Classboxes – An Experiment in Modeling Compositional Abstractions using Explicit Contexts

Markus Lumpe
Iowa State University
Department of Computer Science
113 Atanasoff Hall
Ames, USA
lumpe@cs.iastate.edu

Jean-Guy Schneider
Faculty of Information & Communication Technologies
Swinburne University of Technology
P.O. Box 218
Hawthorn, VIC 3122, AUSTRALIA
jschneider@swin.edu.au

ABSTRACT

The development of flexible and reusable abstractions for software composition has suffered from the inherent problem that reusability and extensibility are hampered by the dependence on position and arity of parameters. In order to address this issue, we have defined $\lambda\mathcal{F}$, a substitution-free variant of the λ-calculus where names are replaced with first-class namespaces and parameter passing is modeled using explicit contexts. We have used $\lambda\mathcal{F}$ to define a model for classboxes, a dynamically typed module system for object-oriented languages that provides support for controlling both the visibility and composition of class extensions. This model not only illustrates the expressive power and flexibility of $\lambda\mathcal{F}$ as a suitable formal foundation for compositional abstractions, but also assists us in validating and extending the concept of classboxes in a language-neutral way.

1. INTRODUCTION

In recent years, component-oriented software technology has become increasingly popular to develop modern, large-scale software systems [17]. The primary objective of component-based software is to take elements from a collection of reusable software components (i.e., *components-off-the-shelf*), apply some required domain-specific incremental modifications, and build applications by simply plugging them together. Moreover, with each reuse, it is expected that a component's quality improves, as potential defects are discovered and eliminated [15].

However, in order to be successful, the component-based software development approach needs to provide abstractions to represent different component models and composition techniques [1]. Furthermore, we need a canonical set of preferably language-neutral composition mechanisms that allows for building applications as compositions of reusable software components [14]. However, precise semantics is essential if we are to deal with multiple component models within such a common, unifying framework. As a consequence, we argue, any abstractions suitable for component-based software development need to be based on an appropriate formal foundation [8].

We have previously been studying a substitution-free variant of the λ-calculus, called $\lambda\mathcal{F}$, where names are replaced by *forms* and parameter passing is modeled using *explicit contexts* [8]. Forms are first-class namespaces that, in combination with a small set of purely asymmetric operators, provide a core language to define extensible, flexible, and robust software abstractions [9]. Explicit contexts, on the other hand, mimic λ-calculus substitutions, which are used to record named parameter bindings. For example, the $\lambda\mathcal{F}$-term `a[b]` denotes an expression `a`, the meaning of which is refined by the context `[b]`. That is, all occurrences of free variables in `a` are resolved using form `b`. Thus, the context `[b]` expresses the requirements posed by the free variables of `a` on its environment [13].

But is the $\lambda\mathcal{F}$-calculus a suitable formal foundation for defining compositional abstractions? As a proof of concept, we have used the $\lambda\mathcal{F}$-calculus to define a model for class extensions based on *classboxes* [2, 3]. Classboxes constitute a kind of *module system* for object-oriented languages that defines a packaging and scoping mechanism for controlling the visibility of isolated extensions to portions of complex, class-based systems. More precisely, classboxes define *explicitly named scopes* within which classes, methods, and variables are defined. In addition, besides the "traditional" operation of *subclassing*, classboxes support also the *local refinement* of imported classes by adding or modifying their features without affecting the originating classbox. Thus, classboxes allow one to import a class and apply some extensions to it without breaking the protocol defined between clients of that class in other classboxes. Consequently, classboxes and their associated operations provide a better control over changes, as they strictly limit the impact of changes to clients of the extending classbox.

The rest of this paper is organized as follows: in Section 2, we give a brief introduction into the $\lambda\mathcal{F}$-calculus, followed by a summary of the main characteristics of classboxes in

Section 3. In Section 4, we use $\lambda\mathcal{F}$ to define a general model of classes and classboxes. In Section 5, we present our $\lambda\mathcal{F}$ encodings of classbox operations. We conclude this paper in Section 6 with a summary of the main observations from our classbox modelings and outline future work in this area.

2. THE $\lambda\mathcal{F}$ CALCULUS

The design of the $\lambda\mathcal{F}$-calculus is motivated by our previous observations that the definition of general purpose compositional abstractions is hampered by the dependence on position and arity of parameters [7, 16]. Requiring parameters to occur in a specific order, to have a specific arity, or both, imposes a specification format in which programming abstractions are characterized not by the parameters they effectively use, but by the parameters they *declare.* This, however, limits our ability to combine them seamlessly with other, possibly unknown or weakly specified programming abstractions, because any form a parameter mismatch has to be resolved explicitly and, in general, manually.

To address this issue, we champion the concept of *dynamic binding* that allows for a software development approach in which new functionality can be added to an existing piece of code without affecting its previous behaviour [5]. Consequently, the $\lambda\mathcal{F}$-calculus is an attempt to combine the expressive power of the λ-calculus with the flexibility of position-independent data communication as well as late binding in expressions. In the following, we will briefly illustrate the main abstractions of $\lambda\mathcal{F}$; the interested reader is referred to [8] for further details.

The syntax of the $\lambda\mathcal{F}$-calculus is given in Figure 1. We presuppose a countably infinite set, $\mathcal{L}$, of *labels*, and let l, m, n range over labels. We also presuppose a countably infinite set, $\mathcal{V}$, of *abstract values*, and let a, b, c range over abstract values. We think of an abstract value as a representation of any programming value like integers, objects, types, and even forms themselves. However, we do not require any particular property except that equality and inequality be defined for elements of $\mathcal{V}$. We use F, G, H to range over the set of forms and M, N to range over the set of $\lambda\mathcal{F}$-terms.

Every form is derived from the empty form $\langle\rangle$, a form that does not define any bindings. A form F can be extended by adding a binding for a label l with a value V, written $F\langle l = V\rangle$. With projections we recover variable references of the λ-calculus. We require, however, that the subject of a projection denote a form. For example, the meaning of `F.l` is the value bound by label `l` in form `F`. A projection `a.l`, where `a` is not a form yields $\mathcal{E}$, which means "no value."

