A Form-based Framework for Class Extensions

Markus Lumpe

Department of Computer Science
Iowa State University
113 Atanasoff Hall
Ames, IA, 50011-1040, USA
lumpe@cs.iastate.edu

TR #05-08
March 2005

Abstract. Class extensions allow for a modular addition of new behavior to an existing class hierarchy. However, the reliance on position-dependent parameters in mainstream programming languages has often a negative impact on the way new behavior can be specified. This observation has led us to explore the concept of *forms*, which are first-class extensible records that, in combination with a small set of purely asymmetric operators, provide a core language for an extensible, flexible, and robust software development approach. In this paper, we present a refinement of our recent work on a substitution-free lambda calculus with forms. More precisely, we develop the $\lambda\mathcal{F}_{@}$-calculus in which names are replaced with *shared forms* and parameter passing is modeled using *explicit contexts* and show, how this calculus can be used to model *open classes*, a key mechanism for class extensions.

A Form-based Framework for Class Extensions

Markus Lumpe
Department of Computer Science
Iowa State University
113 Atanasoff Hall
Ames, IA-50011, USA
lumpe@cs.iastate.edu

ABSTRACT

Class extensions allow for a modular addition of new behavior to an existing class hierarchy. However, the reliance on position-dependent parameters in mainstream programming languages has often a negative impact on the way new behavior can be specified. This observation has led us to explore the concept of *forms*, which are first-class extensible records that, in combination with a small set of purely asymmetric operators, provide a core language for an extensible, flexible, and robust software development approach. In this paper, we present a refinement of our recent work on a substitution-free lambda calculus with forms. More precisely, we develop the $\lambda\mathcal{F}_{@}$-calculus in which names are replaced with *shared forms* and parameter passing is modeled using *explicit contexts* and show, how this calculus can be used to model *open classes*, a key mechanism for class extensions.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; F.3.2 [**Semantics of Programming Languages**]: Operational semantics; D.2.13 [**Software Engineering**]: Reusable Software

1. INTRODUCTION

A general engineering principle is to base the development of new products on accumulated, generally available system knowledge and experience. By putting emphasis on reuse and evolution, the component-oriented software technology, which has become the major approach to develop modern software systems [33, 39, 44], can be considered a veritable incarnation of this principle. The primary objective of the component-based software technology is to take elements from a collection of reusable software components (i.e., take *components-off-the-shelf*), apply some required domain-specific incremental modifications, and build applications by simply plugging them together.

However, in order to be truly successful, the component-based software development approach not only needs to provide abstractions to represent different component models and composition techniques, but must also incorporate a systematic method for constructing large software systems [5, 28]. In particular, we need a canonical set of preferably language-neutral composition mechanisms that allows for building applications as compositions of reusable software components [41]. Moreover, in order to guarantee flexible, reliable, and verifiable software composition, these mechanisms have to be based on a suitable formal foundation [17, 26, 34, 41]. A precise semantics is essential if we are to deal with multiple architectural styles and component models within a common, unifying framework.

The component-based approach has emerged from the object-oriented approach that has already shown a positive influence on software reuse. In fact, most component-oriented systems are still built using mainstream object-oriented techniques and languages. However, evolution and reuse of component-based software do not depend on the object-oriented programming approach alone [39]. Moreover, when using object-oriented techniques to specify component composition, one often faces the *extensibility problem* that arises from the fact that mainstream object-oriented languages only provide limited support for modular addition of new behavior to existing classes. In particular, the inheritance relationships in mainstream object-oriented and class-based systems are not adequate for expressing many useful forms of incremental modifications [7].

To address the extensibility problem several approaches have emerged (e.g., CLOS [22], MultiJava [14], and AspectJ [23]) that focus on a particular technique: *class extensions*. Class extensions allow for a modular addition of both new classes and new operations to an existing class hierarchy. Thus, class extension provide a controllable mechanism to support unanticipated changes [7].

The key mechanism that enables class extensions is the concept of *open classes* [13]. An open class is one that can be extended with new behavior without editing the class directly. Open classes are of great interest for component-oriented software development. Software components are sufficiently self-contained, prefabricated units of independent deployment [44] that encapsulate some domain-specific abstractions. As such, they exist, for example, in binary form without source code (e.g., ActiveX components [38]).

Thus, in order the build applications, software components, in general, have to be configured and adapted to meet actual deployment requirements. Component configuration and adaptation can, therefore, be considered instances of unanticipated changes. Support for unanticipated changes is a prerequisite for reuse. However, the extensions applied to the components in questions should be confined to the units that introduces them, should be visible to collaborating clients, and should provide support to resolve conflicts due to competing extensions [7].

There is, however, a second aspect that affects our ability to define flexible, extensible, and reusable abstractions for software composition. Mainstream program techniques often hamper the definition of general purpose compositional abstractions, as they impose a dependence on position and arity of parameters [18, 42]. For example, in the standard λ-calculus the functions λ`(x, y).x` and λ`(y, x).y` are equivalent, but λ`(x, y).x` and λ`(y, x).x` are different, as position matters in λ-calculus. Moreover, if we use de Bruijn indices [19], then names disappear totally, as arguments to functions are uniquely identified by their positions. Thus, if we abstract from position and use instead the parameter names as keys functions like $\lambda\langle$`x, y`$\rangle$`.x` and $\lambda\langle$`y, x`$\rangle$`.x` become indistinguishable.

This observation has led us to explore the concept of *forms* [24, 27, 41]. Forms are first-class extensible records that define mappings from labels to values, which, in combination with a small set of purely asymmetric operators, provide a core language to define extensible, flexible, and robust software abstractions.

Forms gain their specific value due to two asymmetric operators: *form extension* and *form restriction*. Form extension, written `F` $\cdot$ `G`, allows one to simultaneously add or redefine a set of services, whereas form restriction, written `F`$\backslash$`G`, 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`. Both operators are the main building block in a fundamental concept for defining adaptable, extensible, and more robust software abstractions [24, 28, 41].

Programmatically, forms are both compile-time and runtime entities. As compile-time entities, forms can be used to denote components, component interfaces, and component composition. At run-time, on the other hand, forms provide uniform and language-neutral access to component services and support runtime composition on demand. However, forms are not bound to a particular computational model. They are an environment-independent framework that has to be combined with a concrete target system like the λ-calculus or the π-calculus.

We have been studying a substitution-free variant of the λ-calculus, called $\lambda\mathcal{F}$, where names are replaced with forms and parameter passing is modeled using explicit contexts [25]. Explicit contexts mimic λ-calculus substitutions. However, unlike λ-calculus in which substitutions are meta-level operations [2], explicit contexts have a form-based syntactic representation to record named parameter bindings. For example, the $\lambda\mathcal{F}$-term `a[b]` denotes an expression `a`, which meaning 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 the environment [32].

The $\lambda\mathcal{F}$-calculus is a calculus in which parameters are identified by names rather than positions. Position-independent parameter specification allows for the development of extensible, flexible, and reliable component-based application. The resulting flexibility of a form-based programming model can also be seen, for example, in XML/HTML forms [48], where fields are encoded as named (rather than positional) parameters, in Python [47] and Common Lisp [43], where functions can be defined to take arguments by keywords, and in Perl [49] where it is a common technique to pass a map of name/value pairs as arguments to a function or method.

We can use the $\lambda\mathcal{F}$-calculus to model open classes in a purely functional way. However, the purely functional encoding makes it cumbersome to propagate extensions to collaborating clients. Locally, it is always possible to build the fixed-point of the extensions and update all collaborating clients, respectively. Unfortunately, this technique does not work on a larger scale. In particular, it does not allow for an approach in which extensions are only propagated to a subset of clients. To solve this problem, we propose *shared forms*. Shared forms denote locations of forms and add, therefore, an imperative element to the calculus. Clients that maintain a reference to a shared form can communicate extensions through its value. For example, we can view the methods of an object as collaborating clients that may alter the state of its instance. In a purely functional setting, updating the state results in a new instance, that is, strictly speaking, no collaboration between the methods of the object at all. However, if we encode the state of an object as a shared form and provide each method with a reference to its location, then any state change is visible to all method instantaneously. The reader should note that we can also control the visibility of a shared form, if we place it inside an explicit context. Therefore, the concepts of shared forms and explicit contexts manifest a feasible mechanism to support unanticipated changes in class-based systems.

2. A MODEL FOR OPEN CLASSES

Object- and component-oriented abstractions can most easily be modeled if classes are represented as first-class entities [28]. It is however important to note that the framework, in which these abstractions are modeled, does not need to be class-based. In Simula, for example, classes are represented as procedures returning a pointer to an activation record (i.e., an instance of the class).

A strength of the object-oriented paradigm lies in its uniform representation of access to different kinds of data. However, whereas mainstream object-oriented programming models typically only provide a fixed and, in general, position-dependent set of mechanisms for constructing software systems, forms allow for the definition of an open and therefore extensible object-oriented development approach [3, 28, 32]. In particular, forms composition, that is, form extension and form restriction are the key mechanisms for a seamless integration of various programming abstractions found in modern object-oriented programming systems.

We begin our study of a form-based model for open classes by adopting an approach proposed by Cook and Palsberg [15] and Bracha and Cook [9] in which classes are modeled by means of *incremental modifications*, *generators*, and *wrappers*. An incremental modification, denoted by Δ, captures the extensions of both state and behavior defined by a class. A generator, denoted by G, defines a constructor-like abstraction that yields an instance of a class with an unbound `self`-reference. In other words, a generator G builds a prototype instance of a class, which is the behavioral building block for a class and the primary abstraction to enable code reuse and class extensions. To close a prototype instance, that is, to establish a correct binding of `self`, we need to apply a wrapper, denoted by W, to the prototype instance of a class. In other words, a wrapper represents the fixed-point operator for the corresponding generator.

