



A form-based meta-model for software composition

M. Lumpe^{a,*}, J.-G. Schneider^b

^a*Department of Computer Science, Iowa State University, 113 Atanasoff Hall, Ames, IA 50011, USA*

^b*Faculty of Information and Communication Technologies, Swinburne University of Technology, P.O. Box 218, Hawthorn, VIC 3122, Australia*

Received 9 November 2003; received in revised form 16 August 2004; accepted 6 September 2004
Available online 13 December 2004

Abstract

In recent years considerable progress has been made in facilitating the specification and implementation of software components. However, it is far less clear what kind of language support is needed to enable a flexible and reliable software composition approach. Object-oriented programming languages seem to already offer some reasonable support for component-based programming (e.g., encapsulation of state and behavior, inheritance, late binding). Unfortunately, these languages typically provide only a fixed and restricted set of mechanisms for constructing and composing compositional abstractions.

In this article, we will present a generic meta-level framework for modeling both object- and component-oriented programming abstractions. In this framework, various features, which are typically merged in traditional object-oriented programming languages, are all replaced by a single concept: the composition of *forms*. Forms are first-class, immutable, extensible records that allow for the specification of compositional abstractions in a language-neutral and robust way. Thus, using the meta-level framework, we can define a compositional model that provides the means both to bridge between different object models and to incorporate existing software artifacts into a unified composition system.

© 2004 Elsevier B.V. All rights reserved.

Keywords: Object models; Meta-level framework; Compositional abstractions

* Corresponding author.

E-mail addresses: lumpe@cs.iastate.edu (M. Lumpe), jschneider@swin.edu.au (J.-G. Schneider).

1. Introduction

Component-oriented software technology has become the major approach to facilitating the development and evolution of modern software systems [6,35,39,43]. The objective of this technology is to take elements from a collection of reusable software components, that is, take *components off-the-shelf*, and build applications by simply plugging them together. Thus, the component-based software development approach enables the practical reuse of valuable software assets and amortization of investments over multiple applications [43]. By reconfiguring, adapting existing, or introducing new components we can bring applications in line with changing requirements more easily than is possible using “traditional” development approaches [35].

A successful component-based development approach, however, not only needs to provide abstractions to represent different component models and composition techniques, but also has to provide a systematic method for constructing large software systems. This is because software composition takes place at two different levels: a system level and a language level [6]. At the system level we have components, composition rules and techniques, and a special-purpose language for describing composites. The language level should be structured similarly. That is, it should be defined in a component-oriented way. This allows not only for a seamless integration of different component models and composition techniques, but also for extending the language level with new compositional abstractions on demand.

Object-oriented programming languages seem to already offer some reasonable support for component-based programming (e.g., encapsulation of state and behavior, inheritance, late binding). Nevertheless, these languages are not powerful enough to provide flexible and type-safe component composition and evolution mechanisms [47]. We can identify especially a lack of abstractions for building and adapting class-like components in a framework or domain specific way, a lack of abstractions for defining cooperation patterns, and a lack of support for checking the correctness of compositions.

Although all of these problems are of importance for component-based programming, addressing all of them is beyond the scope of this work. In this article, we will therefore focus on the first issue, that is, the specification of class-like components in a framework or domain specific way. In particular, we will present a generic meta-level framework for modeling both object- and component-oriented programming abstractions. This meta-level framework provides the means to define composition specifications (or *composition recipes* [6]) that incorporate abstractions both to bridge between different object models and to incorporate existing software artifacts into one unified composition system.

Several different models have been proposed for defining the semantics of object- and component-oriented programming abstractions. However, even though most of these models define objects and/or object-oriented abstractions as primitives, they either use a hard-wired inheritance model [37], integrate concepts in a non-orthogonal way [7,19], or do not incorporate important features found in object-oriented programming languages (e.g., they lack inheritance [1]).

In our meta-level framework, various features which are typically merged in traditional object-oriented programming languages are replaced by a single concept: the composition of *forms*. Forms [24,25] are first-class, immutable, extensible records that define

variable-free mappings from labels to values, which, in combination with a small set of purely asymmetric operators, provide a core language for defining compositional abstractions. We argue that forms are the key concept for extensibility, flexibility, and robustness in component-based application development and enable the definition of a canonical set of compositional abstractions in a uniform framework.

Forms address the inherent problem of limited reusability and extensibility of position-dependent parameters in component interfaces and compositional abstractions [16,17,24]. For example, in the standard λ -calculus, the functions $\lambda(x, y).x$ and $\lambda(y, x).y$ are equivalent, but $\lambda(x, y).x$ and $\lambda(y, x).x$ are different, as position matters in the λ -calculus. Moreover, even though the functions $\lambda(x, y).x$ and $\lambda(x, y, z).x$ both yield x , they are different, because a function in the λ -calculus is characterized not by the parameters that it effectively uses, but by the parameters that it declares. Thus, the need to use position-dependent parameters results in a limited reusability and extensibility of the defined abstractions.

Whereas the idea of having key/value pairs is nothing new (cf. XML/HTML forms [45], Python [27], Common Lisp [42], Perl [46]), the specific added value of the expressive power of forms can be found in two composition operators: *polymorphic extension* and *polymorphic restriction*. If applied to two forms F and G , polymorphic extension allows one to combine the features of both F and G simultaneously, but giving G 's features precedence in the result. On the other hand, the polymorphic restriction can be used to define a form that is restricted to all bindings of F that do not occur in G . Experience has shown that the two operators are the main building blocks in a fundamental concept for defining adaptable, extensible, and more robust software abstractions [24,25,40]. We argue that a combination of these two operators in a single concept of form composition is quite unique and gives our approach the expressive power needed to define compositional abstractions in a language-neutral and robust way [25].

The rest of this article is organized as follows. In Section 2, we present the semantic model of forms, illustrate their syntax, and sketch out their interpretation. In Section 3, we show how forms can be used to model object-oriented abstractions and identify the key concepts for extension and generalization of this model in Section 4. Motivated by the results of the previous sections, we define a generic meta-level framework for modeling both object- and component-oriented programming abstractions in Section 5. We conclude this article with a discussion of related work in Section 6 and a summary of the main observations and outline of future work in Section 7.

2. Forms

Forms are first-class, immutable, extensible records that define variable-free mappings from an infinite set of labels, denoted by \mathcal{L} , to an infinite set of abstract values, denoted by \mathcal{V} . Programming values such as *Strings* and *Integers* or even *Objects*, *Classes*, and *forms* themselves are elements of \mathcal{V} . We do not require any particular properties for \mathcal{L} and \mathcal{V} except that equality and inequality are defined for both.

The set \mathcal{V} of abstract values contains a distinguished element \mathcal{E} – the empty value. In the context of software composition, the empty value denotes the lack of a *component*

$F, G, H ::= \langle \rangle$ <i>empty form</i> $F\langle l=V \rangle$ <i>abstract binding extension</i> $F \cdot G$ <i>polymorphic extension</i> $F \setminus G$ <i>polymorphic restriction</i> $F \rightarrow l$ <i>form dereference</i>	$V ::= S$ <i>abstract scalar value</i> F <i>nested form</i> $S ::= \mathcal{E}$ <i>empty value</i> a <i>abstract value</i> $F.l$ <i>abstract projection</i>
--	---

Fig. 1. The syntax of forms.

service. That is, if a given label, say l , is bound to the empty value, then the corresponding component service is either currently unavailable or not defined at all.

