A Nominal Theory of Objects with Dependent Types

Martin Odersky, Vincent Cremet, Christine Röckl, Matthias Zenger

École Polytechnique Fédérale de Lausanne
INR Ecublens, 1015 Lausanne, Switzerland

Abstract. We design and study νObj, a calculus and dependent type system for objects and classes which can have types as members. Type members can be aliases, abstract types, or new types. The type system can model the essential concepts of JAVA's inner classes as well as virtual types and family polymorphism found in BETA or GBETA. It can also model most concepts of SML-style module systems, including sharing constraints and higher-order functors, but excluding applicative functors. The type system can thus be used as a basis for unifying concepts that so far existed in parallel in advanced object systems and in module systems. The paper presents results on confluence of the calculus, soundness of the type system, and undecidability of type checking.

1 Introduction

The development in object and module systems has been largely complementary. Module systems in the style of SML or CAML excel in abstraction; they allow very precise control over visibility of names and types, including the ability to partially abstract over types. Object-oriented languages excel in composition; they offer several composition mechanisms lacking in module systems, including inheritance and unlimited recursion between objects and classes. On the other hand, object-oriented languages usually express abstraction only in a coarse grained way, e.g. through modifiers *private* or *protected* which limit accessibility of a name to some predetermined part of a system. There is usually no analogue to the signatures with abstract types in module systems, which can hide information about a binding outside the unit defining it.

Recently, we see a convergence of the two worlds. Module systems have acquired a form of inheritance through mixin modules [18,2,3,9,7], first-class modules [37] can play a role similar to objects, and recursive modules are also being investigated [16]. On the object side, nested classes with virtual or abstract types [30,39,12] can model the essential properties of signatures with abstract types in ML-like module systems [29]. In principle, this is not a new development. Class nesting has been introduced already in SIMULA 67 [17], whereas virtual or abstract types are present in BETA [31], as well as more recently in GBETA [19], RUNE [42] and SCALA [33]. An essential ingredient of these systems are objects with type members. There is currently much work that explores the uses of this concept in object-oriented programming [38,40,20,36]. But its type theoretic foundations are just beginning to be investigated.

As is the case for modules, dependent types are a promising candidate for a foundation of objects with type members. Dependent products can be used to represent functors in SML module systems as well as classes in object systems with virtual types [26]. But where the details in ML module systems build on a long tradition, the corresponding foundations of object systems with abstract and virtual types have so far been less well developed. One possible approach would be to extend the formalizations of ML module systems to object systems, but their technical complexity makes this a difficult task. An alternative would be to apply the intuitions of dependent types to a smaller calculus of objects and classes, with the aim of arriving on a combined foundation for objects and classes as well as modules. This is what we want to achieve in this paper. Our main contribution is a formal study of a type theory for objects based on dependent types. The theory developed here can be used as a type-theoretic foundation for languages such as BETA, GBETA or SCALA, as well as for many concepts that have so far been presented only in an informal way.

A characteristic of our calculus and type system is that it is *nominal*. Nominality comes into play in two respects. First, objects are given unique names in the reduction system. It is always the name of an object which is passed, instead of a copy of the object itself. A name passing strategy for objects is necessary because our regime of dependent types is based on object identity: If L is a type label then $x.L$ and $y.L$ are the same type only if x and y can be shown to refer to the same object. If objects were copied, type equalities would not be maintained during reduction.

Second, we introduce a nominal binding for types: $L \prec T$ defines L as a name of a new type which unfolds to type T. Two such definitions always define two different types, even if they unfold to the same type. This corresponds closely to the notion of interfaces in a language like JAVA. An interface defines a new type whose structure is completely known. It is possible to define values of an interface type by giving implementations for all members of the interface. In our type system we represent the members of an interface by a record type T. The relationship between the interface name I and its unfolding T is then neither an equality $I = T$ (because then I would not represent a new type), nor is I an abstract type $I <: T$ (because then one could not create new values of I from implementations of type T). Hence, the need of the third type binding $I \prec T$.

A perhaps more standard alternative to our nominal new-type bindings would be *branding*. That is, one would define type equality and subtyping structurally and introduce a binder to create new type names. Branding then means creating a new type by combining a structurally defined type and a freshly created type name. An advantage of the branding approach is that it is orthogonal to traditional structural type systems for objects or modules. A disadvantage is that it corresponds less well to the definitions and implementations of existing object-oriented languages (with the exception of MODULA-3 [13]).

A more technical reason for abandoning the structural types with brands approach has to do with recursion: In a system with dependent types, type recursion can involve terms, which means that recursive types are not necessarily regular trees. For instance if p is a qualified identifier of an object with a term member l and a type member L, then the type $p.L$ might depend on the type $p.l.L$. The resulting tree would then not be regular. There is little hope that practical semi-algorithms for checking equality and subtyping of non-regular trees can be found. To sidestep these problems we follow the strategy of many existing programming languages: we restrict ourselves to non-recursive type aliases, and introduce a new kind of type definition that makes the defined type a subtype of its right-hand side. Note that similar problems for type-checking are caused by parameterized algebraic types where recursive use of a type constructor can also lead to non-regular trees. The common approach to deal with such types is again to make them nominal.

In summary, we design and study in this paper νObj, a core calculus and type system for objects and classes with type members. Type members can be aliases, abstract types, or new types. Classes are first-class and can be composed using mixin-composition. Our type system supports via encodings:

- Most concepts of SML-style module systems, including sharing constraints and higher-order functors, but excluding applicative functors.
- System $F_{<:}$ [14], with the full subtyping rule.
- Virtual types and family polymorphism [20].

Because all these constructs are mapped to the same small language core, it becomes possible to express unified concepts. In particular, our theory promotes the following identifications.

$$\begin{array}{lll} \textit{Object} & = \textit{Module} & \\ \textit{Object type} & = \textit{Signature} & \\ \textit{Class} & = \textit{Method} & = \quad \textit{Functor} \end{array}$$

The same identifications are made in BETA and GBETA, where classes and methods are subsumed under the notion of "patterns". Our own language SCALA follows the same approach, except that it maintains a distinction between methods and classes on the syntactical level. Generally, many of our intuitions are inspired by BETA and by the work of Erik Ernst [19] and Mads Torgerson, which build on it. A contribution of our work is the definition and study of these ideas in a formal calculus and type system. The main technical results of the paper are

- Confluence of the reduction relation.
- Undecidability of type checking by reduction to the problem in $F_{<:}$.
- Type soundness – a well-typed program that does not diverge reduces to an answer of the same type.