The expressive power of forms is achieved by the two asymmetric operators *form extension* and *form restriction*, written `F` $\oplus$ `G` and `F`$\backslash$`G`, respectively. Form extension allows one to add or redefine a set of bindings simultaneously, whereas form restriction can be seen as a dual operation that denotes a form, which is restricted to all bindings of `F` that do not occur in `G`. In combination, these operators provide the main building block in a fundamental concept for defining adaptable, extensible, and more robust software abstractions [10].

Forms can also occur as values in binding extensions, denoted as *nested forms.* As in the case of binding extensions, nested forms are bound by labels. However, rather than using a projection `F.l` to extract the nested form bound by label `l`, we use `F` $\rightarrow$ `l`, called *form dereference.* The reason for this is that we want to explicitly distinguish between components, which are encoded as forms, and plain component services, which are denoted by some values other than forms. If the binding involving label `l` does not actually map a nested form, then the result of `F` $\rightarrow$ `l` is $\langle\rangle$.

$$\begin{array}{rcll}
F, G, H & ::= & \langle\rangle & \textit{empty form} \\
 & | & X & \textit{form variable} \\
 & | & F\langle l = V\rangle & \textit{binding extension} \\
 & | & F \oplus G & \textit{form extension} \\
 & | & F\backslash G & \textit{form restriction} \\
 & | & F \rightarrow\ l & \textit{form dereference} \\
 & | & F[G] & \textit{form context} \\
 & & & \\
V & ::= & \mathcal{E} & \textit{empty value} \\
 & | & a & \textit{abstract value} \\
 & | & M & \lambda\mathcal{F} - \textit{value} \\
 & & & \\
M, N & ::= & F & \textit{form} \\
 & | & M.l & \textit{projection} \\
 & | & \lambda(X)\ M & \textit{abstraction} \\
 & | & M\ N & \textit{application} \\
 & | & M[F] & \lambda\mathcal{F} - \textit{context}
\end{array}$$

Figure 1: Syntax of the $\lambda\mathcal{F}$-Calculus.

A *form context* `F[G]` denotes a closed form expression that is derived from `F` by using `G` as an environment to look up what would otherwise be free variables in `F`. We use form dereference to perform the lookup operation and a free variable is reinterpreted as a label. For example, if `X` is a free variable in `F` and `[G]` is a context, then the meaning of `X` in `F` is determined by the result of evaluating `G` $\rightarrow$ `X`. In the case that `G` does not define a binding for `X`, the result is $\langle\rangle$, which effectively removes the set of bindings associated with `X` from `F`. This allows for an approach in which a sender and a receiver can communicate open form expressions. The receiver of an open form expression can use its local context to close (or *configure*) the received form expression according to a site-specific protocol.

Forms and *projections* replace variables in $\lambda\mathcal{F}$. A form can be viewed as an *explicit namespace*, which can comprise an arbitrary number of bindings. The form itself can contain free variables, which will be resolved in the deployment environment or evaluation context, allowing for a computational model with *late binding.*

Both *abstraction* and *application* correspond to the notions used in the λ-calculus, that is, `X` in λ(`X`) `a` stands for the parameter in an abstraction. But unlike the λ-calculus, we do not use substitution to replace free occurrences of this name in the body of an abstraction – parameter passing is modeled by *explicit contexts.*

A $\lambda\mathcal{F}$-context is the counterpart of a form-context. A *$\lambda\mathcal{F}$-context* denotes a lookup environment for free variables in a $\lambda\mathcal{F}$-term. Moreover, $\lambda\mathcal{F}$-contexts provide a convenient mechanism to retain the bindings of free variables in the body of a function. For example, let λ(`X`) `a` be a function and [F] be a creation context for it. Then we can use [F] to build a *closure* of $\lambda(\mathtt{X})\ \mathtt{a}$. A closure is a package mechanism to record the bindings of free variables of a function at the time it was created. That is, the closure of $\lambda(\mathtt{X})\ \mathtt{a}$ is $\lambda(\mathtt{X})\ (\mathtt{a[F]})$.

Denotational semantics is used to formalize the interpretation of forms and $\lambda\mathcal{F}$-terms. The underlying semantic model of forms is that of *interacting systems* [11]. Informally, the interpretation of forms (i.e., their observable behavior) is defined by an evaluation function $[\![]\!]^F$, which guarantees that feature access is performed from right-to-left [8]. In contrast to standard records, however, a given binding may not be observable in a form and, therefore, may not be used to redefine or hide an existing one. A binding is not observable if it cannot be distinguished from $\mathcal{E}$ or $\langle\rangle$. For example, the forms $\langle\rangle\langle\mathtt{m} = \mathcal{E}\rangle$, $\langle\rangle\langle\mathtt{m} = \langle\rangle\rangle$, and $\langle\rangle$ are all considered to be equivalent. Furthermore, the meaning of a $\lambda\mathcal{F}$-term depends on its *deployment context*. We write $[\![\mathtt{a}]\!]^{LF}[H]$ to evaluate the $\lambda\mathcal{F}$-expression a in a deployment context H. Consider, for example, the following deployment context B that provides a Church encoding of Booleans. This context defines three bindings: True, False, and Not:

$$\begin{aligned}\mathtt{B} = {} & \langle\rangle\langle \mathtt{True} = \lambda(\mathtt{X})\ \mathtt{X.true}\rangle\\ & \langle \mathtt{False} = \lambda(\mathtt{X})\ \mathtt{X.false}\rangle\\ & \langle \mathtt{Not} = \lambda(\mathtt{B})\ \lambda(\mathtt{V})\ \mathtt{B}\ \mathtt{V}\langle \mathtt{true} = \mathtt{V.false}\rangle\langle \mathtt{false} = \mathtt{V.true}\rangle\rangle\end{aligned}$$

