FOO: A Minimal Modern OO Calculus

Prodromos Gerakios George Fourtounis Yannis Smaragdakis

Department of Informatics
University of Athens, 15784, Greece
{pgerakios,gfour,smaragd}@di.uoa.gr

Abstract

We present the Flyweight Object-Oriented (FOO) calculus for the modeling of object-oriented languages. FOO is a simple, minimal class-based calculus, modeling only essential computational aspects and emphasizing larger-scale features (e.g., inheritance and generics). FOO is motivated by the observation that recent language design work focuses on elements not well-captured either by traditional object calculi or by language-specific modeling efforts, such as Featherweight Java. FOO integrates seamlessly both nominal and structural subtyping ideas, leveraging the latter to eliminate the need for modeling object fields and constructors. Comparing to recent formalization efforts in the literature, FOO is more compact, yet versatile enough to be usable in multiple settings modeling Java, C#, or Scala extensions.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; D.3.3 [*Programming Languages*]: Language Constructs and Features—Inheritance, Polymorphism

General Terms Languages

Keywords object-oriented programming, formal semantics, type system, structural types, nominal types

1. Introduction and Motivation

Modeling programming languages via concise formalisms has a long history. Languages are complex artifacts whose informal specifications often weigh in at many hundreds of pages, rivaling in length and topping in complexity all but few human-produced texts. Yet, a diminutive formalism, often under a couple of pages in length, can capture key insights on the language's design. By establishing properties of the formalism, researchers can reason about the correctness of important language elements. Formal modeling has often helped identify design errors, has been used to settle algorithmic questions (e.g., decidability of core typing), and has aided the community's understanding of language features. Researchers routinely leverage a formalism in order to propose language extensions in their purest form, and to quickly test new ideas and their interaction with a core model.

The literature landscape of language formalisms contains several core object-oriented calculi [1, 4, 6] as well as more targeted language modeling efforts [10, 15, 21]. The core calculi are typically object-based and attempt to capture elements of language design that have gradually become less studied in the past 15 years. Language-specific modeling efforts are often non-minimal and carry significant baggage that slows down further formal development. It is telling, for instance, that the reference model for Scala [21] contains elements such as dependent types, and has an undecidable type system.

In this work, we present the Flyweight OO (FOO) calculus for modeling object-oriented languages. FOO is inspired by Featherweight Java (FJ) [10]: it is also a class-based formalism and emphasizes inheritance. Indeed, the motivation for FOO stems from our own past language modeling efforts [2, 7–9] which were based on FJ. We found that the language features that are most pertinent to our language extensions had little to do with key elements of Featherweight Java, such as casts, fields, or constructors. (The modeling of casts, including the hallmark "stupid cast" problem of FJ, has, to our knowledge, rarely arisen in the literature subsequent to the original Featherweight Java publications.) The same observation holds regarding the work of others. Recent language models in the literature focus on high-level features, and not on the structure of expressions or low-level computation in general. Such high-level features include mixins and traits [3, 17, 18], polymorphism and gradual typing [12, 22]), modules [11, 14], rich type constraints [23], interactions between different kinds of subtyping [16], and domain-specific extensions [5, 19].

Therefore, we believe there is a need for a minimal calculus that abstracts away low-level computation to its essence (much like a foundational calculus) yet fully supports high-level typing elements (e.g., nominal and structural subtyping, classes, generics). The FOO calculus attempts to strike such a balance. FOO models nominal class-based inheritance, as well as anonymous classes. The latter enable emulation of fields and constructors as well as of (breadth) structural subtyping. FOO tries to be language agnostic, however FOO programs directly map to Scala programs (modulo simple, local syntax transformations). Furthermore, FOO has a straightforward runtime semantics and a simple type-system that makes its algorithmic properties (sound and complete subtyping algorithm) elementary, without a need for external assumptions (e.g., hierarchy cycle checking). Informal inspection of the recent literature suggests that FOO could be leveraged for a large number of language modeling efforts that include a formalism, in the context of either Java, C#, or Scala, and would yield consistent conciseness benefits.

