Composing New Abstractions From Object Fragments

Adrian Kuhn and Oscar Nierstrasz

Software Composition Group
University of Bern,Switzerland
{akuhn, oscar}@iam.unibe.ch

Abstract

As object-oriented languages are extended with novel modularization mechanisms, better underlying models are required to implement these high-level features. This paper describes CELL, a language model that builds on delegation-based chains of object fragments. Composition of groups of cells is used: 1) to represent objects, 2) to realize various forms of method lookup, and 3) to keep track of method references. A running prototype of CELL is provided and used to realize the basic kernel of a Smalltalk system. The paper shows, using several examples, how higher-level features such as traits can be supported by the lower-level model.

1. Introduction

In order to extend object-oriented languages with new means of abstraction, better intermediate representations are required (16). Current approaches typically flatten new abstractions down to existing concepts (9; 6; 15). This is mainly due to limitations of current intermediate models that cannot represent other modularization mechanisms than objects and classes. Flattening of abstractions limits their usefulness. For example, it is often hard, if not impossible, to manipulate flattened abstractions at runtime. Sometime, even meta-information about the abstractions is stripped away and thus not available at runtime, so that many runtime optimizations cannot take these abstractions into account (5). And worst of all, the flattening process might cause existing abstractions to impose severe restrictions upon the new abstractions and thus render them partially useless. For example, Java's erasure of generics is well known for introducing several such flaws.

Sometimes language extensions are realized using language hooks, such as the *method_missing* hook (2; 10; 13) which is often found in dynamic languages. Given that these hooks are typically exposed as methods that must be overridden, their usefulness in a practical setting is limited. For example, consider two libraries that both override the same hook method. The second library that loads will override and thus uninstall the hook installed by the first library, putting the entire system in an unspecified state. Obviously, language extension points that provide better means of modularization than a simple hook method are required.

A third, seemingly unrelated motivation for this work is better means for analysis and reverse engineering of running applications. However, as these tasks often require the introduction of novel abstractions, they are clearly related to the above motivations. Consider for example a back-in-time debugger (11). To implement such a tool, novel abstractions such as persistent objects (17) and aliases (12) must be introduced.

This paper describes CELL, an intermediate language model that breaks objects into a "sea of fragments" (16). Each object, each class, and any other modularization mechanisms, is represented as a group of one or more collaborating cells. The CELL model is message-based (20) and uses delegation-based cell composition to realize method lookup as in the Aspect Machine (8). Everything is a cell. There is no asymmetric distinction that separates actual objects and mere fragments. Rather, being an object is an emergent property of collaborating cells—sometimes with sharp boundaries, sometimes blurring into each other and thus giving rise to novel modularization mechanisms.

As a proof of concept, we provide a running prototype of CELL. The prototype is written in Java and Python and has been deliberately limited on first getting the concepts right, before any performance optimizations are to be considered. When running a cell-based system with the prototype, the following layers are present:

- C3, a running program in the high-level language,
- C2, the kernel of the high-level language, done using the CELL model. We refer to this as a cell-based kernel,
- C1, the kernel of CELL, currently implemented in Java.

In this paper, cell-based kernels of class-based languages are investigated. We present CELLTALK, a basic Smalltalk sys-

VMIL'08, October 19, 2008, Nashville, TN.

tem[1]. Celltalk is further extended with additional language features: frist traits (19), then object-references (12). We show how this is done purely in terms of the Cell model.

The remainder of this paper is structured as follows: Section 2 introduces the CELL model. Sections 3 to 5 present examples of increasing complexity, cumulating in the presentation of CELLTALK in Section 6, which is extended with traits in Section 7, and with aliases in Section 8. Current limitations of CELL are discussed in Section 9, related work in Section 10, and Section 11 concludes.

2. The Language Model of CELL

The language model that we describe here introduces an additional layer of abstraction *below* objects, to be used as an intermediate model for representing and implementing object-oriented languages. The key motivation of the described language model is to support emerging modularization mechanisms such as aspects, traits, or *method_missing* hooks.

The basic building blocks of the CELL model are object fragments (16) which we refer to as *cells*. Objects are built up from groups of cells with delegation relationships between them (8). The CELL model is message-oriented (20), so all interaction between cells is based on message sending. Messages and message replies are also cells, thus "everything is a cell" .

Cells themselves are simple structures, mainly consisting of

- a lookup function,
- a delegation pointer[2], and
- an (optional) payload.

Cell composition can be used to realize various forms of method dispatch, including novel modularization mechanisms. Compared to delegation-based AOP in the Aspect Machine (8), delegation in CELL is associated to a pointer instead of a delegate function. On the other hand, the lookup function of CELL is more powerful, taking into account all information provided by a `Message` cell and not just the message name.

The payload of a cell can either be binary data (used to store information such as integer values, *etc.*) or a list of references to other cells. The payload of each cell instance is private to that instance and can only be exposed through the means of primitive functions that act as closures over the payload.

[1] The choice of Smalltalk as a running example for this paper has been rather arbitrary and was mainly motivated by the author's proficiency with Smalltalk's object model. We could have chosen, without loss of generality, almost any other class-based language, such as Ruby, Python, *etc.*

[2] In the remainder of the paper, we use the terms "delegation pointer" and "`next`" interchangeably, and thus often refer to the delegate of a cell as "the next cell (after that cell)".

The current prototypes of CELL are written in high-level languages (Java and Python), as we decided to first get the concepts right rather than to strive for optimal performance up-front. So far, we identified the following types of cells that seem likely to be used in most language realizations:

- `Alias` and `Head` cells serve to handle identity issues. Head cells delegate only; they have no payload and do not respond to any messages. Head cells are most useful as anchor points for objects (or other modularization mechanisms) by providing a cell identity that can be used as the object's identity as well (16). As such, head cells serve the same purpose as object proxies do in the Aspect Machine (8). Alias cells, on the other hand, are a refinement of head cells used to keep track of object references (12). In Section 8 we show an example that makes use of alias cells.
- `Slot` cells respond to two messages, a setter and a getter message, and contain a reference to another cell, the value of the slot. Slot cells have a predecessor in Self's slots (21), which also have two implicit messages.

  The actual naming convention of the accessor messages is up to the implemented language. In this paper, we refer to slot names using an at sign (`@`) in order to distinguish them from function names, which start with a hash (`#`).
- `Function` cells, where the payload consists of a string and a callable cell. If the name of a sent message matches the string, the callable cell is executed.
- `WInteger` cells, where the payload consists of the binary data of an integer number. Responds to a common set of primitive operations on the integer payload.
- `WString` cells, where the payload consists of the binary data of a string. Responds to a common set of primitive operations on the integer payload.
- `WArray` cells, where the payload consists of a variable-sized array of references to other cells. Responds to a common set of primitive operations on the array payload.
- `CustomLookup` cells raise lookup from the level of the intermediate language to the high-level language. Upon lookup, another cell (that must respond to `#lookup`) is used to carry out the actual lookup.