Suppose, we want to define a class `Point` using the specification shown in Table 1. This class defines two private instance variables `x` and `y`, both initialized to zero, two public functions `getX` and `getY` to access the values of `x` and `y`, two public functions `setX` and `setY` to update the values of `x` and `y`, and two public methods `move` and `double` to change the values of the `x`- and `y`-coordinates of a `Point` instance. Moreover, we assume that all functions are position-independent. So, the notation `move⟨dx, dy⟩` means that the function `move` takes two keyword-based arguments `dx` and `dy`.

```
class Point =
  var x = 0, y = 0

  fun getX⟨⟩        = x
  fun setX⟨nx⟩      = x ← nx
  fun getY⟨⟩        = y
  fun setY⟨ny⟩      = y ← ny
  fun move⟨dx, dy⟩  = x ← x + dx, y ← y + dy
  fun double⟨⟩      = self.move⟨dx=x, dy=y⟩
end
```

Table 1: The class `Point`.

To express the incremental modification $\Delta_{\texttt{Point}}$ defined by class `Point`, we adopt the Self approach [45] and represent $\Delta_{\texttt{Point}}$ as a set of names slots or *traits* [40]. However, to maintain a strict encapsulation of state, we do not blend state and behavior. Instead, we model state as *explicit contexts* that provide an environment to resolve the occurrences of private instance variables and the `self`-reference.

$$\Delta_{\texttt{Point}}\langle\texttt{State}\rangle = \left(\begin{array}{|c|c|}\hline \texttt{getX} & \texttt{return x} \\ \hline \texttt{setX} & \texttt{update x} \\ \hline \texttt{getY} & \texttt{return y} \\ \hline \texttt{setY} & \texttt{update y} \\ \hline \texttt{move} & \texttt{move points} \\ \hline \texttt{double} & \texttt{double points} \\ \hline \end{array}\right) \texttt{[State]}$$

The effect of applying the context `[State]` to the body of $\Delta_{\texttt{Point}}$ is that each slot is provided with an interpretation of the current state. In other words, we can think of the slots in $\Delta_{\texttt{Point}}$ as collaborating clients and the context `[State]` as an abstraction to propagate a current set of class extensions.

The purpose of the generator $G_{\texttt{Point}}$ is to build `Point` prototype instances. A `Point` prototype instance is constructed by combining its constructor arguments with the `Point` state template slots and applying the result to $\Delta_{\texttt{Point}}$.

$$G_{\texttt{Point}}\langle\texttt{Args}\rangle = \Delta_{\texttt{Point}}\langle \left(\begin{array}{|c|c|}\hline \texttt{x} & 0 \\ \hline \texttt{y} & 0 \\ \hline \end{array}\right) \oplus \texttt{Args} \rangle$$

Finally, the wrapper $W_{\texttt{Point}}$ yields a `Point`-object by building the fixed-point for the prototype's `self`-reference and applying it as a context to $G_{\texttt{Point}}$.

$$W_{\texttt{Point}} = \mathit{fix}_{\texttt{self}}\langle\, G_{\texttt{Point}}\ [\texttt{self}]\ \rangle$$

To construct a `Point`-object, where `x` is set to 3 and `y` is set to 5, we write $W_{\texttt{Point}}\langle \texttt{x} = 3,\ \texttt{y} = 5\rangle$.

A specialization of the class `Point` can be defined in a similar way by encoding inheritance by means of delegation like in Self [45]. That is, the original inherited behavior can be accessed through a designated reference `super`. Suppose, we want the `y`-coordinate of a `Point`-instance never to exceed a given upper bound. To specify this property, we can define the class `BoundedPoint` as a subclass of `Point` and introduce an instance variable `bound` in `BoundedPoint` that represents this upper bound. The specification of the class `BoundedPoint` is shown in Table 2.

```
class BoundedPoint extends Point =
  var bound = 0

  fun getBound⟨⟩   = bound
  fun move⟨dx, dy⟩ = if self.getY⟨⟩ + dy < bound
                     then super.move⟨dx, dy⟩
end
```

Table 2: The class `BoundedPoint`.

The behavior of the class `BoundedPoint` is again captured by our three main abstractions. However, class extension by means of inheritance requires some extra effort to link super- and subclasses together. Furthermore, the link process should not require that we have to alter the superclass(es) in any way. The solution to this problem a combination of delegation [45], aggregation [38], and explicit contexts. First, the generator $G_{\texttt{BPoint}}$ for class `BoundedPoint` calls the super-class generator $G_{\texttt{Point}}$ to create a `Point` prototype instance using a subclass context that provides a `self`-reference. Then, this prototype instance is extended with the result of applying $\Delta_{\texttt{BPoint}}$ to a combination of $G_{\texttt{BPoint}}$'s constructor arguments, the `BoundedPoint` state template slots, and a reference `super` that points to the `Point` prototype instance. The result is a `BoundedPoint` prototype instance that provides the inherited behavior, the new (overridden) behavior, and access to original behavior through `super`.

The corresponding abstractions for class `BoundedPoint` are defined as follows.

$$\Delta_{\texttt{BPoint}}\langle \texttt{State} \rangle = \left(\begin{array}{|l|l|} \hline \texttt{getBound} & \texttt{return bound} \\ \hline \texttt{move} & \texttt{move bpoints} \\ \hline \end{array} \right) [\texttt{State}]$$

$$\begin{array}{l} G_{\texttt{BPoint}}\langle \texttt{Args} \rangle = \\ \quad \textbf{let} \\ \quad\quad \texttt{Parent} = G_{\texttt{Point}}\langle \texttt{Args} \rangle \\ \quad \textbf{in} \\ \quad\quad \texttt{Parent} \oplus \Delta_{\texttt{BPoint}}\langle \left(\begin{array}{|l|l|} \hline \texttt{bound} & \texttt{0} \\ \hline \texttt{super} & \texttt{Parent} \\ \hline \end{array} \right) \oplus \texttt{Args} \rangle \end{array}$$

$$W_{\texttt{BPoint}} = \mathit{fix}_{\texttt{self}}\langle\, G_{\texttt{BPoint}}\ [\texttt{self}] \,\rangle$$

To construct a `BoundedPoint`-object, where the `x`-field is set to 3, the `y`-field is set to 5, and `bound` is set to 100, we write $W_{\texttt{BPoint}}\langle \texttt{x} = 3,\ \texttt{y} = 5,\ \texttt{bound} = 100 \rangle$.

The reader should note that the actual meaning of both the generator and the wrapper can be manipulated through an explicit context as well. For example, suppose we want to replace the class `Point` with a new version `ExtendedPoint` in class `BoundedPoint`. Instead of defining a new set of abstractions for class `BoundedPoint`, we can evaluate them in a context than maps $G_{\texttt{Point}}$ to $G_{\texttt{ExtendedPoint}}$. We write

$$W_{\texttt{BPoint}}\langle \texttt{x} = 3,\ \texttt{y} = 5,\ \texttt{bound} = 100 \rangle\ [\langle\, \texttt{G}_{\texttt{Point}} = \texttt{G}_{\texttt{ExtendedPoint}} \,\rangle]$$

to create an object for the extended `BoundedPoint` class. This approach roughly corresponds to the concept of *mixin application* [46]. The class `BoundedPoint` now represents an *abstract subclass*, which is instantiated with the super-class `ExtendedPoint`. When using this technique, it is however important to take special care, as it gives rise to the so-called *fragile base class problem* [29].

3. RELATED WORK

Several researchers have proposed foundational approaches that can serve as a model for open classes. However, stylistic differences make a rigorous comparison difficult. For example, some models represent object-oriented abstractions as translations from a high-level syntax into 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 [11], Reddy [36], and Cook [16], existential encodings proposed by Pierce and Turner [35], Bruce's model based on existential and recursive types [10], and the type-theoretic encoding of a calculus of primitive objects defined by Abadi and Cardelli [1]. Further work in this area includes a calculus for delegation-based languages by Fisher and Mitchell [21] and a calculus of classes and mixins proposed by Bono et al. [8].

To overcome some of the problems related to multiple inheritance, the concept of mixins [9, 46] has emerged. Van Limberghen and Mens [46], for example, give a denotational semantics of a model in which mixins, *mixin composition*, *mixin application*, and *encapsulation* are primitive. However, their approach does not incorporate an explicit notion of classes, as in the calculus of classes and mixins proposed by Bono et al. [8].

Lumpe and Schneider [28] have proposed a form-based meta-level framework for modeling both object- and component-oriented programming abstractions. This framework defines a hierarchy of meta-level abstractions to model *meta-classes*, *classes*, and *objects*. The core of the meta-level framework is a *meta-meta-class model* that can be used to instantiate a specific semantic model (i.e., a specific class-based infrastructure). A semantic model, say, for example, of C++, Java, or Beta, is captured by so-called *model generators*, *model wrappers*, and *model composers*. Each semantic model is guaranteed to interoperate with any other semantic model instantiated within the framework. The framework can, therefore, be viewed as a meta-object protocol [22] for open classes, as it provides the means for both to bridge between different object models and to incorporate reusable software artifacts.