The set \mathcal{F} of forms is the smallest set that satisfies the specification given in Fig. 1. We use F, G, H to range over the set \mathcal{F} of forms, l, m, n to range over the set \mathcal{L} of labels, and a, b, c to range over the set \mathcal{V} of abstract values.

Every form is derived from $\langle \rangle$, the *empty form*, denoting a component (or component interface) that does not define any services. To extend a given form F , two distinct constructs exist: *abstract binding extension* and *polymorphic extension*. The abstract binding extension, $F\langle l=V \rangle$, represents a form that defines at least one binding $\langle l=V \rangle$. The effect of extending F with $\langle l=V \rangle$ is that either a fresh service, named l , is added or an existing service in F becomes redefined. On the other hand, *polymorphic extension* can be used to add or redefine a set of services. Please note that the polymorphic extension $F \cdot G$ not only guarantees that the services of both F and G are properly composed, but also will actually combine arbitrary sets of services. No service that occurs either in F or G may eventually be discarded. The effect of polymorphic extension is similar to asymmetric record concatenation [13]. Thus, if two forms F and G both define a binding for the label l , then only G 's binding may be accessible. The reader should note, however, that in contrast to records, some labels in a form may not be observable. Therefore, it depends on the actual bound value whether F 's or G 's binding for a given label, say l , is accessible in $F \cdot G$.

The *polymorphic restriction*, written as $F \setminus G$, allows for information hiding and yields a form that is restricted to the bindings of F , which do not occur in G . Therefore, it can be considered as a dual operation to polymorphic extension. The fundamental purpose of the polymorphic restriction operator is the definition of *feature encapsulation* [40].

Finally, to facilitate the specification of structured component interfaces, forms can also contain *nested forms*. Like values, nested forms are bound by labels. However, a projection of a nested form yields the special value \mathcal{E} , which denotes the lack of a component service. The reason for this is that we want to distinguish between components and component services. To extract a nested form bound by a label l in a form F , we use *form dereference*, denoted by the expression $F \rightarrow l$. If the binding involving label l does not actually denote a nested form, then the actual value of $F \rightarrow l$ is $\langle \rangle$ —the *empty form*.

In form expressions, an abstract binding extension has precedence over a polymorphic extension, a polymorphic extension has precedence over a polymorphic restriction, which in turn has precedence over form dereference. A sequence of two or more polymorphic

extensions is left associative, that is, $F_1 \cdot F_2 \cdot F_3$ is equivalent to $(F_1 \cdot F_2) \cdot F_3$. The same applies to polymorphic restriction. Parentheses may be used in form expressions in order to enhance readability or to overcome the default precedence rules.

The underlying semantic model of forms is that of interacting systems [32]. Forms and in particular the equivalence of forms are characterized in terms of their *observable behavior*. Thus, whether or not a concrete key-value pair is observable depends solely on the value. Moreover, the main operations on forms are purely asymmetric and, as a result, forms lack a canonical normal form.

Informally, the interpretation of forms (that is, their observable behavior) is defined by an evaluation function $\llbracket \cdot \rrbracket^F$, which guarantees that feature access is performed from right to left. The specific difference in the interpretation of forms, with respect to classical records, has its roots in the way *projections* and *form dereferences* are evaluated within the function $\llbracket \cdot \rrbracket^F$. In contrast to standard records, a given label may not be observable in a form and, therefore, may not be used to redefine or hide an existing binding. A label (that is, a binding) is observable if its value is neither \mathcal{E} nor $\langle \rangle$. More precisely, given a form F and some label l , then the evaluations of $F.l$ and $F \rightarrow l$ must not yield \mathcal{E} and $\langle \rangle$, respectively. The reader should note that this characterization of a label's observability is required in order to define a form equivalence relation that is preserved by all form operations.

To illustrate the effects of form evaluation, consider the following example. Suppose we want to give a designated service of F bound by label m precedence over a service bound by the same label m in G . This operation represents a *compositional style* [2] that defines a *conditional update*. It can be defined using the following expression: $F \cdot (G \setminus \langle \rangle \langle m = F.m \rangle)$. Depending on the actual features defined by F and G , we can distinguish three different meanings that $\llbracket \cdot \rrbracket^F$ can assign to the expression $F \cdot (G \setminus \langle \rangle \langle m = F.m \rangle)$:

- If the label m does not occur either in F or G , then the label m does not occur in the composition of F and G .
- If the label m does not occur in F , but in G , then G 's binding for label m occurs in the composition of F and G .
- If the label m occurs in F , then F 's binding for label m occurs in the composition of F and G .

To facilitate the presentation of our meta-level framework, we will use the following notation that can be thought of as syntactic sugar on top of forms. First, the expression $\{l_1 = v_1, \dots, l_n = v_n\}$ denotes a form with the labels l_1, \dots, l_n binding the values v_1, \dots, v_n . We use $m = \lambda(X) : b$ to denote a method m with the method body b and the (keyword-based) formal argument X . If the meaning of an expression $F.m$ is a method, then we write $F.m(G)$ to denote a method call where the actual arguments (including a self-reference) are encoded in the form expression G . Furthermore, we use ‘;’ to specify sequences of operations. For example, $C_1; C_2$ denotes a sequence where C_1 is evaluated prior to C_2 . Additionally, we employ the syntactic form “**let** $v \leftarrow e_1$ **in** e_2 ” to define a private binding v with the initial value e_1 in expression e_2 . Within the expression e_2 the value of the private binding v may be changed using the assignment operator ‘ \leftarrow ’. Finally, we use $fix_X[f(X)]$ to denote the least fixed-point of the function f .

3. Towards a common meta-model

In earlier work, we have shown that objects and components can most easily be modeled if classes are represented as first-class entities [26,41]. Using this approach, it is possible to incorporate various abstractions found in object-oriented programming. Unfortunately, due to position-dependent parameters, the reusability and the extensibility of the defined abstractions are limited at both the base level and the meta-object level. To overcome these shortcomings, we propose to use *forms* as the formal basis for modeling object-oriented abstractions. Furthermore, we consolidate the definition of our framework by incorporating approaches proposed by Bracha and Cook [8], Cook and Palsberg [15], Van Limberghen and Mens [44], and Rossie et al. [38].

Cook and Palsberg [15] have proposed an approach for modeling classes, mixins, inheritance, etc. using the notion of *generators* and *wrappers*. A *generator*, denoted by G , defines a constructor-like abstraction for a particular class. The important aspect of a generator is that it does not bind an object's **self**-reference. To establish the correct binding of **self**, a *wrapper*, denoted by W , is used. A generator can be considered as a unary function over **self** whereas a wrapper represents the *fixed-point operator* for the corresponding generator. Reddy [37] proposed a similar approach, but he omits the explicit separation between generators and wrappers.

This technique can be further refined using the method defined by Bracha and Cook [8]. A generator for a class can be modeled as the composition of its parent-class generator (or a collection of parent-class generators in the case of multiple derivation) and an *incremental modification*, written as Δ . We shall use the term *intermediate object* to denote the value that is constructed by the evaluation of the composite $G_1 \cdot \dots \cdot G_n \cdot \Delta$, where G_1, \dots, G_n are parent-class generators.

As an example, consider the Java code for the class `Point` (cf. Fig. 2). This class defines two private instance variables `x` and `y`, two public functions `getX` and `getY` to access the values of `x` and `y`, two public methods `move` and `double` to change the values of the `x`- and `y`-coordinates of a `Point` instance, and a constructor `Point` to create a `Point`-object and to initialize its instance variables.