  Custom-lookup is supposed to replace the *method_missing* hook as it is found in many dynamic languages. The advantage of custom-lookup is that it offers a modular extension mechanism rather than a single point of extension. For example, using custom-lookup multiple libraries can extend the same class without interfering with each other.
- `Branching` cells have *two* delegation pointers. Lookup is delegated to both delegates: first to the branch, and afterward to the next cell. Branches can easily be mapped to

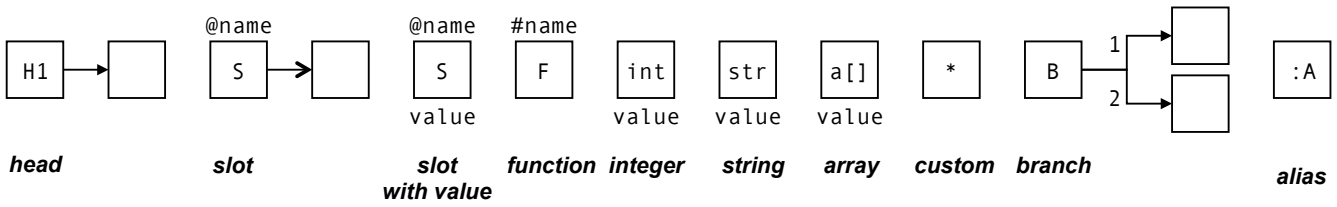

Figure 1. Notation of cell instances: (from left to right) a `Head` cell with delegation arrow; a `Slot` cell with a reference arrow, above the name of the slot; another `Slot` cell, above again the name, below the primitive value stored in the slot (reference arrow and primitive value obviously exclude each other); a `Function` cell, above the name of the function; an `WInteger`, a `WString`, an `WArray` cell, below the value of each; an `CustomLookup` cell; a `Branch` cell with branches b_1 and b_2; and an `Alias` cell (see Section 8).

multiple delegates, as in the Self programming language (21).

- `Message` cells, where the payload consists of a receiver, a name, an array of arguments, and an integer which indicates the "order of resend" (see Subsection 2.1). Message cells are used when sending messages from one cell to another (20).
- `Callable` cells that can be executed by the underlying CELL machine. Some callables have an optional payload, that is, they are closures. Callable cells are typically returned upon successful lookup of a message. Some callables close over the private payload of the returning cell.

Furthermore, the same three cell instances are globally used whenever we need to refer to `nil`, `true`, and `false`. They will typically delegate to further cells that will provide functionality required for these datatypes in the realized high-level language.

When drawing a diagram of cell instances, we use the symbols presented in Figure 1. Cell instances are represented by squares that are marked with letters to denote different cell types. Different kind of arrowheads are used to indicate delegation and reference. Please refer to the caption of Figure 1 for details.

2.1 On Lookup of Messages

The CELL model is message-oriented (20), as such, all interaction between cells is based on message sending. The lookup mechanism iterates over the delegation tree of a receiver and stops when a cell in the tree responds to the sent message.

Resending messages is treated specially: a resend starts at the receiver again and uses "response counting" to skip responses. For example, given a message `#foo` with a resend order of r=2, lookup does not return the first response, but rather the second response from the tree of visited cells.

The lookup mechanism is implemented using the following set of kernel primitives (listed here as UML method signatures), for which a Python implementation is given in Appendix A:

```
Callable.call(Message): Cell
Cell.accept(Visitor)
Cell.full_lookup(Message): Callable
Cell.lookup(Message): Callable
Cell.send(Message): Cell
Lookup.visit(Cell)
Message.resend(): Cell
```

The lookup mechanism works as follows:

1. A cell S sends a message (consisting of name and arguments) to cell R. We refer to S as the *sender* and to R as the *receiver*.
2. A new message cell M_r with receiver R, sender S, name, arguments, and a resend order of $r \geqslant 1$ is created. Normal messages use r=1, resent messages use r_{n+1}=r_n + 1.
3. The kernel calls `full_lookup` on the receiver R with message M_r as parameter. This creates a new lookup visitor V_n where count n=1, which is accepted by the receiver.
4. The lookup visitor V_n iterates over the delegation tree of the receiver, that is following delegations pointers. For branching cells, the branch pointer is followed first and then the delegation pointer. On each cell, `lookup` is called with M_r as parameter. If the visitor has not stopped, it continues.
5. If a cell C responds to the message M, the index n of V_n is compared to order r of M_r. If they are equal, the visitor stops and returns with the callable K provided by the lookup function of C. Otherwise, the index n of V_n is increased by one, and lookup continues.
6. If lookup was successful, the callable K is executed with message M_r as parameter, the returned value is returned to the sender S.

Why response counting rather than doing resends by simply continue lookup at the successor of the implementing cell? To answer that, we must distinguish between two kind of cells, branching cells and other cells. Branching cells have two delegation pointers (branch and trunk), others cells have one delegation pointer only.

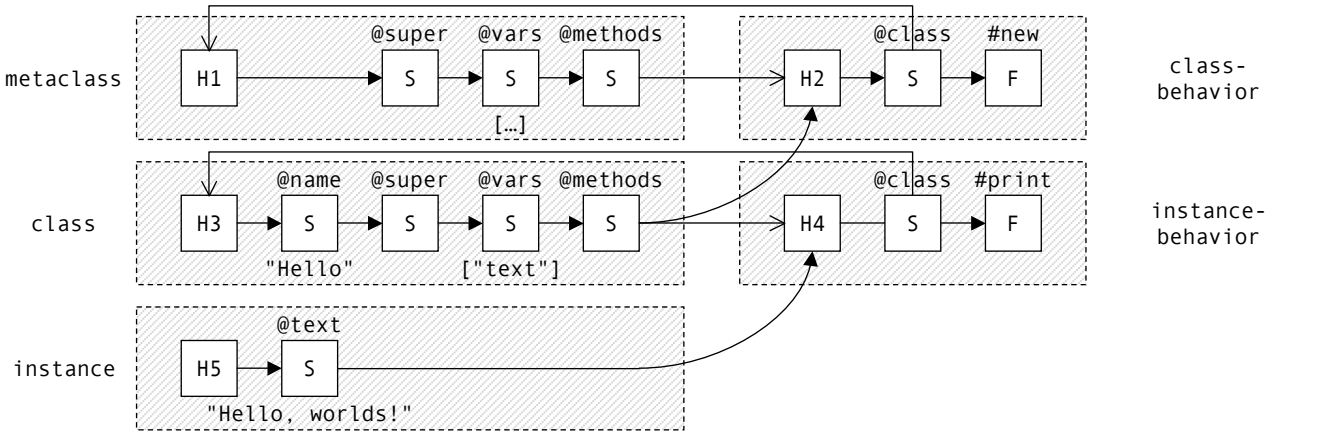

Figure 2. "Hello World" made of cells to illustrate the idiomatic structure of a class-based language realization with CELL: (from top left to bottom right) `H1` represents the metaclass, `H2` provides class behavior, `H3` represents the class, `H4` provides instance behavior, and eventually, `H5` represents an instance of the class.

