutable.

At the moment, tuples are implemented directly as instances of Array and therefore mutable. This is a bug.

5.1.8 Closure Literals

A closure literal denotes a newly created instance of class Closure. When a closure is created, it is associated with a *current instance* and a *current class*, whose values are those of the current instance and current class (respectively) of the current activation (3.6) if it exists. Otherwise, both the current instance and current class are **nil** (5.1.3).

The syntax of closure literals is given below:

```
block = lbracket, blockParameters opt, codeBody, rbracket.
blockParameters = blockParameter plus, vbar.
blockParameter = colon, slotDecl.
where
slotDecl = identifier.
codeBody = temporaries opt, statements.
temporaries = slotDecls.
slotDecls = seqSlotDecls |
                simSlotDecls.
statements = returnStatement |
             statementSequence |
```

empty.

statementSequence = expression, furtherStatements opt.

furtherStatements = dot, statements.

Syntax note: the delimiters for closures will almost certainly change from square brackets to curly braces, to keep them aligned with method and class delimiters, when these are changed to conform to mainstream notation.

Closures are special - they know about the continuation of their lexically enclosing method and its receiver, as well as having a continuation object given to them by their caller. Thus the behavior of return statements in a closure is "non-local" (they return from the nearest enclosing method) , and the result of executing a body without executing a return statement will return the last expression's value, not ***self****).*

A closure takes zero or more parameters. An n-ary closure b may be invoked by sending it a value message of the appropriate arity. This will cause instantation of a closure activation a derived from b in response to the message. Then

- If the closure body is empty, control is passed back to a's continuation object (3.6). The value returned is **nil** (5.1.3).
- Otherwise, the closure body is executed in the context of a. If execution of the closure body completes, let e be the value of the last statement in the closure body. In this case, control is passed back to a's continuation object. The value returned is e.

If the closure body completes, its statement is necessarily an expression statement, since otherwise it would have been a return statement which would have passed control elsewhere before the closure completed.

5.1.9 Pattern Literals

Experimental. A pattern literal is either a wildcard pattern, a literal pattern or a keyword pattern. The exact meaning depends on the binding of the message Pattern in the environment where the literal is evaluated. By default, Pattern is defined in class Object and denotes class Pattern of the Newspeak library.

pattern = (tokenFromChar: "<"), patternLiteral, (char: ">").

patternLiteral = wildcardPattern | literalPattern | keywordPattern.

Wildcard Patterns A wildcard pattern matches any object. It is equivalent to evaluating Pattern wildcard.

wildcardPattern = tokenFromChar: '_'.

Literal Patterns A literal pattern matches a particular Newspeak literal. The pattern matches any object whose #= method returns **true** when invoked upon the literal named in the pattern. A literal pattern $< l >$ is equivalent to evaluating Pattern literal: l, where l is either a number literal, a symbol literal, a character literal, a string literal or a tuple literal.

literalPattern = tokenFor: number | symbolConstant | characterConstant | string | tuple.

Keyword Patterns A keyword pattern is equivalent to evaluating Pattern keywords: kws patterns: pats , where *kws* is a list of symbols denoting keywords, and *pats* is a list of expressions. Sending a message with the selector *kws* to the pattern should return a *binding object* describing if and how the pattern matches the message arguments.

keywordPattern = kwPatternPair plus.

kwPatternPair = keyword, kwPatternValue opt.

kwPatternValue = wildcardPattern | literalPattern | variablePattern | nestedPatternLiteral.

variablePattern = tokenFor: (('?'), id).

nestedPatternLiteral = tokenFor: pattern.

A keyword pattern may contain a *variable pattern* of the form ?x nested within it.

5.1.10 Object Literals

Unimplemented. An *object literal* expression e evaluates to a newly allocated object o.

if it's a value class, how can we tell?

The class C_o of o is implicitly declared by e. The expression e specifies the superclass of C_o, what parameters are to be passed to the superclass' factory method, and what the mixin M_e of C_o is. In particular, e specifies what slots, methods and nested classes the mixin declares and how it initializes its instances.

Each evaluation of e produces a new class C_o, but its mixin M_e is identical for all evaluations of e.

The enclosing object (3.3) of C_o is the current activation, if there is one; otherwise it is **nil** (5.1.3).

If the enclosing object is **nil**, the superclass clause must be implicit, and the superclass defaults to Object. Otherwise, if no superclass is explicitly specified by the object literal, the superclass defaults to the result of evaluating the implicit receiver message Object.

If we say that the default superclass is Object as defined by the underlying platform, we would have the strange situation that explicitly writing Object could give a different result then using the default. This is what would happen if a module had its own binding of Object.

For literals in general, we have a choice as discussed in the beginning of section 5.1, but consistency pushes in the direction of flexiblility.

objectLiteral = (identifier, keywordMsg opt) opt, classBody.

5.2 self

When not part of a self send (5.8), the reserved word **self** denotes the current instance of the currently executing activation (3.5, 3.6).

5.3 Parenthesized Expressions

Evaluating an expression in parentheses (e) is equivalent to evaluating e.

parenthesizedExpression = lparen, expression, rparen.

5.4 Message Send Expressions

5.4.1 Evaluation of Message Sends

A message send expression e defines a receiver and a message to that receiver.

and whether the message is synchronous or asynchronous?

The receiver may be given explicitly or it may be implicit. The message is always given explicitly by a *message clause* (5.4.2).

Evaluation of the message send proceeds as follows:

- If the receiver is implicit, then e is an *implicit receiver send* and it is evaluated as described in section 5.7.
- If the receiver is the reserved word **self**, enclosed in zero or more pairs of parentheses, then e is a *self send* and it is evaluated as described in section 5.8.
- If the receiver has the form outer N (where N is an identifier), then e is an *outer send* and it is evaluated as described in section 5.9.
- If the receiver is the reserved word **super**, then e is a *super send* and it is evaluated as described in section 5.10.
- Otherwise, e is an *ordinary send* and it is evaluated as described in section 5.5.

5.4.2 Message Clauses

Message clauses come in three syntactic forms.

message = keywordMsg | unarySelector | binaryMsg.

Unary Message Clauses A *unary message clause* consists of a selector that is an identifier.

unarySelector = identifier.