Now, assume we want to determine the value denoted by the $\lambda\mathcal{F}$-expression (Not True). We can use B as a lookup environment for the free occurrences of the names Not and True, respectively. Thus, we have to evaluate

$$\begin{aligned}& [\![(\mathtt{Not\ True})]\!]^{LF}[B]\\ = {} & (\lambda(\mathtt{B})\ \lambda(\mathtt{V})\ \mathtt{B}\ \mathtt{V}\langle \mathtt{true} = \mathtt{V.false}\rangle\langle \mathtt{false} = \mathtt{V.true}\rangle)\ \lambda(\mathtt{X})\ \mathtt{X.true}\\ = {} & \lambda(\mathtt{V})\ \mathtt{B}\ \mathtt{V}\langle \mathtt{true} = \mathtt{V.false}\rangle\langle \mathtt{false} = \mathtt{V.true}\rangle\ [\langle\rangle\langle \mathtt{B} = \lambda(\mathtt{X})\ \mathtt{X.true}\rangle]\\ = {} & \lambda(\mathtt{V})\ (\lambda(\mathtt{X})\ \mathtt{X.true})\ \mathtt{V}\langle \mathtt{true} = \mathtt{V.false}\rangle\langle \mathtt{false} = \mathtt{V.true}\rangle\end{aligned}$$

which is a function that is equivalent to False. Due to lack of space, we omit the details of the definition of $[\![]\!]^F$; the interested reader is referred to [8].

3. CLASSBOXES IN A NUTSHELL

In order to address some of the problems of object-oriented programming languages with regard to incrementally changing the behaviour of existing classes, Bergel et al. have proposed the concept of *classboxes* [2, 3]. In their approach, a classbox can be considered as a kind of *module* that defines a controllable context in which incremental changes are visible. Besides the "traditional" operation of *subclassing*, classboxes support the operations of class *import* and class *extension*, respectively. In essence, a classbox exhibits the following main characteristics [3]:

1. It is a unit of scoping where classes (and their associated methods) are defined. A class belongs to the classbox it is first *defined*, but it can be made visible to other classboxes by either *importing* or *extending* it. When a classbox imports a class from another classbox (i.e., the *originating* classbox), the class behaves as if it was directly defined in this classbox. In order to resolve any dependencies, all ancestors of this class are *implicitly* imported also. A class extension can be viewed as an encapsulated import (i.e., self calls are bound early), combined with adding and/or overriding any of the original methods or instance variables.

2. Any extensions to a class are only visible to the classbox in which they are defined and any classboxes that either explicitly or implicitly import the extended class. Hence, overriding a particular method of a class in a given classbox will have no effect in the originating classbox.

3. Although class extensions are only locally visible, their effect extends to all collaborating classes within a given classbox, in particular to any subclasses that are either explicitly imported, extended, or implicitly imported.

In order to illustrate the concept of classboxes, consider the three classboxes *OriginalCB*, *LinearCB*, and *ColorCB*, respectively, given in Figure 2. The class Point defined in *OriginalCB* contains two protected instance variables x and y, a method move which moves a point by a given offset (dx,dy), and a method double that doubles the values of the x and y coordinates. The reader should note that the method move is invoked by double (using a self call). The class BoundedPoint is a direct specialization of Point. It ensures that the y coordinate of an instance never exceeds a given upper bound by. This bound is a constant in BoundedPoint, although this behaviour can be altered (as shown below).

The classbox *LinearCB imports* the class BoundedPoint from *OriginalCB*. As a consequence, Point is also *implicitly* imported, making it visible as the direct superclass of BoundedPoint. In order to define a non-constant bound, the class LinearBoundedPoint specializes BoundedPoint in *LinearCB* by overriding the method bound in an appropriate way (i.e., move checks if the y coordinate is smaller than the x coordinate).

The classbox *ColorCB extends* the class Point from *OriginalCB* by adding a protected instance variable c and a corresponding accessor method getColor. As a consequence, all instances of Point in *ColorCB* as well as the instances of any of its subclasses posses this additional behaviour. Therefore, the class LinearBoundedPoint *imported* from *LinearCB* also possesses the color property.

ColorCB also contains a *new* class BoundedPoint as a specialization of the extended class *Point* (restricting the x coordinate of its instances). Although it has the same name as BoundedPoint defined in *OriginalCB*, the two classes are *not* related. Hence, BoundedPoint defined in *ColorCB* is not the direct superclass of LinearBoundedPoint – the direct superclass of LinearBoundedPoint in *ColorCB* remains BoundedPoint defined in *OriginalCB*. This class is *implicitly* imported by LinearBoundedPoint and *co-exists* with BoundedPoint defined in *ColorCB*.

The semantics of extension operator deserves some further analysis. In [3, p. 118], it is stated that importing a class into a classbox is the same as extending this class with an empty set of methods. Furthermore, to guarantee the *locality of changes*, class extensions are purely local to the classbox within which they occur. Hence, it should be possible to use the operator extend to add a tracing mechanism

