Flexible Object Layouts

Enabling Lightweight Language Extensions by Intercepting Slot Access

Toon Verwaest Mircea Lungu
Oscar Nierstrasz
Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch

Camillo Bruni
RMoD, INRIA Lille - Nord Europe, France
http://rmod.lille.inria.fr

Abstract

Programming idioms, design patterns and application libraries often introduce cumbersome and repetitive boilerplate code to a software system. Language extensions and external DSLs (domain specific languages) are sometimes introduced to reduce the need for boilerplate code, but they also complicate the system by introducing the need for language dialects and inter-language mediation.

To address this, we propose to extend the structural reflective model of the language with *object layouts*, *layout scopes* and *slots*. Based on the new reflective language model we can 1) provide behavioral hooks to object layouts that are triggered when the fields of an object are accessed and 2) simplify the implementation of state-related language extensions such as stateful traits. By doing this we show how many idiomatic use cases that normally require boilerplate code can be more effectively supported.

We present an implementation in Smalltalk, and illustrate its usage through a series of extended examples.

Categories and Subject Descriptors D.3.4 [*Programming Language*]: Processors—Interpreters, Runtime environments; D.3.3 [*Programming Language*]: Language Constructs and Features; D.3.2 [*Programming Language*]: Language Classifications—Very high-level languages

General Terms Reflection

Keywords Smalltalk, Structural Reflection, Metaobject Protocol, Traits

1. Introduction

Object-oriented programming languages (OOPL) are highly effective as modeling languages. Features including classes and inheritance can be used to model concepts at a high level of abstraction, normally leading to compact and concise code. Unfortunately there are many situations in which idiomatic programming constructs nevertheless lead to verbose boilerplate code.

Consider, for example, the need in certain applications to model first-class relationships between objects. Since no mainstream OOPL provides first-class relationships as a programming construct, they must be laboriously simulated in code. Even if we were to develop a library to implement relationships, a relatively large amount of boilerplate code will be required to properly configure the relationships and to ensure that all access to the related fields triggers the library. A possible solution is to extend our programming language to provide first-class support for relationships [6]. Although feasible, this is undesirable for various reasons: Language extensions that change the syntax and semantics of a programming language make it harder for programmers to grasp the entire language. Moreover, language developers are burdened with ensuring that all development tools of the host language properly support the language extension.

If we look more closely at this use case, we see that all boilerplate code concerning the management of first-class relationships is related to field initialization and access. By intercepting access to the fields that are used to represent relationships, we can trigger the required behavior and avoid the need for boilerplate code without the need to modify the syntax or semantics of the host language.

We argue that boilerplate code is a common problem arising from idiomatic programming practices that attempt to compensate for the lack of missing meta-level abstractions. One such missing abstraction is the reification of the fine-grained composition of the object layouts as they are known to the language and its runtime.

In this paper we present as solution a first-class model of *object layouts*, *layout scopes* and *slots* that combines

the primitive object representation from the virtual machine with the implicit representation from the language and reifies it. The resulting model is sufficiently fine-grained to provide reflective hooks at all levels of abstraction and allows us to avoid boilerplate code that results from the lack of sub-class-level reuse.

The contributions of this paper are:

- proposing the flexible object layouts, our approach to modelling instance variables through first-class slots,
- presenting a classification and examples of customized slots and their associated behavior,
- introducing and discussing an implementation of the flexible object layouts in Smalltalk.

Outline In Section 2 we analyze related work and conclude that boilerplate code can be avoided by intercepting field access. Section 3 introduces our approach of *flexible object layouts* in which object fields are represented by first-class *slots*. In Section 4 we present a series of examples of different kinds of slots with their associated behavior. Section 5 illustrates how first-class layout scopes are used to control the visibility of slots. Section 6 shows how *stateful traits* are implemented using first-class layouts. In Section 7 we shed light on how to build and migrate classes based on layouts. Finally we sum up our findings in Section 8.

2. State of the Art

Various techniques in software engineering exist to to address problems that stem from the lack of appropriate programming language abstractions. In this section we list several techniques whose *raison d'être* is, at least partially, to address the lack of adequate abstractions for object state manipulation, access, and composition.

2.1 Language Extensions

When application concerns cannot be properly expressed in a programming language, this leads to crosscutting boilerplate code. External DSLs [14, 16] are often used to address this problem. When external DSLs are tightly integrated into our programming language they essentially become language extensions [18, 24]. However, application-specific extensions to a language limit understandability, usability and portability.

Mixins [9] and traits [4, 11] are language extensions built for reuse below the class-level. They promote removal of boilerplate code by extracting it to the introduced reusable components. Traits improve over mixins by requiring explicit conflict resolution and avoiding lookup problems resulting from multiple inheritance through the flattening property. While both approaches support reuse related to the state of objects, the abstractions themselves are fairly heavy-weight and they require *glue code*, another form of boilerplate code, to configure the final class.

Aspect oriented programming [17], a language extension in itself, addresses the problem of cross-cutting concerns related to behaviour in a system. However, it does not address the cross-cutting problems regarding state.

2.2 Meta Modeling

As opposed to language extensions meta modeling focuses on describing data by generally relying on existing language features. These meta descriptions don't interfere with the core language and thus are generally decoupled from the actual objects they describe. However by only accessing data only through the meta model it is possible to alter access using first-class objects.

Magritte [23] is a meta modelling framework mainly used together with Seaside [5]. Magritte is used to describe attributes, relationship and their constraints. All descriptions are provided as first-class Smalltalk objects. Unlike the previous two examples, Magritte provides a complete interface to read and write attributes of an instance through its meta descriptions. A favorable property of Magritte is, that it is meta-descriptions are described in terms of themselves. This way it is possible to rely on the same tools to work with instances and with the model themselves.

Magritte and meta-modelling tools in general overlap in many regions with our approach of first-class layouts, scopes and slots. However these tools are built on top of an existing language and not into it. For instance Magritte's meta description are decoupled from the classes of an object. Hence the objects themselves won't directly benefit from their added meta-descriptions. For instances it is still possible to use direct instance variable access inside an instance and assign values which conflict with the well-defined meta-description. Thus meta-modelling frameworks show only the same behavior as first-class slots when attributes of objects are accessed solely trough the meta-descriptions. But due to the decoupled implementation this is not enforced and rather relies on the discipline of the programmer.

2.2.1 Annotations

Several programming languages support annotations to attach metadata to program structure. Java annotations, available since version 5.0 of the language, are probably the most prominent example. Annotations are generally a way to directly attach meta information to source elements. Later on the information in the annotations can be queried using a reflection API. In this sense annotations cannot be used directly to alter state access. However it is possible to provide new tools which use annotation to control access and validate the state of a model. In Java, annotations can be supplied for classes, methods and instance variables. Generally the annotations are only used for adding meta-descriptions to the code. This metadata is then later accessed at runtime using reflection. Example use-cases of annotations include unit-tests [3] and compile-time model constraints verification [12].