Other related work This paper extends a previous workshop contribution [35]. Nominal type systems have also been formalized in the Java context, examples are [21,27,32]. A difference between these approaches and ours is that they rely on a global class graph that describes membership and inheritance. Another difference is that these systems are almost completely nominal, in the sense that most types can be described by a name (exceptions are only array types and generic types in FGJ [27]). By contrast, classes can be local in νObj and nominal types are just one construction in an otherwise structural type system.

There are two other attempts at formalizations of virtual or abstract types in object-oriented programming that we are aware of. The first, by Torgersen [41], sketches a nominal type system for virtual types. It argues informally that if certain restrictions are imposed on the usage of virtual types (which in fact makes them equivalent to abstract types in our terminology), type soundness can be ensured. Igarashi and Pierce [25] proposed a foundation of virtual types using a type system that adds dependent types to an $F_{<:}$ core. However, no formal study of the type system's properties was attempted, and in fact their initial formalization lacked the subject reduction property (that formalization was dropped in the journal version of their paper [26]).

The rest of this paper is structured as follows. Section 2 presents context-free syntax, operational semantics, and type assignment rules of our object calculus, νObj. Section 3 illustrates in a series of examples how the calculus expresses common object-oriented idioms. Section 4 presents the type structure of νObj types, including derivation rules for well-formedness, equality and subtyping. Section 5 presents an encoding of $F_{<:}$ in νObj. Section 6 presents the meta-theory of νObj with results on confluence, soundness and undecidability. Section 7 concludes.

2 The νObj Calculus

We now present a core language for objects and classes. Compared to the standard theory of objects [1], there are three major differences. First, we have classes besides objects as a primitive concept. Classes are even "first-class" in the sense they can result from evaluation of a term and they may be associated with a label. Second, the calculus has a notion of object identity in that every object is referenced by a name and it is that name instead of the object record which is passed around. Third, we can express object types with type components, and some of these components can be nominal.

2.1 Context-Free Syntax

Figure 1 presents the νObj calculus in terms of its abstract syntax, and its structural equivalence and reduction relations. There are three alphabets. Proper term names x, y, z are subject to α-renaming, whereas term labels l, m, n and type labels L, M, N are fixed.

Syntax

x, y, z	Name		
l, m, n	Term label	L, M, N	Type label
s, t, u ::=	Term	S, T, U ::=	Type
x	Variable	$p.\mathbf{type}$	Singleton
$t.l$	Selection	$T \bullet L$	Type selection
$\nu x \leftarrow t \,;\, u$	New object	$\{x \| \overline{D}\}$	Record type (=:: R)
$[x : S \| \overline{d}]$	Class template	$[x : S \| \overline{D}]$	Class type
$t \;\&_S\; u$	Composition	$T \;\&\; U$	Compound type
d ::=	Definition	D ::=	Declaration
$l = t$	Term definition	$l : T$	Term declaration
$L \preceq T$	Type definition	$L \preceq: T$	Type declaration
p ::=	Path	$\preceq:$ ::=	Type binder
$x \mid p.l$		$=$	Type alias
v ::=	Value	$\prec$	New type
$x \mid [x : S \| \overline{d}]$		$<:$	Abstract type
		$\preceq$ ::=	Concrete type binder
		$= \mid \prec$	

Structural Equivalence α-renaming of bound variables x, plus

(extrude) $$e\langle \nu x \leftarrow t \,;\, u\rangle \equiv \nu x \leftarrow t \,;\, e\langle u\rangle \quad \textbf{if } x \notin fn(e),\ bn(e) \cap fn(x, t) = \emptyset$$

Reduction

(select) $$\nu x \leftarrow [x : S|\, \overline{d}, l = v] \,;\, e\langle x.l\rangle \to \nu x \leftarrow [x : S|\, \overline{d}, l = v] \,;\, e\langle v\rangle \quad \textbf{if } bn(e) \cap fn(x, v) = \emptyset$$

(mix) $$[x : S_1|\, \overline{d}_1] \;\&_S\; [x : S_2|\, \overline{d}_2] \to [x : S|\, \overline{d}_1 \uplus \overline{d}_2]$$

where evaluation context

$$e ::= \langle\rangle \mid e.l \mid e \;\&_S\; t \mid t \;\&_S\; e \mid \nu x \leftarrow t \,;\, e \mid \nu x \leftarrow e \,;\, t \mid \nu x \leftarrow [x : S|\, \overline{d}, l = e] \,;\, t$$

Fig. 1. The νObj Calculus

A *term* denotes an object or a class. It can be of the following five forms.

- A *simple name* x, which denotes an object.
- A *selection* $t.l$, which can denote either an object or a class.
- An *object creation* $\nu x \leftarrow t \,;\, u$, which defines a fresh instance x of class t. The scope of this object is the term u.
- A *class template* $[x : S|\, \overline{d}]$ where $\overline{d}$ is a sequence of *definitions* which associate term labels with values and type labels with types. This acts as a template to construct objects with the members defined by the definitions. The name

x of type S stands for "self", i.e. the object being constructed from the template. Its scope is the definition sequence $\overline{d}$. A term or type can refer via $x.l$ to some other member of that object. No textual sequence constraint applies to such references; in particular it is possible that a binding refers to itself or to bindings defined later in the same record. This distinguishes our type system from earlier type systems for records [15] or modules [23].
– A *mixin composition* $t \ \&_S \ u$, which forms a combined class from the two classes to which t and u evaluate. Here, S is the type of "self" in the combined class.

A *value* is a simple name or a class template. A *path* p is a name x followed by a possibly empty sequence of selections, e.g. $x.l_1. \ \ldots \ .l_n$.

The syntax of *types* in our system closely follows the syntax of terms. A type can be of the following five forms.

– A *singleton type* $p.\mathbf{type}$. This type represents the set of values which has as only element the object referenced by the path p. Singleton types are the only way a type can depend on a term in νObj.
– A *type selection* $T \bullet L$, which represents the type component labelled L of type T.
– A *record type* $\{x| \ \overline{D}\}$ where $\overline{D}$ is a sequence of *declarations* which can be value bindings or type bindings. A *value binding* $l : T$ associates a term label l with its type T. *Type bindings* come in three different forms:
First, the binding $L = T$ defines L to be an *alias* for T. Second, the binding $L \prec T$ defines L to be a *new type* which *expands* to type T. That is, L is a subtype of T which has exactly the members defined by T; furthermore, one can create objects of type L from a class which defines all members of T. Third, the binding $L <: T$ defines L to be an *abstract type* which is known to be a subtype of its bound T.
We let the meta-variable $\preceq$ range over $=$ and $\prec$, and let $\preceq:$ range over $=$, $\prec$, and $<:$. The name x stands for "self"; its type is assumed to be the record type itself. We let the letter R range over record types.
– A *compound type* $T \ \& \ U$. This type contains all members of types T and U. The subtyping relation for compound types is the same as the one for intersection types [4], but the formation rules are more restrictive. Where T and U have a member with the same label, the compound type contains the member defined in U. That member definition must be more specific (see Section 4) than the corresponding member definition in T.
– A *class type* $[x : S| \ \overline{D}]$, which contains as values classes that instantiate to objects of type $\{x| \ \overline{D}\}$, or some subtype of it. x is again the name for "self". It now comes with an explicit type S which may be different from $\{x| \ \overline{D}\}$. Definitions in S which are missing from $\overline{D}$ play the role of abstract members. Such members can be referred to from other definitions in the class, but they are not defined in the class itself. Instead, these members must be defined in other classes which are composed with the class itself in a mixin composition.