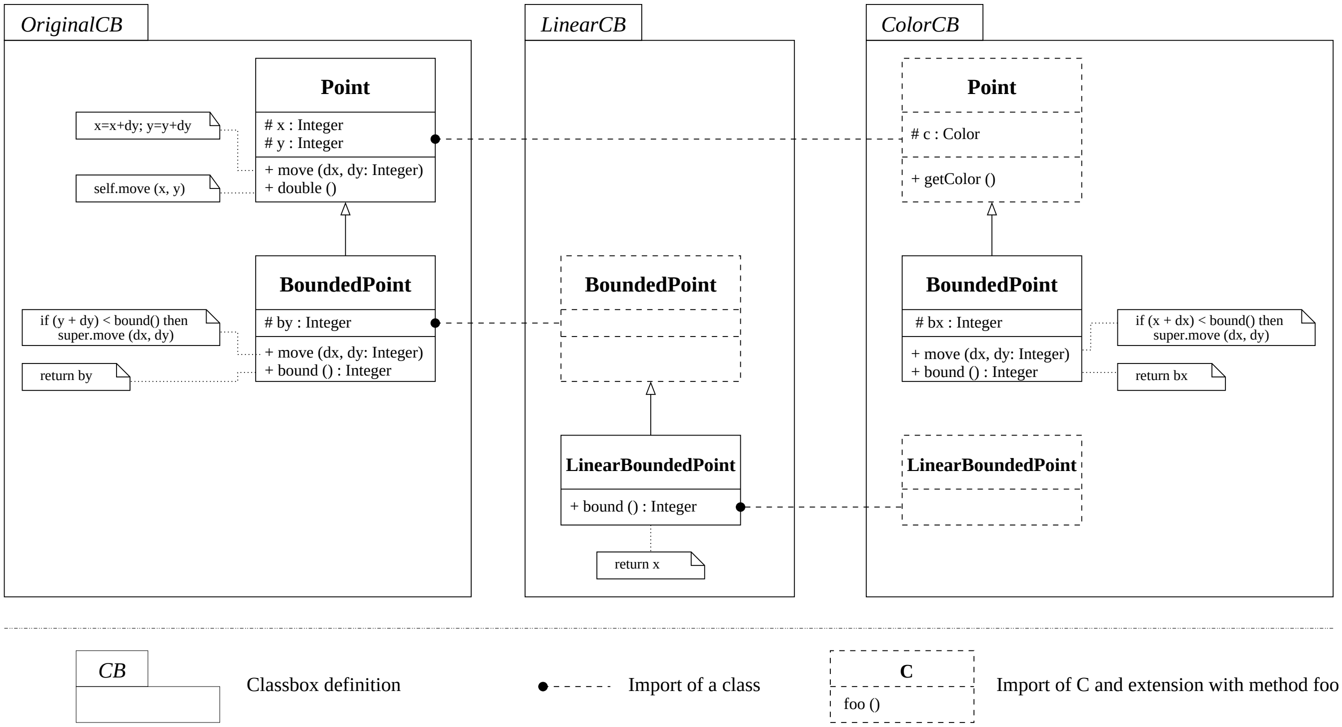

Figure 2: Sample classboxes.

to the class Point that logs all invocations of move in a new classbox, say *TraceCB*.

However, considering the way extend is formally defined in [3], we come to the conclusion that the semantics is slightly different and that extend should behave like C#'s new specifier [12], which can be used to hide any superclass method by declaring a new method with the same signature in a subclass. The effect is similar to binding self calls in superclass methods *early* (i.e., confine self calls occurring within superclass methods to the superclass). As a consequence, any instance derived from *TraceCB*'s Point class will not invoke the most recent definition of method move when required, but rather the original move defined in *OriginalCB*.

Therefore, to clarify the semantics of the classbox operators, we propose a revised notion of extending classes, which incorporates two separate extension operations: (i) *extension* of classes in which self calls are encapsulated to the context in which they occur, and (ii) *inclusion* of new behaviour by means of late binding of self calls. Such a separation will also allow us to seamlessly integrate the concept of *accessing the original method* (i.e., accessing the original implementation of method being redefined) presented in [2].

4. THE MODEL

As we have shown in earlier work [14], object- and component-oriented abstractions can most easily be modeled if classes are represented as first-class entities. This approach can be further generalized using a form-based framework (see Figure 3), which defines a hierarchy of meta-level abstractions to model *meta-classes*, *classes*, and *objects* [10]. The core of this meta-level framework is MetaModel, an abstraction that provides support for the instantiation of an object-oriented programming infrastructure. The underlying semantics of a specific programming infrastructure is captured by so-called *model generators*, *model wrappers*, and *model composers*, denoted by G_m, W_m, and C_m, respectively. The model abstractions G_m, W_m, and C_m define the rules by which a concrete object-oriented programming system (e.g., the Java programming model) is governed. For example, to construct a Java-like programming infrastructure, we need to specify a generator G_m^{Java}, which defines the mechanism required for dynamic method lookup, and the single inheritance abstractions W_m^{Class} and C_m^{Class}. To instantiate the Java-like infrastructure, we apply these three abstractions to MetaModel. The result is an infrastructure meta-object that can be used to create classes that adhere to Java semantics.

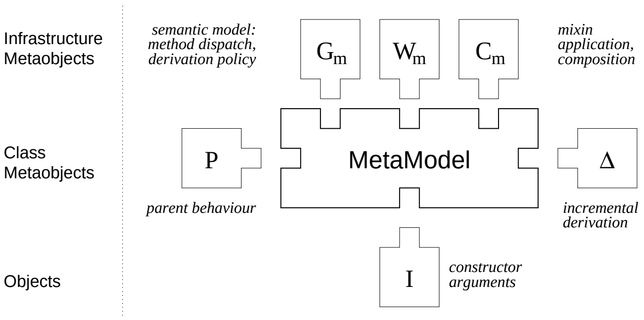
Infrastructure Metaobjects
semantic model: method dispatch, derivation policy
G_m
W_m
C_m
mixin application, composition
Class Metaobjects
P
MetaModel
Δ
parent behaviour
incremental derivation
Objects
I
constructor arguments

Figure 3: A form-based meta-level framework.

The behaviour of a class, on the other hand, is captured by an *incremental modification*, denoted by Δ, and by a (possibly empty) *parent behaviour*, denoted by P, that captures the behaviour of its superclass(es). For example, to create a new class C, one has to define a new class generator G_C[1] that combines C's incremental modification Δ_C with C's parent behaviour P. To instantiate objects of the class C, one has to apply G_C to some suitable *constructor arguments*, denoted by I. The result is a prototype instance that has to be passed to a model-specific wrapper, in order to establish the desired binding of self references within the newly created object.