Using forms, the class `Point` can be represented as follows:

```
class Point
{
    private int x, y;

    public Point (int ix, int iy) { x = ix; y = iy; }           // constructor
    public int getX() { return x; }
    public int getY() { return y; }
    public void move (int dx, int dy) { x = x + dx; y = y + dy; }
    public void double () { move (x, y); }
}
```

Fig. 2. Java code of class `Point`.

$$\begin{aligned}
\Delta_{Point}(I) &= \mathbf{let} \ x \leftarrow I.ix; \ y \leftarrow I.iy \\
&\quad \mathbf{in} \\
&\quad \{ \mathit{getX} = \lambda() : x, \ \mathit{getY} = \lambda() : y, \\
&\quad \quad \mathit{move} = \lambda(\mathit{Args}) : x \leftarrow x + \mathit{Args}.dx; \ y \leftarrow y + \mathit{Args}.dy, \\
&\quad \quad \mathit{double} = \lambda() : (I \rightarrow \mathit{self}).\mathit{move}(\{dx=x, dy=y\}) \} \\
G_{Point}(I) &= \Delta_{Point}(I) \\
W_{Point}(I) &= \mathit{fix}_s [G_{Point} (I \langle \mathit{self} = s \rangle)] \\
Point &= \{ G = \lambda(I) : G_{Point}(I), \ W = \lambda(I) : W_{Point}(I) \}
\end{aligned}$$

The behavior of the class `Point` is captured by Δ_{Point} denoting the incremental modification defined by this class. We use G_{Point} to define a generator for `Point`, W_{Point} to stand for its class wrapper, and I as the constructor arguments required to correctly initialize instances of the class `Point`. The expression $\mathit{fix}_s [G_{Point} (I \langle \mathit{self} = s \rangle)]$ yields a `Point`-object with a bound `self`-reference, which is expressed by the binding $\langle \mathit{self} = s \rangle$. The class `Point` is represented as a form, which defines bindings for its generator and wrapper. In order to create an instance of the class `Point`, one needs to call the method W with appropriate constructor arguments. For example, a `Point`-object where the x-coordinate is set to 3 and the y-coordinate is set to 5 is denoted by the expression $Point.W(\{ix=3, iy=5\})$.

Please note that private instance variables are not addressed through `self`: private instance variables are encoded in a separate form whose scope is restricted to the enclosing object. In this way, private instance variables are protected against uncontrolled access by both clients and subclasses.

A specialization of the class `Point` can be defined in a similar way. Suppose, for example, that we want the y-coordinate of a `Point` instance never to exceed a given upper bound. We can define the class `BoundedPoint` as a subclass of `Point` and introduce an instance variable `bound` in `BoundedPoint`, which encodes this bound. The Java source of the class `BoundedPoint` is shown in Fig. 3.

```

class BoundedPoint extends Point
{
    private int bound;

    public BoundedPoint (int ix, int iy, int ibound)    // constructor
        { super(ix, iy); bound = ibound; }
    public int getBound() { return bound; }
    public void move (int dx, int dy)
        { if ( (getY() + dy) < getBound() ) super.move( dx, dy ); }
}

```

Fig. 3. Java code of class `BoundedPoint`.

The behavior of the class `BoundedPoint` is modeled by the incremental modification Δ_{BPoint} , the generator G_{BPoint} , and the wrapper W_{BPoint} :

$$\begin{aligned}
 \Delta_{BPoint}(I) &= \mathbf{let} \ b \leftarrow I.b \\
 &\quad \mathbf{in} \\
 &\quad \{ \mathit{getBound} = \lambda() : b, \\
 &\quad \quad \mathit{move} = \lambda(\mathit{Args}) : \\
 &\quad \quad \quad \mathbf{if} \ ((I \rightarrow \mathit{self}).\mathit{getY}() + \mathit{Args}.dy) \\
 &\quad \quad \quad \quad < (I \rightarrow \mathit{self}).\mathit{getBound}() \\
 &\quad \quad \quad \quad \mathbf{then} \ (I \rightarrow \mathit{super}).\mathit{move}(\mathit{Args}) \} \\
 G_{BPoint}(I) &= \mathbf{let} \ \mathit{Parent}_I \leftarrow \mathit{Point}.G(I) \\
 &\quad \mathbf{in} \\
 &\quad \mathit{Parent}_I \cdot (\Delta_{BPoint}(I(\mathit{super} = \mathit{Parent}_I))) \\
 W_{BPoint}(I) &= \mathit{fix}_s [G_{BPoint}(I(\mathit{self} = s))] \\
 \mathit{BoundedPoint} &= \{ G = \lambda(I) : G_{BPoint}(I), W = \lambda(I) : W_{BPoint}(I) \}
 \end{aligned}$$

The composition of classes using inheritance requires that `self`-calls are correctly dispatched. Therefore, we have to be able to access inherited behavior such that code reuse is feasible. Thus, the abstraction Δ_{BPoint} not only requires a binding for `self` in the parameter I , but also a binding for `super` to access the inherited behavior of its direct super-class, that is, the class `Point`.

The constructor arguments encoded in the form I , which are passed to Δ_{BPoint} , provide initial values for `ix`, `iy`, and `bound`. The generator G_{BPoint} is defined as a composition of the intermediate object Parent_I (instantiated by the generator G_{Point} , which uses the bindings `ix` and `iy` in I , while ignoring any other non-relevant bindings with respect to G_{Point}) and the incremental modification Δ_{BPoint} . Finally, the fixed-point operator in W_{BPoint} ensures a correct binding of `self` within the resulting object in both the `self`-context `BoundedPoint` and the `super`-context `Point`.

Analyzing the encodings of the classes `Point` and `BoundedPoint`, we can observe that the corresponding wrappers W_{Point} and W_{BPoint} differ only in the generators used. Furthermore, the composition of the intermediate object (that is, Parent_I) and the incremental modification (that is, Δ_{BPoint}) in G_{BPoint} is very similar to the way inheritance is implemented in Java, and it can therefore be assumed that the underlying inheritance mechanism being used to define a particular set of classes will not differ in the same semantic model. Thus, by appropriately parametrizing the corresponding generators and wrappers, the same abstractions for G and W can be used to represent different classes.

This motivates a first preliminary specification of the abstraction `Class` that defines a Java-like class model:

$$\begin{aligned}
 \mathit{Class} = \lambda(A) : \mathit{fix}_{M\mathit{self}} [\{ G = \lambda(I) : \mathbf{let} \ \mathit{Parent}_I \leftarrow (A \rightarrow \mathit{super}).G(I) \\
 &\quad \mathbf{in} \ \mathit{Parent}_I \cdot (A.\Delta(I(\mathit{super} = \mathit{Parent}_I))), \\
 W = \lambda(I) : \mathit{fix}_s [M\mathit{self}.G(\{\mathit{init} = I, \mathit{self} = s\})]]
 \end{aligned}$$

Class is a *function* with one formal keyword-based argument A . If **Class** is applied to an actual argument that defines appropriate bindings for both an incremental modification Δ and a parent-class representation, then the result of this invocation is a meta-object that represents a class with Java-like inheritance and method dispatch semantics. Thus, the class **Point** can be constructed using the following expression:

$$\text{Point} = \text{Class} (\{\Delta = \Delta_{\text{Point}}, \text{super} = \{G = \lambda(I) : \langle \rangle, W = \lambda(I) : \langle \rangle\}\})$$

where the form $\{G = \lambda(I) : \langle \rangle, W = \lambda(I) : \langle \rangle\}$ denotes the *root* class of the corresponding class hierarchy. Conceptually, the definition of the root class depends on the underlying semantic model. For example, in C++ we do not have such a distinguished root class. Therefore, the sole purpose of the form $\{G = \lambda(I) : \langle \rangle, W = \lambda(I) : \langle \rangle\}$ in a model of C++ is to terminate the recursion in class specifications.