Annotations can be used to avoid manually written boilerplate code by generating code from the annotations. Java 6 features pluggable annotation processors that can hook into the compiler and transform the AST. However it is not possible to directly modify the annotated sources. Using this infrastructure it is only possible to create new class definitions that take slot definitions into account. Due to this limitation it would be required to use the generated sources.

2.2.2 Object Relational Mapping

A special case of meta modelling worth mentioning is the use of structural and semantic meta information to model object relational mapping [2]. Meta information is needed to provide a meaningful mapping from the objects to the database. However generally the objects should stay fully functional thus some part of the semantics described in the meta information has to be available.

Several object oriented front ends for relational databases support slot-like structures to describe the database fields. Django [10] provides several types of fields to describe and constrain what kind of data can be stored in the different instance variables. This metadata is further used to create the table description. Although the field descriptors could be directly used to generate getters and setters which dynamically validate the assigned data, this is only done when serializing the object to the database. As such relationship are only indirectly usable by storing and loading objects from the database.

In the Active Record implementation used with Ruby on Rails class-side methods are used to create descriptions of the fields used in a table. These methods use Ruby's reflective capabilities to install getters and setters. In this sense there are no slots objects but class-side methods to create slot descriptions.

2.3 First-Class Slots

The Common Lisp Object System (CLOS) [7, 20] provides support for first-class slots[1]. Upon defining a class slots are described as part of the class definition. Internally CLOS uses this information to decide which slot class to use. Standard CLOS always relies on the default slot class. In Persistent CLOS (PCLOS) [19] the lookup was customized, to decide based on an extra keyword whether the default or the persistent slot class should be used. On accessing slots the `slot-value` function is called. This is a generic function similar to the `instVarAt:` and `instVarAt:put:` methods in Smalltalk which can be used to directly access the fields of an object using indices. It can be overridden to specialize slot access for the entire class. Internally this function relies on a class-side method `slot-value-using-class`. This method can finally specialize variable access to the type of class and the type of slot.

[1] What we call fields is called slots in CLOS. Slot is named `slot-descriptor` in CLOS.

While CLOS already provides slots as one of the main missing reifications, standard CLOS does not provide a way to specialize instance variable access. As PCLOS shows it would however be fairly easy to hook into the protocol and allow programmers to provide custom slot metaobjects. CLOS however does not reify any instance structure beyond the level of slots.

The E programming language [25] provides slots as objects representing the location where values of instance variables are stored for specific instances. This model is the closest to what is presented in the paper. However it requires the system to generate a multitude of objects for each user-level object, as all instance variables of a single instance need their own metaobject.

Since C and C++ provide references which can be used to mimic the availability of first-class slots. However such references are simply *lvalues* providing direct access to the raw memory. They cannot influence any access semantics, nor do they provide a higher-level abstraction that can be reused by other instance variables by bundling accessor methods.

2.4 Slots as Methods

In Self [26] and Newspeak [8] everything is a message send. Slots are just special methods that return a value. This forces the user to always access values through a standardized interface that can flexibly react to change. Even more interesting is that (Selfs version of subclasses) can change the behavior of slots just like methods can be specialized. Data becomes completely public however since objects cannot be hidden. Since the accessors and initialization code have to be overridden separately, this implies that their specialization has to be done over and over again for each individual slot. There is no standard way to bundle these methods in a specialized metaobject and install them as a single unit.

3. Flexible Object Layouts in a Nutshell

Tools like compilers and class builders are needed to support programming languages. They are necessarily linked to the runtime that they target. If they are however too tightly coupled to the assumptions made in the VM they become less extensible. By introducing flexible object layouts as a new layer of abstraction between the programming language and its runtime, we decouple language tools and the runtime. This layer consists of three main concepts, directly related to the low-level view of how classes are constructed: layouts, layout scopes and slots.

Layouts are the direct reification of the object headers known to the VM. Just like VMs generally relate an object header to the class of an object, we relate a single layout instance to each class. A class knows its layout, and the layout knows the class to which it belongs. As shown in Figure 1, layouts are installed in `Class Behavior`, the superclass of both `Class` and `Metaclass`, in the `layout` instance vari-

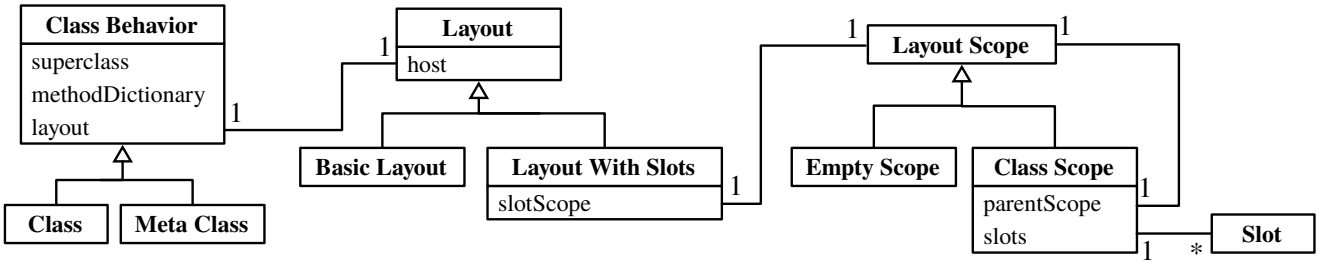

Figure 1. Flexible Object Layouts Overview

able. The layout itself links back to the `Class Behavior` through the `host` instance variable.

The number of available layout types depends on the VM. Our prototype implementation is implemented in the Pharo Smalltalk [21] dialect and provides a fairly typical set of object types: words, bytes, pointers, variable sized, weak pointers, compiled methods, and small integers [15]. Pharo additionally relies on compact classes to save memory for the instances of widely used classes by not keeping a pointer from an instance to a class. All this information encoded in the object header, which is normally only available to the VM, is now directly available in the first-class layout.

Layout scopes group instance variables that are declared in the same scope. Apart from a few special cases, most classes declare a collection of instance variables. Such classes are related to a *layout with slots*. A class inherits instance variables from its superclass and potentially adds several itself. In our abstraction layer this is directly modeled using layout scopes. The different layout scopes are nested in a hierarchy parallel to that of the class structure. As Figure 1 shows, layout scopes are contained by layouts with slots.

Slots are a first-class representation of instance variables and their corresponding fields[2]. They are referred to by a program's source code when their name is mentioned in an instance variable access. As such they can modify read and write access to fields. In our current implementation the access semantics defined by the slots are directly inlined by the compiler. As Figure 1 shows, slots are contained by layout scopes.

Figure 2 illustrates our model using the layout of `Dictionary`. This particular class builds instances with a total of two fields, related to the instance variables `#tally` and `#buckets`. Classes that build such instances with a fixed size have a `Pointer Layout`. Since this layout is a subclass of `Layout With Slots`, it is related to a *class scope*. Since `Dictionary` is a subclass of `Hashed Collection`, the class scope of `Dictionary` has as `parentScope` the class scope of `Hashed Collection`. Because `Hashed Collection` has two direct instance variables `#tally` and

[2] For clarity we refer to the memory location in an object as the *field*. The token in source code that refers to the field we call *instance variable*.