Given a delegation chain without branchings, both resending approaches are equivalent. Given a delegation chain with a branching cell, that is a delegation tree, the approaches differ. A "successor resend" in the branch will never reach the cells in the trunk since the last cell of branch does not point back to the trunk. Response counting however restarts the (re)send from the receiver and thus takes both branch and trunk. As such, response counting is conceptually related to performing resends with a continuation of the original lookup.

3. Example 1: Hello World

This section provides an introductory example on how to realize a class-based language with CELL. In a class-based language there are three kinds of messages:

- Class messages can be sent without having an instance; they are understood by the class itself rather than the instance.
- Instance messages can only be sent if you have an instance; they are understood by each instance of the class.
- Instance-specific messages can only be sent to one instance; they are understood by one specific instance only.

This example exercises how to model a class that defines both class- and instance methods.

Figure 2 shows the structure of `Hello`, a class made of cells. There are two kind of cell groups: the groups on the left represent objects, and the groups on the right represent method dictionaries. Typically we denote groups by the name of their head cell. The cell groups are (from top left to bottom right):

- `H1` represents the metaclass of Hello. The main purpose of a metaclass is to store the class-method dictionary `H2` in its `#methods` slot.
- `H2` represents the class-methods of Hello. It is referred to by the `#methods` slot of the metaclass `H1`, and delegated to by the last cell of the class `H3`.
- `H3` represent the Hello class. Classes are instances of their metaclasses. The last cell of Hello delegates the class-method dictionary `H2`. Each class can create new instances of itself and stores the instance-method dictionary `H4` in its `#methods` slot.
- `H4` represents the instance-methods of Hello. It is referred to by the `#methods` slot of the class `H3`, and delegated to by the last cell of any instance, including `H5`.
- `H5` represents an instance of Hello. The instance group contains a slot for each instance variable, and the last cell typically delegates to the instance-method dictionary `H4`. As such, each instances has a separate state, while all share the same behavior.

  To install instance-*specific* behavior on an instance, one would insert additional function cells at the end of an instance group.

Actually printing Hello World works as follows (given in Python-like pseudo-code for better readability[3]).

```
# Prints "Hello, Worlds!"
hello = Hello.send("new")
hello.send("text=", "Hello, Worlds!")
hello.send("print")
```

The above structure distinguishes classes and behavior. Why? If the instance `H5` would simply delegate to its class `H3` rather than, as it is done now, to behavior `H4`, then, both class and instances would respond to the same messages—

[3] For better readability, any code examples in this paper are provided as Python code. Nevertheless, all examples presented in this paper are taken from the running examples of the current Java prototype, available under GPL license at `http://smallwiki.unibe.ch/cells`.

which is obviously *not* how class-based languages are supposed to work.

3.1 Creating New Instances of a Class

New instances of Hello are created by sending the message `#new` to the head cell `H3`. The Hello World example provided with the CELL prototype, implements instance creation as follows:

```
@bind(name = "new", arity = 0)
def create_instance(message):
    h3 = message.send("receiver")
    h5 = Head()
    for name in h3.send("vars"):
        h5.append(Slot(name))
    h4 = self.send("methods")
    head.append(h4)
    return h5
```

The above Python code does the following.

1. get the receiver of the message, that is, class Hello,
2. create the head cell of the to-be-created instance group,
3. get the list of instance variable names from Hello's `vars` slot,
4. for each name, append a new slot cell to the instance group,
5. get the instance-method dictionary, that is cell group `H4`, from Hello's `methods` slot,
6. append the dictionary to the the instance group, and
7. eventually, return the head of the instance group.

4. Cell Injection

Adding new cells to an existing group is either done by appending the new cell to the last cell of the group, or by inserting the new cell between two existing cells. We refer to this as *cell injection*.

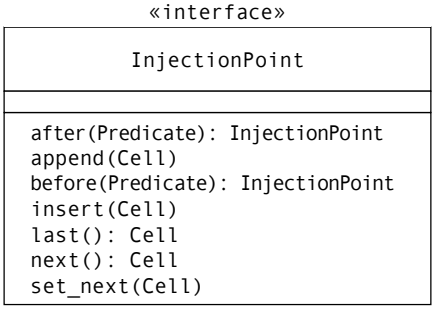

Figure 4. The InjectionPoint interface provides an API that can be used to find the correct "point of injection" for to-be-added cells.

The `InjectionPoint` interface provides an API that can be used to find the correct "point of injection" for to-be-added cells. Injection of new cells typically works in two steps as follows:

1. the *point of injection* is defined to be either before or after a cell that satisfies a given predicate,
2. the to-be-added cell is inserted at that point.

`before` injection points are typically used to inject cells that override or change existing behavior. Whereas `after` injection points are most often used to install cells with novel behavior just after the head cell of an object (or other unit of abstraction).

5. Example 2: Dwemthy's Array

This section illustrates how to extend a CELL-made high-level language using a `CustomLookup` cell. We considers Dwemthy's Array (13) as an example. Dwemthy's Array is a text based adventure game with a built-in coding challenge that uses the *method_missing* hook. Our implementation however will not use such a hook, but rather a CustomLookup cell.

In the game, a rabbit and creatures fight to death. One particular creature is Dwemthy's Array, an array filled with creatures that initially delegates all received messages to its first creature. If the creature dies, it is removed and thus the next creature appears. The array does not die until all its members are dead.

Figure 3 illustrates Dwemthy's Array made of cells: (from top left to bottom right) `H1` models Dwemthy's Array using a custom lookup cell, `H2` and `H4` are creatures contained in Dwemthy's Array, `H3` models the class behavior of the Creature class, `H5` models the rabbit, and eventually, `H6` models the instance behavior of the Creature class.