A full-scale programming language that supports both open classes and symmetric multiple dispatch is MultiJava [13, 14]. In MultiJava, an open class is a class to which new methods can be added without editing the class directly. The new behavior is visible to the package that implements it and to the packages importing that package. Moreover, MultiJava allows for modular typechecking of class extensions. But, in order to prevent occurrences of *message-not-understood* and *message-ambiguous* errors, MultiJava imposes some requirements that ensure that extended classes are properly implemented. However, MultiJava still uses a position-dependent parameter passing mechanism, which is due to one of MuliJava's goals, that is, to be backward-compatible to Java.

Recently, Bergel et al. [7] have proposed *classboxes*, a dynamically typed module system for object-oriented languages that provides support for controlling the visibility of both method addition and method replacement. That is, a classbox defines a scope within which classes, methods, and variables are defined. A classbox may also import definitions form other classboxes and extend them without affecting the originating classbox. In other words, 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.

4. THE CALCULUS

In this section, we define $\lambda\mathcal{F}_{@}$, a calculus in which names are replaced with *shared forms* and parameter passing is modeled using *explicit contexts*. Our main goal is to find a minimal set of language elements that preserves the expressive power of forms, while retaining the main functional attributes of forms, which are essential in an approach that provides visibility control for class extensions.

Parameter passing in the λ-calculus is defined by the β-rule and substitution, which can be interpreted as an operation on the equivalence classes of λ-terms [6]. However, substitution, as used in the classical λ-calculus, is actually a

meta-level concept and not part of the language. By making it part of the language, Abadi et al [2] argue that we can achieve a better correspondence between the language theory and its implementation. For example, in the classical λ-calculus we write $\mathtt{a\{b/x\}}$ to denote the term `a` where all free occurrences of `b` have been replaced with `x`. However, substitution can be a very expensive term-rewriting operation, as a term may contain many free occurrences of a variable in question. To address this issue, we propose a solution in which substitutions are mimicked by *explicit contexts* [2, 4] that have a form-based syntactic representation. To resolve occurrences of free variables, we shall use explicit contexts for both forms and $\lambda\mathcal{F}_{@}$-terms.

Assignment enables the sharing of values between different parts of a program. The actual information exchange takes place through common references to locations containing values. Unfortunately, if one adds assignment to a language, then, in general, software abstractions defined in this language lack referential transparency, which would it make hard if not impossible to define an approach that provides support for visibility control. Therefore, rather than adding an assignment operator to forms, we introduce the concept of *shared forms*. A shared form denotes a location of a form, which, taken for itself, is immutable.

Consider, for example, two programs `A` and `B`. To execute these programs, we may need to provide two applications domains $\mathtt{D_A}$ and $\mathtt{D_B}$, which we may derive from some domain prototype `D`. Furthermore, the program `B` may require a specific event-handling mechanism $\mathtt{E_B}$. Now, both $\mathtt{D_A}$ and $\mathtt{D_B}$ may already define a standard mechanism to handle events. This default handler is defined as a shared form, written `@E`. To override $\mathtt{D_B}$'s event handler, we can combine its handler `@E` with $\mathtt{E_B}$ using form extension, written $(\mathtt{D_B} \rightarrow \mathtt{E}) \cdot \mathtt{D_B}$. The subterm $(\mathtt{D_B} \rightarrow \mathtt{E})$ extracts the default handler. However, since it has been specified as a shared form, $(\mathtt{D_B} \rightarrow \mathtt{E})$ actually yields a reference to a fresh location initialized with `E`. Combining this location with $\mathtt{E_B}$ results in an updated handler in $\mathtt{D_B}$. To run the programs `A` and `B`, we combine them with there corresponding application domains, written $\mathtt{(A\ [D_A])}$ and $\mathtt{(B\ [D_B])}$, respectively. The changes that we have applied to `B`'s domain do not affect `A`'s behavior, that is, the changes are confined to `B`'s domain. However, every client that refers to `B`'s domain can see this modification.

We presuppose a countable infinite set, $\mathcal{L}$, of *labels*, and let l, m, n to range over labels. We also presuppose a countable 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, or even classes, and types. However, we do not require any particular property except that equality and inequality are defined for elements of $\mathcal{V}$. Furthermore, 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. The syntax of the $\lambda\mathcal{F}_{@}$-calculus is given in Table 3.

Every form is derived from the *empty form* $\langle\rangle$, which denotes an empty set of services. The services that a form offers are specified as *binding extensions*. A binding extension, written $\langle\, \mathtt{l} = \mathtt{s} \,\rangle$, denotes a form's capability to allow access to a service `s` that is published under the name `l`. For example,

F, G, H	::=	$\langle\rangle$	*empty form*
	|	X	*form variable*
	|	$@F$	*shared form*
	|	$F\langle\, l = V \,\rangle$	*binding extension*
	|	$F \cdot G$	*form extension*
	|	$F \backslash G$	*form restriction*
	|	$F \rightarrow\ l$	*form dereference*
	|	$F[G]$	*form context*
V	::=	$\mathcal{E}$	*empty value*
	|	a	*abstract value*
	|	M	$\lambda\mathcal{F}_{@} - value$
M, N	::=	F	*form*
	|	$M.l$	*projection*
	|	$\lambda(X)\ M$	*abstraction*
	|	$M\ N$	*application*
	|	$M[F]$	$\lambda\mathcal{F}_{@} - context$

Table 3: Syntax of the $\lambda\mathcal{F}_{@}$-Calculus.

we write `F.a` to access the service that is bound by label `a` in form `F`.

A *shared form* `@F` defines a location of a form initialized with `F`. We can use shared forms to organize "cross-cutting" services. That is, if two forms `G` and `H` both contain a reference to the location of `F`, then `G` and `H` share `F`'s services. Moreover, if an operation involving either `G` or `H` alters the value in this location, then this change is visible to the other form instantaneously.

Form extension and *form restriction* allow for the manipulation of sets of services. Using form extension, written $\mathtt{F \cdot G}$, one can simultaneously add or redefine a set of services, whereas form restriction, written $\mathtt{F \backslash G}$, denotes a form that is restricted to all bindings of `F` that do not occur in `G`. Both operators provide a convenient way to define *compositional styles* [3]. Assume, for example, that we want to compose two forms `F` and `G`, but we want to give a specific 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 that defines a *conditional update*, which can be specified using both form extension and form restriction: $\mathtt{F \cdot (G \backslash \langle\rangle \langle\, m = F.m \,\rangle)}$. Depending on the actual bindings defined by `F` and `G`, we can distinguish three different meanings of $\mathtt{F \cdot (G \backslash \langle\rangle \langle\, m = F.m \,\rangle)}$:

- If the label `m` does occur neither in `F` nor `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`.

Forms can also occur as values in binding extensions. These forms are called *nested forms* and they facilitate the specification of structured service sets. To extract a nested form

bound by a label `l` in a form `F`, we use $\texttt{F} \rightarrow \texttt{l}$. Note, however, that if the binding involving label `l` does not actually map a nested form, then the result of $\texttt{F} \rightarrow \texttt{l}$ is $\langle\rangle$ – the *empty form*. The reason for this is that we want to distinguish between forms, which denote sets of services, and plain services.

A *form context* `F[G]` defines a refinement of `F` by using `G` as an environment to look up what would otherwise be a free variable in `F`. We use form dereference to perform the lookup operation. That is, 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 $\texttt{G} \rightarrow \texttt{X}$. In the case that `G` does not define a binding for `X`, the result is $\langle\rangle$, which effectively removes the set of services associated with `X` in `F`. This allows for an approach in which a sender and a receiver can communicate open form expressions. The receiver of this open form expression can use its local context to close (i.e., configure) the received form expression according to a site-specific protocol, but may also chose to ignore it (e.g., the configuration of a Web-browser to run an application associated with a specific MIME-type).

Forms and *projections* replace variables in $\lambda\mathcal{F}_@$. A form stands for an *explicit namespace* [4] or module [20], which can comprise an arbitrary number of services. The form may itself contain free variables, which will be resolved by means of the deployment environment or an evaluation context. In other words, free variables in a form expression allow for a computational model with *late binding*.

With projections we recover variable references of λ-calculus. We require, however, that the subject of a projection denotes 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".

Both *abstraction* and *application* correspond to the notions used in λ-calculus. As in λ-calculus, the `X` in λ(`X`) `a` stands for the parameter. But unlike 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 current context. We can then use `[F]` to build a *closure* of λ(`X`) `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 λ(`X`) `a` is λ(`X`) `(a[F])`.

In form expressions, binding extension has precedence over form extension, form extension has precedence over a form restriction, form restriction has precedence over form deference, which in turn has precedence over form contexts. Furthermore, all form operators are left associative. Similarly, in $\lambda\mathcal{F}_@$-terms, projection has precedence over application, which in turn has precedence over $\lambda\mathcal{F}_@$-contexts. Moreover, application is left associative. Parenthesis may be used in $\lambda\mathcal{F}_@$-expressions in order to enhance readability or to overcome the default precedence rules.

There are three forms of binding: $\lambda(X)\ M$, $M[F]$ and $F[G]$. Conventionally, we write $fv(M)$ and $bv(M)$ to denote the set of free variables of M and the set of bound variables of M, respectively. Additionally, we write $fp(M)$ and $bp(M)$ to denote the set of free projections of M and the set of bound projections of M, respectively, which are defined as follows:

$$\begin{array}{rcl} fp(M) & = & \{\ (X,p) \mid X \in fv(M) \wedge X.p \text{ occurs in } M\ \} \\ bp(M) & = & \{\ (X,p) \mid X \in bv(M) \wedge X.p \text{ occurs in } M\ \} \end{array}$$

These two sets allow for a detailed analysis of a translation of the λ-calculus into the $\lambda\mathcal{F}_@$-calculus, as they actually capture the λ-calculus' definitions of free- and bound-names.