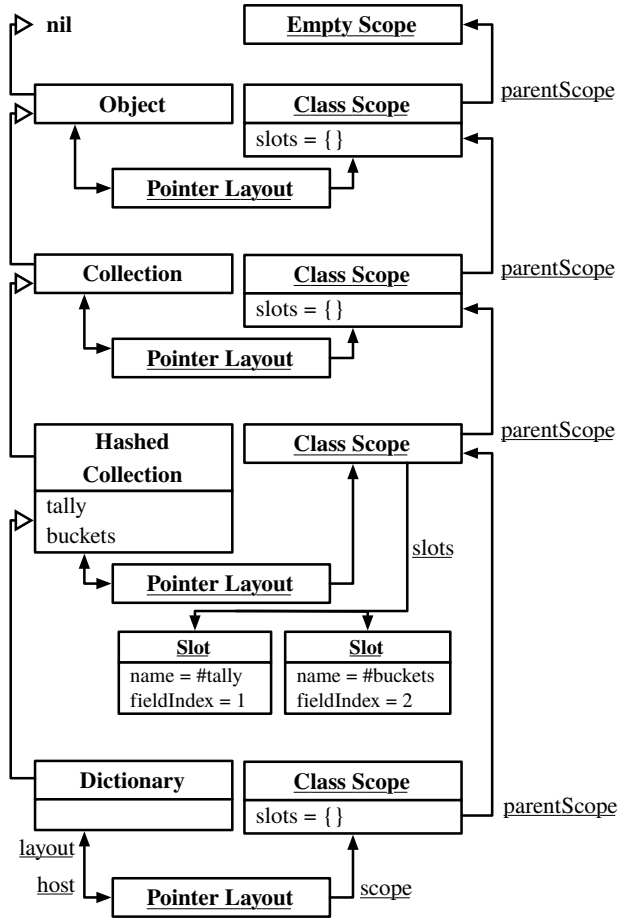

Figure 2. Scopes related to `Dictionary`

`#buckets`, they are linked as slots from the related class scope. Since `Object` has no slots, its class scope is empty. A list of scopes generally ends in the *empty scope*, just like lists end in `nil`.

4. First-Class Slots

While normally it is the compiler that is solely responsible for mapping instance variables to fields, slots provide an abstraction that can assume this responsibility. This allows the slots to influence the semantics of accessing instance variables. We distinguish between four types of actions: initial-

ization, reading, writing and migration. Slots can specialize the semantics of any of these four actions by overriding the related method in the slot class definition.

We classify slots as follows:

- *Primitive slots* are the direct reification of the link between instance variables and fields.
- *Customized slots* define custom semantics for the four main actions related to slots: initialization, reading, writing and migration.
- *Virtual slots* have no direct representation in the related objects but rather read state from an aliased field or derive their state in another way.

4.1 Primitive Slots

Primitive slots are metaobjects that simply bind an instance variable to a field index.

```
Object subclass: #Slot
    layout: PointerLayout
    slots: {
        #index => Slot.
        #name => Slot.
    }.

Slot >> initializeInstance: anInstance
    self write: nil to: anInstance

Slot >> read: anInstance
    ↑ anInstance instVarAt: index.

Slot >> write: aValue to: anInstance
    ↑ anInstance instVarAt: index put: aValue.
```

Listing 1. Default Slot Implementation

Instance variables are by default replaced by standard `Slot` instances. Listing 1 shows the core implementation of the default Slot with the three actions for slots:

- *Initialize:* The method named `initializeInstance:` is called during object instantiation for all Slots. As in most languages the fields of newly created object are initialized with `nil`.
- *Read:* The `read:` takes the object instance as an argument and uses the low-level `instVarAt:` to directly access the field in the instance.
- *Write:* The `write:to:` method works similar to the `write:` method and delegates the write access to the low-level `instVarAt:put:` operation.

Notice that the slots used by the definition of `Slot` are such standard metaobjects, making the `Slot` definition a recursive one. As shown in Listing 1, to read out a standard slot we need to first access the `index` slot of the slot. But to access the `index` slot, we need to be able to access the `index` slot, and so on. [3] This circularity is however easily broken by letting the VM directly execute it. Slots break the recursion by directly inlining accessor code into the methods that refer to them.

[3] This is similar to methods. They are conceptually instances of the `Method` class. While this class could have a method telling the runtime how to execute the method, this equally recurses infinitely.

4.2 Customized Slots

Whereas accesses to primitive slots immediately translate into accesses to the related field, it is advantageous to be able to customize the semantics of accessing the slot into something more elaborate. There are four main types of actions related to slots: initialization, read, write and migration.

In standard object-oriented code initialization of slots is handled directly in constructor methods. This implies that initialization code needs to be duplicated for similar but different instance variables, independent of the instance variables being present on the same class. By providing an initialization mechanism on the slot metaobject this initialization code is shared between all instance variables related to the same type of slot. The initialization procedure can be further customized towards the class and finally the instance.

By customizing the reading and writing of slots we directly influence all source code that mentions the related instance variable. A slot is read by using it as an *rvalue*. This triggers the protocol `slot read: anInstance`. Slots are written to by using it as an *lvalue*, triggering the protocol `slot write: aValue to: anInstance`. This allows developers to create reusable components at the level of instance variables that avoid the need for boilerplate code to access them.

Finally slots are related to class updates. Whenever instance variables of a class are removed or added this directly impacts the class and its subclasses, their methods and all their instances. While full-blown solutions to class updates are outside the scope of this paper, it is important to mention that our model supports the construction of solutions for class updates. Slots can determine how instances should be migrated at the field-level.

4.2.1 Type-checked Slots

As a first example Listing 2 shows how we can easily build type-checked slots.

```
Slot subclass: #TypedSlot
    layout: PointerLayout
    slots: {
        #type => TypedSlot type: Class.
    }.

TypedSlot >> write: aValue to: anInstance
    (aValue isNil or: [aValue isKindOf: type])
        ifFalse: [ InvalidTypeError signal ]
    ↑ super write: aValue to: anInstance.
```

Listing 2. Typed Slot Implementation

Although it is possible to provide the same functionality as slots by implementing accessor methods, this does not provide the same level of abstraction. It is not possible to enforce that all code indirectly accesses the state over a getter

method. Each instance variable requiring preconditions to be fulfilled can also be used in a direct way, circumventing the tests. By relying on slots however, the programmer has only one single way access the instance variable. Here the guard can be enforced for all methods.

A second advantage of using typed slots rather than relying on modified setter functions is that the semantics of the slot are reified. In the case of typed-checked slots this already provides metadata to gradually add typing to the partly dynamically-typed application, a technique also known as hardening [27].

By encapsulating type checks in slots we can avoid code that would otherwise duplicated. In an untyped language type checks would either occur at instance variable write or in setter methods. Since we can even create a specific slot class for a specific, for example `PositiveIntegerSlot`, there is no need to explicitely type check instance variables anymore.