Evaluation of a unary message clause consists of constructing a message object with no arguments and a selector that is a symbol derived from the identifier given by the message clause.

Binary Message Clauses A *binary message clause* consists of a selector that consists of special characters as defined in section 4.2, along with a single argument given by a unary expression.

binaryMsg = binarySelector, unaryExpression.

Evaluation of a binary message clause consists of evaluating its unary expression to yield an object a, and constructing a message object with the argument a and the selector given by the message clause.

Keyword Message Clauses A *keyword message clause* consists of one or more keywords each followed by a binary expression. A keyword is defined as an identifier suffixed by a colon character.

keywordMsg = (keyword, binaryExpression) plus.

Evaluation of a keyword message clause consists of evaluating its binary expressions in the order they appear, starting at the left, to yield objects $a_1 \ldots a_n, 1 \leq n$, and constructing a message object with arguments $a_1 \ldots a_n$ and a selector that is a symbol representing the concatenation of all the keywords given in the message clause.

5.4.3 Message Send Syntax

Message sends come in four syntactic forms.

Unary Expressions A unary expression u consists of either:

- A primary expression p which is either a literal (5.1), a parenthesized expression (5.3) or **self** (5.2). In this case the value of u is the value of p.

or one of the following:

- A unary message, which is directed at the implicit receiver.
- A receiver given by a unary expression, **outer**, or **super** followed by a unary message

In these cases, u is a message send expression, and it is evaluated as described in section 5.4.1.

primary = unarySelector |
literal |
block |
parenthesizedExpression.

unaryExpression = primary, unarySelector star.

Binary Expressions A binary expression b consists of one of the following:

- A unary expression u. In this case the value of b is the value of u.
- A receiver given by a binary expression followed by a binary message. In this case, b is a message send expression, and it is evaluated as described in section 5.4.1.

  Binary sends cannot have an implicit receiver (unlike the Self language). This gives us a lot more flexibility in parsing any new constructs. It also avoids code that looks like reverse Polish notation.

binaryExpression = unaryExpression, binaryMsg star.

Keyword Expressions A keyword expression k consists of either:

- A binary expression b. In this case the value of k is the value of b.

or one of the following:

- A keyword message, which is directed at the implicit receiver.
- A receiver given by a binary expression followed by a keyword message.

In these cases, k is a message send expression, and it is evaluated as described in section 5.4.1.

```
keywordExpression =  binaryExpression, keywordMsg opt.
implicitKeywordSend = keywordMsg.
```

Setter Sends A setter message has the form id:: e. It is equivalent to the expression [:p | id:p. p] value:(e) where $p \neq id$. Tools may treat it as a single message send however.

```
sendExpression = implicitKeywordSend | cascadedMessageExpression.
expression = setterKeyword opt, sendExpression.
setterKw = kw, (char: ":").
setterKeyword = tokenFor: setterKw.
```

The rationale for this formulation is to allow setter messages to play a role similar to traditional assignment. In particular:

- *To eliminate excess parentheses, for example,* w:: x at: y put: z *instead of* w:(x at:y put: z). *For this purpose alone, it would have sufficed to specify that* id:: e *be equivalent to* id:(e).
- *To enable chaining of setter sends, e.g.,* w::x::y, *similar to* w := x := y.

5.4.4 Compound Message Send Expressions

Cascades A *cascade* has the form e μ_0; ... μ_n where $\mu_i, i \in 0..n$ are message clauses and e is an expression. It is equivalent to [:p | p μ_0. ... p μ_n] value:(e).

```
nontrivialUnaryMessages = unarySelector plus, binaryMsg star,  keywordMsg opt.
nontrivialBinaryMessages = binaryMsg plus, keywordMsg opt.
keywordMessages = keywordMsg.
nonEmptyMessages = nontrivialUnaryMessages |
                   nontrivialBinaryMessages  |
                   keywordMessages.
cascadeMsg = semicolon, (keywordMsg | binaryMsg | unarySelector).
msgCascade = nonEmptyMessages, cascadeMsg star.
cascadedMessageExpression = primary, msgCascade opt.
```

Chains *Unimplemented.* A *chain* has the form e :| μ. It is equivalent to (e) μ.

Chains are a new proposed feature that has not yet been implemented. They are intended as a sugar that allows one to chain sends without excess parentheses (similar to the $ operator in Haskell), as in:

label: "foo" :| color: Color red :| font: #Courier

which otherwise might have to be written as

((label: "foo") color: Color red) font: #Courier

Other examples include:

index between: 1 and: string size :| ifTrue: [string at: index]

collection select: [:each | each > 0] :| collect: [:each | each factorial]

While in some cases, a cascade could be used, this doesn't work if one is coding in a functional style, where each expression produces a new value. One can consider chains as "cascades for functional programming". Is this worthwhile? Or are we falling into a cesspool of sweetener? It looks like the idea of chains can be generalized in a powerful and uniform way. Both chains and cascades are a form of syntactic combinator. While chains in isolation may seem excessive, we may yet be able to generalize this in an attractive way. It's an open question if chains stay in the language.

5.5 Ordinary Sends

An ordinary send consists of an explicit receiver expression and a message clause (5.4.2). The receiver expression is evaluated first, yielding an object o. Then the message clause is evaluated, yielding a message μ with selector s.

Let $f = lookupPublic(s, R)$ where R is the class of o and $lookupPublic(n, C)$ is defined as

- Undefined, if $C =$Top.
- m, if the mixin of C defines a method m with selector n and **public** access.
- Undefined, if the mixin of C defines a method m with selector n and **protected** access.
- $lookupPublic(n, superclass(C))$ otherwise.

If f is defined, then the value of the send is the result of invoking f in response to (3.5) message μ on o.

Otherwise, the #doesNotUnderstand: method defined for (3.5) the class of o is invoked with an argument that is a message mirror on μ, and the result returned by the corresponding method activation is the value of the message send.

The class Object must provide a default implementation of #doesNotUnderstand: as a **protected** method which causes a MessageNotUnderstood exception to be thrown.

This ensures that there always is a #doesNotUnderstand:, method defined for the class. Subclasses may override it to customize behavior. For example, a class can choose to have its instances forward messages to other objects.