Creatures are created by sending `#new` to the cell `H3`. This creates a new head cell, appends four slot cells, initializes the slots, and eventually, links them to `H6` which provides the instance behavior of creatures.

The special behavior of Dwemthy's Array is implemented using a custom lookup cell that leverages lookup from the C2-level of the CELL kernel to the C3-level of the high-level language. Therefore, we can implement the lookup of Dwemthy's Array purely in terms of high-level sends:

```
def lookup(w_message):
    message = w_message.send("arg", 0)
    self = message.send("receiver")
    first = self.send("first")
    if first.send("life").value == 0:
        self.send("shift")
        if self.send("empty?").value:
            return Callable.constant(0)
        first = self.send("first")
    return first.full_lookup(message)
```

Please note, that by the protocol of `CustomLookup`, custom handlers receive the actual message as the first argument of a wrapper message named `#lookup`. Thus, our custom handler first unwraps the actual message, and then uses a series of high-level sends to dispatch the actual message on the first element of the array.

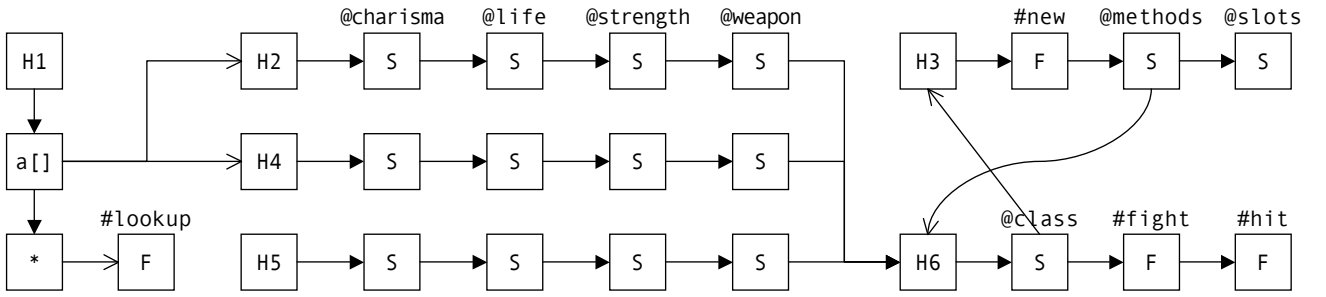

Figure 3. Dwemthy's Array made of Cells: (from top left to bottom right) `H1` models Dwemthy's Array using a custom lookup cell, `H2` and `H4` are creatures contained in Dwemthy's Array, `H3` models the class behavior of the Creature class, `H5` models the rabbit, and eventually, `H6` models the instance behavior of the Creature class.

6. Example 3: Smalltalk Made of Cells

This section provides a somewhat more complex example. A minimal Smalltalk system is made of cells, which is then considered by the two following sections as a case-study for introducing traits (19) and first-class object references (12).

"Smalltalk made of Cells" (or short CELLTALK) consists of the following four classes: `Object`, `Behavior`, `Class`, and `Metaclass`, each with an corresponding metaclass. Behavior inherits from Object, both Class and Metaclass inherit from Behavior. All four classes are the sole instances of their corresponding metaclasses. The inheritance of metaclasses parallels that of classes, with the addition that Object's metaclass inherits from Class. All metaclasses are an instance of Metaclass.

If you are unfamiliar with this setup, or even with Smalltalk's object model in general, please refer to Chapter 5 of the "Squeak by Example" book (4) for an introduction to the subject.

6.1 The Class Point and an Instance of Point

Figure 5 illustrates an (almost complete) cell graph of the CELLTALK kernel together with a Point class and a Point instance. On the left, the high-level abstractions of Smalltalk, *i.e.,* classes and objects, are shown as a UML class diagram, whereas the main part of the figure is taken up by cells that actually realizes these abstractions.

The Smalltalk abstractions are (from top to bottom):

- Everything is an `Object`. The classes of both classes and object ultimately inherit from Object. Of particular interest is the custom lookup handler at the end of Object's method dictionary. This cell implements the method-missing hook of Smalltalk-80 as follows:

```
def lookup(w_message):
    dnu = "doesNotUnderstand:"
    message = w_message.send("arg", 0)
    self = message.send("receiver")
    return self.send(dnu, message)
```

  In Smalltalk-80 and many other dynamic languages, hooks such as *method_missing* or *method_added* are popular extension points for custom language extensions. In CELLTALK however, this kind of hook is rendered obsolete as CELL's decomposition of objects into smaller parts offers better modularization and separation of concerns.

- `Behavior` is the common superclass of classes and metaclasses. Behavior implements `#new`, so instances of Behavior can create new instances of themselves. Instances of behavior have no state. Thus, `#new` is implemented as follows.

```
def new(message):
    self = message.send("receiver")
    method_dictionary = self.send("methods")
    head = Head()
    head.append(method_dictionary)
    return head
```

- `Class` extends Behavior with a name and named instance variables. It overrides `#new` using a resend as follows.

```
def new(message):
    self = message.send("receiver")
    head = message.resend()
    for name in self.send("vars").values:
        head.insert(Slot(name))
    return head
```

- `Object_class` is the metaclass of Object. It implements class-behavior of Object, which is in this case empty.

- `Point_class` is the metaclass of Point. It implements class-behavior of points, that is the constructor `#x:y:`. The constructor takes two arguments and is implemented as follows.

```
def x_y_(message):
    self = message.send("receiver")
    point = self.send("new")
    point.send("x:", message.send("arg", 0))
    point.send("y:", message.send("arg", 1))
    return point
```