In this short paper, we present our language design informally, via examples (Section 2) and detail a formalism that captures the essence of our approach (Section 3).

2. FOO, Informally

Before presenting our formalism, we illustrate its syntactic features in a more palatable form, with the help of some syntax sugar. All examples are valid Scala code, but map straightforwardly to concepts in our calculus.

FOO is a class-based calculus. Type expressions are hybrid, consisting of a nominal part and an anonymous set of method signatures. This is quite similar to a feature of the Scala language [20], allowing on-the-fly extension of an existing named class with an anonymous part.

Example 1 (**On-the-fly and anonymous classes**). An existing Scala class, `Employee`, can be extended with extra functionality by adding method `extra`.

```
(new Employee
 { def extra() = println("add-on") }).extra;
```

In FOO all types are of the above hybrid form. The anonymous part of a type can be empty, resulting in purely nominal typing, while the nominal part of the type can be `Object` (the root of all class hierarchies) allowing subtyping to be determined structurally, by the contents of anonymous method sets.

Example 2 (**Structural types for anonymous classes**). The following example illustrates subtyping based on the structure of anonymous classes.

```
def fun1(e : { def extra() }) = e.extra
...
fun1(new Object { def extra() =
                    println("subtyping") })
```

Indeed, inheritance itself (i.e., the extension of a class by another) can be viewed as merely the naming of a hybrid type, consisting of a nominal part (the named superclass) and the anonymous extension part.

Example 3 (**Inheritance defined in terms of anonymous classes**). We see below the typical syntax for declaring a subclass—the combination of superclass and subclass body can be viewed as just a hybrid type. This is precisely the view that our formalism encodes.

```
class EnhancedEmployee extends
  Employee { def extra() = println("more") }
```

Anonymous classes allow us to simulate several syntactic conveniences without integrating them in the core language. FOO relies on such encodings, omitting explicit support for fields, constructors, or multiple method parameters.

Example 4 (**Encoding of fields**). We see in this example the construction of `Employee` objects with different field values. The simulation of (final) fields with methods is faithful.

```
abstract class Employee {
  def id():  Integer
  def name(): String
  def salary() : Integer
};

def newEmployee ( cid : Integer,
                  cname : String,
                  csalary : Integer ) : Employee
 = new Employee { def id() = cid
                  def name() = cname
                  def salary() = csalary }
```

Similarly to the encoding of fields, we can encode multiple function parameters. This is a conventional encoding, listed here for completeness.

Example 5 (**Encoding of multiple formal parameters**). Let us assume that we want to encode a function adding two integers as follows:

```
class Add {
  def apply(x : Integer, y : Integer) = x + y
}
(new Add).apply(5, 10)
```

The above snippet can be encoded by capturing the first parameter inside an instance of `Add`, using the technique of the previous example, and by invoking method `apply` which now takes only the the second argument.

```
abstract class Add {
  def x (): Integer
  def apply(y : Integer) = x() + y
}
(new Add{ def x() = 5 }).apply(10)
```

3. Language Description

In this section, we introduce the syntax of the FOO calculus and present its formal semantics.

Our formalism captures the salient features of class-based OO languages with nominal and structural subtyping elements, but eliminates unnecessary complexity: we do not model redundant language features such as fields, constructors, multiple parameters as they can be encoded in our calculus.

3.1 Syntax

The syntax of FOO is presented below:

Member type	Ψ	$::=$	$\mathtt{m} : \mathtt{N} \longrightarrow \mathtt{N}$
Hybrid Type	$\mathtt{N}$	$::=$	$\mathtt{C} \,\&\, \overline{\Psi}$
Member	$\mathtt{M}$	$::=$	$\mathtt{m}(\mathtt{x})\ \mathtt{e}$
Program Value	$\mathtt{v}$	$::=$	$\mathtt{new}\ \mathtt{N}\,\{\overline{\mathtt{M}}\} \mid \mathtt{x}$
Expression	$\mathtt{e}$	$::=$	$\mathtt{v} \mid \mathtt{v.m(e)}$
Top-level classes	$\mathtt{P}$	$::=$	$\overline{\mathtt{class}\ \mathtt{C} = \mathtt{N}\,\{\overline{\mathtt{M}}\}}$

We adopt many of the notational conventions of well-known formal calculi such as FJ [10]: $\mathtt{C}$ denotes constant class names, $\mathtt{N}$ denotes object types, $\mathtt{m}$ denotes method names and $\mathtt{x}$ denotes argument names. Classes and methods are explicitly typed in our calculus.