It is important that #doesNotUnderstand: is ***protected****. Otherwise it would be possible to distinguish between an object and a proxy.*

Let o be an object, and let p be a proxy for that object, that operates using #doesNotUnderstand: to forward all messages to o. Given a ***public*** *#doesNotUnderstand: method, one can compare the behavior of o and p. Say o supports a message #foo that returns 3.*

```
o foo. (* 3 *)
p foo. (* 3 *)
o doesNotUnderstand: #foo. (* message not understood *)
p doesNotUnderstand: #foo. (* 3 *)
```

However, because #doesNotUnderstand: is protected, the last call fails. If, on the other hand, o decides to make its #doesNotUnderstand: method ***public****, both calls will return 3, and so o and p are indistinguishable.*

Subclasses should not, as a matter of good practice, reduce the accessibility of inherited methods, but there is nothing to prevent them from doing so. In particular, one could override #doesNotUnderstand: with a protected or private method. However, the wording of the semantics above ensures that the class' implementation of #doesNotUnderstand: will be invoked by the system regardless of its accessibility, and so such an attempt to restrict access would be pointless.

An alternative semantics would be to do an ordinary send of #doesNotUnderstand:. In that case, changing the access would have an effect, and we would have to specify what happened if no #doesNotUnderstand: method was defined (presumably a run time error). I see no advantage to this, as an attacker can always determine what messages an object supports.

A consequence of the above definitions is that message mirrors are freely available to all objects. This is not a security issue, as the only capabilities provided by a message mirror are the name of the method the sender intended to invoke, and the arguments it was going to pass. Since the sender intended to pass these arguments to the object that is receiver of #doesNotUnderstand:, and the name is an immutable symbol, we see no risk in providing this capability universally.

5.6 Asynchronous Sends

An asynchronous send consists of an explicit receiver expression followed by the asynchronous send token <-: followed by a message clause. The receiver expression is evaluated first, yielding an object o. Then the message clause is evaluated, yielding a message μ.

If o is a promise, then at some point after o is resolved to an object o', the send will be processed further as if the receiver were o'. If o is a near reference, then the send will immediately be processed further as if the receiver were the remote representation of o.

If o is a far reference (3.1.2) then let A be the actor associated with the referent of o. We will often say that the message μ is sent to A in this case. If A is the current actor $A_{current}$, then let μ' be μ, otherwise let μ' be a new message in the heap of A, equivalent to μ except that for each argument a in μ, the remote representation of a is placed in the corresponding position in μ'.

The message μ' is later placed on A's message queue for subsequent evaluation. The timing of this placement is constrained by the following rules:

Let A_1, A_2 and A_3 be actors.

1. If A_1 sends a message m_1 to A_2, and subsequently sends message m_2 to A_2, then A_2 will receive and process m_1 before m_2.

2. If A_1 sends a message m_1 to a far reference o associated with A_2, and A_1 has not yet passed o to A_3, and A_1 subsequently passes o to A_3, then m_1 will be received and processed by A_2 before any message from A_3 to o.

These constraints enforce the E-Order of [Mil06]

The result of the asynchronous send is a promise p that is immediately returned to the sender.

When A processes μ' , an ordinary send (5.5) o' μ' is executed, where o' is the referent of o.

If the execution of the send did not result in an exception being thrown, then let r be the result of the send. Then if $A \neq A_{current}$ then p is subsequently resolved to the remote representation of r, otherwise p is immediately resolved to r.

Otherwise, the the execution of the send resulted in an exception e being thrown. Then if $A \neq A_{current}$ then p is subsequently broken with the remote representation of e, otherwise p is immediately broken with exception e.

need to specify what broken promises mean.

5.7 Implicit Receiver Sends

An implicit receiver send consists of a message clause (5.4.2) m. Let s be the selector of the message that results from the evaluation of m.

- Let d be the innermost lexically enclosing construct in which a method, slot or class named s is declared, if such a construct exists.
  - If d is a class then let N be the name of d. Then the implicit receiver send is equivalent to an outer send (5.9) of the form outer N m.
  - If d is an object literal, then let o be the value of the object literal, let f be the method with selector s defined on (3.4) o, and let μ be the value of m. The value of the implicit receiver send is the result of invoking f on o in response to (3.5) μ.
  - Otherwise, d is necessarily a method or a closure literal. Then the implicit receiver send is equivalent to a send of the message m to the current activation (3.6) of d.

- Otherwise, the implicit receiver send is equivalent to a self send (5.8) of the form self m.

The curved line in figure 4 indicates the lookup path taken for an implicit receiver send. First, the surrounding lexical scope chain is traversed (as indicated by the broad red arrow), from a class C through its enclosing class OC, and up to the top level class OOC. Only if this fails is the receiver's inheritance chain followed (broad grey arrow).

5.8 Self Sends

A self send consists of the reserved word **self** (5.2), enclosed in zero or more pairs of parentheses, followed by a message clause (5.4.2) m. Let C be the name of the immediately enclosing class declaration. A self send is equivalent to an outer send (5.9) of the form outer C m.

Sadly, a self send is not treated compositionally: ***self*** *m is not evaluated by first evaluating* ***self*** *and then evaluating m. Consider*

```
| x |
x:: self.
x foo
```

Here, x foo is not the same as **self** *foo. The former is an ordinary send, which will succeed only if foo is* ***public****. The latter is a self send, which can access* ***protected*** *members of the the immediately enclosing class, as well as any* ***private*** *members declared within it.*

The option of enclosing a self send in parentheses is of little practical use. However, by specifying that all such forms are all treated as self sends, we avoid an ugly wart.

*If the self send construct did not allow for parentheses, (****self****) m would behave differently from a self send - it would be an ordinary send, and so could not invoke non-public methods. The parentheses would have a semantic effect beyond order of evaluation!*

This would be easily fixed by having a special syntax for self sends, e.g., ***SelfSend*** *m. However, this flies in the face of deeply ingrained habits, and would likely cause many mistakes.*

*No doubt programmers would be surprised that (****self****) m only called public methods (because it was just an ordinary send), though the likelihood of that mistake is negligible. On the other hand, if we were to introduce a special syntax for self sends, programmers would constantly be surprised and annoyed that* ***self*** *m was an ordinary send.*

Instead, the solution is to recognize the parenthesized form as a self send as well. All self sends work as expected, and so do parentheses.