Definitions which are present in $\overline{D}$ but missing in S play in some sense the role of non-virtual members – they are not referred to via "self" from inside the class, so overriding them does not change existing behavior. Definitions present in both S and $\overline{D}$ play the role of virtual members.

Discussion Most notably missing from the core language are functions, including polymorphic ones, and parameterized types. In fact, type variables are missing completely — the only α-renameable identifiers denote ν-bound terms. However, these omitted constructs can still be expressed in νObj using context-free encodings. This will be shown later in the paper. Section 3 explains how named monomorphic functions are encoded. Section 5 generalizes the encoding to system $F_{<:}$.

The type syntax defines a singleton type $p.\mathbf{type}$ and a selection $T \bullet L$ which operates on types T. More conventional would have been a type selection $p.L$ which operates on terms p instead of types. The latter selection operation can be expressed in our syntax as $p.\mathbf{type}\bullet L$. Besides having some technical advantages, this decomposition can express two concepts which the conventional type selection $p.L$ cannot. First, the self-type of a class can be expressed as a singleton type $\mathit{this}.\mathbf{type}$. This can accurately model covariant self-types. For contravariant self-types one would need a matching operation [11,10] instead of – or in addition to – the subtyping relation that we introduce. Second, an inner class of the kind it exists in Java [22,24] can be referenced by a type selection $\mathit{Outer} \bullet \mathit{Inner}$ where Outer and Inner are types. Such a selection risks being non-sensical in the presence of abstract type members in the outer class Outer. Consequently, our typing rules prevent formation of the type $T \bullet L$ if L's definition depends on some abstract member of T. Note that this is not a problem for Java, which does not have abstract type declarations.

Syntactic Sugar

1. The type $p.L$ is a shorthand for $p.\mathbf{type}\bullet L$.
2. The class type $[x|\ \overline{D}]$ is a shorthand for $[x : \{x|\ \overline{D}\}|\ \overline{D}]$.
3. The class template $[x|\ \overline{d}]$ is a shorthand for $[x : \{x|\ \overline{D}\}|\ \overline{d}]$ where $\overline{D}$ is the most specific set of declarations matching definitions $\overline{d}$.
4. The types $\{\overline{D}\}$, $[\overline{D}]$ and the term $[\overline{d}]$ are shorthands for $\{x|\ \overline{D}\}$, $[x|\ \overline{D}]$ and $[x|\ \overline{d}]$ where x does not appear in $\overline{D}$ or $\overline{d}$.
5. new t is a shorthand for $\nu x \leftarrow t\ ;\ x$.
6. $t_1\ \&\ t_2$ is a shorthand for $t_1\ \&_{S_1\ \&\ (S_2\ \&\ \{x|\ D_1 \uplus D_2\})}\ t_2$ if t_i has least type $[x : S_i|\ \overline{D}_i]$ for $i \in 1..2$.

The last shorthand implements an overriding behavior for mixin composition where a concrete definition always overrides an abstract definition of the same label. Furthermore, between two abstract definitions or between two concrete definitions of the same label it is always the second which overrides the first.

This scheme, which corresponds closely with the rules in Zenger's component calculus [44], is often more useful than the straight "second overrides first" rule of systems where mixins are seen as functions over classes [8,21,5].

2.2 Operational Semantics

Figure 1 specifies a structural equivalence and a small-step reduction relation for our calculus. Both relations are based on the notion of an *evaluation context*, which determines where in a term reduction may take place. The grammar for evaluation contexts given in Figure 1 does not yet yield a deterministic reduction relation, but still leaves a choice of a strict or lazy evaluation strategy, or some hybrid in-between. Particular evaluation strategies are obtained by tightening the grammar for evaluation contexts.

Notation We write $\overline{a}$ for a sequence of entities $a_1, \ldots, a_n$. We implicitly identify all permutations of such a sequence, and take the empty sequence ϵ as a unit for $(,)$. The *domain* $dom(\overline{d})$, $dom(\overline{D})$ of a sequence of definitions $\overline{d}$ or declarations $\overline{D}$ is the set of labels it defines. The restriction $\overline{d}|_{\mathcal{L}}$, $\overline{D}|_{\mathcal{L}}$ of definitions $\overline{d}$ or declarations $\overline{D}$ to a set of labels $\mathcal{L}$ consists of all those bindings in $\overline{d}$ or $\overline{D}$ that define labels in $\mathcal{L}$. The $\uplus$ operator on definitions or declarations denotes concatenation with overwriting of common labels. That is, $\overline{a} \uplus \overline{b} = \overline{a}|_{dom(\overline{a}) \backslash dom(\overline{b})}, \overline{b}$.

A name occurrence x is *bound* in a type T, a term t, a definition d, a declaration D, or an evaluation context e if there is an enclosing object creation $\nu x \leftarrow u\ ;\ t$, a class template $[x : S|\ \overline{d}]$, a class type $[x : S|\ \overline{D}]$, or a record type $\{x|\ \overline{D}\}$ which has the occurrence in the scope of the name x. The free names $fn(X)$ of one of the syntactic classes X enumerated above is the set of names which have unbound occurrences in X. The bound names $bn(e)$ of an evaluation context e are all names x bound by a subterm of e such that the scope of x contains the hole $\langle\rangle$ of the context.