On the other hand, object-oriented languages such as Java and C# define a distinguished root class called **Object**, which defines some common behavior that is shared by all classes. Therefore, the expression that denotes the root class is usually enriched with some additional features (e.g., `toString()` or `clone()` in Java) that every class implements and possibly overrides. Note, however, that the root class must not contain any references to `super`.

Summarizing our first observations, we argue that generators and wrappers define the *semantic model* (that is, the underlying inheritance model, the method dispatch strategy, etc.) for related families of classes, whereas incremental modifications (that is, specific Δ 's) and collections of parent-class generators specify the behavior of concrete classes within a particular semantic model.

The concept of generators and wrappers constitutes a form of *meta-level protocol* [23]. A generator creates intermediate objects of the parent-class(es) and Δ , and composes these objects according to a given semantic model. The way in which these intermediate objects are composed may be different for each semantic model (e.g., in a Java-like model, methods of a class have precedence over the methods defined in a parent-class, whereas in a Beta-like model [28], the parent-class methods have precedence), but the “ingredients” of the composition are the same for all semantic models. Hence, it seems appropriate to split the functionality of generators into a static protocol part and a variable model part, denoted as *concrete* and *model* generators, respectively. Moreover, it seems to be sufficient to define only one concrete generator and parametrize it with appropriate model generators [40]. A similar separation can also be achieved for wrappers.

4. Mixins

In an ideal world, there is always a component available that precisely implements a given set of requirements that an application has to satisfy. Unfortunately, this is seldom the case and, so, we might run into the situation where we need to add some aspects of behavior orthogonal to or independent of the original behavior supported by a given component. In the context of object orientation, various authors have proposed the notion of *mixins* to achieve the necessary adjustments [8,44], but the underlying concepts are also applicable in other areas.

From our point of view, the main idea is that a mixin can be composed with (or mixed into) different parent-classes (or components) to create a related family of modified classes (or components). Using the terminology of Van Limberghen and Mens, we will call such a composition a *mixin application* [44]. Applying a mixin M to class A , written $M * A$, results in a new class, which combines the behavior of both M and A with precedence of the behavior of M . The true value of mixins lies in the fact that they can also be composed with other mixins. Such an operation is called *mixin composition*. Mixin composition takes two mixins M_1 and M_2 and yields a composite mixin $M_1 * M_2$, which combines the behavior of M_1 and M_2 , but gives precedence to the behavior of M_1 . By definition, mixin application and composition are associative, that is, $(M_1 * M_2) * A = M_1 * (M_2 * A)$ [8].

It is relatively straightforward to incorporate mixins into our model: a mixin is again represented as an object at the meta-level, that is, as a class abstraction with an unbound parent `super`. In order to handle mixin application and mixin composition uniformly, we define the concept of a *composer abstraction*, denoted by C , which is a meta-level operation that is defined for both classes and mixins. A composer is a binary function (or method) that combines the left-hand side operand (that is, a class or a mixin) with the right-hand side operand (that is, a mixin). For example, a mixin application $M * A$ is modeled as $A.C(M)$, whereas a mixin composition $M_1 * M_2$ is encoded as $M_2.C(M_1)$.

A first specification of a `Mixin` abstraction is given as follows:

$$\begin{aligned} \text{Mixin} &= \lambda(A) : \\ & \text{fix}_{Mself} [\{ G = \lambda(I) : \mathbf{let} \text{Parent}_I \leftarrow (I \rightarrow \text{super}).G(I) \\ & \quad \mathbf{in} \text{Parent}_I \cdot (A.\Delta(I\langle \text{super} = \text{Parent}_I \rangle)), \\ & \quad W = \lambda(I) : \text{fix}_s[Mself.G(\{ \text{init} = I, \text{self} = s, \\ & \quad \quad \quad \text{super} = I \rightarrow \text{super} \})], \\ & \quad C = \lambda(I) : \mathbf{Mixin}(G = \lambda(J) : \\ & \quad \quad \quad I.G(J\langle \text{super} = (J \rightarrow \text{super}).C(Mself) \rangle)) \}] \end{aligned}$$

In contrast to `Class`, `Mixin` is a function that only requires a binding for an incremental modification Δ , but not a parent-class representation. That is, if we want to instantiate a mixin, using a concrete parent-class (e.g., `Point`), this parent-class has to be passed as argument to the generator of the mixin.

To enable interoperability between classes and mixins, the composer abstraction C specified below needs to be added to the abstraction `Class`:

$$C = \lambda(I) : \mathbf{Class} (G = \lambda(J) : I.G(J\langle \text{super} = Mself \rangle))$$

The two composer abstractions C specified above (that is, for both `Mixin` and `Class`) create new meta-objects by solely passing a generator G to either the abstraction `Mixin` or `Class`. Using this approach, we override the default generator behavior of the underlying protocol [40]. As a mixin only specifies partial behavior of objects, the reader should note that, in general, it is not possible to define a wrapper W for the `Mixin` abstraction. However, defining a mixin wrapper as a function over a parent-class representation, say P , enables us to create *anonymous* classes of the form $M * P$. This concept has however limited applicability as it can only be used in combination with single inheritance.

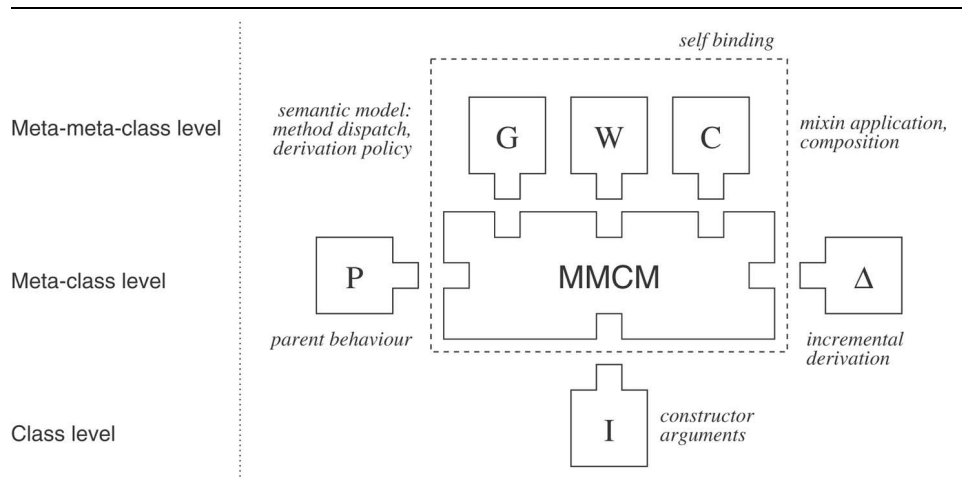


Fig. 4. A conceptual view of the meta-level framework.