In order to define classboxes in $\lambda\mathcal{F}$, we shall adapt an approach that is as close as possible to the original definition of classesboxes defined by Bergel et al. [3] with the exception that we shall define two separate extension operators. Furthermore, the reader should note that our classbox model does not require the composer abstraction which is mainly used for mixin application and composition [4, 19]. Hence, we shall only use incremental modifications, generators, and wrappers, respectively, to define a $\lambda\mathcal{F}$ representation of classes and classboxes.

We use the Greek letters α, β, and γ to denote classboxes, and A, B, C to range over class names. A class C in classbox α is represented by a named form $C_\alpha = \langle G, W\rangle$, where

- C_α is a so-called *decorated class name* for class C [3] in which α identifies the originating classbox;
- G is the generator for class C combining C's incremental modification Δ_C with C's parent behaviour P; and
- W is a wrapper yielding instances of class C when applied to suitable constructor arguments.

Classboxes are actually *open class namespaces*. As a consequence, G and W are also open with respect to the environment being used to invoke them. Therefore, both G and W are parameterized over an *activation classbox*. An activation classbox is the fixed-point of classbox in which the corresponding class $C_\alpha = \langle G, W\rangle$ occurs explicitly by means of either import, subclassing, or extension. Thus, passing the activation classbox to G and W, respectively, closes both abstractions and provides them with an appropriate lookup environment. This technique enables a late binding of G, which is the key mechanism for extending classes. The general structure of a class definition follows a format as shown below:

$$
\begin{array}{l}
C_\alpha = \\
\quad \mathbf{let} \\
\qquad \Delta_C \;=\; \lambda(\mathsf{State})\ (\ \boxed{\ Methods_C\ }\ )\ [\mathsf{State}] \\[1em]
\qquad G_C \;=\; \lambda(\gamma)\ \lambda(I\,)\ P_\beta \oplus \Delta_C\langle\ I \oplus (\ \boxed{\ State_C\ }\ )\ \rangle \\[1em]
\qquad W_C \;=\; \lambda(\gamma)\ \mu_{\mathsf{self}}\langle((\gamma \rightarrow C_\alpha).G\ (\beta \oplus \gamma))\ [\mathsf{self}]\rangle \\
\quad \mathbf{in} \\
\qquad \langle G = G_C, W = W_C\rangle
\end{array}
$$

A class is characterized by the three abstractions Δ_C, G_C, and W_C, respectively, all defined within the scope of C_α. We use the syntactic form "$\mathbf{let}\ v_1 = M_1\ \ldots\ v_n = M_n\ \mathbf{in}\ N$" to define a $\lambda\mathcal{F}$-context containing the required private definitions of Δ_C, G_C, and W_C, respectively, to capture the behaviour of C_α.

The incremental modification Δ_C captures the behaviour defined by class C. In order to represent C's behaviour, we use an approach based on the way *traits* are defined in the language SELF [18]. However, to maintain a strict encapsulation of state, we do not blend state and methods. Instead, we model state as an explicit context, written $[\mathsf{State}]$, that provides an environment for each method to resolve the occurrences of private instance variables and the self reference.

The generator G_C builds a prototype instance of class C. G_C takes an activation classbox to provide a correct lookup environment to the parent behaviour P_β originating from classbox β. Upon receiving the constructor arguments, denoted by I, the generator G_C extends P_β with the result of applying C's incremental modification Δ_C to the combination of the constructor arguments and C's state template.

The wrapper W_C yields an object of class C by building the fixed-point (denoted by μ_{self}) of the prototype instance being created within W_C. In order to create a prototype instance for class C, the wrapper uses the activation classbox γ to a look up $C's$ generator. The expression $(\gamma \rightarrow C_\alpha).G$ denotes the fact that we look up the most recent definition of the named form $C_\alpha = \langle G, W\rangle$ in classbox γ to invoke C's generator. We apply this generator to $(\beta \oplus \gamma)$ that combines the classbox β containing C's parent behaviour P_β with the activation classbox γ. The resulting classbox will contain not only the most recent definitions that class C depends upon, but also the ones that have been implicitly imported. More specifically, $(\beta \oplus \gamma)$ denotes a lookup environment that is a transitive closure of C's dependency graph.

A classbox is also represented as a form. The general structure of a classbox follows a format as shown in the example below:

$$
\begin{array}{l}
ColorCB = \\
\quad \mathbf{let} \\
\qquad \mathsf{Point} = \langle G_{\mathsf{Point}}, W_{\mathsf{Point}}\rangle \\
\qquad \mathsf{BoundedPoint} = \langle G_{\mathsf{BoundedPoint}}, W_{\mathsf{BoundedPoint}}\rangle \\
\qquad \mathsf{LinearBoundedPoint} = \\
\qquad\qquad\qquad \langle G_{\mathsf{LinearBoundedPoint}}, W_{\mathsf{LinearBoundedPoint}}\rangle \\
\quad \mathbf{in} \\
\qquad \langle \mathsf{Point}_{\mathsf{OriginalCB}} = \mathsf{Point}, \\
\qquad\ \mathsf{BoundedPoint}_{\mathsf{ColorCB}} = \mathsf{BoundedPoint}, \\
\qquad\ \mathsf{LinearBoundedPoint}_{\mathsf{LinearCB}} = \mathsf{LinearBoundedPoint}\rangle
\end{array}
$$

A classbox is a form that contains mappings from decorated class names to class definitions. Each decorated class name uniquely identifies the originating classbox of a class, that is, the classbox in which a class was first defined. For example, the classbox *ColorCB*, as shown in Figure 2, contains three classes Point, BoundedPoint, and LinearBoundedPoint, respectively. However, each class has a different originating classbox. Only class BoundedPoint originates in *ColorCB*. The classes Point and LinearBoundedPoint originate in *Orig-*

[1] The reader should note that a class generator G_C defines the protocol between Δ_C and P, whereas a model generator G_m defines the protocol between classes.

inalCB and *LinearCB*, respectively. In fact, both Point and LinearBoundedPoint occur as *extended classes* in *ColorCB*.