4.2.2 First-class Relationships

Using slots it is possible to model first-class relationships that integrate seamlessly into the existing language. We model relationships by modeling both possible sides of *one-to-one*, *one-to-many* and *many-to-many* relationships by defining a `One Slot` and a `Many Slot`. To complete the relationship such slots will then have another one or many slot as *opposite* slot.

Listing 3 implements two classes `Boss` and `Clerk` that are in a one-to-many relationship. A boss has a staff of many clerks, but a clerk just has a single boss. In step 1 of Figure 3 we create one instance of the `Boss` class and N `Clerk` instances. In step 2 we set the boss of `c1` and `c2`. This makes the boss have two clerks as his staff, and the two clerks have a boss. All the other clerks are unaffected. In step 3 we overwrite the staff by the array of clerks `c3` till `cN`. This breaks the relationship between the boss and `c1` and `c2` and creates new relationships with the clerks `c3` till `cN`. If in step 4 we set the staff of the boss to the empty array, all relationships are broken again.

```
Object subclass: #Boss
    layout: PointerLayout
    slots: {
        #staff => ManySlot opposite: #boss
                          class: Clerk.
    }

Boss >> staff: aCollection
    staff := aCollection

Object subclass: #Clerk
    layout: PointerLayout
    slots: {
        #boss => OneSlot opposite: #staff
                         class: Boss.
    }

Clerk >> boss: aBoss
    boss := aBoss
```

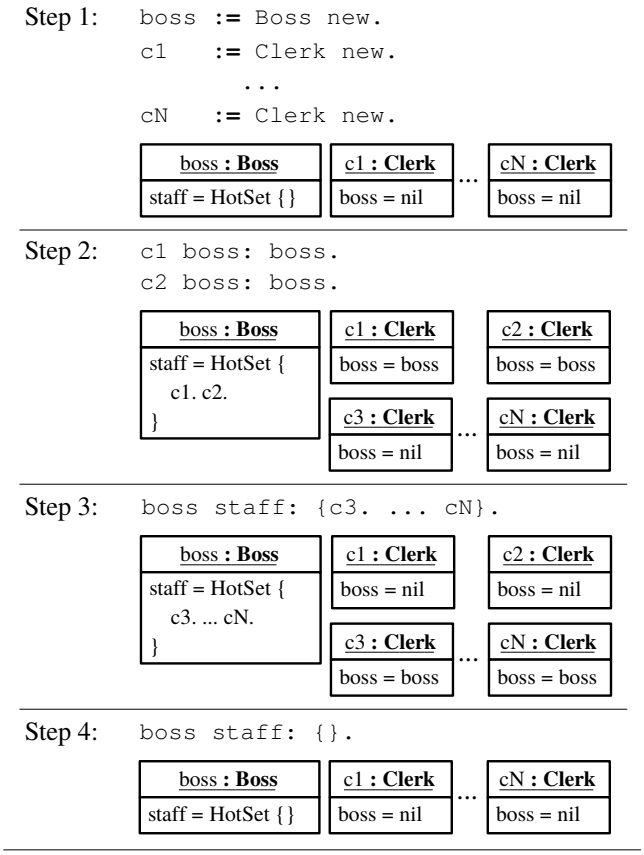

Figure 3. Relationships in Action

Listing 3. Many Relationship Usage

The code in Listing 4 shows the full implementation of the related classes. Both ends of a relationship need to be typed, so we reuse the `Typed Slot` class from Listing 2. We extend it by adding a subclass `Opposite Slot` that knows that both slots that occur in a relationship refer back to each other using the opposite instance variable[4]. Finally the `One Slot` knows that it will contain a single value, while the `Many Slot` has many values. To make the picture complete, in the case of a `Many Slot` we install a *hot collection*. This is a special kind of collection that knows that it has to update the opposite side on every change. This is required since the collection itself is a way to avoid having to directly access the data via the slot.

```
TypedSlot subclass: #OppositeSlot
    layout: PointerLayout
    slots: {
        #opposite => OneSlot opposite: #opposite
                             class: OppositeSlot.
    }
```

[4] Notice that `#opposite` is also declared as a `One Slot`, referring back to itself. This is because a slot `#y` that has slot `#x` as its opposite, is by itself the opposite of `#x`. Slots that are in a relationship at the base-level are also in a relationship on the meta-level.

```
OppositeSlot subclass: #ManySlot
    layout: PointerLayout
    slots: {
        #oppositeHost => Slot.
    }

ManySlot >> initializeInstance: anInstance
    |set|
    set := HotSet new
        oppositeSlot: opposite;
        myself: anInstance;
        type: self oppositeHost.
    super internalWrite: set to: anInstance.


ManySlot >> write: aValueCollection to: anObject
    |hotSet|
    hotSet := self read: anObject.
    hotSet removeAll.

    aValueCollection ensureType: Collection.
    hotSet addAll: aValueCollection.


OppositeSlot subclass: #OneSlot
    layout: PointerLayout
    slots: {}

OneSlot >> write: aValue to: anObject
    (self internalRead: anObject)
        ifNotNilDo: [:oldValue|
            opposite remove: anObject from: oldValue].

    super write: aValue to: anObject.

    aValue ifNotNil: [opposite add: anObject to: aValue].
```

Listing 4. Relationship Slot Implementation

There are several advantages to using slots rather than specific language extensions. A library encapsulates the core behavior of relationships but still requires a significant amount of glue code to invoke all necessary hooks. However it is possible to dispense with glue code altogether by implementing the first-class relationships directly as a language extension. But language changes require all the tools to be changed as well. Hence we argue in favor of an implementation which solely requires first-class slots that intercept read and write access. As shown in Listing 3 it is sufficient to specify the relationship with slots.

4.3 Virtual Slots

Virtual slots do not require a field in the related object but redirect access the data elsewhere.

Alias slots are a trivial kind of virtual slot that simply redirect all accesses to the aliased slots. Listing 5 shows the basic implementation details of the alias slot. The basic access operations `read:` and `write:to:` are forwarded to `aliasedSlot`. This is useful for providing a compatibility interface for legacy or external code. Wrongly named variable accesses can be redirected by specifying an alias to an existing slot. Since accesses to the slot are directly compiled as accesses to the aliased slot there is no extra overhead by using an alias slot over a normal one.

```
VirtualSlot subclass: #AliasSlot
    layout: PointerLayout
    slots: {
        #aliasedSlot => TypedSlot type: Slot.
    }.

AliasSlot >> read: anInstance
    ↑ aliasedSlot read: anInstance

AliasSlot >> write: aValue to: anInstance
    ↑ aliasedSlot write: aValue to: anInstance
```

Listing 5. Alias Slot Implementation

Derived slots are computed from the values of other slots. They can be used for example to provide a dual representation of values without having to duplicate support code or add explicit transformation code. The code in Listing 6 shows a `Color` object which has three standard slots for the three color compounds red, green and blue. The fourth slot is a virtual slot combining the three compounds into a single integer value. The `RGBSlot` internally links to the three other color components, denoted by the slots named `#r`, `#g` and `#b`. Internally the `RGBSlot` uses these slots as sources and transforms the input and output to represent one single integer value.