There are two kinds of type annotations: member types Ψ and hybrid types $\mathtt{N}$. A method signature maps method identifier $\mathtt{m}$ to a method type $\mathtt{N} \longrightarrow \mathtt{N}'$, indicating methods taking a single argument of type $\mathtt{N}$ and returning a value of type $\mathtt{N}'$. A class type $\mathtt{N}$ consists of two components expressing both nominal and structural types: the first component is the parent class $\mathtt{C}$, which is extended by the method signatures defined in $\overline{\Psi}$. The two components are separated by the ampersand (&) symbol.

Type annotations and definitions of methods are placed separately: a class is defined by listing its hybrid type (i.e., its superclass, as well as a list $\overline{\Psi}$ of extra method signatures) and then the definitions, $\overline{\mathtt{M}}$, of these methods (i.e., their bodies, without type annotations) are listed. Both type annotations and definitions are associated with a unique method identifier. Thus, annotations and definitions having the same identifier are associated.

Similarly to FJ we use the bar notation to express an ordered sequence of symbols. For sequences, we use $\bullet$ to denote an empty one, $[\sigma]$ for the sequence holding the single element σ, and the comma operator to add a new element to the front of an existing

Figure 1 Formal semantics, dynamic (top) and static (bottom).

$$\frac{\mathtt{m} \in \mathsf{dom}(\overline{\mathtt{M}})}{\mathtt{mbody}(\mathtt{P}, \mathtt{N}\,\{\overline{\mathtt{M}}\}, \mathtt{m}) = \overline{\mathtt{M}}(\mathtt{m})}\ \textit{(M-O)} \qquad \frac{\begin{array}{c}\mathtt{mbody}(\mathtt{P}, (\mathtt{N}\,\{\overline{\mathtt{M}'}\}), \mathtt{m}) = \mathtt{M} \\ \mathtt{m} \notin \mathsf{dom}(\overline{\mathtt{M}}) \quad \mathtt{P}(\mathtt{C}) = \mathtt{N}\,\{\overline{\mathtt{M}'}\}\end{array}}{\mathtt{mbody}(\mathtt{P}, (\mathtt{C}\ \&\ \overline{\Psi})\,\{\overline{\mathtt{M}}\}, \mathtt{m}) = \mathtt{M}}\ \textit{(M-C)}$$

$$\frac{\mathtt{e} \longrightarrow_{\mathtt{P}} \mathtt{e}'}{\mathtt{new}\ \mathtt{N}\,\{\overline{\mathtt{M}}\}\,.\mathtt{m}(\mathtt{e}) \longrightarrow_{\mathtt{P}} \mathtt{new}\ \mathtt{N}\,\{\overline{\mathtt{M}}\}\,.\mathtt{m}(\mathtt{e}')}\ \textit{(R-C)} \qquad \frac{\mathtt{v}' = \mathtt{new}\ldots \quad \mathtt{mbody}(\mathtt{P}, \mathtt{N}\,\{\overline{\mathtt{M}}\}, \mathtt{m}) = \mathtt{m}(\mathtt{x})\ \mathtt{e}}{\mathtt{new}\ \mathtt{N}\,\{\overline{\mathtt{M}}\}\,.\mathtt{m}(\mathtt{v}') \longrightarrow_{\mathtt{P}} \mathtt{e}[(\mathtt{new}\ \mathtt{N}\,\{\overline{\mathtt{M}}\})/\mathtt{this}, \mathtt{v}'/\mathtt{x}]}\ \textit{(R-I)}$$