Now, reconsider the specification of the classes `Point` and `BoundedPoint`. The class `BoundedPoint` redefines the method `move` of `Point` to guarantee that the y -coordinate never exceeds a given upper bound. The aspect of restricting the y -coordinate in the method `move` is independent of the behavior of the class `Point` and can, therefore, be factored out into a mixin, called `Bound`. The class `BoundedPoint` can now be defined as an application of the mixin `Bound` to the class `Point`, that is `BoundedPoint = Bound * Point`:

$$\begin{aligned} \textit{Bound} &= \textit{Mixin}(\{\Delta = \Delta_{\textit{BPoint}}\}) \\ \textit{BoundedPoint} &= \textit{Point.C}(\textit{Bound}) \end{aligned}$$

5. A meta-level framework

So far, we have illustrated how forms can be used to represent classes, mixins, and objects using generators, wrappers, composers, and incremental modifications. This approach can now be generalized into a *meta-level framework* for modeling object- and component-oriented abstractions. This framework, illustrated in Fig. 4, defines a hierarchy of meta-level abstractions for modeling *meta-classes*, *classes*, and *objects*. It substantially benefits from the features provided by forms. In particular, polymorphic extension is the key feature used to model various types of inheritance, whereas only polymorphic restriction allows us to efficiently model attribute hiding and/or method encapsulation.

The core of the meta-level framework is a *meta-meta-class* level abstraction, denoted by `MMCM` (as an abbreviation for *Meta-Meta-Class Model*). It is used to instantiate *meta-class* and *class* level abstractions of a semantic model and it provides the common behavior of all class and mixin abstractions, the corresponding meta-protocol, and hooks for concrete generators, wrappers, and composers. To instantiate a specific meta-class level abstraction (e.g., for *classes*, *mixins*, or *encapsulations*), a corresponding generator, wrapper, and composer needs to be plugged into `MMCM`.

Any meta-class level abstraction created from MMCM can be further configured by plugging in suitable parent behavior, denoted by P , and incremental derivation, denoted by Δ , in order to create a specific class level abstraction (e.g., the class `Point` or the mixin `Bound`). Finally, to instantiate objects, a constructor argument, denoted by I , has to be provided to the corresponding class level abstraction.

5.1. The MMCM abstraction

The MMCM abstraction defines the common protocol for all meta-class abstractions, that is, it provides a default constructor G , a default wrapper W , and a default composer C . MMCM is a function that returns an instantiation of a meta-class abstraction (e.g., `Class`) for a concrete semantic model. In order for the instantiation process to work correctly, the actual keyword-based argument (denoted by S) has to contain bindings for a model generator G_m , a model wrapper W_m , and a model composer C_m .

The result of MMCM is itself a function, which is used to create concrete model-specific meta-classes. In fact, we can think of MMCM and its result as functions from forms to environments or *explicit namespaces* [3]. When instantiated, MMCM returns an environment that defines the rules by which a concrete component model or object model is governed. Similarly, the instantiation of a meta-class abstraction such as `Class` yields an environment in which one can define new user-defined classes that adhere to the rules defined at the meta-level. The abstraction MMCM is defined as follows:

$$\text{MMCM} = \lambda(S) : (\lambda(Args) : \text{fix}_{Mself}[\{G, W, C\} \cdot Args])$$

where $\{G, W, C\}$ denotes the *static protocol part* (that is, the meta-protocol that is the same for *all* semantic models). The static protocol part has the following structure:

$$\begin{aligned} G &= \lambda(I) : S.G_m(I \cdot Args) \\ W &= \lambda(I) : \text{fix}_s[S.W_m(\{I = I, G = Mself.G, self = s\})] \\ C &= \lambda(M) : \\ &\quad (\text{MMCM}(S)) \\ &\quad (\{G = \lambda(I) : S.C_m(\{mixin = M, I = I, Mself = Mself\})\}) \end{aligned}$$

The expression $\text{fix}_{Mself}[\{G, W, C\} \cdot Args]$ yields the least fixed-point of the static protocol part. Thus, this expression denotes a meta-object with the self-reference `Mself` for accessing its methods.

The default generator G polymorphically extends the constructor arguments denoted by I with the model-specific arguments denoted by $Args$. The resulting form is used as the argument for the model generator G_m . The default wrapper W yields the least fixed-point of a meta-level abstraction (e.g., `Class`). In other words, the result of applying the constructor arguments, denoted by I , is a meta-object that provides model-specific operations for defining class level abstractions. The composer C creates a new class meta-object based on the model abstraction passed to MMCM and the model composer C_m .

Our framework supports also the specification of class members (that is, *static* instance variables and methods as in C++, Java, or C#) at every level. Class members are uniformly treated as meta-operations, which are encoded as additional values in the parameter $Args$.

The proper binding of class-based **self**-references is again guaranteed by the fixed-point operator $fix_{Mself}[\{G, W, C\} \cdot Args]$.

Finally, we use dynamic method binding in the definition of W in order to allow for the composer C to override the default generator. Thus, the generator created by the composer C adds an additional layer to the generator (or defines some overridden behavior) that controls mixin composition and mixin application.

5.2. The Class abstraction

In this section, we discuss the instantiation of the **Class** abstraction. More precisely, we illustrate how to define model specific generators, wrappers, and composers for a single inheritance class model.

Common to all (single inheritance) class abstractions is that they require an incremental modification Δ and a parent-class meta-object **super**. Furthermore, all single inheritance models use the same wrapper and composer denoted by W_m^C and C_m^C , respectively. The corresponding definition is given below:

$$\begin{aligned} W_m^C &= \lambda(I) : I.G(\{init=I \rightarrow init, self=I \rightarrow self\}) \\ C_m^C &= \lambda(M) : (M \rightarrow mixin).G((M \rightarrow I)(super=M \rightarrow Mself)) \end{aligned}$$

The single inheritance model wrapper W_m^C receives a form I that contains bindings for a concrete model generator, the constructor arguments, and **self**, which is the least fixed-point of the object being created. The model composer C_m^C , on the other hand, defines the behavior required for mixin application, that is, when a model specific class object is applied to a mixin.

Using W_m^C and C_m^C , we can now define the abstraction **Class** as a function over a model generator:

$$\mathbf{Class} = \lambda(G) : (\lambda(A) : (\mathbf{MMCM}(G \cdot \{W_m^C, C_m^C\}))(A))$$

When applied to a concrete model generator, **Class** yields an environment for creating model specific instances of classes that support single inheritance.

The main difference between the concrete class abstractions for various single inheritance class models can be found in the way that the corresponding model generators are defined. Consider, for example, the definitions given in Fig. 5. The model generator G_m^{Java} defines a Java-like class model, that is, objects in this model use dynamic method lookup. G_m^{Beta} , on the other hand, defines a Beta-style class model using a *prefix style* of inheritance, whereas G_m^{Stat} defines a model generator for static method dispatch.

Using G_m^{Java} , G_m^{Beta} , G_m^{Stat} , and the abstraction **Class**, we can now create model specific meta-objects as follows:

$$\begin{aligned} JavaClass &= \mathbf{Class}(\{G_m^{Small}\}) \\ BetaClass &= \mathbf{Class}(\{G_m^{Beta}\}) \\ StaticClass &= \mathbf{Class}(\{G_m^{Static}\}) \end{aligned}$$

However, prior to the use of these abstractions, we need to define a *Root*-class called *Object* that does not define any methods. Thus, its sole purpose is to terminate the recursion in

$$\begin{aligned}
G_m^{Java} &= \lambda(I) : \mathbf{let} \text{ Parent}_I \leftarrow (I \rightarrow \text{super}).G(I) \\
&\quad \mathbf{in} \text{ Parent}_I \cdot (I.\Delta(I\langle \text{super} = \text{Parent}_I \rangle)) \\
G_m^{Beta} &= \lambda(I) : \mathbf{let} \text{ Parent}_I \leftarrow (I.\Delta((I \rightarrow \text{super}) \rightarrow \Delta_{inner}) \cdot (I \rightarrow \Delta_{inner}) \cdot I) \\
&\quad \mathbf{in} \text{ Parent}_I \cdot ((I \rightarrow \text{super}).G(I \cdot \text{Parent}_I)) \\
G_m^{Stat} &= \lambda(I) : \text{fix}_{self} [\mathbf{let} \text{ Parent}_I \leftarrow (I \rightarrow \text{super}).G(I\langle \text{self} = \text{self} \rangle) \\
&\quad \mathbf{in} \text{ Parent}_I \cdot (I.\Delta(I\langle \text{self} = \text{self} \rangle)\langle \text{super} = \text{Parent}_I \rangle)]
\end{aligned}$$