5.9 Outer Sends

An outer send consists of the reserved word **outer**, an identifier N, and a message clause (5.4.2). It is a compile time error if N is not the name of

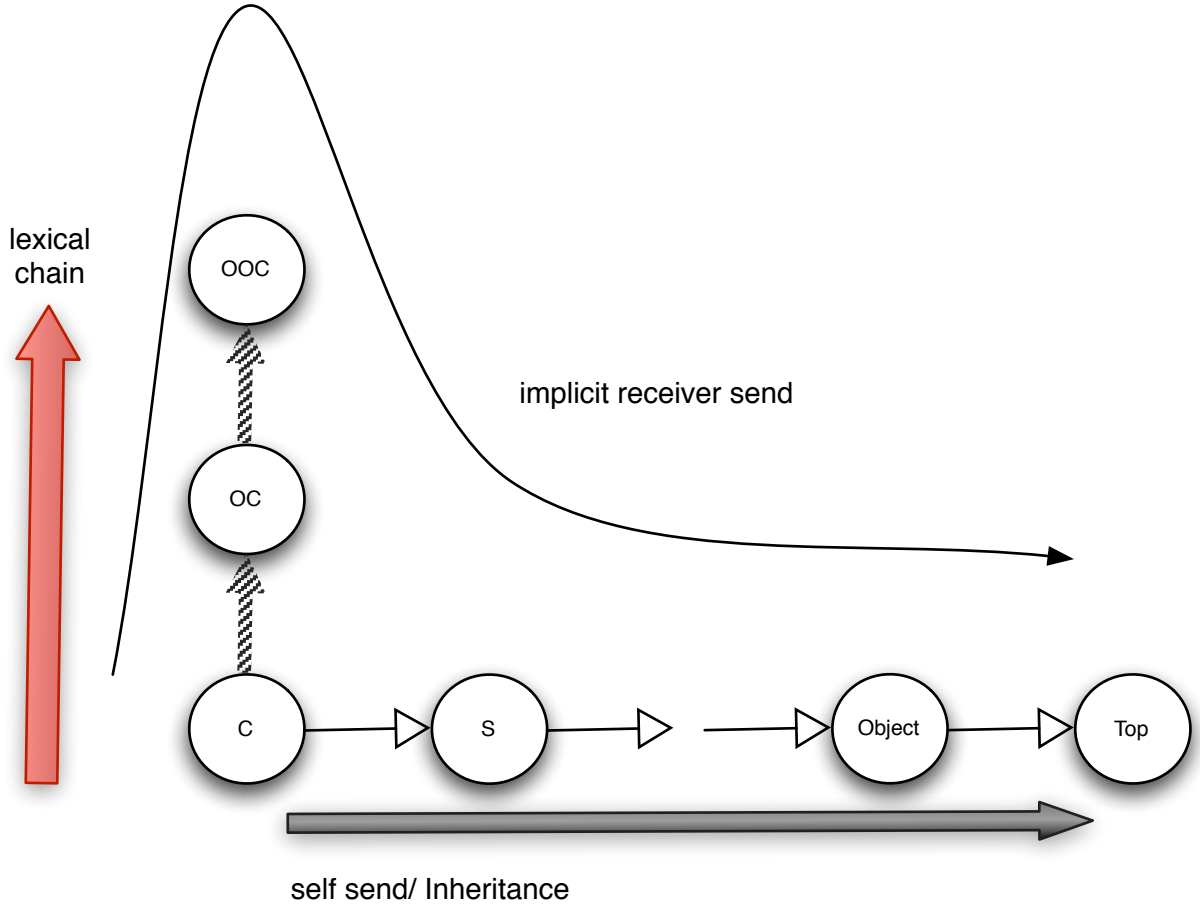

Figure 4: Method lookup

a class whose declaration D lexically encloses the outer send. Let C be the class declaration immediately enclosing the outer send. D is necessarily the kth lexically enclosing class declaration (6) of C for some $k \geq 0$. Let C' be the current class of the current activation, and let o be the kth enclosing object (3.3) of **self** with respect to C'.

The situation is as represented in figure 3, assuming $N = S_k$ for some $k \leq 3$.

A call **outer** *N s is intended to refer to a method s that is defined in an enclosing class N. Usually this can be referred to simply as s, but in some cases it may be hidden due to an intervening lexical definition. In practice this is very rare. Also note that the call cannot be bound statically unless it is* ***private****, since it may be overridden.*

Evaluation of the send begins by evaluating the message clause. Let μ the resulting message and let s be the selector of μ. If N declares a **private** method f named s then the value of the send is the result of invoking f in response to (3.5) message μ on o.

Otherwise, Let $f = lookupProtected(s, R)$ where R is the class of o and $lookupProtected(n, C)$ is defined as

- Undefined, if $C =$Top.
- m, if the mixin of C defines a method m with selector n and either **public** or **protected** access.
- $lookupProtected(n, superclass(C))$ otherwise.

If f is defined, then the value of the send is the result of invoking f in response to (3.5) message μ on o.

Otherwise, the #doesNotUnderstand: method defined for (3.5) the class of o is invoked with an argument that is a message mirror on μ, and the result returned by the corresponding method activation is the value of the message send.

Readers (and especially prospective implementors) may wish to consider what happens when such an ***outer*** *call is executed at run time. We need to identify the code that should be run; to do that, we must determine the object that should receive the message μ. We know the receiver of the currently executing method, and we know what mixin the method was declared in.*

The latter tells us what lexical scope we are in, because the mixin was induced by a class declaration C. Using this information we should be able to locate the nearest enclosing class declaration named N. This need not necessarily be done dynamically.

The appropriate receiver for our outer send must be an instance of N, or more precisely, an instance o_k of some subclass C_k of an application of N mixin*. As you'll see, this is exactly the kth enclosing object of* ***self*** *with respect to C.*

To find o_k, we'll start with the class C_0 of $o_0 =$ ***self****. Somewhere along the superclass chain of C_0 there must be a class S_0, that is an application of* C mixin*. Usually, this is the class where the current method, which contains our outer send, was found (in the case of super calls, this may not be the case, but*

we'll ignore that possibility to keep this example simple; the definition of kth enclosing object handles the situation correctly). From here, we can climb up the lexical chain of C.