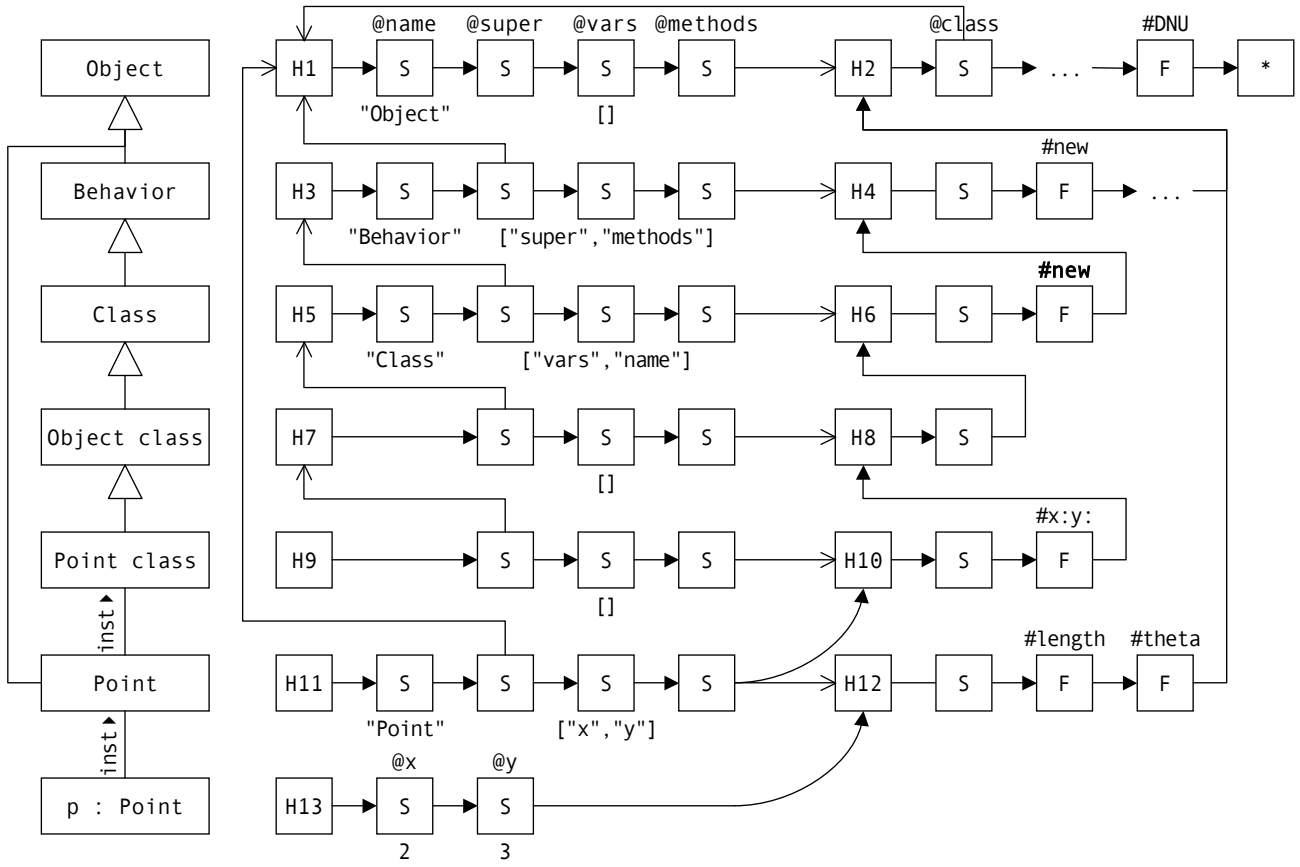

Figure 5. Smalltalk made of Cells: (from top to bottom), class Object, class Behavior, class Class, metaclass of Object, metaclass of Point, class Point, an instance of Point with instance variables x=2 and y=3. Cell groups in the center, that is with odd header numbers, represents objects. Cell groups to the right, that is with even header numbers, represent method dictionaries.

- `Point` is the class of Point. It specifies two instance variables named x and y, and implements instance-behavior of points. As an example, accessors for polar coordinates are shown on Figure 5.
- Eventually, the last item is an instance of Point. The instance is represented by two cell groups. First, `H13` represents the state of the instance, and second, `H12` is part of its class and provides instance-behavior. Thus each instance has its own state, whereas all instances share the same behavior.

However, in contrast to Smalltalk-80, in CELLTALK adding instance-behavior to an instance is straightforward. Just inject a new function cell either after `H13` or before `H12`!

Messages sent to the instance are looked up in `H13`, `H12`, and `H2` in this order. By contrast, lookup of class-methods (such as its constructor) starts at `H11`, and then traverses `H10`, `H8`, `H6`, `H4`, and `H2` in this order.

7. Example 4: Adding Traits

In this section, we extend the above CELLTALK kernel with first-class support for traits. Traits are composable units of behavior, similar to mixins (19), but avoiding fragility problems that may arise when modifying classes or mixins. When a class uses a trait, the trait contributes a set of additional methods that are based on one (or more) existing methods of the class. For example, the `TBound` traits contributes the methods `#bottom`, `#left`, `#right`, `#top`, and requires the method `#bounds`. Traits have no state and can inherit from one or more other traits.

Semantically, traits can be flattened away (15). In practice, this means that traits can be introduced experimentally to an existing class-based language "on the cheap" by flattening them away to the host language (14). This may be done because typical implementations of class-based languages do not allow one to extend the language with other unit of abstractions, such as traits. In CELL however, we can.

Figure 6 illustrates how we extend the CELLTALK kernel with traits. On the left, the high-level abstractions of Smalltalk, *i.e.,* classes, traits and objects, are shown in a UML class diagram, whereas the main part of the figure is taken up by cell-instances that actually realize these abstractions.