Fig. 5. Single inheritance model generators.

the class specification of a single inheritance hierarchy. The class *Object* is defined as follows:

$$\text{Object} = \text{JavaClass}(\{G = \lambda(I) : \langle \rangle, \Delta = \lambda(I) : \langle \rangle\})$$

where $G = \lambda(I) : \langle \rangle$ defines a special generator that creates an intermediate root object. The reader should note that $G = \lambda(I) : \langle \rangle$ will override the *JavaClass*'s generator binding, because projection is performed from right to left. We can now use *JavaClass* to create a *Point*-class meta-object:

$$\text{Point} = \text{JavaClass}(\text{Object}\langle \Delta = \Delta_{\text{Point}} \rangle)$$

In order to create a *Point*-object where the x-coordinate is set to 3 and the y-coordinate is set to 5, we use the following expression:

$$\text{Point}.W(\{ix = 3, iy = 5\})$$

5.3. Model abstractions for mixins and encapsulation

Both mixin application and mixin composition require a distinguished model composer C_m . In the case of mixin application $M * A$, the required model composer is C_m^C , as illustrated in the previous section. Thus, C_m^C yields an intermediate object by passing A as parent to the generator of M .

In the case of mixin composition, the situation is different. The corresponding model composer C_m^M needs to guarantee that **super** is correctly interpreted in all mixins; hence we get

$$\begin{aligned}
C_m^M &= \lambda(M) : \mathbf{let} \text{ Parent}_M \leftarrow (M \rightarrow \text{init}) \rightarrow \text{super}.C(M \rightarrow \text{Mself}) \\
&\quad \mathbf{in} (M \rightarrow \text{mixin}).G((M \rightarrow \text{init})\langle \text{super} = \text{Parent}_M \rangle)
\end{aligned}$$

The subexpression $(M \rightarrow \text{init}) \rightarrow \text{super}.C(M \rightarrow \text{Mself})$ creates a new class meta-object, which defines the application of the rightmost mixin to the class meta-object that the composite mixin will be applied onto. This class meta-object is then being used to bind **super** in the leftmost mixin.

The model wrapper for mixins, denoted as W_m^M , is defined as illustrated below and can be considered as an abstraction for creating anonymous classes of the form $M * P$.¹

$$W_m^M = \lambda(I) : I.G(I \langle \text{super} = (I \rightarrow \text{init}) \rightarrow \text{super} \rangle)$$

Using the model generator G_m^{Java} , the mixin model composer C_m^M , and the mixin model wrapper W_m^M , the abstraction **Mixin** (supporting Java-like semantics) can now be defined as follows:

$$\text{Mixin} = \lambda(A) : (\text{MMCM}(\{G_m^{Java}, W_m^M, C_m^M\}))(A)$$

In order to define a class *BoundedPoint* that is governed by a Java-like semantics, we can combine the class *Point*, defined above, with a mixin *Bound*:

$$\begin{aligned} \text{Bound} &= \text{Mixin}(\{\Delta = \Delta_{BPoint}\}) \\ \text{BoundedPoint} &= \text{Point}.C(\text{Bound}) \end{aligned}$$

Going a step further in our example, consider the mixin **LinearBound**, which is a specialization of the mixin **Bound** that ensures the invariant that the y-coordinate never exceeds the x-coordinate (that is, $y \leq x$):

$$\text{LinearBound} = \text{Mixin}(\{\Delta = \{\text{getBound} = \lambda() : (I \rightarrow \text{self}).\text{getX}()\}\}) * \text{Bound}$$

If we want to define a class *DoubleBoundedPoint*, which combines the behavior of both the mixins *LinearBound* and *Bound* with the class *Point* (that is, the y-coordinate never exceeds both the x-coordinate and a given upper bound), it is not possible to define the class *DoubleBoundedPoint* using the expression $\text{LinearBound} * \text{Bound} * \text{Point}$, because the method *getBound* in the mixin *LinearBound* overrides the *Bound*-mixin's *getBound*-method and, as a consequence, the test for the upper bound will never be performed. This problem can be solved, however, by *encapsulating* the method *getBound* of the mixin *Bound*: we replace dynamically dispatched **self**-calls to *getBound* in *Bound* with static calls and remove *getBound* from the intermediate object created by the corresponding generator. We then compose *LinearBound* with the encapsulated *Bound*-mixin and the class *Point*:

$$\begin{aligned} \text{EncapsBound} &= (\text{Encapsulate}(\{\Delta = \{\text{getBound} = \lambda() : \langle \rangle\}\})).C(\text{Bound}) \\ \text{DoubleBoundedPoint} &= \text{LinearBound} * \text{EncapsBound} * \text{Point} \end{aligned}$$

The concept of method encapsulation can be seen as a variant of the *hide* operator presented by Bracha and Lindstrom [9]. It ensures that encapsulated methods become invisible to any client of a class or mixin that the encapsulation is applied to.

In order to build the **Encapsulate** abstraction used above, we need to define the corresponding model generator G_m^{Encaps} as follows:

¹ This wrapper is applicable only in combination with single inheritance.

$$G_m^{Encaps} = \lambda(I) : \\ \text{fix}_{self'} [\text{let } Parent_I \leftarrow (I \rightarrow \text{super}).G(I \langle self = (I \rightarrow self) \cdot self' \rangle) \\ \text{in } Parent_I \setminus (I.\Delta(I \langle \text{super} = Parent_I \rangle \langle self = (I \rightarrow self) \cdot self' \rangle))]$$

Reusing the Mixin's model composer C_m^M and model wrapper W_m^M , we can now define the abstraction **Encapsulate** as follows:

$$\text{Encapsulate} = \lambda(A) : (\text{MMCM}(\{G_m^{Encaps}, W_m^M, C_m^M\}))(A)$$

The abstraction **Encapsulate**, like **Mixin**, is an abstraction over Δ . Thus, the meta-object created by **Encapsulate** has to be considered as a mixin meta-object and, therefore, it can be applied to classes as well as mixins. However, the purpose of Δ is different, since it is not being used to specify an incremental modification, but to define a set of features to be encapsulated.

6. Related work

Several researchers have proposed foundational models for object- and component-oriented programming. Although most of these models are strongly based on typed λ -calculi with subtyping, stylistic differences make a rigorous comparison difficult. Some models, for example, are presented as translations from a high-level syntax into the syntax of a typed λ -calculus whereas others map high-level syntax directly into a denotational model or focus on the object syntax as a primitive calculus in its own right.

Examples of such foundational models are the recursive-record encodings of Cardelli [12], Reddy [37], and Cook [14], existential encodings proposed by Pierce and Turner [36], Bruce's model based on existential and recursive types [10], and the type-theoretic encoding of a calculus of primitive objects defined by Abadi et al. [1]. Further work in this area includes a calculus for delegation-based languages by Fisher and Mitchell [18] and a calculus of classes and mixins proposed by Bono et al. [7]. A detailed comparison of all these λ -calculus based approaches is beyond the scope of this work (refer to [1] or [11] for further discussions).