Structural Equivalence As usual we identify terms related by α-renaming. We also postulate a scope extrusion rule (extrude), which allows us to lift a ν-binding out of an evaluation context, provided that this does not cause capture of free variable names. Formally, α-renaming equivalence $\equiv_\alpha$ is the smallest congruence on types and terms satisfying the four laws

$$\begin{array}{rll} \nu x \leftarrow t\ ;\ u \equiv_\alpha & \nu y \leftarrow t\ ;\ [y/x]u & \textbf{if } y \notin fn(u) \\ [x : S|\ \overline{d}] \equiv_\alpha & [y : S|\ [y/x]\overline{d}] & \textbf{if } y \notin fn(\overline{d}) \\ [x : S|\ \overline{D}] \equiv_\alpha & [y : S|\ [y/x]\overline{D}] & \textbf{if } y \notin fn(\overline{D}) \\ \{x|\ \overline{D}\} \equiv_\alpha & \{y|\ [y/x]\overline{D}\} & \textbf{if } y \notin fn(\overline{D}) \end{array}$$

Structural equivalence $\equiv$ is the smallest congruence containing $\equiv_\alpha$ and satisfying the (extrude) law in Figure 1.

$$(\textsc{Var})\quad \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad\qquad \frac{\Gamma \vdash t : T, \quad T \ni (l : U)}{\Gamma \vdash t.l : U}\quad(\textsc{Sel})$$

$$(\textsc{VarPath})\quad \frac{\Gamma \vdash x : R}{\Gamma \vdash x : x.\mathbf{type}} \qquad\qquad \frac{\Gamma \vdash t : p.\mathbf{type}, \quad t.l : R}{\Gamma \vdash t.l : p.l.\mathbf{type}}\quad(\textsc{SelPath})$$

$$(\textsc{Sub})\quad \frac{\Gamma \vdash t : T, \quad T \leq U}{\Gamma \vdash t : U} \qquad\qquad \frac{\begin{array}{c}\Gamma \vdash t : [x : S|\, \overline{D}], \quad S \prec \{x|\, \overline{D}\} \\ \Gamma, x : S \vdash u : U \qquad x \notin fn(U)\end{array}}{\Gamma \vdash (\nu x \leftarrow t \,;\, u) : U}\quad(\textsc{New})$$

$$(\textsc{Class})\quad \frac{\begin{array}{c}\Gamma \vdash S \text{ wf} \qquad \Gamma, x : S \vdash \overline{D} \text{ wf}, \quad t_i : T_i \\ t_i \textit{ contractive in } x \qquad (i \in 1..n)\end{array}}{\Gamma \vdash [x : S|\, \overline{D}, l_i = t_i{}^{i \in 1..n}] : [x : S|\, \overline{D}, l_i : T_i{}^{i \in 1..n}]}$$

$$(\&)\quad \frac{\Gamma \vdash t_i : [x : S_i|\, \overline{D}_i] \qquad \Gamma \vdash S \text{ wf}, \quad S \leq S_i \qquad (i = 1, 2)}{\Gamma \vdash t_1 \,\&_S\, t_2 : [x : S \mid \overline{D}_1 \uplus \overline{D}_2]}$$

Fig. 2. Type assignment

Reduction The reduction relation $\rightarrow$ is the smallest relation that contains the two rules given in Figure 1 and that is closed under structural equivalence and formation of evaluation contexts. That is, if $t \equiv t' \rightarrow u' \equiv u$, then also $e\langle t\rangle \rightarrow e\langle u\rangle$. The first reduction rule, (*select*), connects a definition of an object with a selection on that object. The rule requires that the external object reference and the internal "self" have the same name x (this can always be arranged by α-renaming). The second rule, (mix), constructs a class from two operand classes by mixin composition, combining the definitions of both classes with the $\uplus$ operator. Multi-step reduction $\twoheadrightarrow$ is the smallest transitive relation that includes $\equiv$ and $\rightarrow$.

2.3 Type Assignment

Figure 2 presents the rules for assigning types to terms. These are expressed as deduction rules for type judgments $\Gamma \vdash t : T$. Here, Γ is a type environment, i.e. a set of bindings $x : T$, where all bound names x are assumed to be pairwise different.

There are the usual tautology and subsumption rules. Rule (Sel) assigns to a selection $t.l$ the type U provided t's type has a member $l : U$. Rules (VarPath) and (SelPath) assign singleton types $p.\mathbf{type}$ to terms which denote unique objects.

Rule (New) types a ν-expression $\nu x \leftarrow t \,;\, u$. The term t needs to have a class type $[x : S \mid \overline{D}]$ such that the self type S expands to a record type which contains exactly the declarations $\overline{D}$. This means that all declarations present in

S must be defined in D, with the same type. In particular, classes with abstract members cannot be instantiated. The body u is then typed under an augmented environment which contains the binding $x : S$. The type of u is not allowed to refer to x.

Rule (Class) types class templates. All term definitions $l_i = t_i$ in the template are typed under a new environment which includes a binding $x : S$ for the self-name of the class. However, it is required that all terms t_i are contractive in self. This means that they do not access self during the instantiation of an object of the class. Contractiveness is defined formally as follows.

Definition. The term t is *contractive* in the name x if one of the following holds.

- $x \notin fn(t)$, or
- t is a class template $[y : S|\ \overline{d}]$, or
- t is a mixin composition $t_1 \, \&_S \, t_2$ and t_1, t_2 are contractive in x, or
- t is an object creation $\nu y \leftarrow t_1 \,;\, t_2$, $x \notin fn(t_1)$ and t_2 is contractive in x.

The contractiveness requirement prevents accesses to fields of an object before these fields are defined. In conventional object-oriented languages this would correspond to the requirement that self can be accessed only from methods, not from initializers of object fields. More liberal schemes are possible [6], but require additional technical overhead in the type assignment rules. One can also envisage to allow accesses to self without restrictions, preinitializing fields to some default value, or raising a run-time exception on access before definition.

The last rule, (&) types compositions of class terms. The self type S of the composition is required to be a subtype of the self types of both components. The definitions of the composed class are then obtained by concatenating the definitions of the components.

These deduction rules are based on several other forms of judgments on types, specifically the well-formedness judgment $\Gamma \vdash T\ wf$, the membership judgment $\Gamma \vdash T \ni D$, the expansion judgment $\Gamma \vdash T \prec T'$, and the subtyping judgment $\Gamma \vdash T \leq T'$. Deduction rules for these judgments are motivated in Section 4 and given in full in an accompanying technical report [34].

As usual, we assume that terms can be alpha-renamed in type assignments in order to prevent failed type derivations due to duplicate variables in environments. That is, if $\Gamma \vdash t : T$ and $t \equiv_\alpha t'$ then also $\Gamma \vdash t' : T$.

The type assignment judgment is extended to a judgment relating definitions and declarations as follows.

Definition. A declaration D *matches* a definition d in an environment Γ written $\Gamma \vdash d : D$, if one of the following holds:

$\Gamma \vdash (l = t) : (l : T)$ if $\Gamma \vdash t : T$.
$\Gamma \vdash (L \preceq T) : D$ if $\Gamma \vdash (L \preceq T) \leq D$ (see Section 4.5 for a definition of $\leq$ on declarations).