On assignment the `RBG Slot` splits the written integer value into the three compounds and forwards them to the corresponding slots. On read access the single integer value is computed from the three other slots. By reading from the `rgb` instance variable the full combined integer value is read. This has the advantage over a normal method invocation in that it can be directly inlined by the compiler and that it stays private to the class. Whenever this dual number representation is required elsewhere it is sufficient to copy over the `RGB Slot` and thus the slot helps to reduce code duplication.

```
VirtualSlot subclass: #RGBSlot
    layout: PointerLayout
    slots: {
        #redSlot   => TypedSlot type: Slot.
        #greenSlot => TypedSlot type: Slot.
        #blueSlot  => TypedSlot type: Slot.
    }.

RGBSlot >> read: aColor
    ↑  (((redSlot   read: aColor) & 0xFF) << 16)
     + (((greenSlot read: aColor) & 0xFF) << 8)
     +  ((blueSlot  read: aColor) & 0xFF).

RGBSlot >> write: anInt to: aColor
  redSlot   write:((anInt & 0xFF0000) >> 16) to: aColor.
  greenSlot write:((anInt & 0x00FF00) >>  8) to: aColor.
  blueSlot  write: (anInt & 0x0000FF)        to: aColor.


Object subclass: #Color
    layout: PointerLayout
    slots: {
```

```
        #r   => PositiveIntSlot limit: 0xFF.
        #g   => PositiveIntSlot limit: 0xFF.
        #b   => PositiveIntSlot limit: 0xFF.
        #rgb => RGBSlot redSlot: #r
                        greenSlot: #g
                        blueSlot: #b.
    }.
```

Listing 6. RGB Color Slot Implementation

5. First-Class Layout Scopes

In our system a layout consists of layout scopes, which themselves again can contain slots. Layout scopes provide a level of reusable object semantics that is orthogonal to the standard reuse through subclassing. They are responsible for providing access to instance variables and requiring the fields in the final instance. This allows them to influence the visibility of slots while still requiring enough space for all slots in the final instances. By creating custom layout scopes we can implement more complex use-cases which would otherwise require boilerplate code.

The two core scopes, the empty scope and the class scope are shown in Figure 1. As a default for each class a class scope is generated and link to a parent scope which holds the slots form the superclass. These layout scopes form a chain which eventually ends in an empty scope, as shown in Figure 2. With this approach we have a compatible model to represent slot reuse through subclassing. So far we only assumed that the scopes will contain exactly the slots from the class definition. The following examples however, show situations where new slots are introduced depending on the slots specified in the class definition. We introduce additional slots by adding specialized scopes. Hence the class scopes always contain exactly the scops provided with the class definition.

In addition to the empty scope and class scopes two general groups of additional layout scopes exist. *slot hiding scopes* only give access to a part of the actually declared slots, and *slot issuing scopes* give access to more slots than are declared by the scope. The following two examples both introduce a new slots and thus are to be seen as slot issuing scopes.

5.1 Bit Field Layout Scope

In several languages the number of instances variables is restricted, for instance many Smalltalk VMs limit the number of instances variables to something less than 64 on many systems. When using many instance variables that only use booleans it feels natural to combine them into a single field. Normally each instance variable would require a full pointer to store a value that can be represented with a single bit. Combining these variables into a single field helps to reduce the memory footprint of an object. In our implementation we can combine multiple boolean fields into a single bit field. This not only reduces the memory footprint but also helps to speed up the garbage collection. Due to the single field the garbage collector has to traverse fewer fields.

Listing 7 shows the implementation details of the `BitSlot`. Each `BitSlot` knows its storage location in the object denoted by the `bitSlot` instance variable. The `bitIndex` is used to extract the corresponding bit out of the integer stored at the location of the `bitSlot`. In order to read a boolean value, the `BitSlot` reads the full integer value from the `bitSlot` and masks out the corresponding bit.

```
VirtualSlot subclass: #BitSlot
    layout: PointerLayout
    slots: {
        #bitIndex     => PositiveIntegerSlot.
        #bitFieldSlot => TypedSlot type: BitHolderSlot.
    }

read: anInstance
    mask := (0x01 >> index).
    ↑ (bitFieldSlot read: anInstance) & mask == mask.

write: aBoolean to: anInstance
    |int|
    int := bitFieldSlot read: anInstance.
    int := int & (0x01 >> index) invert.   "mask the bit"
    int := int | (aBoolean asBit >> index) "set the bit"
    bitSlot write: int to: anInstance
    ↑ aBoolean
```

Listing 7. Bit Field Slot Implementation

Using bit fields in a normal object is a matter of changing the slot definition. Instead of using the default `Slot` the `BitSlot` has to be used. Listing 8 shows the basic definition of an object using bit slots. When using such an object up to 30 bit slots are combined into a single field. Figure 4 shows a transcript of how the boolean values are written to the single instance variable. If this were implemented without encapsulating the behavior in slots, the code of the `write:to:` or `read:` method would have to be copied at least into a getter or setter. In this case there is a single definition of the extraction semantics in the bit slot, which serves as a template.

```
Object subclass: #BitObject
    layout: PointerLayout
    slots: {
        boolean1 => BitSlot.
        boolean2 => BitSlot.
                ...
        booleanN => BitSlot.
    }
```

Listing 8. Bit Object using Bit Slots

Unlike the previous examples of slots the `BitSlots` require the layout to add a storage slot. As a reminder, the `BitSlots` are virtual and hence do not occupy a field in the instance. The situation is further complication in that the number of storage slot is not fixed and depends on the number of `BitSlots`, since each storage integer fixed number of bits. Instead of changing the current class scope, and thus obfuscating the original slots definition we add specific bit