While formal models based on the λ -calculus emphasize aspects such as encapsulation, classes, inheritance, and incremental modification (e.g., method update), they do not support several important aspects of modern component-based systems, such as concurrency, distribution, active objects, and synchronization. The simplest foundation that seems appropriate to serve as a foundation of concurrent models is that of communicating, concurrent agents, which are based on some process calculus (e.g., CSP [22], CCS [31], π -calculus [33], join-calculus [20]) or on (asynchronous) actor models [4]. For a detailed survey of formal models for object orientation that address concurrency, refer to [29].

The concept of mixins has been proposed by several researchers in order to overcome some of the problems with multiple inheritance [8,14]. Van Limberghen and Mens give a denotational semantics for their mixin model, where mixins, mixin composition, and encapsulation are primitives. However, they do not incorporate an explicit notion of classes [44]. On the other hand, all these concepts are integrated as primitives in the calculus of classes and mixins proposed by Bono et al. [7]. Ancona and Zucca have studied a rigorous

semantic foundation for mixins independently from the notions of classes and objects, starting from an algebraic setting for module composition [5].

A slightly different approach is used by Flatt et al. where a subset of Java is extended with mixins [19]. Their system supports higher-order mixin composition, a hierarchy of named interface types, and resolution of accidental name collisions. In contrast to explicit encapsulation, the collision resolution system allows the “original” and the overriding method definitions to coexist. The two methods are distinguished using so-called *views* on objects, which is carried with the object at runtime and altered at each subsumption step. As a consequence, method lookup is sensitive to an object’s history of subsumption.

In order to model the mechanism for inheritance of several object-oriented languages, Rossie et al. define the notion of inheritance in terms of so-called *subobjects* [38]. From their point of view, a class C represents a collection of members (that is, the methods and instance variables that are shared by all instances of C). When a class D inherits from C , the underlying inheritance mechanism may either attempt to merge the members of C with those of D or collapse members with the same name into a single definition. Alternatively, all members of C are inherited as an indivisible collection. This collection, when instantiated, is known as a subobject. Each instance of the class D has a distinct subobject D/C as well as a subobject D/D ; the latter is also referred to as the *primary subobject* of D . Subobjects are meant to support subclass polymorphism: each subobject represents a different view of an object, allowing it to be viewed as an instance of any of its parent-classes. Subobjects are very similar to intermediate objects defined in our model, and generators can be seen as an abstraction that appropriately composes “subobjects”.

7. Conclusions

In this article, we have discussed forms, a special notion of first-class, immutable, and extensible records, as the main foundation for defining a generic meta-level framework for modeling both object- and component-oriented programming abstractions. Form composition, that is, polymorphic extension and polymorphic restriction, we argue, is the key mechanism for defining the abstractions of the framework using a uniform semantic model. One of the main insights in using forms for modeling objects is that they allow for a *compositional view* of features found in object-oriented programming languages and, therefore, enable us to achieve a stronger separation between functional elements (that is, methods) and their composition. In particular, inheritance does not need to be taken as a fundamental operation, but can be considered as an application of form composition. Thus, forms and the corresponding composition mechanisms can be viewed as an enabling technology for evaluating and implementing object models, which in turn facilitates the mediation between different object and component models needed in a successful, flexible, and reliable approach for component-oriented software development.

Representing classes as meta-objects allows us to integrate programming abstractions such as class variables and methods, various inheritance mechanisms, and different method dispatch strategies. The resulting flexibility is achieved by introducing intermediate objects (that is, objects with an unbound `self`-reference) specifying partial behavior of objects, generators defining compositions of intermediate objects, and wrappers, which apply a

fixed-point operator over composed intermediate objects to establish a sound interpretation of *self*. As a result, different object models can be represented uniformly and, therefore, provide us with the means for a seamless incorporation of software artifacts into one unified composition system. Moreover, the meta-level framework presented in this article is much more general and defines fewer restrictions than similar language approaches (e.g., Common Lisp [42], Smalltalk [21], and Python [27]).

The generalization of the concepts of generators, wrappers, and composition of intermediate objects can be used not only to define classes and class abstractions, but also to model mixins, mixin application, mixin composition as well as method encapsulation. By (i) splitting the functionality of generators and wrappers into a static protocol part and a variable model part and (ii) introducing the concept of a composer abstraction, it is possible to derive all object-oriented abstractions from a single meta-model abstraction. Whereas the resulting meta-model abstraction (that is, MMCM) defines the generic behavior of all meta-class abstractions and a corresponding meta-protocol, so-called *model* generators, wrappers, and composers specify the common behavior of specific semantic models (e.g., *JavaClass*).

Software composition takes place at both a system and a language level [6]. Our meta-level framework is targeted at the language level as it is open with respect to the support of different component models. In fact, there exists no predefined or preferred component model. A new component model can be added to the system by using the meta-meta-class abstraction MMCM and an appropriate generator, wrapper, and composer to instantiate a meta-class level abstraction representing the newly supported component model. A specific set of composition techniques constitutes a specific semantic model. Thus, adding a new technique requires a new set of semantic model abstractions and a corresponding instantiation of MMCM. New component models and composition techniques created in this way enrich the underlying composition system (and language). Therefore, our form-based meta-level framework can be seen as an open, flexible meta-composition language.

One aspect of modeling compositional abstractions is the implementation of code reuse mechanisms. The most prominent mechanism used in object- and component-based technology today is inheritance. Unfortunately, inheritance gives rise to the so-called *fragile base class problem* [30]. Our meta-level framework does not eliminate the occurrence of this problem by default. However, an appropriate definition of the model-specific generator, wrapper, and composer may reduce the risk for the fragile base class problem, or may even eliminate its occurrence completely.

So far, we have defined our meta-level framework without considering any typing issues, mainly due to the fact that forms are untyped. However, the definition of an appropriate type theory for forms and, therefore, for our meta-model will be of major concern for ensuring the *correctness of composition*. Thus, future work in this area will include the definition of an appropriate typing scheme for forms and an investigation of its applicability in the context of object- and component-oriented programming abstractions.

Early results from recent work in this area [24,34] indicate that the definition of a suitable component type system will go beyond “traditional” type theories. In fact, a new type theory for flexible and reliable software composition will have to provide a framework for detecting mismatches at component interface level, ensure component compatibility, and check behavioral properties of component-based software systems. Furthermore, a

suitable component type system should be able to express both the services required by components and those that are provided, even in the presence of incomplete system knowledge.

Acknowledgements

We would like to thank our former colleagues of the Software Composition Group for inspiring discussions on these topics as well as the anonymous reviewers for commenting on earlier drafts.