3 Examples

Before presenting the remaining details of the theory, we demonstrate its usage by means of some examples. Since the νObj calculus is quite different from standard object-oriented notations, we first present each example in the more conventional object-oriented language SCALA [33]. SCALA's object model is a generalization of the object model of JAVA. The extensions most important for the purposes of this paper are abstract types, type aliases, and mixin composition of classes. A subset of SCALA maps easily into νObj, and we will restrict the example code to that subset. Other constructs, such as higher-order functions, generics, or pattern matching can be defined by translation into the subset, and, ultimately, into the object calculus.

3.1 Modules, Classes, and Objects

We start with a class for representing points in a one dimensional space. Class *Point* is defined as a member of the singleton object *pt*. In SCALA, such top-level singleton objects play the role of modules. In addition to the coordinate *x*, class *Point* defines a method *eq* for comparing a point with another point.

```
object pt {
    abstract class Point {
        def x: Int;
        def eq(p: Point): Boolean = (x == p.x);
    }
}
```

In the subset of SCALA used here, classes do not have explicit constructor parameters. Instead, parameters are represented as abstract class members. For creating an object, one has to subclass *Point* and provide concrete implementations for the abstract members. In the following code we do this twice by using a mixin composition of class *Point* with an anonymous class that defines the missing coordinate *x*.

```
val a = new pt.Point with { def x = 0; };
val b = new pt.Point with { def x = 1; };
a.eq(b)
```

We now devise a translation of the previous SCALA code into our calculus. In addition to the syntax defined in Figure 1, we also make use of λ-abstractions and applications. Later in this section we will explain how to encode these constructs in νObj.

$$
\begin{array}{l}
\nu\, pt \leftarrow [pt \mid \\
\quad Point \prec \{x\colon Int,\ eq\colon pt.Point \rightarrow Boolean\}, \\
\quad point = [this\colon pt.Point \mid eq = \lambda\ (p\colon pt.Point)\ p.x == this.x\,] \\
]; \\
\nu\, a \leftarrow pt.point\ \&_{pt.Point}\ [x = 0]; \\
\nu\, b \leftarrow pt.point\ \&_{pt.Point}\ [x = 1]; \\
a.eq(b)
\end{array}
$$

A class is represented by two entities: an object type that is used to type instances of the class and a class value, which is used to construct objects. We use the name of the class as the name of the type and the same name, but starting with a lower-case letter, as the name of the class value. While the type includes the signatures of all class members, the class value only provides implementations for the non-abstract members. In general, abstract members are present in the self-type S of a class $[x : S|\, \overline{d}]$, but are missing from the class definitions $\overline{d}$. Non-abstract members are present in both S and $\overline{d}$.

3.2 Functions

For encoding λ-abstractions and applications we use a technique similar to the one for passing parameters during class instantiations. A λ-abstraction $\lambda(x : T)\ t$ is represented as a class with an abstract member *arg* for the function argument and a concrete member *fun* which refers to the expression for computing the function's result:

$$[x\colon \{arg\colon T\} \mid fun = [res = t']]$$

where t' corresponds to term t in which all occurrences of x get replaced by $x.arg$. As explained in Section 2, we cannot access *arg* directly on the right-hand-side of *fun*. Therefore *fun* packs the body of the function into another class. The instantiation of this class will then trigger the execution of the function body. For instance, function $\lambda\ (p\colon pt.Point)\ p.x == this.x$ could be encoded as a class $[p\colon \{arg\colon pt.Point\} \mid fun = [res = p.arg.x == this.x]]$ of type $[p\colon \{arg\colon pt.Point\} \mid fun\colon [res\colon Boolean]]$ that contains an abstract member *arg* and a concrete member *fun*.

In νObj, an application $g(e)$ gets decomposed into three subsequent steps:

$$\nu\, g_{app} \leftarrow g\ \&\ [\mathrm{arg} = \mathrm{e}];$$
$$\nu\, g_{eval} \leftarrow g_{app}.fun;$$
$$g_{eval}.res$$

First we instantiate function g with a concrete argument yielding a thunk g_{app}. Then we evaluate this thunk by creating an instance g_{eval} of it. Finally we extract the result by querying field *res* of g_{eval}. For instance, the call to function *eq* from the previous code could be encoded as $\nu\, g_{app} \leftarrow \mathrm{a.eq}\ \&\ [\mathrm{arg} = \mathrm{b}]; \nu\, g_{eval} \leftarrow g_{app}.fun;\ g_{eval}.res.$

3.3 Abstract Types

Suppose we would now like to extend the *Point* class for defining a new class *ColorPoint* that includes color information. Since extended classes define subtypes in Scala, we cannot override method *eq* contravariantly such that the parameter of *eq* now has type *ColorPoint*. But exactly this would allow us to compare *ColorPoint*s only with *ColorPoint*s. Instead, we have to refactor our code and abstract over the parameter type explicitly in anticipation of future extensions. The following code fragment defines an abstract type *This* in class *Point* with bound *Point* which gets covariantly refined in subclasses like *ColorPoint*.

```
object pt {
    abstract class Point {
        type This <: Point;
        def x: Int;
        def eq(p: This): Boolean = (x == p.x);
    }
}
object cpt {
    abstract class ColorPoint extends pt.Point {
        type This <: ColorPoint;
        def col: String;
        override def eq(p: This): Boolean = (x == p.x) && (col == p.col);
    }
}
```

We now make use of the two classes and define a *Point* and two *ColorPoint* instances.

```
val c = new pt.Point with
            {type This = pt.Point; def x=0;};
val d = new cpt.ColorPoint with
            {type This = cpt.ColorPoint; def x=1; def col="blue";};
val e = new cpt.ColorPoint with
            {type This = cpt.ColorPoint; def x=2; def col="green";};
```

The type system has to ensure that we are able to compare only compatible objects; i.e. we have to be able to execute $d.eq(e)$ and $e.eq(d)$ as well as $c.eq(d)$ and $c.eq(e)$, whereas terms like $d.eq(c)$ are ill-typed and therefore rejected by the typechecker.

An encoding of the previous two classes in our object calculus is given by the following term.

```
ν pt ← [pt |
    Point ≺ {this | This <: pt.Point, x: Int, eq: this.This → Boolean},
    point = [this: pt.Point | eq = λ (p: this.This) p.x == this.x]
];
ν cpt ← [cpt |
    ColorPoint ≺ pt.Point & {This <: cpt.ColorPoint, col: String},
    colorPoint = [this: cpt.ColorPoint |
        eq = λ (p: this.This) p.x == this.x && p.col == this.col]
];
ν c ← pt.point & [This = pt.Point, x = 0];
ν d ← cpt.colorPoint & [This = cpt.ColorPoint, x = 1, col = "blue"];
c.eq(d)
```

This example does not only explain how to use abstract types, it also shows that our calculus is expressive enough to model virtual types in a type-safe way.