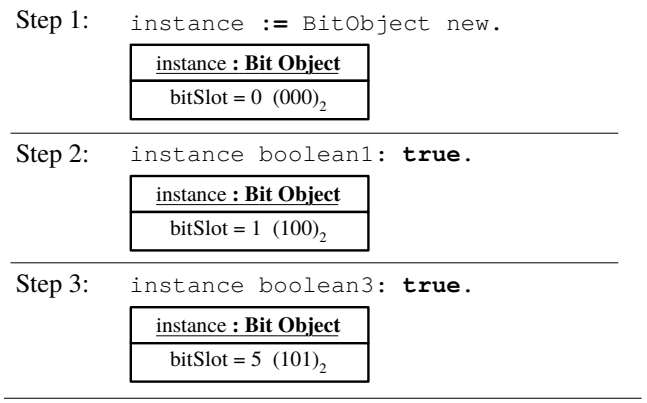

Figure 4. A BitField Instance in Action

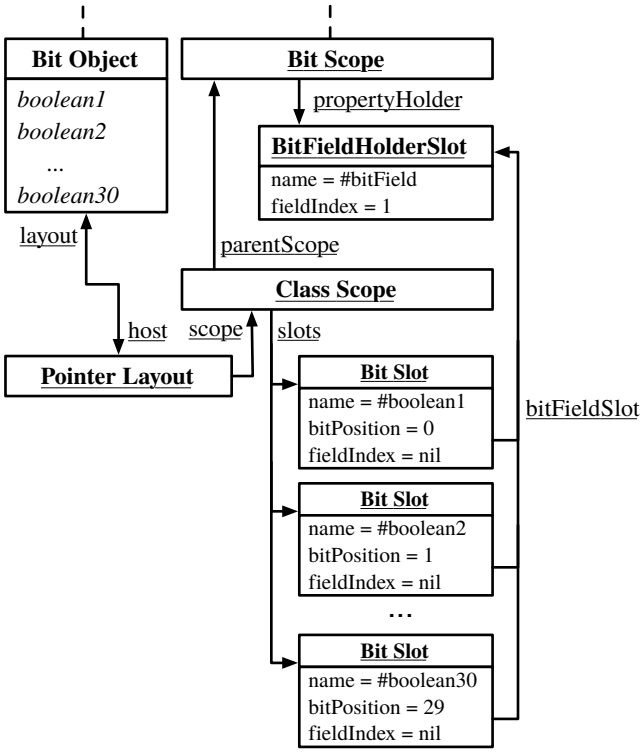

Figure 5. Bit Field Scope Example

scopes. In Figure 5 we see that the layout of the `Bit Object` points to a normal class scope which contains the slot definition mentioned in the class definition. Instead of linking directly to the class scope defining the slots of the superclass the parent scope is set to a special bit field scope. The bit scope internally contains the bit slot which is used to store the different bits in it. Each virtual bit slot points to a non-virtual bit field slot defined in a bit scope.

5.2 Property Layout Scope

The previous example using bit fields displayed that by using slots and slots scopes it is possible to transparently optimize the footprint of an object using boolean instance variables. Here we will show how this technique can be used to selectively but drastically transform the layout of objects.

JavaScript [13] and Python [22] use dictionaries as the internal representation for objects. This enables the dynamic addition of instance variables and saves memory when there are many unused instance variables. The simplicity of the object design comes with two major drawbacks however: 1) typing mistakes in instance variable names are not easily detected, and 2) attribute access is difficult to optimize.

In standard Smalltalk the number of instance variables is fixed up-front in the class. However we easily overcome this limitation by using an intermediate dictionary which holds all the object's instance variables. Without first-class layouts this would enforce us to use unchecked symbols as field names to access the properties. Furthermore each instance variable access would have to be manually replace with a dictionary access, which can be completely avoided in our case. In our implementation it is possible to benefit from both worlds by only enabling dictionary-based storage where it is needed, while still providing syntax checking for slots.

```
Object subclass: #PropertyObject
    layout: PointerLayout
    slots: {
        field     => Slot
        property1 => PropertySlot.
        property2 => PropertySlot.
                  ...
        propertyN => PropertySlot.
    }
```

Listing 9. Property Object using Property Slots

Listing 9 provides a class definition of an object which uses a normal slot, named `field` and an arbitrary number of virtual slots that use a dictionary as storage target. Figure 6 shows an example usage of this property object. Note that the resulting instance uses only two fields. The property holder named `dict` is lazily filled with the values for the different properties.

Similar to the previous bit field example we have to introduce a data holder slot depending on the types of specified slot. In this case we use a special property scope. Figure 7 shows that the property scope holds an instance of a `PropertyHoldertSlot` which is required to reserve one field for the property storage. This field holds a *property dictionary* that maps *property slots* onto their values. Listing 10 shows how accesses are rewritten by property slots such that they access the state via the property dictionary.

```
VirtualSlot subclass: #PropertySlot
    layout: PointerLayout
    slots: {
        #dictSlot => TypedSlot type: PropertyHolderSlot.
    }.

PropertySlot >> read: anInstance
    ↑ (dictSlot read: anInstance)
            at: name ifAbsent: [ nil ].
```

Step 1:
```
instance := PropertyObject new.
```

instance : Property Object
dict = { } field = nil

Step 2:
```
instance field: 'real field'.
```

instance : Property Object
dict = { } field = 'real field'

Step 3:
```
instance property1: 'string'.
instance property2: false.
```

instance : Property Object

dict = {
#property1 → 'string'.
#property2 → false.
}
field = 'real field'

Step 4:
```
instance property1: nil.
instance field: nil.
```

instance : Property Object

dict = {
#property2 → false.
}
field = nil

Figure 6. Property Object in Action

```
PropertySlot >> write: aValue to: anInstance
    ↑ (dictSlot read: anInstance) at: name put: aValue.
```

Listing 10. Property Slot Implementation

This approach has three main advantages over the default behavior in Python or JavaScript. First the overall performance of the system does not suffer since only the accesses of selected property slots are rewritten to go over the dictionary. Secondly converting a property slot into a normal slot is matter of changing the type of slot. The only difficulty being that special care has to be taken to convert the values in property dictionaries of existing live instances back into normal fields. Finally, in contrast to the standard Python or JavaScript approach our model minimizes the risk of runtime errors related to misspelled variable names by requiring the property slots to be explicitly specified in the layout scope up-front. This allows us to provide proper compile-time checks of property slots just like for all other Smalltalk slots.

6. Stateful Traits

In this section we show that by reifying the state of the objects and making it available in the programming language new concepts that revolve around state can be implemented with less effort. As case-study we implement stateful traits [11], a mechanism for sharing behavior and state in standard

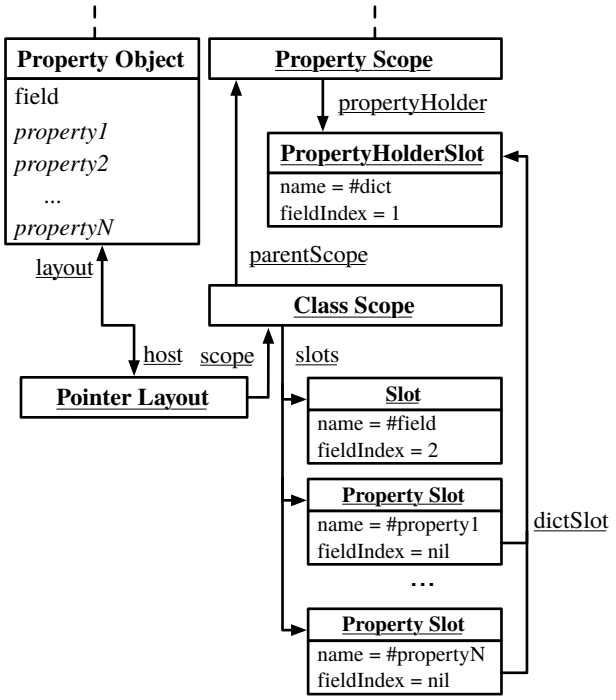

Figure 7. Property Scope Example

object-oriented systems which is orthogonal to subclassing. Stateful traits are components of reuse that are more fine-grained than classes but generally larger than slots.

Although a previous implementation for Smalltalk exists it was more difficult to attain and includes ad-hoc solutions like renaming instance variables to avoid name clashes.