$$\frac{\mathtt{x} \mapsto \mathtt{N} \in \Gamma \quad \vdash_{\mathtt{H}} \mathtt{N}}{\Gamma \vdash_{\mathtt{H}} \mathtt{x} : \mathtt{N}}\ \textit{(T-V)} \qquad \frac{\mathtt{N} = \mathtt{C}\ \&\ \overline{\Psi} \quad \vdash_{\mathtt{H}} \mathtt{N} \quad (\Gamma \setminus \mathtt{this}), \mathtt{this} \mapsto \mathtt{N} \vdash_{\mathtt{H}} \overline{\Psi}\ \overline{\mathtt{M}}}{\Gamma \vdash_{\mathtt{H}} \mathtt{new}\ \mathtt{N}\,\{\overline{\mathtt{M}}\}\ :\ \mathtt{N}}\ \textit{(T-N)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathtt{H}} \mathtt{v}_1 : \mathtt{N}_1 \quad \Gamma \vdash_{\mathtt{H}} \mathtt{e}_2 : \mathtt{N}_2 \quad \vdash_{\mathtt{H}} \mathtt{N}_2 <: \mathtt{N}_3 \\ \vdash_{\mathtt{H}} \mathtt{N}_1 \Rightarrow \overline{\Psi'}; \ldots \quad \overline{\Psi'}(\mathtt{m}) = \mathtt{N}_3 \longrightarrow \mathtt{N}_4\end{array}}{\Gamma \vdash_{\mathtt{H}} \mathtt{v}_1.\mathtt{m}(\mathtt{e}_2) : \mathtt{N}_4}\ \textit{(T-I)} \qquad \frac{\Gamma, \mathtt{x} \mapsto \mathtt{N} \vdash_{\mathtt{H}} \mathtt{e} : \mathtt{N}'' \quad \vdash_{\mathtt{H}} \mathtt{N}'' <: \mathtt{N}'}{\Gamma \vdash_{\mathtt{H}} \mathtt{m} : \mathtt{N} \longrightarrow \mathtt{N}'\ \mathtt{m}(\mathtt{x})\ \mathtt{e}}\ \textit{(T-M)} \qquad \frac{\vdash_{\mathtt{H}} \overline{\Psi}}{\vdash_{\mathtt{H}} \mathtt{Object}\&\overline{\Psi}}\ \textit{(W-O)}$$

$$\frac{\begin{array}{c}[\mathtt{this} \mapsto \mathtt{C}\ \&\ \bullet] \vdash_{\mathtt{H}} \overline{\Psi}\ \overline{\mathtt{M}} \\ \mathtt{H}(\mathtt{C}) = \mathtt{N} \quad \mathtt{N} = \mathtt{C}'\ \&\ \overline{\Psi} \quad \vdash_{\mathtt{H}} \mathtt{C}\ \&\ \bullet\end{array}}{\vdash_{\mathtt{H}} \mathtt{class}\ \mathtt{C} = \mathtt{N}\{\overline{\mathtt{M}}\}}\ \textit{(T-C)} \qquad \frac{\begin{array}{c}\overline{\Psi'} \subseteq \overline{\Psi} \\ \vdash_{\mathtt{H}} \mathtt{N} \Rightarrow \overline{\Psi}; \overline{\mathtt{N}}, \overline{\mathtt{N}'} \quad \vdash_{\mathtt{H}} \mathtt{N}' \Rightarrow \overline{\Psi'}; \overline{\mathtt{N}'}\end{array}}{\vdash_{\mathtt{H}} \mathtt{N} <: \mathtt{N}'}\ \textit{(S-N)} \qquad \frac{\vdash_{\mathtt{H}} \mathtt{C}\ \&\ \overline{\Psi} \Rightarrow \overline{\Psi'}; \ldots}{\vdash_{\mathtt{H}} \mathtt{C}\ \&\ \overline{\Psi}}\ \textit{(W-C)}$$