3.4 Generic Types

We now present a more evolved example that shows how to use νObj to encode generic classes. The following code defines a "module" *lst* which contains an implementation for generic lists consisting of three classes *List*, *Nil*, and *Cons*.

```
object lst {
    abstract class List {
        type T <: scala.Object;
        def isEmpty : Boolean;
        def head : T;
        def tail : List with {type T = List.this.T;};
    }
    abstract class Nil extends List {
        def isEmpty = true;
        def head : T = error;
        def tail : List with {type T = Nil.this.T;} = error;
    }
    abstract class Cons extends List {
        def isEmpty = false;
    }
}
```

Since classes are neither parameterized by values nor types, we model the element type of a list with an abstract type T in class *List*. Similarly, class parameters like the head and the tail of a cons-cell are represented by abstract functions. Note that the type of the *tail* value of a list object is a mixin composition of *List* with a record type which consists of the type binding $\{$**type** $T = List.$**this**$.T\}$. This forces the element type of a list and its tail to be the same. [1] In general, mixin composition with type bindings subsumes in expressive power the sharing constraints of SML module systems [28].

Class *Nil* provides all the abstract functions of its superclass *List*. For the implementation of *head* and *tail* we make use of a predefined value *error* that produces errors at run-time when accessed. *error* is of any type. Even though our formal treatment does not include such a bottom type, adding one would be straightforward.

Class *Cons* only defines function *isEmpty*. The other abstract functions constitute constructor parameters and have to be provided at instantiation time.

Here is an example how the list abstraction is applied. The following code fragment constructs two lists of integers [ ] and [*1*] and returns the *head* of the second list. Again, we use a mixin class composition to emulate parameter passing.

```
val x0 = new lst.Nil with {type T = Int;};
val x1 = new lst.Cons with {type T = Int; def head = 1; def tail = x0;};
x1.head
```

[1] Like in JAVA, *Outer.***this** denotes the identity of an enclosing *Outer* object in the scope of an inner class of *Outer*.

Here is the translation of the previous SCALA code into our object calculus.

```
ν lst ← [lst |
    List ≺ {this |
        T <: {}, isEmpty: Boolean, head: this.T, tail: lst.List & {T = this.T}},
    Nil ≺ lst.List,
    Cons ≺ lst.List,
    nil = [this: lst.Nil | isEmpty = true, head = error, tail = error],
    cons = [this: lst.Cons | isEmpty = false]
];
ν x0 ← lst.nil & [T = Int];
ν x1 ← lst.cons & [T = Int, head = 1, tail = x0];
x1.head
```

We now augment class *List* of the previous example with a function *len* that computes the length of the list. In SCALA, this can be done without changing the source code of *List*, by using a class as a mixin:

```
object llst {
    abstract class ListWithLen extends lst.List {
        def tail: ListWithLen with { type T = ListWithLen.this.T; };
        def len(): Int = if (this.isEmpty) 0 else 1 + this.tail.len();
    }
}
```

Class *ListWithLen* extends class *List*. It adds a new *len* member and narrows the type of the existing *tail* member to *ListWithLen*. To build lists with *len* members, we add this class as a mixin. Here is an example usage:

```
val y0 = new lst.Nil with {
    type T = Int;
    def tail: ListWithLen with {type T = Int;} = error;
} with llst.ListWithLen;
val y1 = new lst.Cons with {
    type T = Int;
    def head = 1;
    def tail = y0;
} with llst.ListWithLen;
y1.len()
```

The translation of this program into νObj is given in the following code fragment. Please note that this time, we encode function *len* directly as a class, similar to the description given before. This time we can use a slightly simpler encoding since our function is not parameterized.

```
ν llst ← [llst |
    ListWithLen ≺ lst.List & {this |
        tail: llst.ListWithLen & {T = this.T},
        len: [res: Int ]
    },
    listWithLen = [this: llst.ListWithLen |
        len = [res = if (this.isEmpty) 0 else 1 + (ν t ← this.tail.len; t.res)]]
];
```

$\nu\, y0 \leftarrow lst.nil\ \&\ [T = Int]\ \&\ llst.listWithLen;$
$\nu\, y1 \leftarrow lst.cons\ \&\ [T = Int,\ head = 1,\ tail = y0]\ \&\ llst.listWithLen;$
$\nu\, l \leftarrow y1.len;$
$l.res$

Note that type *ListWithLen* is represented as a composition of type *List* and a record type containing added and overridden members. This turns type *ListWithLen* into a subtype of type *List*.

4 Type Structure

The type structure of νObj is defined by deduction rules for the following kinds of judgments:

$\Gamma \vdash T\ wf$	Type T is well-formed.
$\Gamma \vdash D\ wf$	Declaration D is well-formed.
$\Gamma \vdash T \ni D$	Type T contains declaration D.
$\Gamma \vdash T = U$	Types T and U are equal.
$\Gamma \vdash T \prec U$	Type T expands to type U.
$\Gamma \vdash T <: U$	Type T is upper-bounded by type U.
$\Gamma \vdash T \leq U$	Type T is a subtype of type U.
$\Gamma \vdash \overline{D}_1 \leq \overline{D}_2$	Declarations D_1 are more specific than declarations D_2.

Compared to standard type systems there are three non-standard forms of judgments: First, the membership judgment $\Gamma \vdash T \ni D$ factors out the essence of path-dependent types. Second, the expansion judgment $\Gamma \vdash T \prec U$ captures the essential relation between a new type and its unfolding. Third, the upper-binding judgment $\Gamma \vdash T <: U$ provides exact type information about which record type is a supertype of a given type. This information is needed for the correct treatment of type bindings in records. The essential typing rules for all these judgments are discussed in the following.

Notation We sometimes write judgments with several predicates on the right of the turnstile as an abbreviation for multiple judgments. E.g. " $\Gamma \vdash T\ wf,\ T'\ wf$ " is an abbreviation for the two judgments " $\Gamma \vdash T\ wf$ " and " $\Gamma \vdash T'\ wf$ ".