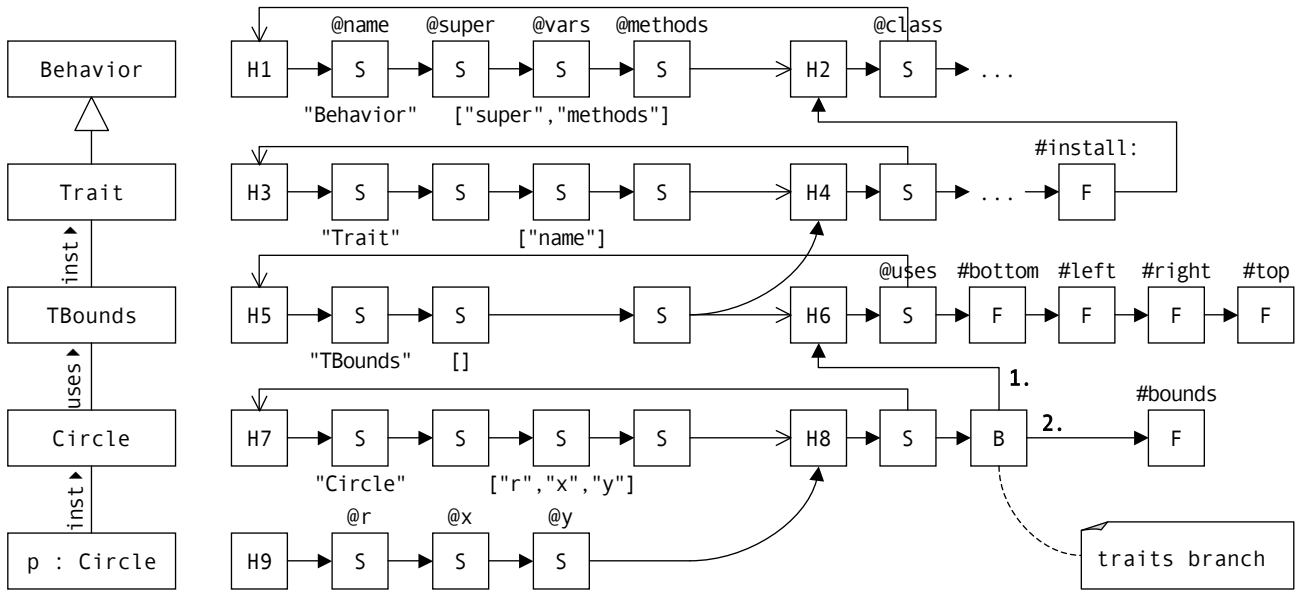

Figure 6. CELLTALK with Traits: (from top to bottom) class Behavior, class Trait, TBounds trait, class Circle that uses TBounds trait, an instance of Circle. The branching in the bottom right installs trait TBounds in the Circle class: messages are first sent to the methods provided by trait TBounds, and second only to the instance-methods provided by class Circle.

The Smalltalk abstractions are (from top to bottom):

- `Behavior` is the common superclass of Class and Metaclass. Instances of Behavior consist of a method dictionary and an inheritance relationship. Behavior is thus the natural extension point to add traits to the kernel.
- `Trait` extends Behavior with a name and functionally to allow classes to use traits using the following method:

```
def install_on(message):
    trait = message.send("receiver")
    class = message.send("arg", 0)
    m_dict = class.send("methods")
    branch = Branch.new(trait)
    m_dict.insert(branch)
```

  This inserts a branching into the method dictionary of the class. Branchings are cells with two delegation pointers: lookup is first delegated to the branch, then to the next cell.
- `TBounds` inherits from Trait. It provides four methods that all depend on the class's `#bounds` method.
- `Circle` is a class that uses the trait `TBounds`. Circle itself provides an `#bounds`.
- Finally, we have an instance of Circle.

When `#top` is sent to the Circle instance, lookup reaches the branching and takes the first branch. In the branch, the message is matched and thus the `#top` method of the trait is executed with the circle instance as receiver. In the body of this method, `#bounds` is called. Again, lookup starts at the circle instance, reaches the branching and takes the first branch. But none of the trait methods match the message, thus, lookup continues within the method dictionary of the class Circle, where the message is eventually matched by the `#bounds` method.

Neither the trait `TBounds` nor the provided methods contain any reference whatsoever to the class Circle or the circle instance. Therefore, the same trait can be used by any number of classes and any number of instances.

7.1 On Resend and Super-send Semantics

It is in the interaction between modules to support traits and class-inheritance that super-send semantics becomes interesting. In the presented traits example, the resend semantics of CELL are used to realize super-sends. Thus, if a trait-method does a super-send, the lookup visitor first visits all super-traits of the current class, and then the super-classes. However, this is not in accordance with the conventional flattening property of traits (15). Since traits are flattened, any super-send from a trait-method should behave as if sent from the class that imports the trait.

It remains open for further investigation how the desired behavior of flattened traits could best be realized at the intermediate level. Even though we introduced response counting to route resent message from tree branches back to the trunk, the resend semantics for CELL are limited. It is most likely best to distinguish between "resend" as a concept of the intermediate layer and "super-send" as a concept of the high-level language, rather than trying to directly map super-sends from the higher-level to resends at the lower-level.

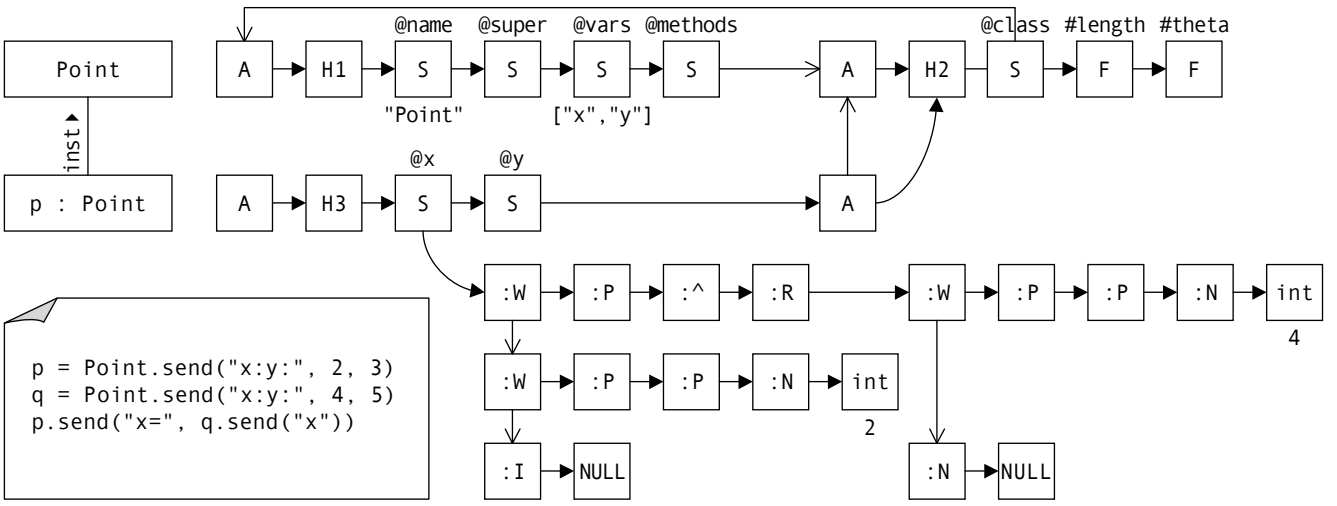

Figure 7. CELLTALK with Aliases: the complete alias tree of one slot cell is shown, together with the sequence of statements that led to the creation of these aliases. The following notation is used: write aliases are denoted as `:W`, read aliases as `:R`, parameter aliases as `:P`, return aliases as `:E`, allocation as `:A` and initialization as `:I`.

8. Example 5: Adding First-Class References

In this section, we extend CELLTALK with first-class object reference. Conceptually, we use the same approach as Adrian Lienhard's OBJECTFLOWVM (12), which extends the virtual machine of Smalltalk-80 with first-class object references.

Lienhard uses aliases as an abstraction to capture how object references propagate through a running system. All objects in the system are indirectly references through an alias. Each time an object is passed along in the running system, a new alias is created. The new alias references the same object as the originating alias, and additionally, maintains a pointer to the originating alias.