To facilitate the specification of $\lambda\mathcal{F}_@$-expression, we will use the following abbreviations:

- We will often omit $\langle\rangle$ in a form, and write for example $\langle \texttt{l} = \texttt{a} \rangle\langle \texttt{m} = \texttt{b} \rangle$ instead of $\langle\rangle\langle \texttt{l} = \texttt{a} \rangle\langle \texttt{m} = \texttt{b} \rangle$.
- We will use a more compact record-like notion to specify forms, and write for example $\langle \texttt{l} = \texttt{a}, \texttt{m} = \texttt{b}, \texttt{n} = \texttt{c} \rangle$ instead of $\langle\rangle\langle \texttt{l} = \texttt{a} \rangle\langle \texttt{m} = \texttt{b} \rangle\langle \texttt{n} = \texttt{c} \rangle$.
- We will write `\() b` to specify a function with body `b` that takes no arguments.
- We will use "**let** $\texttt{l}_1$ `=` $\texttt{v}_1$`, ...,` $\texttt{l}_n$ `=` $\texttt{v}_n$ **in** `e`" to specify a context expression $\texttt{e}[\langle \texttt{l}_1 = \texttt{v}_1, ..., \texttt{l}_n = \texttt{v}_n \rangle]$.

5. OPERATIONAL SEMANTICS

We describe the interpretation of $\lambda\mathcal{F}_@$-expressions in the style of *natural semantics* [31], which provides a reasonably efficient evaluation model for $\lambda\mathcal{F}_@$.

We use the Greek letters σ and γ to denote states and locations, respectively. A state σ is a set of mappings from locations to values. We assume that all locations in σ are pairwise distinct. A location γ is called fresh, if $\gamma \notin dom(\sigma)$. We also have a notion for updating states. We define $\sigma[\gamma \mapsto v]$ to be the state σ' that is as σ except that the value bound to γ is v.

$$(\sigma[\gamma \mapsto v])(\gamma') = \begin{cases} v & \text{if } \gamma' = \gamma \\ \sigma(\gamma') & \text{otherwise} \end{cases}$$

In $\lambda\mathcal{F}_@$, we distinguish between *form values* and *model values*. Form values are finite mappings from a countable infinite set of labels to model values. However, form values do not contain any projections or form dereferences, as they have been replaced with their corresponding denoted values, and shared forms, as they have been replaced with their corresponding locations.

The set of model values depends on the concrete underlying computational model. In $\lambda\mathcal{F}_@$, we distinguish between five different kinds of model values: *empty value*, *abstract value*, *form value*, *closure*, and *abstract application*. Empty value

and abstract value have the same denotation as in forms, whereas the structure of a form value is as specified above.

A closure $\lambda(X)\ (M[\overline{F}])$ denotes an abstraction or function in *head normal form*. The context $[\overline{F}]$ represents the evaluation environment at the time the abstraction $\lambda(X)\ M$ was evaluated. To force the evaluation of the body of a closure (i.e., evaluate $M[\overline{F}]$), it has to be applied to a concrete argument (i.e., a $\lambda\mathcal{F}_{@}$-value).

An abstract application $a\ \overline{M}$ represents the fact that its actual interpretation lies outside the $\lambda\mathcal{F}_{@}$-calculus. The function a is abstract and its actual meaning is unknown. Therefore, the target system, in which the abstract application $a\ \overline{M}$ is embedded, is responsible for the proper handling of this expression. We have chosen this approach, rather than using $\perp$ (i.e., undefined) in this situation, because the meaning of $a\ \overline{M}$ is not really undefined, but merely our knowledge about it is incomplete.

We use $\overline{F}, \overline{G}, \overline{H}$ to range over the set of forms values, and $\overline{M}, \overline{N}$ to range over the set of model values. The set of $\lambda\mathcal{F}_{@}$-values is defined as shown in Table 4.

$\overline{F}, \overline{G}, \overline{H}$	::=	$\langle\rangle$	*empty form value*
	|	γ	*location*
	|	$\overline{F}\langle\, l = \overline{M}\,\rangle$	*binding extension value*
	|	$\overline{F} \cdot \overline{G}$	*form extension value*
	|	$\overline{F} \backslash \overline{G}$	*form restriction value*
$\overline{M}, \overline{N}$	::=	$\mathcal{E}$	*empty value*
	|	a	*abstract value*
	|	$\overline{F}$	*form value*
	|	$\lambda(X)\ (M[\overline{F}])$	*closure*
	|	$a\ \overline{M}$	*abstract application*

Table 4: $\lambda\mathcal{F}_{@}$ Values.

The operational semantics of $\lambda\mathcal{F}_{@}$ is given by a transition system using rules of the form

$$\overline{H} \vdash M, \sigma \Rightarrow \overline{M}, \sigma'$$

indicating that, under an evaluation context $\overline{H}$, a term M in state σ evaluates to a value $\overline{M}$ and a final state σ'. The transition system is shown in Table 5.

The meaning of a $\lambda\mathcal{F}_{@}$-term depends on its deployment environment. A deployment environment, represented by $\overline{H}$, defines a context that may map some of the free variables of M to deployment-specific values. We require, however, that $\overline{H}$ is *well-formed*. That is, if $\overline{H}$ is a deployment environment and σ is an initial state, then

$$\forall\ \gamma \in \overline{H} :\ \gamma \in dom(\sigma).$$

In addition, we assume that $\overline{H}$ is *minimal*, that is, it contains only observable binding extensions or locations. We call such a form value *normalized form*. We write $\overline{\overline{F}}(\sigma)$ to denote the normalized form of $\overline{F}$ under σ. A normalized form $\overline{\overline{F}}(\sigma)$ is behaviorally equivalent to $\overline{F}$ under σ, written $\overline{\overline{F}}(\sigma) \approx \sigma \models \overline{F}$. Moreover, since a normalized form contains solely observable binding extensions it is isomorphic to a classical record, but we still maintain position independency.

$$\overline{\overline{F}}(\sigma) \quad ::= \quad \begin{cases} \langle\rangle & n = 0 \\ \langle l_1 = v_1, l_2 = v_2, \ldots, l_n = v_n\rangle & n > 0 \end{cases}$$

with

$\forall\ i, j \in \{1 \ldots n\} \wedge i \neq j :\ l_i \neq l_j,$

$\forall\ i \in \{1 \ldots n\} :\ v_i \neq \mathcal{E}\ \wedge\ v_i \neq \langle\rangle,$ and

$\forall\ \gamma \in \overline{F} :\ \overline{\sigma(\gamma)}(\sigma) = \langle l_k = v_k, \ldots, l_m = v_m\rangle\ \wedge\ 1 \leq k \leq m \leq n$

Figure 1: Normalized Forms.

The transition system uses two additional, mutually-dependent total functions to resolve *projections* and *form dereferences*. Feature access in forms is performed from right-to-left [27]. However, rather than using classical records as semantic model of forms, we adopt that of interacting systems [30]. Therefore, forms are characterized not by the bindings they define, but by the bindings that are observable in them. We say, a binding is not observable if it cannot be distinguished from $\mathcal{E}$ or $\langle\rangle$. For example, both $\langle\rangle\langle\, \mathtt{m} = \mathcal{E}\,\rangle$ and $\langle\rangle$ are considered equivalent. We write $\overline{F} \parallel l$ to express that label l is not observable in form value $\overline{F}$.

The function $[\![\, \overline{M}, l\,]\!](\sigma)$, called *projection evaluation*, maps projections of the form $\overline{M}.l$ under state σ to a model value $\overline{N}$, and is defined as follows:

$$\left.\begin{array}{l} [\![\, \langle\rangle, l\,]\!](\sigma) \\ [\![\, \mathcal{E}, l\,]\!](\sigma) \\ [\![\, a, l\,]\!](\sigma) \\ [\![\, (\lambda(X)\ M[\overline{F}]), l\,]\!](\sigma) \\ [\![\, a\ \overline{M}, l\,]\!](\sigma) \end{array}\right\} = \mathcal{E}$$

$$[\![\, \gamma, l\,]\!](\sigma) = [\![\, \sigma(\gamma), l\,]\!](\sigma)$$

$$[\![\, (\overline{F}\langle\, m = \overline{M}\,\rangle), l\,]\!](\sigma) = [\![\, \overline{F}, l\,]\!](\sigma) \quad \text{if } m \neq l$$

$$[\![\, (\overline{F}\langle\, l = \overline{M}\,\rangle), l\,]\!](\sigma) = \begin{cases} \overline{M} & \text{if } \overline{M} \not\equiv \overline{F} \\ \mathcal{E} & \text{otherwise} \end{cases}$$

$$[\![\, (\overline{F} \cdot \overline{G}), l\,]\!](\sigma) = \begin{cases} [\![\, \overline{G}, l\,]\!](\sigma) & \text{if } \overline{G} \parallel l \\ [\![\, \overline{F}, l\,]\!](\sigma) & \text{otherwise} \end{cases}$$

$$[\![\, (\overline{F} \backslash \overline{G}), l\,]\!](\sigma) = \begin{cases} \mathcal{E} & \text{if } \overline{G} \parallel l \\ [\![\, \overline{F}, l\,]\!](\sigma) & \text{otherwise} \end{cases}$$