4.1 Membership

The membership judgment $\Gamma \vdash T \ni D$ states that type T has a member definition D. The judgment is derived by the following two rules, which capture the principles of path-dependent types.

$$(\textsc{Single-}\ni)\quad \frac{\Gamma \vdash p.\mathbf{type} <: \{x|\ \overline{D'}, D\}}{\Gamma \vdash p.\mathbf{type} \ni [p/x]D}$$

$$(\textsc{Other-}\ni)\quad \frac{\Gamma, x : T \ \vdash\ x.\mathbf{type} \ni D \qquad x \notin fn(\Gamma, D)}{\Gamma \ \vdash\ T \ni D}$$

Rule (SINGLE-$\ni$) defines membership for singleton types. In this case, the self-reference x in the definition is replaced by the path p. Rule (OTHER-$\ni$) defines membership for arbitrary types in terms of (SINGLE-$\ni$). To determine a member D of a type T which is not a singleton, invent a fresh variable x of type T and determine the corresponding member of type $x.\mathbf{type}$. The resulting member is not allowed to depend on x. Note that, if T is a singleton type, rule (OTHER-$\ni$) either fails or yields the same judgments as rule (SINGLE-$\ni$).

Example Consider the type $T \prec \{x : T \mid L <: \{\}, l_1 : x.L, l_2 : \mathit{Int}\}$. Further consider a path p and some other term t which is not a path, both of type T. Then p contains the definitions $L <: \{\}$, $l_1 : p.L$, and $l_2 : \mathit{Int}$. On the other hand, t contains only the definitions $L <: \{\}$ and $l_2 : \mathit{Int}$ since rule (OTHER-$\ni$) does not derive a binding for l_1. Indeed, substituting t for the self reference x in the binding for l_1 would yield the type $t.L$ which would not be well-formed.

4.2 Equality

The type equality judgment $\Gamma \ \vdash\ T = T'$ states that the two types T and T' are the same or aliases of each other. Type equality is the smallest congruence which is closed under the following two derivation rules.

$$(\textsc{Alias-}=)\quad \frac{\Gamma \ \vdash\ T \ni (L = U),\ \ T\ wf}{\Gamma \ \vdash\ T\bullet L = U} \qquad \frac{\Gamma \ \vdash\ p : q.\mathbf{type}}{\Gamma \ \vdash\ p.\mathbf{type} = q.\mathbf{type}}\quad (\textsc{Single-}=)$$

Rule (ALIAS-$=$) is standard; it states that type $T\bullet L$ is equal to U, provided T has an alias member definition $L = U$. Rule (SINGLE-$=$) expresses the following property: if a path p has a singleton type $q.\mathbf{type}$, we know that p and q are aliases, hence the singleton types $p.\mathbf{type}$ and $q.\mathbf{type}$ should be equal. Without the rule, one would only have that $p.\mathbf{type}$ is a subtype of $q.\mathbf{type}$.

4.3 Expansion

The type expansion judgment $\Gamma \ \vdash\ T \prec T'$ states that type T expands (or: unfolds) into type T'. Expansion is the smallest transitive relation which contains type equality and is closed under the following three derivation rules.

$$(\textsc{Tsel-}\prec)\quad \frac{\Gamma \ \vdash\ T \ni (L \prec U)}{\Gamma \ \vdash\ T\bullet L \prec U} \qquad \frac{\Gamma \ \vdash\ T \prec T',\ \ U \prec U'}{\Gamma \ \vdash\ T \,\&\, U \prec T' \,\&\, U'}\quad (\&\text{-}\prec)$$

$$(\textsc{Mixin-}\prec)\quad \frac{\Gamma, x : \{x \mid \overline{D}_1 \uplus \overline{D}_2\} \ \vdash\ \overline{D}_2 \leq \overline{D}_1|_{dom(\overline{D}_2)}}{\Gamma \ \vdash\ \{x \mid \overline{D}_1\} \,\&\, \{x \mid \overline{D}_2\} \prec \{x \mid \overline{D}_1 \uplus \overline{D}_2\}}$$

Rule (TSEL-$\prec$) expresses expansion of type selections in the usual way. Rule (MIXIN-$\prec$) states that the combination of two record types R_1 and R_2 expands to a record type containing the concatenation of the definitions in R_1 and R_2. If some label is defined in both R_1 and R_2, the definition in R_2 overrides the definition in R_1. In this case we must have that the definition in R_2 is more specific than the definition in R_1.

4.4 Upper Bounds

The upper bound judgment $\Gamma \vdash T <: T'$ states that T' is an expansion of T or a (tight) upper bound of it. The primary use of this relation is in determining for a type T the least record type which is a supertype of T. This information is needed for deriving the membership judgment by rule (SINGLE-$\in$).

Upper-binding is the smallest transitive relation which contains expansion and which is closed under the following three derivation rules.

$$(\textsc{Tsel-}<:) \quad \frac{\Gamma \vdash T \ni (L <: U)}{\Gamma \vdash T \bullet L <: U} \qquad\qquad \frac{x : T \in \Gamma}{\Gamma \vdash x.\mathbf{type} <: T} \quad (\textsc{Var-}<:)$$

$$(\textsc{Sel-}<:) \quad \frac{\Gamma \vdash p.\mathbf{type} \ni (l : U)}{\Gamma \vdash p.l.\mathbf{type} <: U}$$

The first rule (TSEL-<:) defines upper bounds of abstract types in the usual way. The other two rules take as the upper bound of a singleton type $p.\mathbf{type}$ the type which p has in the current environment. Note that we could not have replaced these two rules by a simpler rule which states that $\Gamma \vdash p.\mathbf{type} <: T$, provided $\Gamma \vdash p : T$. The reason is that the subsumption for type assignments would allow one to forget information about a path's type. Hence, one could not guarantee with the simpler rule that upper bounds are tight.

4.5 Subtyping

The subtyping judgment $\Gamma \vdash T \leq T'$ states that T is a subtype of T'. Subtyping is the smallest transitive relation that contains upper-binding (<:) and that is closed under the following four rules.

$$(\&\text{-}\leq) \quad \begin{array}{l}\Gamma \vdash T_1 \;\&\; T_2 \leq T_1 \\ \Gamma \vdash T_1 \;\&\; T_2 \leq T_2\end{array} \qquad\qquad \frac{\Gamma \vdash T \leq T_1, \;\; T \leq T_2}{\Gamma \vdash T \leq T_1 \;\&\; T_2} \quad (\leq\text{-}\&)$$

$$(\textsc{Rec-}\leq) \quad \frac{\Gamma, x : \{x | \; \overline{D}, \overline{D'}\} \vdash \overline{D} \leq \overline{D''}}{\Gamma \vdash \{x | \; \overline{D}, \overline{D'}\} \leq \{x | \; \overline{D''}\}}$$

$$(\textsc{Class-}\leq) \quad \frac{\Gamma \vdash R \text{ wf}, \;\; S \;\&\; R \leq S', \;\; S' \leq S \qquad \Gamma, x : S' \vdash \overline{D} \leq \overline{D'}}{\Gamma \vdash [x : S | \; \overline{D}] \leq [x : S' | \; \overline{D'}]}$$

Rules (&-$\leq$) and ($\leq$-&) state that & behaves like type intersection in subtyping: That is, the type T_1 & T_2 is a subtype of both T_1 and T_2 and to show that a type U is a subtype of T_1 & T_2 one needs to show that U is a subtype of both T_1 and T_2.