Decorated class names are the key ingredient in an approach that provides support for class extensions [3]. However, as labels are not first-class values in the $\lambda\mathcal{F}$-calculus, we cannot directly express decorated class names. But, $\lambda\mathcal{F}$ provides another abstraction that can be used instead: it is possible to denote operations involving decorated class names by so-called *abstract applications*. An abstract application $(\mathsf{a}\ \mathsf{M})$ is an $\lambda\mathcal{F}$-expression in which the function a is abstract, that is, a is defined *outside* $\lambda\mathcal{F}$. The intuition here is that a $\lambda\mathcal{F}$-term (i.e., $(\mathsf{a}\ \mathsf{M})$) has to be embedded into a concrete target system that provides an interpretation of the abstract function. When applied to some argument, an abstract function has to yield a value that is again in $\lambda\mathcal{F}$.

To handle decorated class names in our $\lambda\mathcal{F}$-based model for classboxes, we need four abstract functions:

$$\mathsf{buildDecoratedName}\langle \mathsf{C}, \alpha\rangle = C_\alpha$$

$$\mathsf{lookupDecoratedName}\langle \mathsf{C}, \alpha\rangle = \begin{cases} C_\beta, \text{ if } \exists!\beta, (\alpha \rightarrow C_\beta) \neq \langle\rangle \\ \perp, \text{ otherwise} \end{cases}$$

$$\mathsf{lookupClass}\langle \mathsf{C}, \alpha\rangle = \alpha \rightarrow \mathsf{lookupDecoratedName}\langle \mathsf{C}, \alpha\rangle$$

$$\begin{aligned}&\mathsf{buildClass}\langle \mathsf{C} = \langle G, W\rangle, \alpha\rangle = \\ &\qquad \langle \mathsf{lookupDecoratedName}\langle \mathsf{C}, \alpha\rangle = \langle G, W\rangle\rangle\end{aligned}$$

The function buildDecoratedName takes a class name C and a classbox name α and returns a decorated class name C_α that is a valid $\lambda\mathcal{F}$-label. The function lookupDecoratedName takes a class name C and a classbox name α and returns a $\lambda\mathcal{F}$-label that denotes a valid decorated class name C_β, if such a name exists in the classbox α. The function lookupClass takes a class name C and a classbox name α and returns a form that represents class C, as defined in classbox α. Finally, the function buildClass takes a class $\mathsf{C} = \langle G, W\rangle$ and a classbox name α and yields a binding in which the label denotes a valid decorated class name for C in α.

5. MODELING CLASSBOX OPERATIONS

In this section, we present our $\lambda\mathcal{F}$ encodings of classbox operations. More precisely, we show the encoding of *import* of classes, introduction of *subclasses*, *extension* of classes, and *inclusion* of new behaviour. The latter two operations are deduced from the original extend operator [3] by the refining process outlined in Section 3.

5.1 Import of classes

The import of a class C from classbox β into classbox α is defined as shown below.

$$\begin{array}{l} C_\alpha = \\ \quad \textbf{let} \\ \qquad W_C = \lambda(\gamma)\ (\mathsf{lookupClass}\langle \mathsf{C}, \beta\rangle).W\ (\beta \oplus \gamma) \\ \quad \textbf{in} \\ \qquad (\mathsf{lookupClass}\langle \mathsf{C}, \beta\rangle)\langle W = W_C\rangle \end{array}$$

To import class C, we acquire its definition from classbox β using the expression $(\mathsf{lookupClass}\langle \mathsf{C}, \beta\rangle)$. However, the class C may depend on some behaviour for which an extended definition is given in classbox α (or the activation classbox γ containing the extensions specified by classbox α). Therefore, an imported class requires a new wrapper that combines the definitions of a class' originating classbox and the actual activation classbox. The result (i.e., $(\beta\oplus\gamma)$) is passed to the class' original wrapper that will use it to incorporate pertinent definitions into the class' behaviour. Finally, if not stated otherwise, the decorated class name C_α in this and all following encodings is the result of applying lookupDecoratedName to the class name C and the originating classbox name β (i.e., $C_\alpha = \mathsf{lookupDecoratedName}\langle \mathsf{C}, \beta\rangle$).

5.2 Subclassing

We can define a class C as a subclass of class B originating from classbox β using the following specification:

$$\begin{array}{l} C_\alpha = \\ \quad \textbf{let} \\ \qquad \Delta_C \;=\; \lambda(\mathsf{State})\ (\ \boxed{\mathit{Methods}_C}\ )\ [\mathsf{State}] \\[1ex] \qquad G_C \;=\; \lambda(\gamma)\ \lambda(I) \\ \qquad\qquad\qquad \textbf{let} \\ \qquad\qquad\qquad\quad \mathsf{P} = ((\mathsf{lookupClass}\langle \mathsf{B}, \gamma\rangle).G\ \gamma)\ I \\ \qquad\qquad\qquad \textbf{in} \\ \qquad\qquad\qquad\quad \mathsf{P} \oplus \Delta_C\langle I\ \oplus \left(\begin{array}{|c|}\hline \mathit{State}_C \\ \hline \langle \mathsf{super} = \mathsf{P}\rangle \\ \hline \end{array}\right)\rangle \\[1ex] \qquad W_C \;=\; \lambda(\gamma)\ \mu_{\mathsf{self}}\langle(((\mathsf{lookupClass}\langle \mathsf{C}, \gamma\rangle).G\ (\beta \oplus \gamma))\ [\mathsf{self}]\rangle \\ \quad \textbf{in} \\ \qquad \langle G = G_C, W = W_C\rangle \end{array}$$

To construct class C, we acquire its superclass behaviour P using the expression $((\mathsf{lookupClass}\langle \mathsf{B}, \gamma\rangle).G\ \gamma)\ I$ and combine it with C's incremental modification Δ_C. To acquire C's superclass behaviour, we dynamically look up B's generator with respect to the activation classbox γ, which guarantees that any relevant extensions to B's behaviour are also incorporated in class C. Methods in Δ_C may override methods in the superclass B. Overridden methods are accessible by means of the additional binding $\langle \mathsf{super} = \mathsf{P}\rangle$ passed to Δ_C.