To illustrate the effect of *projection evaluation*, consider the following examples:

$$[\![\, \langle\, \mathtt{l} = \mathtt{a}, \mathtt{m} = \mathtt{b}\,\rangle, \mathtt{m}\,]\!](\emptyset) = \mathtt{b}$$
$$[\![\, (\langle\, \mathtt{l} = \mathtt{a}, \mathtt{m} = \mathtt{b}\,\rangle) \backslash (\gamma), \mathtt{m}\,]\!](\{\gamma \mapsto \langle\, \mathtt{m} = \mathtt{c}\,\rangle\}) = \mathcal{E}$$
$$[\![\, \langle\, \mathtt{l} = \mathtt{a}, \mathtt{m} = \langle\, \mathtt{n} = \mathtt{c}\,\rangle\,\rangle, \mathtt{m}\,]\!](\emptyset) = \mathcal{E}$$
$$[\![\, (\langle\, \mathtt{l} = \mathtt{a}, \mathtt{m} = \langle\, \mathtt{n} = \mathtt{c}\,\rangle\,\rangle) \cdot \gamma, \mathtt{m}\,]\!](\{\gamma \mapsto \langle\, \mathtt{l} = \mathtt{b}, \mathtt{m} = \mathtt{d}\,\rangle\}) = \mathtt{d}$$

$$\text{(F-EMPTY)}\quad \overline{H} \vdash \langle\rangle, \sigma \Rightarrow \langle\rangle, \sigma$$

$$\text{(F-VAR}_{\mathcal{E}}\text{)}\quad \frac{\overline{H} \vdash [\![\, \overline{H}, X \,]\!](\sigma) = \mathcal{E}}{\overline{H} \vdash X, \sigma \Rightarrow \langle\!\langle \overline{H}, X \rangle\!\rangle(\sigma), \sigma}$$

$$\text{(F-VAR}_{\overline{V}}\text{)}\quad \frac{\overline{H} \vdash [\![\, \overline{H}, X \,]\!](\sigma) = \overline{M};\ \ \overline{M} \neq \mathcal{E}}{\overline{H} \vdash X, \sigma \Rightarrow \overline{M}, \sigma}$$

$$\text{(F-SHARED)}\quad \frac{\overline{H} \vdash F, \sigma \Rightarrow \overline{F}, \sigma';\ \ \gamma \neq dom(\sigma')}{\overline{H} \vdash @F, \sigma \Rightarrow \gamma, \sigma'[\,\gamma \mapsto \overline{F}\,]}$$

$$\text{(F-BIND)}\quad \frac{\overline{H} \vdash F, \sigma \Rightarrow \overline{F}, \sigma';\ \ \overline{H} \vdash V, \sigma' \Rightarrow \overline{M}, \sigma''}{\overline{H} \vdash F\langle\, l = V \,\rangle, \sigma \Rightarrow \overline{F}\langle\, l = \overline{M} \,\rangle, \sigma''}$$

$$\text{(F-BIND}_{\gamma}\text{)}\quad \frac{\overline{H} \vdash F, \sigma \Rightarrow \gamma, \sigma';\ \ \overline{H} \vdash V, \sigma' \Rightarrow \overline{M}, \sigma''}{\overline{H} \vdash F\langle\, l = V \,\rangle, \sigma \Rightarrow \gamma, \sigma''[\,\gamma \mapsto \sigma''(\gamma)\langle\, l = \overline{M} \,\rangle\,]}$$

$$\text{(F-EXT)}\quad \frac{\overline{H} \vdash F, \sigma \Rightarrow \overline{F}, \sigma';\ \ \overline{H} \vdash G, \sigma' \Rightarrow \overline{G}, \sigma''}{\overline{H} \vdash F \cdot G, \sigma \Rightarrow \overline{F} \cdot \overline{G}, \sigma''}$$

$$\text{(F-EXT}_{\gamma}\text{)}\quad \frac{\overline{H} \vdash F, \sigma \Rightarrow \gamma, \sigma';\ \ \overline{H} \vdash G, \sigma' \Rightarrow \overline{G}, \sigma''}{\overline{H} \vdash F \cdot G, \sigma \Rightarrow \gamma, \sigma''[\,\gamma \mapsto \sigma''(\gamma) \cdot \overline{G}\,]}$$

$$\text{(F-RES)}\quad \frac{\overline{H} \vdash F, \sigma \Rightarrow \overline{F}, \sigma';\ \ \overline{H} \vdash G, \sigma' \Rightarrow \overline{G}, \sigma''}{\overline{H} \vdash F \backslash G, \sigma \Rightarrow \overline{F} \backslash \overline{G}, \sigma''}$$

$$\text{(F-RES}_{\gamma}\text{)}\quad \frac{\overline{H} \vdash F, \sigma \Rightarrow \gamma, \sigma';\ \ \overline{H} \vdash G, \sigma' \Rightarrow \overline{G}, \sigma''}{\overline{H} \vdash F \backslash G, \sigma \Rightarrow \gamma, \sigma''[\,\gamma \mapsto \sigma''(\gamma) \backslash \overline{G}\,]}$$

$$\text{(F-DEREF)}\quad \frac{\overline{H} \vdash F, \sigma \Rightarrow \overline{F}, \sigma'}{\overline{H} \vdash F \rightarrow\ l, \sigma \Rightarrow \langle\!\langle \overline{F}, l \rangle\!\rangle(\sigma'), \sigma'}$$

$$\text{(F-CXT)}\quad \frac{\overline{H} \vdash G, \sigma \Rightarrow \overline{G}, \sigma';\ \ \overline{\overline{G} \cdot \overline{H}}(\sigma') \vdash F, \sigma' \Rightarrow \overline{F}, \sigma''}{\overline{H} \vdash F[G], \sigma \Rightarrow \overline{F}, \sigma''}$$

$$\text{(V-EMPTY)}\quad \overline{H} \vdash \mathcal{E}, \sigma \Rightarrow \mathcal{E}, \sigma$$

$$\text{(V-ABS)}\quad \overline{H} \vdash a, \sigma \Rightarrow a, \sigma$$

$$\text{(LF-NORM)}\quad \frac{\overline{H} \vdash F, \sigma \Rightarrow \overline{F}, \sigma'}{\overline{H} \vdash F, \sigma \Rightarrow \overline{\overline{F}}(\sigma'), \sigma'}$$

$$\text{(LF-SEL)}\quad \frac{\overline{H} \vdash M, \sigma \Rightarrow \overline{M}, \sigma'}{\overline{H} \vdash M.l, \sigma \Rightarrow [\![\, \overline{M}, l \,]\!](\sigma'), \sigma'}$$

$$\text{(LF-CXT)}\quad \frac{\overline{H} \vdash F, \sigma \Rightarrow \overline{F}, \sigma';\ \ \overline{\overline{F} \cdot \overline{H}}(\sigma') \vdash M, \sigma' \Rightarrow \overline{M}, \sigma''}{\overline{H} \vdash M[F], \sigma \Rightarrow \overline{M}, \sigma''}$$

$$\text{(LF-ABS)}\quad \overline{H} \vdash \lambda(X)\ M, \sigma \Rightarrow \lambda(X)\ (M[\overline{H}]), \sigma$$

$$\text{(LF-APPL}_{\mathcal{E}}\text{)}\quad \frac{\overline{H} \vdash M, \sigma \Rightarrow \mathcal{E}, \sigma'}{\overline{H} \vdash M\ N, \sigma \Rightarrow \mathcal{E}, \sigma'}$$

$$\text{(LF-APPL}_{a}\text{)}\quad \frac{\overline{H} \vdash M, \sigma \Rightarrow a, \sigma';\ \ \overline{H} \vdash N, \sigma' \Rightarrow \overline{N}, \sigma''}{\overline{H} \vdash M\ N, \sigma \Rightarrow a\ \overline{N}, \sigma''}$$

$$\text{(LF-APPL}_{\overline{F}}\text{)}\quad \frac{\overline{H} \vdash M, \sigma \Rightarrow \overline{F}, \sigma';\ \ \overline{\overline{H} \cdot \overline{F}}(\sigma') \vdash N, \sigma' \Rightarrow \overline{N}, \sigma''}{\overline{H} \vdash M\ N, \sigma \Rightarrow \overline{N}, \sigma''}$$

$$\text{(LF-APPL}_{\lambda}\text{)}\quad \frac{\overline{H} \vdash M, \sigma \Rightarrow \lambda(X)\ (M'[\overline{G}]), \sigma';\ \ \overline{H} \vdash N, \sigma' \Rightarrow \overline{N}, \sigma'';\ \ \overline{\overline{G}\langle\, X = \overline{N} \,\rangle}(\sigma'') \vdash M', \sigma'' \Rightarrow \overline{M'}, \sigma'''}{\overline{H} \vdash M\ N, \sigma \Rightarrow \overline{M'}, \sigma'''}$$

Table 5: Operational semantics of the $\lambda\mathcal{F}_{@}$-Calculus.