Different types of aliases are used to represent the different sources of new object references:

- `Allocation` of an object,
- `Initialization` of a slot,
- `Parameter` passing into a method,
- `Read` access to a slot,
- `Return`ing values from a method,
- and `Write` access to a slot.

Write aliases maintain two pointers: one to the originating alias, and another to the alias that was previously contained in the accessed slot.

Lienhard implements aliases as objects of the high-level language that are transparently added by the VM whenever an object reference is passed along. Our implementation however is achieved purely at the level of cell composition, so no changes to the underlying CELL kernel or virtual machine are required. Extending CELLTALK with aliases is simply a matter of changing the high-level methods that create or manipulate high-level instances.

- `Behavior#new` is changed so that it wraps the head of the just created instance group in an Allocation alias.
- `Behavior#putMethod:` is changed so that all installed functions are wrapped to create Parameter and Return aliases whenever they are called.
- `Class#new` is changed so that all created slots are wrapped into a function that creates Read and Write aliases upon access to the slot. Furthermore, all slots are initialized with an Initialize alias.

Figure 7 illustrates an (incomplete) snapshot of a running CELLTALK system with aliases. In the upper part, class Point and an instance of Point are shown. In the lower part, the complete alias tree of cell slot $p.x$ is shown.

In contrast to plain CELLTALK (see Figure 5) the cell groups that represent class Point and its instance are not referenced by their head cells but rather through alias cells. Nevertheless, head cells are still used as the identity of these groups. As a consequence, alias cells respond to the message `#==`, so identity comparison of alias cells can be delegated to the referenced head cells.

The lower part of Figure 7 shows the alias tree stored in the `#x` slot of the Point instance `p`. The sequence of statements that led to this alias cells is as follows (given in Smalltalk syntax):

```
p := Point x: 2 y: 3
q := Point x: 4 y: 5
p.x: q.x
```

Or, as expressed in terms of CELL message sends:

```
p = Point.send("x:y:", 2, 3)
q = Point.send("x:y:", 4, 5)
p.send("x=", q.send("x"))
```

Let's for example follow the flow of the integer 4 on Figure 7: it has first been initialized, then passed to the constructor of q, then passed to the setter of $q.x$ and written to the x slot of q. Later, it has been read from the slot and returned from a call to the reader of $q.x$. This return value has been passed to the setter of $p.x$, and eventually been stored in the x slots of p.

9. Discussion

The current implementation of CELL has been deliberately limited to first getting the concepts right. The semantics of CELL are currently given as Python code only; in the future we would like to provide a operational semantics for the language, potentially as a calculus of evolving objects (7).

The intermediate model presented in Section 2 provides language designers with 10 different types of cell abstractions. However, this set is not necessarily complete. In a language that supports more built-in data types besides integers and strings, the number of different cell types might quickly expand. It remains thus open to further investigation how to best implement the differing capabilities of the different cell types. There is a trade-off between complexity and extensibility. If the cell types are provided as built-in primitives of the intermediate language the set of different cell types will not be extensible. On the other hand, if there is a general abstraction that supports different types of cells the intermediate model will be directly extensible at the cost of an additional layer of complexity.

A practical limitation of CELL is runtime performance. Performing lookup on long delegations chains of cells typically results in a linear search over all understood message names. Therefore, we plan to introduce two new basic cell types, namely `Function`- and `SlotDictionary`. Consecutive runs of either Function or Slot cells can be replaced with a such a dictionary. However, for the sake of cell injection, this optimization must remain transparant to injection-clients. For example, injecting a head cell in the middle of a long run of optimized slots cells must split the according SlotDictionary into two SlotDictionaries separated by the injected head cell. More such optimizations are possible as long as they remain transparent to lookup- and injection-operations. For example, message lookup performance can be further improved by installing caches on all, or at least some cells. Head cells are particular candidates for caches, as most often lookup starts with head cells. However, as lookup in CELL is deliberately not limited to mere message name matching, not all lookup results can be caches. Lookup results that depend on contextual information that might change between two subsequent calls of the same message are not to be cached or else bad things may happen. For example, imagine a lookup that takes context information such as user authentication into account. It is clear that, in this case, caching lookup responses beyond a user session leads to a security leak.

The present prototype of CELL is further limited by not providing a complete interpreter environment. Currently, both call stack and garbage collection of the implementing language are used. The same applies for syntax and byte code. All this is done without loss of generality, as obviously Cells can be extended to provide such functionality on their own. One possible direction for further development of the current prototype is to port its sources to RPython, in order to use the PyPy tool chain (18) to automatically generate full-fledged Cells VMs for different back-ends.

10. Related Work

The use of object fragmentation in CELL is obviously reminiscent of Ossher's "sea of fragments" proposal (16). Ossher proposes to represent objects as collaborating groups of fragments with delegate-relationships between them. Each fragment contributes a part of the object's functionality. The key motivation is to provide a better representation of aspects of the intermediate language of virtual machines. The proposal neither provides a running prototype nor does it go into the details of implementing such a system.

The language model of CELL has predecessors in the Aspect Machine of Haupt and Schippers (8) and the ObjectFlow VM of Adrian Lienhard *et al.* (12). Both approaches split objects into at least two fragments, a head fragment and the actual object.

In the Aspect Machine of Haupt and Schippers (8) each object is represented by two or more fragments: a head fragment, zero or more proxy fragments, and a body with the entire object. The proxy fragments are used to install aspects in the running system. The authors propose to build a virtual machine based on that model, with common byte-codes for object manipulation and dedicated byte-code set to support proxy manipulation. One key difference is that the CELL model does not distinguish between fragments and actual objects. As a consequence the same set of operations (or byte-codes) can be used to manipulate both objects and aspects.

In the ObjectFlow VM of Lienhard *et al.* (12) each object is represented by two or more fragments: one or more head fragments (*i.e.,* aliases) and a body with the actual object. Section 8 presents a CELL implementation of their approach. The head fragments are introduced to keep track of object references. The authors use their system to realize a back-in-time debugger.

Composing classes from smaller fragments is not novel. Composition filters similarly enhance class abstraction in a modular way by decomposing into smaller fragments (1). In the model of CELL, objects are decomposed into fragments as well, and furthermore, both classes and objects are decomposed using the same set of fragments.

Also related is Daniel Bardou's work on split objects (3). The author uses head fragments to represent different roles and viewpoints of the same object. All head fragments associated with, *i.e.,* referring to, the same object have the same identity. This is analogous to what Cell does with aliases, since all aliases pointing to the same head cell have the same identity.