6.1 Traits

Fundamentally traits are used as collections of reusable methods that are installed on classes. Normally, installing a trait is implemented by flattening out the collection in the method dictionary of the target class. All conflicts resulting from installing a trait have to be resolved by the developer. This includes renaming methods, rejecting methods and overriding methods.

On installing a trait, the trait object is copied before installation. The trait methods are recompiled on the receiving class to ensure correct semantics for superclass sends. Whenever a trait-related method is modified, the trait and all its users are notified of the change and updated accordingly. A trait is uninstalled by removing all methods introduced by the trait. Finally whenever a class is updated, its trait composition is copied over from the previous version of the class to the new version of the class.

6.2 Stateful Traits

Stateful traits [4] add the possibility to define state related to traits. When composing a stateful trait with a class not only the methods are installed, but also the associated state is added to the class. Stateful traits are closely related to mixins

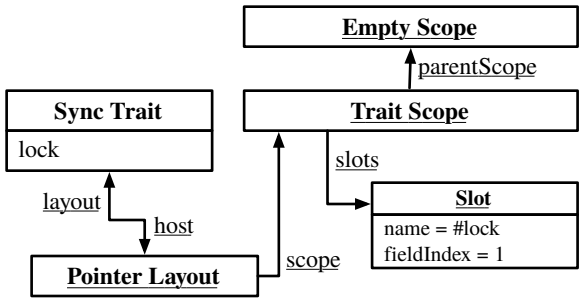

Figure 8. Stateful Trait Example

[9] except that they follow the conflict-avoiding composition strategy of standard traits.

6.3 Installing State of Stateful Traits

By relying on our model of first-class layouts and scopes it becomes straightforward to extend traits with state. A stateful trait is in essence a subclass of Trait which is extended with a layout. Where we previously declared a class to use a trait, we can now allow it to equally rely on a stateful trait. The behavior has to be mixed in the exact same location as standard traits. The only additional step required is the mixing of the state declared by the trait with the state declared by the class. Figure 8 shows an example of such a stateful trait, the `Sync Trait`. In addition to the provided methods, the stateful trait has a layout. This layout is linked to the related *trait scope* that contains a single slot `lock`.

The class builder is the tool responsible for installing the state of a stateful trait. During this process we want to avoid name clashes with the state of the target class. To avoid complex renaming required by the original stateful trait work, we introduce a new kind of layout scope in our model, the *fork scope*. A fork scope is a scope that does not only have a parent scope, just like a normal class scope, but also a list of side scopes. The side scopes contribute to the final number of fields that an associated object has, but they do not provide any visible slots. Their state in the resulting object is essentially private to the owner of the scope. The trait scopes are then installed in the fork scope as side scopes. Figure 9 shows how the trait from Figure 8 is applied to the `Sync Stream` class.

The modelling challenge when installing stateful traits is to correctly update the field indices and scope instance variable access. The index calculation has to take into account the fields in the superclass and other installed traits.

For the normal operations on a class (*e.g.*, compiling a method inside the class) the visible slots will be computed by recursively traversing the parent links of the scopes, aggregating the slots from the class scopes, but ignoring the side scopes of fork scopes. During compilation of the trait-specific behavior the trait providing the behavior is used as compilation target. This way, at compilation time, the class methods do not have access to the trait state and vice-versa.

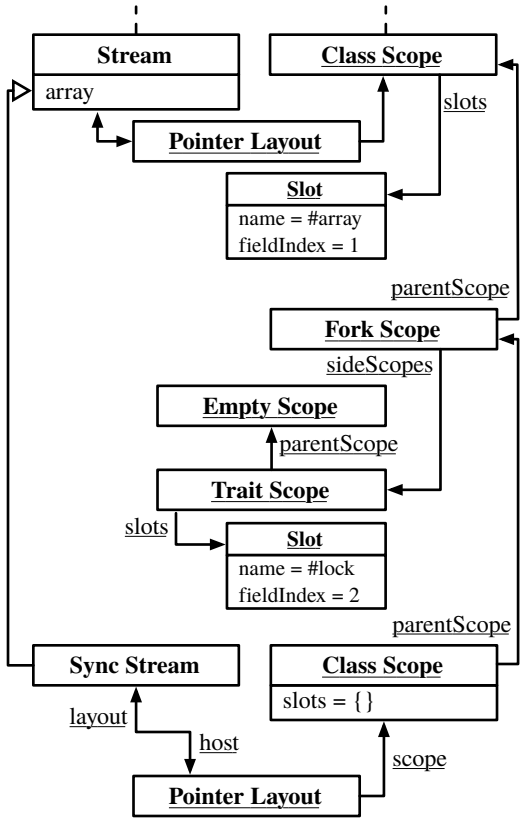

Figure 9. Stateful Trait with Fork Scopes

6.4 Installing Behavior of Stateful Traits

After the class has been successfully composed by the class builder, the methods of the used traits are installed. Stateful trait methods are installed by updating them in the context of the installed trait copy, meaning in the context of the trait scope that was installed rather than the original trait scope. The indices of the slots in the installed scope are already updated to reflect their installed offset. Recompiling methods in the installed scope will equally update them to reflect this modification. This is a simplification of the *copy down* technique [1] in that it does not try to save memory but always installs methods by copy. Bytecode modification can be used to reduce the runtime overhead of installing traits.

7. Class Building

In Smalltalk class definitions are themselves objects just like anything else. Modifying the class definitions implies changing the layout of the class objects. Since a class is part of a class hierarchy this layout change has to be propagated throughout the hierarchy.

The class builder is the programming language tool that supports this evolution of class definitions. It requires the most changes to support our layout-based model. However, after the changes the implementation becomes much simpler since its responsibilities are now distributed over the differ-

ent types of layouts, layout scopes and slots, making it more adept to change:

- A layout knows the rules the VM imposes on it, for example that it is not allowed to subclass a class with byte array slots.
- Slot scopes know exactly how many slots they have and which slots are supposed to be visible.
- Slots themselves know their access semantics and how they are related to fields in instances.

By delegating the responsibility to those specific objects, we allow libraries to provide new types of layouts or slots without modifying the class builder. The class builder will gladly support such metaobjects as long as they are polymorph to the primitive slot and layout.

7.1 Class Modification Model

The class builder itself is responsible for building or changing classes for a *class installer*. The builder constructs new classes from their definition and notifies the installer about the new class. In case the class installer already had a version of the new class, the class builder builds up a *class modification model* of the difference between the old class and the new class. In case the old class had subclasses and if the change impacts the layout of the final instances or its methods, this has to be propagated to the subclasses. For each subclass of a class modification, a *class modification propagation* is created. These propagations are cheaper than the normal propagation since the fields they add have to be shifted by the difference in the number of fields of the superclass. All the other changes are the same for the modified class as for any of its subclasses, so it only needs to be calculated once for each class hierarchy.

The class modification is validated by checking if all new layouts are consistent. This means that no slots should be introduced that mask or will be masked by existing slots, and that all layout types in the hierarchy are still compatible.