$$\frac{\vdash_{\mathtt{H}} \overline{\Psi}}{\vdash_{\mathtt{H}} \mathtt{Object}\&\overline{\Psi} \Rightarrow \overline{\Psi}\ ;\ [\mathtt{Object}\&\overline{\Psi}]}\ \textit{(H-O)} \qquad \frac{\begin{array}{c}\mathtt{H}(\mathtt{C}) = \mathtt{N} \quad \vdash_{\mathtt{H}} \mathtt{N} \Rightarrow \overline{\Psi'}; \overline{\mathtt{N}} \quad \vdash_{\mathtt{H}} \overline{\Psi} \\ \mathtt{for\ all}\ \mathtt{m} \in \mathsf{dom}(\overline{\Psi}) \cap \mathsf{dom}(\overline{\Psi'})\ \ \overline{\Psi}(\mathtt{m}) = \overline{\Psi'}(\mathtt{m})\end{array}}{\vdash_{\mathtt{H}} \mathtt{C}\ \&\ \overline{\Psi} \Rightarrow \overline{\Psi} \cup \overline{\Psi'}\ ;\ \mathtt{C}\ \&\ \overline{\Psi}, \overline{\mathtt{N}}}\ \textit{(H-C)} \qquad \frac{\vdash_{\mathtt{H}} \mathtt{N}, \mathtt{N}'}{\vdash_{\mathtt{H}} \mathtt{m} : \mathtt{N} \longrightarrow \mathtt{N}'}\ \textit{(W-M)}$$

sequence. We also treat ordered sequences as functions when applicable. Therefore, $\overline{\Psi}(\mathtt{m})$ performs a look up on the elements of $\overline{\Psi}$ and returns the method signature associated with m. Similarly, $\mathsf{dom}(\overline{\Psi})$ returns the set of method names in $\overline{\Psi}$. Furthermore, we implicitly overload all predicates applicable to members of a sequence to apply to the sequence itself, in the usual way.

Class members M consist only of method definitions of the form $\mathtt{m}(\mathtt{x})\ \mathtt{e}$, where m is a method name and x is the formal parameter of m, that is bound for the scope of its body e.

A value v can either be a variable x or an object instantiation expression $\mathtt{new}\ \mathtt{N}\,\{\overline{\mathtt{M}}\}$, where N is the class to be instantiated and $\overline{\mathtt{M}}$ denotes the set of additional methods extending N. Expressions can either be values or method invocations of the form $\mathtt{v.m(e)}$, where v is the receiver of method m and e an expression. Without loss of generality, we restrict method receivers to values in order to simplify our formal semantics.

Finally, P is a set of class declarations mapping class names to their definitions. The special class Object is considered to be the root of the nominal class hierarchy and contains no members.

3.2 Operational Semantics

Figure 1 defines the formal semantics of FOO. Our operational semantics is defined in terms of the reduction relation $\longrightarrow_{\mathtt{P}}$, which transforms expressions from e to $\mathtt{e}'$, given the entire program P that is passed as an implicit parameter. The reduction relation is defined by two rules, a congruence rule *R-C* and a reduction rule *R-I*. The former rule is standard and applies the reduction relation recursively until a reducible expression of the form $\mathtt{v.m(v')}$ is reached. The purpose of the latter rule *R-I* is to reduce method invocation expressions to the corresponding method bodies. The first premise of *R-I* requires that the argument $\mathtt{v}'$ passed to method m is an object instantiation expression. The second premise employs function mbody to lookup the definition of method m using the receiver object $\mathtt{N}\,\{\overline{\mathtt{M}}\}$ and the entire program P.