In the form $\langle\, \mathtt{l} = \mathtt{a}, \mathtt{m} = \mathtt{b} \,\rangle$, label `m` refers to the abstract value `b`. Therefore, $[\![\langle\, \mathtt{l} = \mathtt{a}, \mathtt{m} = \mathtt{b} \,\rangle, \mathtt{m}]\!](\emptyset)$ yields `b`. The second example yields $\mathcal{E}$, because $\langle\, \mathtt{m} = \mathtt{c} \,\rangle$, the value in location γ, restricts $\langle\, \mathtt{l} = \mathtt{a}, \mathtt{m} = \mathtt{b} \,\rangle$ by hiding the binding involving label `m`. In the third example, the label `m` binds a nested form value. Therefore, the result of the projection evaluation is $\mathcal{E}$. Finally, since the form $\langle\, \mathtt{l} = \mathtt{b}, \mathtt{m} = \mathtt{d} \,\rangle$ in location γ is used as a form extension of $\langle\, \mathtt{l} = \mathtt{a}, \mathtt{m} = \langle\, \mathtt{n} = \mathtt{c} \,\rangle \,\rangle$, the result in this example is `d`, which is the actual observable right-most value bound by label m.

Secondly, the function $\langle\!\langle \overline{F}, l \rangle\!\rangle(\sigma)$, called *form dereference evaluation*, maps form deferences of the form $\overline{F} \rightarrow\ l$ under state σ to a form value $\overline{G}$, and is defined as follows:

$$
\begin{array}{lcll}
\langle\!\langle \langle\rangle, l \rangle\!\rangle(\sigma) & = & \langle\rangle & \\
\langle\!\langle \gamma, l \rangle\!\rangle(\sigma) & = & \langle\!\langle \sigma(\gamma), l \rangle\!\rangle(\sigma) & \\
\langle\!\langle \overline{F}\langle\, m = \overline{M} \,\rangle, l \rangle\!\rangle(\sigma) & = & \langle\!\langle \overline{F}, l \rangle\!\rangle(\sigma) & \text{if } m \neq\ l \\
\langle\!\langle \overline{F}\langle\, l = \overline{M} \,\rangle, l \rangle\!\rangle(\sigma) & = & \begin{cases} \langle\rangle & \text{if } \overline{M} \not\equiv \overline{F} \\ \overline{\overline{M}}(\sigma) & \text{otherwise} \end{cases} & \\
\langle\!\langle \overline{F} \cdot \overline{G}, l \rangle\!\rangle(\sigma) & = & \begin{cases} \langle\!\langle \overline{G}, l \rangle\!\rangle(\sigma) & \text{if } \overline{G} \parallel l \\ \langle\!\langle \overline{F}, l \rangle\!\rangle(\sigma) & \text{otherwise} \end{cases} & \\
\langle\!\langle \overline{F} \backslash \overline{G}, l \rangle\!\rangle(\sigma) & = & \begin{cases} \langle\rangle & \text{if } \overline{G} \parallel l \\ \langle\!\langle \overline{F}, l \rangle\!\rangle(\sigma) & \text{otherwise} \end{cases} &
\end{array}
$$

To illustrate the effect of *form dereference evaluation*, consider the following examples:

$$
\begin{array}{l}
\langle\!\langle \langle\, \mathtt{l} = \mathtt{a}, \mathtt{m} = \mathtt{b} \,\rangle, \mathtt{m} \rangle\!\rangle(\emptyset) = \langle\rangle \\
\langle\!\langle (\langle\, \mathtt{l} = \mathtt{a}, \mathtt{m} = \mathtt{b} \,\rangle) \backslash (\gamma)), \mathtt{m} \rangle\!\rangle(\{\gamma \mapsto \langle\, \mathtt{m} = \mathtt{c} \,\rangle\}) = \langle\rangle \\
\langle\!\langle \langle\, \mathtt{l} = \mathtt{a}, \mathtt{m} = \langle\, \mathtt{n} = \mathtt{c} \,\rangle \,\rangle, \mathtt{m} \rangle\!\rangle(\emptyset) = \langle\, \mathtt{n} = \mathtt{c} \,\rangle \\
\langle\!\langle (\langle\, \mathtt{l} = \mathtt{a}, \mathtt{m} = \langle\, \mathtt{n} = \mathtt{c} \,\rangle \,\rangle) \cdot \gamma, \mathtt{m} \rangle\!\rangle(\{\gamma \mapsto \langle\, \mathtt{l} = \mathtt{b}, \mathtt{m} = \mathtt{d} \,\rangle\}) = \langle\rangle
\end{array}
$$

In the form $\langle\, \mathtt{l} = \mathtt{a}, \mathtt{m} = \mathtt{b} \,\rangle$, label `m` binds the abstract value `b`. Therefore, the form derefence evaluation yields $\langle\rangle$. The second example yields $\langle\rangle$ also, because $\langle\, \mathtt{m} = \mathtt{c} \,\rangle$, the value in location γ, restricts $\langle\, \mathtt{l} = \mathtt{a}, \mathtt{m} = \mathtt{b} \,\rangle$ by hiding the binding involving label `m`. In the third example, the label `m` binds a nested form value $\langle\, \mathtt{n} = \mathtt{c} \,\rangle$, which is, therefore, the result in this example. Finally, since the form $\langle\, \mathtt{l} = \mathtt{b}, \mathtt{m} = \mathtt{d} \,\rangle$ in location γ is used as a form extension of $\langle\, \mathtt{l} = \mathtt{a}, \mathtt{m} = \langle\, \mathtt{n} = \mathtt{c} \,\rangle \,\rangle$, the result in this example is $\langle\rangle$, because, now, `m` actually maps the abstract value `d` and not a nested form.

Finally, the transition rule LF-APPL$_{\overline{\mathrm{F}}}$, as shown in Table 5, deserves some additional comments. This rules captures the situation in which the operator M of an application $M\ N$, under an evaluation context $\overline{H}$, reduces not to a closure, but to a form value $\overline{F}$. In such a case $\overline{F}$ constitutes a local refinement of $\overline{H}$ with scope N. This approach is similar to way the so-called sandbox expressions are handled in the Piccola-calculus [3]. When evaluating a sandbox expression `A;B` the term left to the semicolon defines a *root* context or *controlled environment* for the right-hand side agent. However, `A` in `A;B` may not evaluate to a form. In this case the agent `A;B` is *stuck* and identified with $\mathcal{E}$. In $\lambda\mathcal{F}_{@}$, on the other hand, an application $M\ N$ can only reduce to $\mathcal{E}$, if M reduces to $\mathcal{E}$, which means that the application $M\ N$ has "no value" and will simply be discarded without reducing the argument N.

6. ENCODING CLASS EXTENSIONS

We have illustrated an approach to model open classes and class extensions in Section 2. In this section we shall use the $\lambda\mathcal{F}_{@}$-calculus to encode this model in two steps. In the first step, we will give a purely functional interpretation of the model. That is, we will not use shared forms. We shall then use the second step to develop a refinement of our encoding scheme to illustrate the added benefit of shared forms.

To facilitate the presentation of the encodings, we assume the existence of form-based binary operators and conditional expressions. We write `x + y` to denote the application of the binary position-independent function `+` (i.e., addition) to the arguments `x` and `y`. So, the term `x + y` has to read as an application `+` $\langle$ `left = x, right = y` $\rangle$ returning the sum of `x` and `y`.

Similarly, we write `if b then` $\mathtt{e}_1$ `else` $\mathtt{e}_2$ to denote the application of the ternary position-independent function `if` to the arguments `b`, $\mathtt{e}_1$, and $\mathtt{e}_2$. The function `if` first evaluates `b`, and, depending on the result, either continues to evaluate $\mathtt{e}_1$ or $\mathtt{e}_2$. We can omit the `else`-part in an application of `if` without any consequences. This is due to the fact that a function in the $\lambda\mathcal{F}_{@}$-calculus is not characterized by the parameters it declares, but by the parameters it effectively uses.

In a purely functional model, we need an explicit fixed-point operator to model both dynamic method lookup and state update. The operational semantics of the $\lambda\mathcal{F}_{@}$-calculus is strict. We need, therefore, a strict (or *call-by-value*) fixed-point operator. In the λ-calculus, this operator is defined as

```
fix = λf.((λx.f (λy.(x x) y)) (λx.f (λy.(x x) y)))
```

To reuse this definition, we need to embed it into $\lambda\mathcal{F}_{@}$. The translation of a closed λ-calculus term M into $\lambda\mathcal{F}_{@}$ is defined by the function $[\![M]\!]$ specified below:

$$
\begin{array}{lcl}
[\![x]\!] & = & x.arg \\
[\![\lambda x.\ M]\!] & = & \lambda(x)\ [\![M]\!] \\
[\![M\ N]\!] & = & ([\![M]\!]\ \langle\, arg = [\![N]\!] \,\rangle)
\end{array}
$$

Here, λ-calculus variables are encoded as projections to extract the actual value associated with those variables. The encoding of abstractions maps a position-dependent function to a position-independent functions, whereas the encoding of application builds a form expression for the λ-term in argument position. The encoding of `fix` into $\lambda\mathcal{F}_{@}$ is shown in Figure 2.