References

- [1] Martín Abadi, Luca Cardelli, *A Theory of Objects*, Springer, 1996.
- [2] Franz Achermann, *Forms, agents and channels: defining composition abstraction with style*, Ph.D. Thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 2002.
- [3] Franz Achermann, Oscar Nierstrasz, *Explicit namespaces*, in: Jürg Gutknecht, Wolfgang Weck (Eds.), *Modular Programming Languages*, LNCS, vol. 1897, Springer, 2000, pp. 77–89.
- [4] Gul Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.
- [5] Davide Ancona, Elena Zucca, *An algebraic approach to mixins and modularity*, in: Michael Hanus, Mario Rodríguez-Artalejo (Eds.), *Proceedings of the 5th International Conference on Algebraic and Logic Programming*, ALP'96, September 1996, LNCS, vol. 1139, Springer, 1996, pp. 179–193.
- [6] Uwe Assmann, *Invasive Software Composition*, Springer, 2003.
- [7] Viviana Bono, Amit J. Patel, Vitaly Shmatikov, *A core calculus of classes and mixins*, in: Rachid Guerraoui (Ed.), *Proceedings ECOOP'99*, June 1999, LNCS, vol. 1628, Springer, 1999, pp. 43–66.
- [8] Gilad Bracha, William Cook, *Mixin-based inheritance*, in: Norman Meyrowitz (Ed.), *Proceedings OOPSLA/ECOOP'90*, October 1990, ACM SIGPLAN Notices, vol. 25, 1990, pp. 303–311.
- [9] Gilad Bracha, Gary Lindstrom, *Modularity meets inheritance*, in: *Proceedings of International Conference on Computer Languages*, April 1992, IEEE Computer Society, 1992, pp. 282–290.
- [10] Kim B. Bruce, *A paradigmatic object-oriented programming language: design, static typing and semantics*, *Journal of Functional Programming* 4 (2) (1994).
- [11] Kim B. Bruce, Luca Cardelli, Benjamin C. Pierce, *Comparing object encodings*, in: *Proceedings of Theoretical Aspects of Computer Software*, TACS'97, August 1997, LNCS, vol. 1281, Springer, 1997, pp. 415–438.
- [12] Luca Cardelli, *A semantics of multiple inheritance*, *Information and Computation* 76 (1988) 138–164.
- [13] Luca Cardelli, John C. Mitchell, *Operations on records*, in: Carl Gunter, John C. Mitchell (Eds.), *Theoretical Aspects of Object-Oriented Programming*, MIT Press, 1994. Also appeared as SRC Research Report 48, and in *Mathematical Structures in Computer Science* 1 (1) (1991) 3–48.
- [14] William R. Cook, *A denotational semantics of inheritance*, Ph.D. Thesis, Department of Computer Science, Brown University, Providence, RI, May 1989.
- [15] William Cook, Jens Palsberg, *A denotational semantics of inheritance and its correctness*, *Information and Computation* 114 (2) (1994) 329–350.
- [16] Laurent Dami, *Software composition: towards an integration of functional and object-oriented approaches*, Ph.D. Thesis, Centre Universitaire d'Informatique, University of Geneva, CH, 1994.
- [17] Laurent Dami, *A lambda-calculus for dynamic binding*, *Theoretical Computer Science* 192 (1998) 201–231.
- [18] Kathleen Fisher, John C. Mitchell, *A delegation-based object calculus with subtyping*, in: *Proceedings FCT'95*, LNCS, vol. 965, Springer, 1995, pp. 42–61.
- [19] Matthew Flatt, Shriram Krishnamurthi, Matthias Felleisen, *Classes and mixins*, in: *Proceedings POPL'98*, January 1998, San Diego, ACM Press, 1998, pp. 171–183.
- [20] Cédric Fournet, Georges Gonthier, *The reflexive chemical abstract machine and the join-calculus*, in: *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, January 1996, ACM, 1996, pp. 372–385.

- [21] Adele Goldberg, David Robson, *Smalltalk-80: The Language*, Addison-Wesley, 1989.
- [22] Charles A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.
- [23] Grégor Kiczales, Jim des Rivières, Daniel G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.
- [24] Markus Lumpe, A π -calculus based approach to software composition, Ph.D. Thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.
- [25] Markus Lumpe, Jean-Guy Schneider, Form-based software composition, in: Mike Barnett, Steve Edwards, Dimitra Giannakopoulou, Gary T. Leavens (Eds.), *Proceedings of ESEC'03 Workshop on Specification and Verification of Component-Based Systems, SAVCBS'03*, September 2003, Helsinki, Finland, 2003, pp. 58–65.
- [26] Markus Lumpe, Jean-Guy Schneider, Oscar Nierstrasz, Using metaobjects to model concurrent objects with PICT, in: *Proceedings of Languages et Modèles à Objets'96*, October 1996, Leysin, 1996, pp. 1–12.
- [27] Mark Lutz, David Ascher, *Learning Python*, 2nd edition, O'Reilly & Associates, 2003.
- [28] Ole Lehmann Madsen, Birger Møller-Pedersen, Kristen Nygaard, *Object-Oriented Programming in the BETA Programming Language*, Addison-Wesley, 1993.
- [29] Tom Mens, A survey on formal models for OO. Technical Report vub-tinf-tr-94-03, Department of Computer Science, Vrije Universiteit Brussel, Belgium, 1994.
- [30] Leonid Mikhajlov, Emil Sekerinski, A study of the fragile base class problem, in: Eric Jul (Ed.), *Proceedings ECOOP 1998*, July 1998, Brussels, Belgium, LNCS, vol. 1445, Springer, 1998, pp. 355–382.
- [31] Robin Milner, *Communication and Concurrency*, Prentice Hall, 1989.
- [32] Robin Milner, *Communicating and Mobile Systems: The π -Calculus*, Cambridge University Press, 1999.
- [33] Robin Milner, Joachim Parrow, David Walker, A calculus of mobile processes, Part I/II, *Information and Computation* 100 (1992) 1–77.
- [34] Oscar Nierstrasz, Contractual types. Technical Report IAM-03-004, University of Bern, Institute of Computer Science and Applied Mathematics, August 2003.
- [35] Oscar Nierstrasz, Laurent Dami, Component-oriented software technology, in: Oscar Nierstrasz, Dennis Tschirtzis (Eds.), *Object-Oriented Software Composition*, Prentice Hall, 1995, pp. 3–28.
- [36] Benjamin C. Pierce, David N. Turner, Simple type-theoretic foundations for object-oriented programming, *Journal of Functional Programming* 4 (2) (1994) 207–247.
- [37] Uday S. Reddy, Objects as closures: abstract semantics of object oriented languages, in: *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, July 1998, ACM, 1988, pp. 289–297.
- [38] Jonathan G. Rossie, Daniel P. Friedman, Mitchell Wand, Modeling subobject-based inheritance, in: Pierre Cointe (Ed.), *Proceedings ECOOP'96*, July 1996, LNCS, vol. 1098, Springer, 1996, pp. 248–274.
- [39] Johannes Sametinger, *Software Engineering with Reusable Components*, Springer, 1997.
- [40] Jean-Guy Schneider, Components, scripts, and glue: a conceptual framework for software composition, Ph.D. Thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.
- [41] Jean-Guy Schneider, Markus Lumpe, Synchronizing concurrent objects in the Pi-Calculus, in: Roland Ducournau, Serge Garlatti (Eds.), *Proceedings of Languages et Modèles à Objets'97*, October 1997, Roscoff, Hermes, 1997, pp. 61–76.
- [42] Guy L. Steele, *Common Lisp the Language*, 2nd edition, Digital Press, Thinking Machines, Inc., 1990.
- [43] Clemens Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd edition, Addison-Wesley, ACM Press, 2002.
- [44] Marc Van Limberghen, Tom Mens, Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems, *Object-Oriented Systems* 3 (1) (1996) 1–30.
- [45] W3C. Extensible Markup Language (XML) 1.0, 3rd edition, W3C Recommendation, February 2004. <http://www.w3.org/TR/REC-xml>.
- [46] Larry Wall, Tom Christiansen, Jon Orwant, *Programming Perl*, 3rd edition, O'Reilly & Associates, 2000.
- [47] Markus Zenger, Type-safe prototype-based component evolution, in: Magnusson Boris (Ed.), *Proceedings ECOOP 2002*, June 2002, Malaga, Spain, LNCS, vol. 2374, Springer, 2002, pp. 470–497.