We can now look at o_1, the enclosing object of S_0. The class of o_1 is C_1. We can apply the same procedure we applied to C_0 to C_1, except that now we're looking for an application of a mixin that corresponds to the next step in the lexical chain (C's immediately enclosing class). There has to be one (why? exercise for the reader).

After k times (and k might be zero) we will reach o_k. We don't have to precompute k - that is simply an optimization. Instead, we might, at every stage, check if C_n is in fact an application of a mixin named N.

*If the method is **private**, it cannot be overridden and so the code could be predetermined - but the receiver could not. If the method is not **private**, a standard lookup can be done on o_k. This completes the exegesis of this particularly thorny issue.*

5.10 Super Sends

A super send consists of the reserved word **super**, followed by a message clause (5.4.2) m. Let a be the current activation (3.6) and C be a's current class. It is a run time error if C is Object or Top. Otherwise, let S be the superclass of C. Evaluation of the send begins by evaluating the message clause. Let μ be the resulting message, let s be the selector of μ and let o be the current instance of a. If a **public** or **protected** method f named s is defined for (3.5) S, the value of the send is the result of invoking f in response to (3.5) message μ on o. The method activation's current class is set to $definingClass(s, S)$. Otherwise, the #doesNotUnderstand: method defined for S is invoked with an argument that is a message mirror on μ, and the result returned by the corresponding method activation is the value of the message send.

5.11 Class Expressions

Class expressions are used to define classes, and are described in the following section.

In practice, class expressions are restricted to the top level and as nested classes in current syntax and implementations. The intent is to relax this restriction at some point, but this has not proven at all necessary so far.

6 Classes

A class expression begins with the word **class**, followed by an identifier that is the name of the class being created by the expression. The identifier may be followed by a method signature for the *primary factory* method of the class. If the signature is omitted it defaults to new. Next is a specification of the class' superclass and mixin.

The mixin can be specified via a *class body* (6.3) that declares instance and class members and specifies how to initialize an instance, based on the parameters given to the primary factory. The formal parameters named in the primary factory signature are in scope in the instance initializer but not in the rest of the class body.

Alternatively, the mixin can be specified along with the superclass via a *mixin application* (6.2), and in that case, the formal parameters named in the primary factory signature are in scope throughout the mixin application.

The superclass may be specified as part of a mixin application as noted above, or it may be specified alongside a class body. If a class body appears, the superclass can be specified implicitly, or it can be specified explicitly via a *superclass clause*. The superclass clause defines how to compute the superclass and how to invoke its initializer. The superclass clause may be an *inheritance clause* (6.1) or a mixin application. If the superclass clause is given implicitly it defaults to Object. The formal parameters named in the primary factory signature are in scope in the superclass clause.

```
classDecl = ((tokenFromSymbol: #class), identifier, messagePattern, equalSign,
                inheritanceListAndOrBody) |
            ((tokenFromSymbol: #class), identifier, empty, equalSign,
                inheritanceListAndOrBody).
inheritanceListAndOrBody = defaultSuperclassAndBody |
                           explicitInheritanceListAndOrBody.
defaultSuperclassAndBody = classBody.
explicitInheritanceListAndOrBody = inheritanceClause, mixinSpec.
mixinSpec =  classBody | mixinAppSuffix.
mixinAppSuffix = ((tokenFromSymbol: #<:),  inheritanceClause) plus,
                 (dot | classBody).
```

A class expression is always evaluated in the context of an enclosing object *eo* (3.3). If a class expression is directly nested in a method (3.5) or closure, the enclosing object *eo* is the activation (3.6) *a* in which evaluation takes place.

Unimplemented. The current syntax does not allow a class declaration to appear as an ordinary expression, but the intent is that it will be permitted.

If a class expression is lexically nested directly within another class or within an object literal, then the declaration is necessarily evaluated in response to a message (3.4) whose selector is the class' name. The enclosing object *eo* is then the receiver of that message.

Otherwise, the class expression is necessarily evaluated at the top level, and *eo* is **nil** (5.1.3).

The class expression evaluates to a class. This class is an instance of a metaclass that is the application of the metaclass mixin defined by the class' class side to the class Class as defined by the underlying platform.

The class has a name, given by the class expression, and an enclosing object which is *eo*.

The class is an application of the mixin defined by the class' instance side to its superclass. If *eo* is **nil** the superclass clause must be implicit, and the

superclass is Object.

Every instance of a class C has a distinct nested class for each nested class declaration given in the declaration of C and its superclasses.

The nested classes are distinct, because they implicitly refer to the instance; every instance created by a nested class refers to the enclosing object via lexical scoping. This information comes to it from its class (see the discussion of enclosing objects). Hence, each nested class has one piece of state - it's enclosing object, and so there must be a distinct class per instance of the enclosing class. Also, the superclass of a nested class might be distinct in different instances, as it might be a parameter to a module.

A declaration of a nested class N nested directly within another class OC implicitly defines a method with the same name and accessibility in OC. The first time this method is invoked on a given receiver r, it evaluates the class declaration, using r as the enclosing object, and returns the resulting class as its result. Subsequent calls on r always return the same class object.

Ergo, the class is effectively cached within r.

Since a class object has no slots of its own, the only state it can access is that which is available through its enclosing object. This implies that a module definition has no state, since its enclosing object is ***nil****.*

A class declaration C is its own 0th lexically enclosing class declaration. For $n > 0$, a class C_n is the nth lexically enclosing class declaration of a class declaration C_0, if it is the immediately enclosing class declaration of the $n-1$st lexically enclosing class declaration of C_0.

6.1 Inheritance Clauses

An inheritance clause consists of an expression e_c denoting a class, optionally followed by a message clause m. If omitted, m defaults to new. It is a run time error if e_c does not evaluate to a class object C. It is a run time error if the message selector specified by the inheritance clause is not the selector of the primary factory of the class C.

```
inheritanceClause = inheritancePrefix opt, unarySelector, message opt.
inheritancePrefix = outerReceiver |
                    (tokenFromSymbol: #self) |
                    (tokenFromSymbol: #super).
outerReceiver = (tokenFromSymbol: #outer) , identifier.
```

The expression e_c is syntactically restricted to a unary send whose receiver is either implicit, **self**, **super**, or the receiver of an outer send.