Analyzing the result, we observe that an expression of the kind $\langle\, \mathtt{arg} = \mathtt{X.arg} \,\rangle$ is the same as `X`. We can, therefore, rewrite `h` as follows:

$$\mathtt{h}\ = \lambda(\mathtt{x})\ \mathtt{f.arg}\ \langle\, \mathtt{arg} = (\lambda(\mathtt{y})\ (\mathtt{x.arg}\ \mathtt{x})\ \mathtt{y}) \,\rangle$$

We can now use some more meaningful names and obtain a strict fixed-point combinator, written `FIX`, in the $\lambda\mathcal{F}_{@}$-calculus as shown in Table 6.

$$\texttt{FIX} \;=\; \lambda(\texttt{Fun})\ (\texttt{h}\ \langle\,\texttt{self} = \texttt{h}\,\rangle)\ [\langle\,\texttt{h} = \lambda(\texttt{Fix})\ (\texttt{Fun.f}\ (\lambda(\texttt{Args})\ (\texttt{Fix.self Fix})\ \texttt{Args}))\,\rangle]$$

Table 6: A Strict Fixed-point Combinator in $\lambda\mathcal{F}_{@}$.

$$\begin{array}{rcl}
[\![\texttt{fix}]\!] & = & \lambda(\texttt{f})\ [\![\texttt{g g}]\!] \\
& & \quad \text{where } \texttt{g} = \lambda\texttt{x.f}\ (\lambda\texttt{y.(x x)}\ \texttt{y}) \\
& = & \lambda(\texttt{f})\ [\![\texttt{g}]\!]\ \langle\,\texttt{arg} = [\![\texttt{g}]\!]\,\rangle \\
& & \quad \text{where } \texttt{g} = \lambda\texttt{x.f}\ (\lambda\texttt{y.(x x)}\ \texttt{y}) \\
& = & \lambda(\texttt{f})\ \texttt{h}\ \langle\,\texttt{arg} = \texttt{h}\,\rangle \\
& & \quad \text{where } \texttt{h} = \lambda(\texttt{x})\ \texttt{f.arg}\ \langle\,\texttt{arg} = \texttt{h}'\,\rangle \\
& & \qquad\quad \texttt{h}' = (\lambda(\texttt{y})\ (\texttt{x.arg}\ \langle\,\texttt{arg} = \texttt{x.arg}\,\rangle) \\
& & \qquad\qquad\qquad\qquad \langle\,\texttt{arg} = \texttt{y.arg}\,\rangle)
\end{array}$$

Figure 2: Translation of `fix`.

With the help of the fixed-point operator `FIX`, we can now specify a $\lambda\mathcal{F}_{@}$-encoding of the model abstractions discussed in Section 2. We start with the encoding of class `Point` as illustrated in Figure 3.

```
ClassPoint =
 let
   PointBehavior =
     ⟨ getX = \() State.x,
       setX = \(Args) self State⟨x=Args.nx⟩,
       getY = \() State.y,
       setY = \(Args) self State⟨y=Args.ny⟩,
       move = \(Args) self State⟨x=State.x+Args.dx,
                                 y=State.y+Args.dy⟩,
       double = \() (self ⟨⟩).move ⟨dx=State.x,
                                    dx=State.x ⟩⟩

   PointState = ⟨ x = 0, y = 0⟩

   Δ_Point = \(State) PointBehavior

   G_Point = \(Args)
      Δ_Point Args·(PointState\⟨x=Args.x, y=Args.y⟩)

   W_Point = FIX ⟨f = \(Self) (G_Point[⟨self=Self⟩])⟩
 in
   ⟨W = W_Point, G = G_Point ⟩
 end
```

Figure 3: Encoding of Class `Point`.

In a purely functional representation every object is a recursive specification (i.e., a recursive function). To instantiate a `self`-reference of an object, that is, to "unroll" the recursive specification, we write `self` $\langle\rangle$. The value associated with `self` is an unary function. If applied to $\langle\rangle$, then this function is the *identity* function, that is, it yields the current instance. This behavior enables *dynamic dispatch* of methods. For example, the expression `(self` $\langle\rangle$`).move` in the body of method `double` yields a function that is bound to name `move` in the context denoted by the expression `self` $\langle\rangle$.

On the other hand, if the function associated with `self` is applied to a non-empty form, say `State`$\langle$`x = Args.nx`$\rangle$ as in the method `setX`, then the evaluation of the application `self State`$\langle$`x = Args.nx`$\rangle$ yields a new instance, where the instance variable `x` has been set to the value of `Args.nx` in `setX`. This behavior corresponds to *state update* (or assignment).

The incremental modification $\Delta_{\texttt{Point}}$ defined by class `Point` is captured by the function `\(State) PointBehavior`. This function acts in close concert with the generator $G_{\texttt{Point}}$. That is, when creating a new `Point`-object, the generator $G_{\texttt{Point}}$ combines its arguments with the `PointState` template using conditional update. The result, which denotes the image of the `Point`-object currently being created, is then passed to $\Delta_{\texttt{Point}}$, resulting in an extended evaluation context for `PointBehavior`, where the label `State` maps the state of the associated `Point`-instance. Moreover, it important to note that the protocol between $\Delta_{\texttt{Point}}$ and $G_{\texttt{Point}}$ establishes a proper encapsulation of an object's private state, as the visibility of label `State` is restricted to the scope of `PointBehavior`.

The purpose of $W_{\texttt{Point}}$ is to bind the `self`-reference in `Point`-objects. The value of the `self`-reference is actually the fixed-point of the generator $G_{\texttt{Point}}$. That is, the wrapper $W_{\texttt{Point}}$ invokes $G_{\texttt{Point}}$ in a context `[`$\langle$`self=Self`$\rangle$`]` in which the name `Self` is bound to `FIX` $\langle$`f =` $G_{\texttt{Point}}\,\rangle$.

Finally, the expression denoted by `ClassPoint` is the $\lambda\mathcal{F}_{@}$-representation of class `Point`. `ClassPoint` is actually a form defining bindings for two labels W and G, respectively. The binding for label `W`, which maps $W_{\texttt{Point}}$, denotes the *constructor* for class `Point`. Clients use this binding to acquire instances of class `Point`.

The second binding in `ClassPoint`, that is, label `G`, denotes a *class extension access point*, which is the main ingredient in an approach that provides support for open classes. In order to extend class `Point` with new behavior, implementers use this binding to acquire a `Point` prototype instance and place it in new, extended context. Access to the actual class code is not required, as illustrated in Figure 4, which shows the $\lambda\mathcal{F}_{@}$-encoding of the class `BoundedPoint` that is a specialization of class `Point` defined using inheritance.

We use a combination of aggregation and delegation to implement the class `BoundedPoint`. The semantics of inheritance in our encoding of class `BoundedPoint` is defined in the generator $G_{\texttt{BPoint}}$. The construction of `BoundedPoint` prototype instances takes place in four steps. First, we acquire a `Point` prototype instance and assign it to label `Parent`.

```
ClassBoundedPoint =
 let
   BPointBehavior =
     ⟨ getBound = \() State.bound,
       move =
         \(Args)
           if ((self <>).getY <>) + dy < State.bound
           then (State → super).move Args          ⟩

   BPointState = ⟨bound = 0⟩

   Δ_BPoint = \(State) BPointBehavior

   G_BPoint =
     \(State)
       let
         Parent = PointClass.G Args
         newState = (PointState\⟨bound=Args.bound⟩)
         newDelta = Δ_BPoint Args·newState⟨super=Parent⟩
       in
         Parent·newDelta
       end

   W_BPoint = FIX ⟨f = \(Self) (G_BPoint[⟨self=Self⟩])⟩
 in
   ⟨W = W_BPoint, G = G_BPoint⟩
 end
```

Figure 4: Encoding of Class `BoundedPoint`.

`Parent` is a self-contained `Point`-object in which the `self`-reference refers to the `BoundedPoint`-object currently being constructed.

In the second step, we eliminate the default values defined in the `BPointState` template, if $G_{\texttt{BPoint}}$'s constructor arguments contain new definitions for them.

The third step yields `newDelta`, which represents a prototype instance that contains only the behavior defined in class `BoundedPoint`, plus a static, delegation-based link to the `Parent`-object in the scope of `BPointBehavior`.

An actual `BoundedPoint` prototype instance is created in the last step. We use form extension to override the inherited `Point`-behavior with the new `BoundedPoint`-behavior. That is, we extend `Parent` with `newDelta`, which yields a form that contains all bindings of `Parent` and `newDelta`, respectively, except `Parent`'s binding for label `move`.

The remaining definitions for class `BoundedPoint` are similar to the ones used to define class `Point`.

Both `ClassPoint` and `ClassBoundedPoint` represent utilizable solutions in a straightforward, purely functional model for open classes. However, the need to use an explicit fixed-point operator limits somewhat our ability to express more sophisticated collaboration patterns. For example, two methods $\texttt{m}_1$ and $\texttt{m}_2$ cannot collaborate by means of their corresponding object's state if one of them attempts to alter it, because the other method is unable to observe this change. However, if the state of an object is implemented as a shared form, say `@State`, then state changes are visible in all methods belonging to that object.

In an imperative object model both the `self`-reference and the object's state are represented by shared forms. The required modifications for class `Point` and `BoundedPoint` are summarized in Figure 5 and 6, respectively.

```
  PointBehavior =
    ⟨ getX = \() State.x,
      setX = \(Args) State⟨x=Args.nx⟩,
      getY = \() State.y,
      setY = \(Args) State⟨y=Args.ny⟩,
      move = \(Args) State⟨x = State.x + Args.dx,
                           y = State.y + Args.dy⟩,
      double = \() self.move ⟨dx = State.x,
                              dx = State.x ⟩⟩

  G_Point = \(Args)
    let
      State = @(Args·(PointState\⟨x=Args.x,y=Args.y⟩))
    in
      Δ_Point State
    end

  W_Point = \(Args) let
                      self = @⟨⟩
                    in
                      self·(G_Point Args⟨self=self⟩)
                    end
```

Figure 5: Modifications for Class `Point`.