3.3 Static Semantics

The typing rules of FOO are presented in Figure 1. There are two environments used in typing judgments, the typing context Γ that maps method variables to hybrid types and the program type schema H, which maps class names to hybrid types for the given program P. In contrast with the context Γ, the type schema H is constant during type checking and is provided as input. The type checker also assumes that the types residing in H are well-formed, that is, their definitions satisfy the judgment established by rule *T-C*. The type schema H is defined as follows:

$$\mathtt{H} \equiv \{\mathtt{C} : \mathtt{N} \mid \mathtt{class}\ \mathtt{C} = \mathtt{N}\,\{\overline{\mathtt{M}}\}\}$$

The typing relation is defined as $\Gamma \vdash_{\mathtt{H}} \mathtt{e} : \mathtt{N}$. The type schema H is placed as a subscript of the entails symbol to indicate that it is a constant context, as opposed to Γ, which may expand. Given the typing environments, the typing relation assigns a unique type N to expression e.

The static semantics rules can be divided into five groups:

Hierarchy computation. The hierarchy computation relation is $\vdash_{\mathtt{H}}\ \mathtt{N} \Rightarrow \overline{\Psi}; \overline{\mathtt{N}}$ and is realized by rules *H-O* and *H-C*. The first component of the hierarchy (a sequence of method type signatures) computes the intermediate "interface" of a hybrid type. It is crucially used in rule *T-I* to guarantee type-safe method lookup. The second component of the hierarchy records the chain of superclasses (full hybrid types) towards the Object root class. This chain is used (in the subtyping rule, *S-N*) to distinguish between two different hybrid types, even if these happen to have the same method signatures. Given a class type N and an implicit type schema H, the hierarchy rules compute the full set of method signatures in a class hierarchy $\overline{\Psi}$, as well as the hierarchy $\overline{\mathtt{N}}$ itself.

This relation guarantees the following invariants: (a) there exist no cyclic definitions in the class hierarchy (no separate cycle checking is required or implied), (b) the returned $\overline{\Psi}$ contains all methods of the hierarchy, and (c) methods having the same name also have the same type signature. Most importantly, the hierarchy computation rule simplifies subtyping, which involves both structural and nominal types.

Well-formedness. Rules *W-O* and *W-C* are the class type well-formedness rules, which ensure a non-cyclic hierarchy. Rule *W-M* lifts well-formed class types to well-formed method types. We use abbreviations for sequences of elements: for instance, $\vdash_{\mathtt{H}} \overline{\Psi}$ is an abbreviation for $\forall \mathtt{m} : \mathtt{N} \longrightarrow \mathtt{N}' \in \overline{\Psi}.\ \vdash_{\mathtt{H}} \mathtt{m} : \mathtt{N} \longrightarrow \mathtt{N}'$.

Subtyping. The subtyping relation (rule *S-N*) is defined as $\vdash_{\mathtt{H}}$ $\mathtt{N} <: \mathtt{N}'$ and holds when N is a subtype of $\mathtt{N}'$. Our subtyping relation does not permit depth structural subtyping; we only allow width subtyping for structural types.

Expression typing. The expression typing relation is $\Gamma \vdash_{\mathtt{H}} \mathtt{e} : \mathtt{N}$, where Γ is the method variable typing context, H is the type schema. There are three rules for typing expressions: the method invocation rule, *T-I*, the object instantiation rule, *T-N*, and the variable typing rule, *T-V*. Notice that the ability to define new classes within methods has the following consequences: (a) method signatures $\overline{\Psi}$ and method definitions $\overline{\mathtt{M}}$ must be validated at the instantiation point, (b) the outer `this` variable binding is replaced by the new type binding of `this` (i.e., $(\Gamma \setminus \mathtt{this}), \mathtt{this} \mapsto \mathtt{N}$) and (c) methods may capture variables from the enclosing context, since a method can see the formals of all enclosing methods.

Declaration typing. There are two kinds of declarations. Method declarations are of the form Ψ M and are validated by $\Gamma \vdash_{\mathtt{H}} \Psi\ \mathtt{M}$ (rule *T-M*), while class declarations are validated by rule *T-C*.