In the future, the receiver may also be a parenthesized expression.

Parenthesized expressions are not yet supported because they would introduce ambiguities in parsing. These will be resolved when the syntax is changed so that class bodies are delimited via curly braces.

We also considered allowing e_c to be an arbitrary expression, but that is inherently ambiguous. Consider:

class C foo: x = S m: x ...

is m:x the factory message to the superclass S, or is S m: x as a whole a message that returns the superclass, to which we will send the message new implicitly?

We could of course change the syntax more throughly such that the superclass and factory call are syntactically separate; we may consider this in future revisions.

If the inheritance clause occurs within an enclosing class E the expression e_c and then the message pattern m are each evaluated, separately, as if they occurred inside an instance method of E where the enclosing object of the class expression, *eo*, is the method's receiver.

If the inheritance clause does not appear within an enclosing class, the expression e_c and then the message pattern m are each evaluated, separately, as if they were top level expressions.

Consequently, they will almost certainly fail since no interesting classes are available at top level.

The value of e_c is known as the *class determined by the inheritance clause.*

The value of the message pattern m is known as *the message determined by the inheritance clause.*

6.2 Mixin Application

A mixin may be *applied* to a superclass to produce a new class that is an application of the mixin.

Syntactically, a mixin application consists of an explicit superclass clause denoting a class S, followed by the mixin application operator <:, followed by an inheritance clause that specifies a class C whose mixin M will be applied to the superclass. The inheritance clause's message clause m specifies how to initialize the subobject defined by the mixin.

The result of a mixin application expression is a class I with mixin M and superclass S. The enclosing object of I is the enclosing object of C. Class I has a primary factory method (6) with the same selector as m. When invoked, the primary factory of I creates a new instance o of I, and runs the instance initializer (6.3) of I on o; then it returns o.

Let m_s be the message pattern defined by the superclass clause. The instance initializer for I evaluates m_s to μ_s, the message determined by the superclass clause. Next, the instance initializer of I invokes the instance initializer of S on o with the arguments of μ_s. Then, m is evaluated to μ_m, the message determined by the inheritance clause. The instance initializer of M is then evaluated on o with the arguments of μ_m.

Any change made to the structure or code of a mixin effects all of its applications.

This is because they all share their code and structural definition with the mixin.

6.3 Class Bodies

A class body consists of an *initializer* and two *sides*, the *instance side* and the *class side*.

Each side defines a *mixin*. The methods and nested classes of the class' mixin are defined via the instance side. The initializer declares the slots (6.3.2) of the class' mixin and an optional initialization statement. The class side, together with the primary factory method, defines the mixin for the metaclass, whose methods are typically *secondary factory methods.*

The syntax of a class body is is defined as follows

classBody = classHeader, sideDecl, classSideDecl opt.

where

classHeader = lparen, classComment opt,
slotDecls opt, initExprs, rparen.

classComment:: whitespace, comment.

initExprs = (expression, (dot expression) star, dot opt) opt.

sideDecl = lparen, nestedClassDecl star, transientSlotDecl star, category star, rparen.

nestedClassDecl = accessModifier opt, classDeclaration.

category:: empty, methodDecl star.

classSideDecl = colon, lparen, category star, rparen.

The use of parentheses as delimiters for the sides and initializer derives from the syntax of Self, via certain Smalltalk dialects of nordic heritage. However, they will be replaced by curly braces, in conformance with widespread custom. sniff.

Class comments and categories are temporary measures. They should be subsumed by the systematic use of metadata (4.3).

A class body implicitly induces an instance initialization method, whose arguments are the same as those of the primary factory method of the class.

When a class object receives a message μ whose selector is the name of its primary factory method, the instance initializer is invoked on an instance o of the class, whose identity is distinct from all other instances of the class. The object o has distinct distinct copies of all slots declared by the class and its superclasses. All slots have **nil** as their initial value.

The message used for the instance initializer invocation is μ, so that the arguments to the instance initializer are the same arguments given to the primary factory method. The invocation gives rise to an activation i, per sections 3.5 and 3.6. Execution of the instance initializer begins with evaluation of the message clause given in the class' superclass clause in the context of i. The superclass' instance initializer is then invoked on o with the resulting message. Then, all the slot clauses (6.3.2) given in the class' initializer are executed in the context of i, in the order they were declared. Afterwards, the (possibly empty) statement given in the class' initializer is executed. Finally, i returns o to the primary factory method, which returns it to its caller.

Note that this formulation implies that the body of a class has no access to the actual parameters passed during instance creation - only the initializer does. This encourages a style where all external dependencies of a module are

listed explicitly in the initializer, as slots that are extracted from the factory parameters. These act as "imports", and make it easy to see what a module's requirements from its environment are. In addition, it allows the implementation to garbage collect any unused parameters, instead of retaining them for the life of the instance.

The parameters are available when computing the arguments to be passed to the superclass initializer, but not when computing the superclass. This is deliberate, as otherwise, yet another form of mixins can occur, with confusing puzzlers resulting if the name of a parameter was used as the name of the superclass.

6.3.1 Access Control

Members of a class may optionally be declared with an *access modifier*, which may be one of **public**, **protected** or **private**. If no access modifier is declared, the default is **protected**. The access level of a top-level class is always **public**.

6.3.2 Slots

Slots are declared by *slot declarations.* Slots may be mutable or immutable. Immutable slots must be initialized explicitly. Mutable slots may be initialized explicitly or implicitly.

Slots may be initialized sequentially or simultaneously. A *sequential slot clause* has the form

seqSlotDecls = vbar, slotDefs, vbar.

slotDefs = slotDef star.

The clause consists of a (possibly empty) series of *slot declarations.* A slot declaration declares a slot, and possibly initializes it to the value of the optional expression at the end of the slot declaration.

slotDef = accessModifier opt, slotDecl,
    ((equalSign | (tokenFromSymbol: #'::=')), expression, dot) opt.

When the clause is executed, the slot declarations within it are executed in sequence, one after the other.

A *simultaneous slot clause* has the form

simSlotDecls = vbar, vbar, slotDefs, vbar, vbar.