Since class B may also depend on some behaviour visible only to classbox β, we need to provide a reference of β to C's wrapper W_C. This approach not only guarantees that references to class B can be resolved, but also that references to any superclasses of B can be resolved in classbox α without adding them to the visible scope of classbox α. For example, the class LinearBoundedPoint in classbox *LinearCB*, as shown in Figure 2, implicitly depends on class Point defined in classbox *OriginalCB*. This dependency is resolved by providing the originating classbox of BoundedPoint to the wrapper of LinearBoundedPoint.

Finally, the classbox model as defined by Bergel et al. [3] requires that when defining a subclass, its class name must occur *fresh* in the defining classbox. Therefore, the decorated class name C_α is not derived from classbox β, but constructed with respect to the defining classbox α (i.e., $C_\alpha = \mathsf{buildDecoratedName}\langle \mathsf{C}, \alpha\rangle$).

5.3 Extending imported classes

To specify the semantics of the *refined* extend operation, we need to define an information hiding protocol that, when applied to a concrete class, renders the features of the extensions invisible to the class' behaviour. Hence, the extend operation yields a membrane for a class that permits super calls originating from extensions, but prevents the class' behaviour to see the extensions. This protocol is established by confining self calls to the context within which they occur (i.e., the original class or the extensions). The extension of class C with some behaviour B can be defined as follows:

```
$B^E_\beta =$
  let
    $\Delta_B = \lambda(\mathsf{State})\ (\ \boxed{\ \mathit{Methods}_B\ }\ )\ [\mathsf{State}]$

    $G_B = \lambda(\mathsf{Class})$
             $\lambda(\gamma)\ \lambda(I\,)$
               let
                 $\mathsf{P} = \mu_{\mathsf{self}}\langle\ (\mathsf{Class}.G\ \gamma)\ [\mathsf{self}]\ \rangle\ I$
               in
                 $\mathsf{P} \oplus \Delta_B\langle\ I \oplus \left(\boxed{\begin{array}{c} \mathit{State}_B \\ \hline \langle \mathsf{super} = \mathsf{P} \rangle \end{array}}\right)\rangle$

  in
    $\langle G = G_C \rangle$
```

$$C_\alpha = (\mathsf{lookupClass}\langle \mathsf{C}, \alpha' \rangle)\langle G = B^E_\beta.G\ (\mathsf{lookupClass}\langle \mathsf{C}, \alpha' \rangle)\rangle$$

The abstraction B^E_β captures the behaviour of the extension B being used to modify class C. The extend operator requires that we *encapsulate* C's behaviour in order to protect it from any changes defined by B. That is, we have to combine the fixed-point of C's prototype instance with the incremental modification Δ_B defined by extension B.

The structure of B^E_β is similar to the one required to define subclassing. However, an extension cannot be instantiated independently. Therefore, no wrapper is needed. When combined with a concrete class, the class' wrapper is responsible for providing a suitable environment to create objects of that class. Moreover, the definition of the revised extend operator guarantees that the extensions are local to the classbox in which they occur and that they do not affect the class' original behaviour, as it is shielded from the extensions by binding self calls in the class' methods early.

The purely functional $\lambda\mathcal{F}$-based encoding of the extend operator is, however, a source for a serious problem. Functional update of state yields a new object. The new object is created by passing the new state values to the wrapper of the object's class. However, the wrapper of an extended class (i.e., $\mu_{\mathsf{self}}\langle(\mathsf{Class}.G\ \gamma)[\mathsf{self}]\rangle$) does not include the extensions. Therefore, functional update yields an instance of the original class, not one of the extended class.

5.4 Include behaviour into imported classes

Inclusion is a new operator that enables *down calls* to class extensions. The inclusion operator is like the extension operator, excepted that we do not encapsulate the class' behaviour. This approach roughly corresponds to the concept of *mixin application* [19]. That is, if we apply an extension B to class C, then the class C stands for an *abstract subclass*, which is instantiated with the superclass B. The inclusion of extension B into class C is defined as follows:

```
$B^I_\beta =$
  let
    $\Delta_B = \lambda(\mathsf{State})\ (\ \boxed{\ \mathit{Methods}_B\ }\ )\ [\mathsf{State}]$

    $G_B = \lambda(\mathsf{Class})$
             $\lambda(\gamma)\ \lambda(I\,)$
               let
                 $\mathsf{P} = (\mathsf{Class}.G\ \gamma)\ I$
               in
                 $\mathsf{P} \oplus \Delta_B\langle\ I \oplus \left(\boxed{\begin{array}{c} \mathit{State}_B \\ \hline \langle \mathsf{original} = \mathsf{P} \rangle \end{array}}\right)\rangle$

  in
    $\langle G = G_B \rangle$

$C_\alpha =$
  let
    $G_C = \lambda(\gamma)\ (B^I_\beta.G\ \mathsf{lookupClass}\langle \mathsf{C}, \alpha' \rangle)\ \gamma$
    $W_C = \lambda(\gamma)\ (\mathsf{lookupClass}\langle \mathsf{C}, \alpha' \rangle).W\ (\beta \oplus \gamma)$
  in
    $\langle G = G_C, W = W_C \rangle$
```

Inclusion extension is an operation that combines extend and import. However, unlike extension, we do not build the fixed-point of the parent behaviour P in B^I_β to enable down calls to the extensions. In addition, methods in Δ_B may override methods in class C. The overridden methods of C are, however, accessible by means of the additional binding $\langle \mathsf{original} = \mathsf{P} \rangle$ passed to Δ_B. This approach was recently proposed by Bergel et al. [2] in their Classbox/J model. Finally, the functions G_C and W_C define the protocol required to properly link class C and the inclusion extension B.