Changing the underlying representation of an object's state does change the way its information can be accessed inside method bodies. However, the specification of state change becomes easier. For example, to assign the instance variable `x` in method `setX` the value of `Args.nx`, we can write simply `State⟨x = Args.nx⟩`. That is, we use that fact the `State` actually denotes a location, which value is extended with the binding `⟨x = Args.nx⟩`. To illustrate this further, assume that the value denoted by `Args.nx` is `3` and $\sigma(\texttt{State}) = \gamma$, then under some evaluation context $\overline{H}$ we have

$$\overline{H} \vdash \texttt{State}\langle \texttt{x} = \texttt{Args.nx} \rangle, \sigma \Rightarrow \gamma, \sigma[\gamma \mapsto \sigma(\gamma)\langle \texttt{x} = \texttt{3} \rangle]$$

Secondly, the value associated with the `self`-reference is now readily accessible and denotes the current object instance. Therefore, to lookup the method `move` in the body of the `double` method, we simply write `self.move`. We can do so, because the wrapper for both class `Point` and `BoundedPoint` creates now a location for `self`. That is, in the beginning each wrapper introduces a new location containing value $\langle\rangle$ and associates it with `self`. We use $\langle\rangle$, because we are only interested in `self`'s location at this point. The actual instantiation of it takes place later, namely in the expression

self·(G Args⟨ self=self ⟩), where G stands for the corresponding class generator.

Applying these modifications to our initial purely functional model we have now obtained an imperative model in which methods can truly perform side-effects on the internal state of objects.

```
BPointBehavior =
  ⟨ getBound = \() State.bound,
    move =
      \(Args)
        if (self.getY <>) + dy < State.bound
        then (State → super).move Args      ⟩

G_BPoint =
  \(State)
    let
      Parent = PointClass.G Args
      newState = (PointState\⟨ bound=Args.bound ⟩)
      State = @(Args·newState⟨super=Parent ⟩)
      newDelta = Δ_BPoint State
    in
      Parent·newDelta
    end

W_BPoint = \(Args) let
                     self = @⟨⟩
                   in
                     self·(G_BPoint Args⟨ self=self ⟩)
  end
```

Figure 6: Modifications for Class BoundedPoint.

7. CONCLUSION AND FUTURE WORK

We have presented the $\lambda\mathcal{F}_{@}$-calculus, a substitution-free variant of the λ-calculus that combines three distinguishing concepts: position-independent parameter specification, shared forms, and explicit contexts. The Integration of these concepts into one unifying framework provides us with a convenient tool to model open classes, a key mechanism that enables class extensions.

We have demonstrated that the $\lambda\mathcal{F}_{@}$-calculus can be used to define straightforward utilizable solutions for both purely functional and imperative models for open classes. We have further used inheritance as an example to illustrate, how extensions to classes can be specified. It is, however, important to note that our approach is not restricted to inheritance. Incremental modifications of classes are defined independently of the mechanisms used to apply them to those classes. Moreover, our underlying conceptual framework is based neither on classes nor objects, but on shared forms and explicit contexts, which are the key abstractions in an approach that provides support for the definition of flexible, extensible, and reusable software abstractions.

We have implemented a proof-of-concept prototype language based on the $\lambda\mathcal{F}_{@}$-calculus in Java. Even though the calculus itself is untyped, we use Java's underlying type system to perform runtime type checks to guarantee that extended classes are properly implemented.

Unfortunately, our current prototype does not allow for the import of existing classes, which is essential if we want to define a $\lambda\mathcal{F}_{@}$-based software development approach for real-world applications. In the future we will, therefore, explore the concept of *peer forms* [3], which are wrapper-like abstractions that mediate between the language representation (e.g., the $\lambda\mathcal{F}_{@}$-calculus) and the host representation of software entities.

A key challenge in future work on the definition of a formal model for open classes based on the $\lambda\mathcal{F}_{@}$-calculus is the formulation of suitable type system that incorporates both form extension and form restriction. The form extension operator is similar to *asymmetric record concatenation* [12, 37, 50]. The specific nature of this operation makes it hard to find a suitable type assignment. Some type systems [12, 50] cannot assign a type to this operator at all. In type systems that incorporate a *subsumption rule* the form extension operator requires an additional set of constraints that limits the number of applicable subtypes. Early results [24, 32] indicate that the definition of a suitable type system will go beyond "traditional" type theories.

8. REFERENCES

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[2] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lèvy. Explicit Substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.

[3] F. Achermann. *Forms, Agents and Channels: Defining Composition Abstraction with Style*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, Jan. 2002.

[4] F. Achermann and O. Nierstrasz. Explicit Namespaces. In J. Gutknecht and W. Weck, editors, *Modular Programming Languages*, LNCS 1897, pages 77–89. Springer, Sept. 2000.

[5] U. Assmann. *Invasive Software Composition*. Springer, 2003.

[6] H. P. Barendregt. *The Lambda Calculus - Its Syntax and Semantics*. Elsevier Science B.V., revised edition, 1985.

[7] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling Visibility of Class Extensions. *Computer Languages, Systems and Structures*, Sept. 2005. to appear.

[8] V. Bono, A. J. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In R. Guerraoui, editor, *Proceedings ECOOP '99*, LNCS 1628, pages 43–66. Springer, June 1999.

[9] 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.

[10] K. B. Bruce. A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics. *Journal of Functional Programming*, 4(2), Apr. 1994.

[11] L. Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, 76:138–164, 1988.

[12] L. Cardelli and J. C. Mitchell. Operations on Records. In C. Gunter and J. C. Mitchell, editors, *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):3–48, March 1991.

[13] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, Oct. 2000.

[14] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. Technical Report 04-01b, Iowa State University, Dept. of Computer Science, Dec. 2004. Accepted for publication, pending revision.

[15] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, 1994.

[16] W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Department of Computer Science, Brown University, Providence, RI, May 1989.

[17] F. Curbera, S. Weerawarana, and M. J. Duftler. On Component Composition Languages. Proceedings of ECOOP 2000 Workshop on Component-Oriented Programming, June 2000.

[18] L. Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. PhD thesis, Centre Universitaire d'Informatique, University of Geneva, CH, 1994.

[19] N. G. de Bruijn. Lambda Calculus Notation with Nameless Dummies. *Indagationes Mathematicae*, 34:381–392, 1972.

[20] F. DeRemer and H. H. Kron. Programming in the Large versus Programming in the Small. *IEEE Transactions on Software Engineering*, SE–2(2):80–86, June 1976.

[21] K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proceedings FCT '95*, LNCS 965, pages 42–61. Springer, 1995.

[22] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, *Proceedings ECOOP 2001*, LNCS 2072, pages 327–353. Springer, June 2001.

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

[25] M. Lumpe. A Lambda Calculus With Forms. In T. Gschwind, U. Aßmann, and O. Nierstrasz, editors, *Proceedings of Fourth International Workshop on Software Composition, Edinburgh, Scotland*, pages 73–88, Apr. 2005.

[26] M. Lumpe, F. Achermann, and O. Nierstrasz. A Formal Language for Composition. In G. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 4, pages 69–90. Cambridge University Press, Mar. 2000.

[27] 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.

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

[29] L. Mikhajlov and E. Sekerinski. A Study of The Fragile Base Class Problem. In E. Jul, editor, *Proceedings ECOOP 1998*, LNCS 1445, pages 355–382, Brussels, Belgium, July 1998. Springer.

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

[31] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, 1992.

[32] 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.

[33] O. Nierstrasz and L. Dami. Component-Oriented Software Technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.

[34] O. Nierstrasz and T. D. Meijler. Requirements for a Composition Language. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pages 147–161. Springer, 1995.

[35] B. C. Pierce and D. N. Turner. Simple Type-Theoretic Foundations for Object-Oriented Programming. *Journal of Functional Programming*, 4(2):207–247, Apr. 1994.

[36] U. S. Reddy. Objects as closures: Abstract semantics of object oriented languages. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 289–297. ACM, July 1988.

[37] D. Rémy. Typing Record Concatenation for Free. Technical Report RR-1739, INRIA Rocquencourt, Aug. 1992.

[38] D. Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[39] J. Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.

[40] N. Schärli, O. Nierstrasz, S. Ducasse, R. Wuyts, and A. Black. Traits: The Formal Model. Technical Report IAM-02-006, Institut für Informatik, University of Berne, Switzerland, Nov. 2002. Also available as Technical Report CSE-02-013, OGI School of Science & Engineering, Beaverton, Oregon, USA.

[41] J.-G. Schneider. *Components, Scripts, and Glue: A conceptual framework for software composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, Oct. 1999.

[42] 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.

[43] G. L. Steele. *Common Lisp the Language*. Digital Press, Thinking Machines, Inc., 2nd edition, 1990.

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

[45] 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.

[46] 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.

[47] G. van Rossum. Python Reference Manual. Technical report, Corporation for National Research Initiatives (CNRI), Oct. 1996.

[48] W3C. *Extensible Markup Language (XML) 1.0 (Third Edition)*. W3C Recommendation, Feb. 2004. http://www.w3.org/TR/REC-xml.

[49] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, 2nd edition, Sept. 1996.

[50] M. Wand. Type Inference for Record Concatenation and Multiple Inheritance. *Information and Computation*, 93:1–15, 1991. Preliminary version appeared in *Proc. 4th IEEE Symposium on Logic in Computer Science* (1989), pp. 92–97.