A simultaneous slot declaration with a right hand side expression e initializes the slot to the value of p computing: e, where p is the class Past‘Future. The result is a *future* that will compute the expression e on demand. All these futures are resolved once the last slot declaration in the simultaneous slot clause has been executed.

Past‘Future implements a pipelined promise so that any well founded mutual recursion between simultaneous slots will resolve properly.

Immutable Slots Immutable slots are introduced via the syntax

identifier = expression.

The above form is an *immutable slot declaration.*

When an instance is created, its immutable slots are set to the value of their initialization expression - the expression following the = sign in the slot's declaration.

An immutable slot may be accessed exclusively by invoking a getter method via a unary message consisting of the name of the slot. The getter method is implicitly defined by the slot declaration, and returns the value of the slot. The accessibility of the method is the same as the slot's. There is no setter method associated with the slot and slot initialization is not accomplished by means of a setter method.

Consequently, defining a setter will not impact such an initialization, unless the method is explicitly invoked in the course of executing the initializer.

Mutable Slots Mutable slots are introduced in two ways.

- With an initializer, via the syntax identifier ::= expression.

  Slots declared in this manner are initialized to the value of their initialization expression when an instance of the class declaring the slot is created.

- Without an initializer, via the syntax identifier

  Slots declared in this manner are initialized to **nil** (5.1.3) when an instance of the class declaring the slot is created.

The above forms are both *mutable slot declarations.*

Access to the value of a mutable slot is via a getter method, following the same rules given above for immutable slots. Except for initialization as part of its declaration, a mutable slot may be set exclusively by a invoking a setter method via a single argument keyword message consisting of the slot name with : appended to it. The initialization itself is not accomplished by the setter method.

Consequently, overriding a setter will not impact such an initialization, unless the method is explicitly invoked in the course of executing the initializer.

The setter method is implicitly defined by the slot declaration. It sets the value of the slot to be the incoming argument. It returns the receiver of the message. The accessibility of the method is the same as the slot's.

In some cases, one would prefer that the setter method return its argument rather than its receiver. In such cases, one may use a setter send (5.4.3). One might quibble that this choice couples the syntactic convenience of setter sends with a policy choice regarding their result. This is true, but supporting the cross product of desired return type and desired precedence seems too complex. In practice, the current specification seems satisfactory.

6.3.3 Transient Slots

Experimental. A *transient slot* is a slot that is initialized lazily, upon first use.

transientModifier = (tokenFromSymbol: #transient), whitespace.
transientSlotDecl = accessModifier opt, transientModifier, slotDecl,

```
((tokenFromSymbol: #=) |
 (tokenFromSymbol: #'::=')
), expression, dot.
```

A transient slot declaration has the form transient m ::= e. or transient $m = e$., possibly prefixed with an access modifier. A transient slot always has an initialization expression e, but the slot value is initially **nil**. When the slot is read by invoking its getter method m, if the slot contains **nil**, e is evaluated to an object o, the slot's value is set to o (but not by means of a setter method), and o is returned as the result of the send. Otherwise, the result of getter method invocation m is the value the slot contains.

Transient slots are designed to help support orthogonal synchronization *[Bra]. The intent is that when synchronization occurs, all transient slots will be set to* ***nil****, and recomputed on demand based on persistent values that are synchronized. Transient slots themselves need never be persisted or synchronized.*

It is certainly possible to use transient slots as lazily initialized slots, but they should not be seen as the primary source of truth; they are by design, derivative values, such as caches.

Note that transient slots may be immutable. The changes to a slot as a result of synchronization should be seen as reflective program changes, not as imperative mutations.

At this stage, transient slots are in an experimental implementation, and no synchronization or persistence support is in place.

6.3.4 Method Declarations

The syntax of method declarations is as follows:

```
methodDecl = accessModifier opt, messagePattern, equalSign,
             lparen, codeBody, rparen.
accessModifier = ((tokenFromSymbol: #private) |
                  (tokenFromSymbol: #public) |
                  (tokenFromSymbol: #protected)), whitespace.
messagePattern:: unaryMsgPattern |
                 binaryMsgPattern |
                 keywordMsgPattern.
unaryMsgPattern = unarySelector.
binaryMsgPattern = binarySelector, slotDecl.
keywordMsgPattern = (keyword, slotDecl) plus.
```

6.4 Module Declarations

A module declaration is a class expression that is not nested in any other class expression or object literal. The class object a module declaration evaluates to is known as a *module definition.* Module definitions are value objects. Instances of a module definition are referred to as *modules.*

7 Statements

Statements are units of code that are *executed in the context of an activation* (3.6) .

7.1 Expression Statements

An expression statement consists of an expression e (5). Execution of an expression statement e in the context of an activation a consists of evaluating e in the context of a.

7.2 Return Statements

A return statement has the form $\hat{}e$, where e is an expression (5). Execution of such a return statement in the context of an activation a causes e to be evaluated in the context of a. Then, if a is a method activation, control is passed to a's continuation object (3.6), with the result of the evaluation of e. Then a is marked uncontinuable (3.6).

Otherwise, a is a closure activation; let a_m be a's enclosing method activation, and let c be the continuation object of a_m. Control is passed to c. Then a_m is marked uncontinuable.

If execution of a return statement causes control to be passed to **nil**, a cannotReturn exception is thrown.

returnStatement = hat, expression, dot opt.

We may replace this syntax with a more conventional one. Logically, return is a message send to the current activation, which gives it a result to hand to its caller. We may represent is as such, via a send of the form return: e, which is close to mainstream yet conceptually consistent with our model.

7.3 Statement Sequences

A statement sequence has the form $s_1. \ldots s_n$. Execution of $s_1. \ldots s_n$ in the context of an activation a consists of the execution of s_i, $1 \leq i \leq n$, in the order they appear, in the context of a.

8 Pragmatics

8.1 Compilation Units

A source program (3.8) can be used to define a compilation unit - a completely self contained expression (5) that can be evaluated and serialized into a binary form. A compilation unit begins with a *language id*, which identifies which language the compilation unit is written in.

Currently expressions or statements may be evaluated in the IDE. Only module declarations may be used as compilation units. This is indicated in the syntax

below. Eventually, a compilation unit will consist of a language id followed by a top level expression.

compilationUnit = languageId, toplevelClass.
languageId:: identifier.
toplevelClass = classCategory, classDeclaration.
classCategory = string opt.