6. CONCLUSIONS, FUTURE WORK

In this paper, we have used the $\lambda\mathcal{F}$-calculus to define a model for *classboxes*, a dynamically typed module system for object-oriented languages that provides support for controlling both the visibility and composition of class extensions, and validated our model using a prototype implementation of the $\lambda\mathcal{F}$-calculus.

This work has shown that $\lambda\mathcal{F}$ is a powerful tool to model compositional abstractions such as classes, classboxes as well as their associated operations. Replacing λ-calculus names by first-class namespaces and parameter passing by explicit contexts, we argue, are the key concepts in obtaining the resulting flexibility and extensibility. Both asymmetric form extension as well as the late binding of free variables in form expressions due to explicit form contexts are essential features to express the model in such an elegant way. It has also shown that the meta-level framework we defined in previous work [10] where object-oriented abstractions were modeled as compositions of appropriately parameterized generator, wrapper, and composer abstractions offers enough flexibility to incorporate classboxes.

As we have discussed in Section 3, the formal definition of class extension presented in [3] does not fully match its informal description. As a consequence, extend has limited applicability when *hook methods* [6] are to be extended by independent behavioural properties such as, for example, extending the class Point with a tracing mechanism on the method move. Therefore, we have proposed a revised notion of class extension, incorporating two separate extension operations: (i) *extension* of classes in which self calls are encapsulated to the context in which they occur, and (ii) *inclusion* of new behaviour by means of late binding of self calls. Such a separation has allowed us to clarify the semantics of the classbox operators and seamlessly integrate the concept of *accessing the original method* as defined in the Classbox/J model [2]. In this context we have also illustrated that the early binding of self calls is the source of a serious problem when object-oriented abstractions are modeled in a purely functional setting such as $\lambda\mathcal{F}$.

However, this work has also shown that not all abstractions needed to define a model for classboxes can be expressed within $\lambda\mathcal{F}$, that is, we were forced to use *abstract applications* to model the decoration of class names. It is however not yet fully understood whether this is a limitation of $\lambda\mathcal{F}$ or a result of how classboxes have been formalized both in [3] and in our model. Therefore, we will investigate whether the expressiveness of classboxes can also be achieved without using explicitly decorated classnames, allowing us to model classboxes entirely in $\lambda\mathcal{F}$.

Furthermore, the concept of classboxes does not allow for an *explicit* co-existence of both the original of a class as well as an extension thereof. For example, it could become necessary for the class Point defined in *OriginalCB* and its extension with the color property to co-exist within the classbox *ColorCB*. Currently, both classes may *implicitly* co-exist within *ColorCB* and it is possible to create instances of the extended version of Point, but not of Point defined in *OriginalCB*. It is, however, not yet clear how to extend classboxes with this feature and whether this extension is of real practical value.

Acknowledgements

We would like to the members of the Centre for Component Software and Enterprise Systems at Swinburne for inspiring discussions on these topics as well as Alexandre Bergel and the anonymous reviewers for commenting on earlier drafts.

7. REFERENCES

[1] U. Aßmann. *Invasive Software Composition.* Springer, 2003.

[2] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. In *Proceedings OOPSLA '05*, San Diego, USA, Oct. 2005. ACM Press. To appear.

[3] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling Visibility of Class Extensions. *Journal of Computer Languages, Systems & Structures*, 31(3–4):107–126, May 2005.

[4] G. Bracha and W. Cook. Mixin-based Inheritance. In N. Meyrowitz, editor, *Proceedings OOPSLA/ECOOP '90*, volume 25 of *ACM SIGPLAN Notices*, pages 303–311, Oct. 1990.

[5] L. Dami. A Lambda-Calculus for Dynamic Binding. *Theoretical Computer Science*, 192:201–231, Feb. 1998.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[7] M. Lumpe. *A π-Calculus Based Approach for Software Composition.* PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, Jan. 1999.

[8] M. Lumpe. A Lambda Calculus With Forms. In T. Gschwind, U. Aßmann, and O. Nierstrasz, editors, *Proceedings of the Fourth International Workshop on Software Composition*, LNCS 3628, pages 83–98, Edinburgh, Scotland, Apr. 2005. Springer.

[9] M. Lumpe and J.-G. Schneider. Form-based Software Composition. In M. Barnett, S. Edwards, D. Giannakopoulou, and G. T. Leavens, editors, *Proceedings of ESEC '03 Workshop on Specification and Verification of Component-Based Systems (SAVCBS '03)*, pages 58–65, Helsinki, Finland, Sept. 2003.

[10] M. Lumpe and J.-G. Schneider. A Form-based Metamodel for Software Composition. *Science of Computer Programming*, 56:59–78, Apr. 2005.

[11] R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[12] H. Mössenböck. *C# to the Point.* Addison-Wesley, 2005.

[13] O. Nierstrasz and F. Achermann. A Calculus for Modeling Software Components. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Proceedings of First International Symposium on Formal Methods for Components and Objects (FMCO 2002)*, LNCS 2852, pages 339–360, Leiden, The Netherlands, 2003. Springer.

[14] O. Nierstrasz, J.-G. Schneider, and M. Lumpe. Formalizing Composable Software Systems – A Research Agenda. In *Proceedings the 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems*, pages 271–282. Chapmann & Hall, 1996.

[15] R. S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, sixth edition, 2005.

[16] J.-G. Schneider and M. Lumpe. Synchronizing Concurrent Objects in the Pi-Calculus. In R. Ducournau and S. Garlatti, editors, *Proceedings of Langages et Modèles à Objets '97*, pages 61–76, Roscoff, Oct. 1997. Hermes.

[17] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley / ACM Press, Second edition, 2002.

[18] D. Ungar and R. B. Smith. SELF: The Power of Simplicity. In *Proceedings OOPSLA '87*, volume 22 of *ACM SIGPLAN Notices*, pages 227–242, Dec. 1987.

[19] M. Van Limberghen and T. Mens. Encapsulation and Composition as Orthogonal Operators on Mixins: A Solution to Multiple Inheritance Problems. *Object-Oriented Systems*, 3(1):1–30, Mar. 1996.