11. Concluding Remarks

In this paper, we have presented an intermediate model for representing object-oriented languages. The model breaks classes and objects into smaller fragments referred to as *cells*. A basic set of 10 cell types is given. Composition is used to realize various forms of method lookup and modularization mechanisms.

Since UML has not been designed with cells in mind, a novel graphical notation for the depiction of cells is introduced and used throughout the paper.

A running prototype of CELL is provided, together with examples that demonstrate the following capabilities of the model:

- CELLTALK a basic Smalltalk kernel is introduced in Section 6. We show how to model a class-based language made of cell composition. In particular, the separation of classes and behavior is identified as being idiomatic for such languages.
- Custom lookup is presented in Section 5 as a replacement of the *method_missing* hook often found in dynamic languages.
- Traits are added to CELLTALK to demonstrate the usefulness of cell composition for implementing novel modularization mechanisms (in Section 7).
- Aliases are added to CELLTALK to demonstrate the usefulness of cell composition for back-in-time debugging (in Section 8).

A mechanism for re-sending of message is presented. However, the traits example reveals that a general approach for realizing super-sends remains in need of further investigation.

The present prototype has been deliberately limited on first getting the concepts right, before optimizing for performance. It is however our conviction, that — now that the concepts are stabilizing — CELL can be tuned to become a performant and powerful intermediate machine, that can be used for realistic implementation of high-level languages with novel modularization mechanisms.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008) and "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010).

The authors thanks Adrian Lienhard for his help with the implementation of aliases, and Michael Haupt and Hans Schippers, who granted early access to their MDSOC implementation, from which parts of the CELL implementation are derived.

A. Pseudo-Code of CELL lookup

The following pseudo-code illustrates how lookup is implemented in CELL. Please refer to Subsection 2.1 for more information.

```
class Cell(object):
    def __init__(self):
        self.next = None

    "Accept lookup- or injection-client."
    def accept(self, client):
        client.visit(self);
        if client.stopped: return
        if self.next is None: return
        self.next.accept(client);

    "Lookup message on self and all delegates."
    def full_lookup(self, message):
        lookup = Lookup(message)
        self.accept(lookup)
        if not lookup.stopped: return None
        return lookup.fun

    "Check if self (excluding delegates) responds."
    def implements(self, message):
        return self.lookup(message) is not None

    "Lookup message on self (excluding delegates).
    Must reply with a callable or None."
    def lookup(self, message):
        raise "Subclass responsability"

    "Convenience method for message sending."
    def send(self, name, args):
        w_args = [System.wrap(x) for x in args]
        message = Message(self,1,name,w_args)

    "Lookup message and execute the reply."
    def send(self, message):
        fun = self.full_lookup(message)
        if fun is None: raise Exception
        return fun.call(message)

    "Check if self or any delegate responds."
    def responds_to(self, message):
        return self.full_lookup(message) is not None

class Callable(Cell):
    def __init__(lambda):
        self.lambda = lambda

    "Execute this callable."
    def call(self, message):
        return self.lambda(message)

class Message(Cell):
    def __init__(self, R, order, name, args):
        Cell.__init__(self)
```

```
        self.receiver = R
        self.order = order
        self.name = name
        self.args = args

    "Resend this message."
    def resend(self):
        m = Message(
            self.receiver,
            self.order + 1,
            self.name,
            self.args)
        return self.receiver.send(m)

class Lookup(object):
    def __init__(self, message):
        self.message = message
        self.count = message.order
        self.stopped = False

    "Refer to Section 2.1 for more details."
    def visit(self, cell):
        self.fun = cell.lookup(self.message)
        if self.fun is not None:
            if self.count == 1:
                self.stopped = True
            else:
                self.count = self.count - 1
```

References

[1] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting object interactions using composition filters. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming*, volume 791 of *LNCS*, pages 152–184. Springer-Verlag, 1994.

[2] Michael Bächle and Paul Kirchberg. Ruby on Rails. *IEEE Software*, 24(6):105–108, 2007.

[3] Daniel Bardou and Christophe Dony. Split objects: a disciplined use of delegation within objects. In *Proceedings of OOPSLA '96*, pages 122–137, October 1996.

[4] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Squeak by Example*. Square Bracket Associates, 2007. http://SqueakByExample.org/.

[5] Christoph Bockisch and Mira Mezini. A flexible architecture for pointcut-advice language implementations. In *VMIL '07: Proceedings of the 1st workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, page 1, New York, NY, USA, 2007. ACM.

[6] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices*, pages 183–200. ACM Press, 1998.

[7] Mariangiola Dezani-Ciancaglini, Paola Giannini, and Oscar Nierstrasz. A calculus of evolving objects. In *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL 2008)*, 2008.

[8] Michael Haupt and Hans Schippers. A machine model for aspect-oriented programming. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *LNCS*, pages 501–524. Springer Verlag, 2007.

[9] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM.

[10] Adrian Kuhn. Collective behavior. In *Proceedings of 3rd ECOOP Workshop on Dynamic Languages and Applications (DYLA 2007)*, August 2007.

[11] Bill Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG'03)*, October 2003.

[12] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ECOOP distinguished paper award.

[13] Why Malsky. *Why's (poignant) Guide to Ruby*. Creative Commons, 2005. http://www.poignantguide.net.

[14] Oscar Nierstrasz, Stéphane Ducasse, Stefan Reichhart, and Nathanael Schärli. Adding Traits to (statically typed) languages. Technical Report IAM-05-006, Institut für Informatik, Universität Bern, Switzerland, December 2005.

[15] Oscar Nierstrasz, Stéphane Ducasse, and Nathanael Schärli. Flattening Traits. *Journal of Object Technology*, 5(4):129–148, May 2006.

[16] Harold Ossher. A direction for research on virtual machine support for concern composition. In *VMIL '07: Proceedings of the 1st workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, page 5, New York, NY, USA, 2007. ACM.

[17] Frédéric Pluquet, Stefan Langerman, Antoine Marot, and Roel Wuyts. Implementing partial persistence in object-oriented languages. In *Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX 2008, San Francisco, California, USA, January 19, 2008*. ACM-SIAM, 2008.

[18] Armin Rigo and Samuele Pedroni. PyPy's approach to virtual machine construction. In *Proceedings of the 2006 conference on Dynamic languages symposium, OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 944–953, New York, NY, USA, 2006. ACM.

[19] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[20] Dave Thomas. Message oriented programming. *Journal of Object Technology*, 3(5):7–12, May 2004.

[21] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, December 1987.