The use of class category is also a temporary expedient. It should be subsumed by metadata (4.3).

8.2 Reflection

Newspeak platforms provide the ability to both introspect and modify the running program via mirrors [BU04]. Mirrors provide the ability to examine and modify the structure of class declarations and methods, including the bodies of methods. They also provide for examining and modifying individual objects. In particular:

- Multiple reflective changes to the program can be applied simultaneously and atomically. Such changes are transactional - if any change fails, none take place.
- It is possible to change the class of an individual object.
- It is possible to gain access to the current activation object and its properties, including its continuation object.

Atomic install is vital to avoid brittleness due to order dependencies among individual reflective modifications.

Access to reflective functionality must be tightly controlled for security reasons, which is why the use of mirror based reflection is specified. Mirrors serve as capabilities for reflection.

Reflective access to activations supports applications that otherwise require continuations.

8.3 Accessing the Host Platform

Every Newspeak implementation provides a *platform object* that supports functionality provided by the host platform, such as GUI, file system and network access etc. This may also include platform specific functionality.

Newspeak applications are given access to a platform object via a parameter of their main:args: *method. See 8.4 below.*

In the current implementation, one can obtain a platform object in a Newspeak workspace via the expression platform.

The platform object provides accessors to various modules. These modules in turn provide access to various classes, some of which are standard and mentioned in this specification, and others which are platform specific.

8.3.1 Accessing the Virtual Machine

The system provides an accessor for the Newspeak virtual machine itself via a a mirror on the currently running virtual machine, known as the *VM mirror*.

The VM mirror supports the set of operations traditionally implemented as primitives in Smalltalk. There is no syntax for a primitive call in Newspeak. A primitive call is a message send to the VM mirror. It is important to distinguish the notion of a a system primitive from the notion of a foreign function call (see below). These are often conflated (e.g., Java native methods), but the notions are distinct.

Caveat: The VM mirror API is not mature, and the set of primitives not yet standardized. Some primitives will also be platform specific. In practice, we use our access to Squeak to call methods that involve primitives. This is a bug.

8.4 Running and Deploying Applications

A Newspeak application is an object *o* with a method main:args: that accepts a platform object (8.3) as its first parameter, and a list of any environment-specific (e.g., command line) arguments as its second. The role of main:args: is typically to instantiate and run a module (6.4) that contains the application code. This will often involve instantiating and interlinking several modules; these modules usually require some access to the underlying platform, which is available via the platform object that was passed into main:args:. The modules themselves might be members of *o*, or they may be obtained from the platform object - either directly or by by loading them dynamically (e.g., from the network or file system), or even by creating them dynamically.

8.5 Communicating with Other Languages

Newspeak programs can interact with software written in other languages. Newspeak code can *call out* to foreign languages by means of *alien objects*.

There is no notion of an external/foreign/native function/method declaration or call in Newspeak. Calling a function written in another language is done by sending a message to an alien object.

Different alien classes can be introduced to support different foreign languages. The most common aliens are C aliens. A C alien object can be produced upon request by the Newspeak platform object, which should support methods that, e.g., take a list of files, denoting dynamically linked libraries (DLLs), and a list of C header files, and produce an alien object. The alien object responds to messages corresponding to the set of functions supported by the DLLs in question. The object also supports messages that produce classes corresponding to the various datatypes defined by the DLL. These classes can be used to produce or interpret data being sent to or received from the alien object.

C aliens are fully operational, but the mechanism for generating them automatically is not yet implemented.

In particular, closures may be exchanged with alien objects, allowing idioms such as call backs. Inherent is such a model is the concept of Newspeak objects made available in the context of foreign languages. These objects are known as *expats*, which allow foreign languages to *call in* to Newspeak.

Newspeak IDEs should provide the ability to export a Newspeak module definition (6.4) as a DLL, or to export a Newspeak application as an executable, depending on the environment.

Other examples of an alien classes of interest are ObjectiveCAlien, which can be used to communicate with Objective C code on MacOS based platforms such as Macs and iPhones, and AliensForV8'Alien which allows interaction with Javascript on web browser based implementations.

One can imagine the introduction of classes such as CLRAlien to facilitate interoperability with .Net based languages, JavAlien which would allow interaction with Java code (say, on Android).

Another mode of communicating with foreign languages is asynchronous message passing. Here, Newspeak actors (3.7) communicate with actors written in other languages. This mode is safer but less performant.

8.6 Exception Handling

Because Newspeak provides reflective access (8.2) to the activation (3.6), exception handling is purely a library issue. The platform provides a standard library that supports throwing, catching and resuming exceptions, much as in Smalltalk.

References

[Bra] Gilad Bracha. Objects as software services. Invited talk at OOPSLA 2005 Dynamic Languages Symposium; updated video available at http://video.google.com/videoplay?docid=-162051834912297779. Unpublished manuscript available at http://bracha.org/objectsAsSoftwareServices.pdf.

[Bra04] Gilad Bracha. Pluggable type systems, October 2004. OOPSLA Workshop on Revival of Dynamic Languages. Available at http://pico.vub.ac.be/%7Ewdmeuter/RDL04/papers/Bracha.pdf.

[Bra07a] Gilad Bracha. Executable grammars in Newspeak. *Electron. Notes Theor. Comput. Sci.*, 193:3–18, 2007.

[Bra07b] Gilad Bracha. On the interaction of method lookup and scope with inheritance and nesting, July 2007. ECOOP 2007 3rd Workshop on Dynamic Languages and Applications.

[BU04] Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In

Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications, October 2004.

[BvdAB+10] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in Newspeak. In *European Conference on Object-Oriented Programming*, June 2010.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, Reading, Massachusetts, 2005.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.

[Mad99] Ole Lehrmann Madsen. Semantic analysis of virtual classes and nested classes. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 114–131, New York, NY, USA, 1999. ACM.

[Mil06] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, Mountain View, California, 2008.

[SD03] Matthew Smith and Sophia Drossopoulou. Inner Classes visit Aliasing. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP 2003)*, 2003.

[US87] David Ungar and Randall Smith. SELF: The power of simplicity. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, October 1987.