4. Conclusions

We presented a minimal calculus for class-based OO languages that permits hybrid representation of nominal and structural types. There have been several related calculi either combining structural and nominal systems or providing core formal bases for language design work. However, such calculi either lack some key features or add significant complexity with features such as external dispatch, traits, or dependent types. An interesting quick comparison is with the recent Tinygrace calculus [13], which also aims for minimality while capturing different core features. In contrast to Tinygrace, FOO does not model casts and has classes introduce new types, thus capturing both nominal and structural typing elements. Also FOO is more compact (e.g., 2 reduction rules instead of 6). Based on its feature set, its small size and its minimal set of rules, we believe that FOO can be employed as a basis for the modeling of common language extension efforts with small extra baggage.

We seek input on avenues for improving FOO or validating its suitability for language modeling. In this direction, we are currently completing a Coq proof of soundness for FOO.

Acknowledgments. We thank the anonymous reviewers for their constructive comments. Our work was funded by the Greek Secretariat for Research and Technology under the "MorphPL" Excellence (Aristeia) award.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996.

[2] J. Altidor, C. Reichenbach, and Y. Smaragdakis. Java wildcards meet definition-site variance. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, June 2012.

[3] V. Bono, F. Damiani, and E. Giachino. Separating type, behavior, and state to achieve very fine-grained reuse. In *Electronic proceedings of FTfJP'07 (`http://www.cs.ru.nl/ftfjp`)*, 2007.

[4] V. Bono and K. Fisher. An imperative, first-order calculus with object extension. In E. Jul, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 462–497. Springer Berlin Heidelberg, 1998.

[5] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 831–850, New York, NY, USA, 2012. ACM.

[6] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, Mar. 1994.

[7] P. Gerakios, A. Biboudis, and Y. Smaragdakis. Forsaking inheritance: Supercharged delegation in DelphJ. In *Object Oriented Programming, Systems, Languages & Applications*, OOPSLA '13, pages 233–252, New York, NY, USA, 2013. ACM.

[8] S. S. Huang and Y. Smaragdakis. Morphing: Structurally shaping a class by reflecting on others. *ACM Transactions on Programming Languages and Systems*, 33(2):1–44, Feb. 2011.

[9] S. S. Huang, D. Zook, and Y. Smaragdakis. cJ: Enhancing Java with safe type conditions. In *Aspect-Oriented Software Development conference (AOSD'07)*, pages 185–198, Mar. 2007.

[10] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[11] H. Im, K. Nakata, J. Garrigue, and S. Park. A syntactic type system for recursive modules. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 993–1012, New York, NY, USA, 2011. ACM.

[12] L. Ina and A. Igarashi. Gradual typing for generics. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 609–624, New York, NY, USA, 2011. ACM.

[13] T. Jones and J. Noble. Tinygrace: A simple, safe, and structurally typed language. In *Proceedings of 16th Workshop on Formal Techniques for Java-like Programs*, FTfJP'14, pages 3:1–3:6, New York, NY, USA, 2014. ACM.

[14] C. Kästner, K. Ostermann, and S. Erdweg. A variability-aware module system. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 773–792, New York, NY, USA, 2012. ACM.

[15] A. Kennedy and D. Syme. Transposing F to C#: Expressivity of parametric polymorphism in an object-oriented language. *Concurrency and Computation: Practice and Experience*, 16:707733, June 2004.

[16] D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, ECOOP '08, pages 260–284, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] W. Miao and J. Siek. Pattern-based traits. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1729–1736, New York, NY, USA, 2012. ACM.

[18] W. Miao and J. Siek. Compile-time reflection and metaprogramming for Java. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 27–37, New York, NY, USA, 2014. ACM.

[19] H. Miller, P. Haller, E. Burmako, and M. Odersky. Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In *Object Oriented Programming, Systems, Languages, & Applications*, OOPSLA '13, pages 183–202, New York, NY, USA, 2013. ACM.

[20] M. Odersky. The Scala Language Specification v 2.9, 2014.

[21] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In L. Cardelli, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer Berlin Heidelberg, 2003.

[22] A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Gradual typing for first-class classes. *SIGPLAN Not.*, 47(10):793–810, Oct. 2012.

[23] O. Tardieu, N. Nystrom, I. Peshansky, and V. Saraswat. Constrained kinds. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 811–830, New York, NY, USA, 2012. ACM.