A class modification is calculated by asking the new layout how it differs from the old layout. The layouts rely all of their slots, hidden and visible, to calculate the exact model. The layouts know which slots have been removed, added or were modified in place. Once the class modification is ready we have all the information required to build an *instance migration model* and a *method modification model*.

Once the modification model is validated the class builder will tell the class installer to perform the migration. This consists of copying over the related methods and updating their code, migrating instances to the new classes and finally fully replacing the old version of the class hierarchy by the new hierarchy. Since everything is validated up front we know that this change will not raise any conflicts and can apply the migration in a single transaction.

7.2 Instance Migration

Instance migration is a fairly low-level operation that builds instances from a class-based on other instances from an old version of the class. Our instance migration model is built directly from the class modification model-based on the knowledge of how slots have changed since the old version of the class.

Since instances are built at the level of fields, our instance migration model is built up using *field modification* objects, specifying the semantics of how to initialize the fields in the migrated instances. There are five different types of field modification objects: added, removed, modified, shifted and unmodified.

Since the field modification objects have access to the slot objects, they can forward this decision back to the slot. This provides another new hook into the meta-system that allows users to control exactly what happens to their instances when software is updated.

7.3 Method Updating

Similar to the instance migration model, we build a method modification model. Depending on how the instances changed (*e.g.*, a field was moved) the method sources have to be updated as well. These modifications are again calculated from the class modification model. Whereas the instance migration builds new instances from old instances, the method modification model is used by a *method field updater* that applies bytecode rewriting on existing methods to reflect the new layout of the instance.

Whenever a field is removed, we replace the access to the field by a message send to the removed field object. This will throw an exception at runtime when someone tries to read from the removed field instead of maybe reading from an invalid location. By inserting the removed field object we also keep enough metadata to later on retract the inserted code and replace it by another access when the slot is reintroduced.

8. Conclusion

The lack of proper abstractions reifying object state is often the reason for the introduction of boilerplate code. To address this problem we propose to extend the structural reflective model of the language with *object layouts*, *layout scopes* and *slots*. Layouts and slots are first-class representations of the assumptions which conventionally exist only as implicit contracts between the virtual machine and the compiler. Layouts describe the object layout of instances of a class while slots represent the conceptual link between instance variables and fields. Layout scopes reify how classes extend the layout of their superclass.

We have shown

- that first-class slots encapsulate the definition of custom semantics for instance variable *initialization*, *access* and

migration (*e.g.*, first-class relationships), promoting consistent fine-grained reuse,

- and that layout scopes support language extensions (*e.g.*, stateful traits) that influence layout composition.

We have classified slots into *primitive slots*, *customized slots* and *virtual slots* and provided examples for each. The programming language tool that requires the most fundamental change to support our layout-based model is the class builder, and we have shown how even its implementation becomes simpler by using slots.

In the future we would like to investigate further potential uses of our model (*e.g.*, dynamic software updates) and to see whether we can provide better tool support for software development based on the new reflective model (*e.g.*, specialized object inspectors and debuggers).

Code Availability. We have implemented all the concepts that we have presented in this article in Pharo Smalltalk. The code is released under an BSD licence and is available at `http://www.squeaksource.com/PlayOut.html`

Acknowledgments

We would like to thank Erwann Wernli and Niko Schwarz for kindly reviewing earlier drafts and providing constructive feedback.

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Synchronizing Models and Code" (SNF Project No. 200020-131 827, Oct. 2010 - Sept. 2012).

References

[1] L. Bak, G. Bracha, S. Grarup, R. Griesemer, D. Griswold, and U. Hölzle. Mixins in Strongtalk. In *ECOOP '02 Workshop on Inheritance*, June 2002.

[2] R. Barcia, G. Hambrick, K. Brown, R. Peterson, and K. Bhogal. *Persistence in the enterprise: a guide to persistence technologies*. IBM Press, first edition, 2008. ISBN 9780768680591.

[3] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.

[4] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. *Journal of Computer Languages, Systems and Structures*, 34(2-3):83–108, 2008. ISSN 1477-8424. doi: 10.1016/j.cl.2007.05.003.

[5] A. Bergel, S. Ducasse, and L. Renggli. Seaside – advanced composition and control flow for dynamic web applications. *ERCIM News*, 72, Jan. 2008.

[6] G. Bierman. First-class relationships in an object-oriented language. In *ECOOP*, pages 262–286. Springer-Verlag, 2005. doi: 10.1007/11531142_12.

[7] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. Commonloops: Merging lisp and object-oriented programming. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 17–29, Nov. 1986.

[8] G. Bracha. Executable grammars in Newspeak. *Electron. Notes Theor. Comput. Sci.*, 193:3–18, 2007. ISSN 1571-0661. doi: 10.1016/j.entcs.2007.10.004.

[9] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, Oct. 1990.

[10] Django. Django. http://www.djangoproject.com.

[11] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, Mar. 2006. ISSN 0164-0925. doi: 10.1145/1119479.1119483.

[12] G. Ferreira, E. Loureiro, and E. Oliveira. A java code annotation approach for model checking software systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 1536–1537, New York, NY, USA, 2007. ACM. ISBN 1-59593-480-4. doi: 10.1145/1244002.1244330.

[13] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, second edition, Jan. 1997. ISBN 1-56592-234-4.

[14] M. Fowler. Language workbenches: The killer-app for domain-specific languages, June 2005. `http://www.martinfowler.com/articles/languageWorkbench.html`.

[15] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983. ISBN 0-201-13688-0.

[16] P. Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[18] E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0. doi: 10.1145/1142473.1142552.

[19] A. Paepcke. PCLOS: Stress testing CLOS experiencing the metaobject protocol. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 194–211, Oct. 1990.

[20] A. Paepcke. User-level language crafting. In *Object-Oriented Programming: the CLOS perspective*, pages 66–99. MIT Press, 1993.

[21] Pharo. Pharo. http://www.pharo-project.org.

[22] Python. Python. http://www.python.org.

[23] L. Renggli, S. Ducasse, and A. Kuhn. Magritte — a meta-driven approach to empower developers and end users. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*, pages 106–120. Springer, Sept. 2007. ISBN 978-3-540-75208-0. doi: 10.1007/978-3-540-75209-7_8.

[24] L. Renggli, T. Gîrba, and O. Nierstrasz. Embedding languages without breaking tools. In T. D'Hondt, editor, *ECOOP'10:*

Proceedings of the 24th European Conference on Object-Oriented Programming, volume 6183 of *LNCS*, pages 380–404, Maribor, Slovenia, 2010. Springer-Verlag. ISBN 978-3-642-14106-5. doi: 10.1007/978-3-642-14107-2_19.

[25] M. Stiegler. The E language in a walnut, 2004. www.skyhunter.com/marcs/ewalnut.html.

[26] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, Dec. 1987. doi: 10.1145/38765.38828.

[27] T. Wrigstad, P. Eugster, J. Field, N. Nystrom, and J. Vitek. Software hardening: a research agenda. In *Proceedings for the 1st workshop on Script to Program Evolution*, STOP '09, pages 58–70, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-543-7. doi: 10.1145/1570506.1570513.