The remaining two rules (Rec-$\leq$) and (Class-$\leq$) determine subtyping for record and class types. For record types, subtyping is covariant in the declarations $\overline{D}$, and declarations in the subtype may be dropped in the supertype. For class types, subtyping is contravariant in the self-type S and covariant in the declarations $\overline{D}$. However, both premises are restricted for type checking reasons.

First, unlike for record types, a class type always declares the same labels as its supertypes, so declared labels may not be forgotten. This ensures that the type of labels in a composition is fully determined. For instance, in $[l = 1]\ \&_{\{\}}\ [l = \text{“}abc''\text{]}$ the label l is always known to be bound to a string, not an integer. If labels could be forgotten, the second operand of the composition could be widened via subsumption to the empty class, which would assign l the integer in an alternative typing derivation of the composite class term.

Second, contravariance of self types is limited so that the smaller self type S' must result from the larger self type S composed with some record type. On the other hand, it is not allowed to take as S' some nominal subtype of S. This restriction is necessary to ensure that there is always a least type that can be assigned to instances created from a class in a ν-expression.

The ($\leq$) relation is also defined between declarations. $D \leq D'$ means that declaration D is *more specific* than declaration D'. This predicate is expressed by the following two derivation rules.

$$(\textsc{Bind-}\leq)\quad \frac{\Gamma \vdash T \leq T'}{\Gamma \vdash (l : T) \leq (l : T')} \qquad \frac{\Gamma \vdash T \leq T'}{\Gamma \vdash (L \preceq: T) \leq (L <: T')}\quad (\textsc{Tbind-}\leq)$$

Subtyping on value declarations is defined as usual. For type labels one has that an arbitrary type declaration $L \preceq: T$ is more specific than an abstract type declaration $L <: T$, provided $T \leq T'$. Hence, abstract types can be overridden with other abstract or concrete types as long as the overriding type conforms to the abstract type's bound. Aliases and new types, on the other hand, cannot be overridden.

4.6 Well-formedness

The well-formedness judgment is of the form $\Gamma \vdash T\ \mathit{wf}$. Roughly, a type is well-formed if it refers only to names and labels which are defined and if it does not contain any illegal cyclic dependencies. These requirements are formalized in the four rules given below. The remaining rules propagate these requirements over all forms of types; they are given in full in the accompanying technical report [34].

$$(\textsc{Single-wf})\frac{\Gamma \vdash p : R}{\Gamma \vdash p.\mathbf{type}\ \mathit{wf}} \qquad \frac{\Gamma \vdash T\ \mathit{wf},\ \ T \ni (L = U),\ \ U\ \mathit{wf}}{\Gamma \vdash T\bullet L\ \mathit{wf}}\ (\textsc{Tsel-wf}_1)$$

$$(\textsc{Tsel-wf}_2)\ \frac{\Gamma \vdash T\ \mathit{wf},\ \ T \ni (L \prec U),\ \ U \prec R}{\Gamma \vdash T\bullet L\ \mathit{wf}}$$

$$(\textsc{Tsel-wf}_3)\ \frac{\Gamma \vdash T \ni (L <: U),\ \ U <: R}{\Gamma \vdash T\bullet L\ \mathit{wf}}$$

Rule (Single-wf) states that $p.\mathbf{type}$ is well-formed if p is a path referring to some object. The next three rules cover well-formedness of a type selection $T\bullet L$. They distinguish between the form of definition of L in T.

If L is defined to be an alias of some type U, $T \bullet L$ is well-formed only if U is well-formed. This requirement excludes recursive types, where a type label is defined to be an alias of some type containing itself. Such a recursive type would not have a finite proof tree for well-formedness. On the other hand, if L is defined to be a new type which expands to some type U, one requires only that U in turn expands to some record type. This requirement excludes cyclic definitions such as $\{x|\ L \prec x.L\ \&\ R\}$. But recursive references to the label from inside a record or class are allowed; e.g. $\{x|\ L \prec \{next : x.L\}\}$. Finally, if L is defined to be an abstract type bounded by U, one requires that U in turn is bounded by a record type. This requirement excludes situations where a type is bounded directly or indirectly by itself, such as in $\{x|\ L_1 <: x.L_2, L_2 <: x.L_1\}$. But it admits F-bounded polymorphism, where the abstract type appears inside its bound, as in $\{x|\ L <: \{next : x.L\}\}$.

5 Relationship with $F_{<:}$

System $F_{<:}$ can be encoded in νObj by the translation $\langle\!\langle\cdot\rangle\!\rangle$, which is defined on types, terms, and environments. The translation of $F_{<:}$ types into νObj types is defined as follows.

$$\begin{array}{lll}
\langle\!\langle \forall X <: S.T \rangle\!\rangle & = & \{\mathit{val} : [\ X : \{\mathit{Arg} <: \langle\!\langle S\rangle\!\rangle\}|\ \mathit{fun} : [\mathit{res} : \langle\!\langle T\rangle\!\rangle]\ ]\} \\
\langle\!\langle T \to U \rangle\!\rangle & = & \{\mathit{val} : [\ x : \{\mathit{arg} : \langle\!\langle T\rangle\!\rangle\}|\ \mathit{fun} : [\mathit{res} : \langle\!\langle U\rangle\!\rangle]\ ]\} \qquad (x \text{ fresh}) \\
\langle\!\langle X \rangle\!\rangle & = & X.\mathit{Arg} \\
\langle\!\langle \top \rangle\!\rangle & = & \{\,\}
\end{array}$$

The translation of $F_{<:}$ terms into νObj terms is defined as follows.

$$\begin{array}{lll}
\langle\!\langle \lambda x : T.t \rangle\!\rangle & = & \text{new}\ [\ \mathit{val} = [\ x : \{\mathit{arg} : \langle\!\langle T\rangle\!\rangle\}|\ \mathit{fun} = [\mathit{res} = \langle\!\langle t\rangle\!\rangle]\ ]\ ] \\
\langle\!\langle t\ u \rangle\!\rangle & = & \nu\, x \leftarrow \langle\!\langle t\rangle\!\rangle.\mathit{val}\ \&\ [\mathit{arg} = \langle\!\langle u\rangle\!\rangle]\ ;\ \nu\, y \leftarrow x.\mathit{fun}\ ;\ y.\mathit{res} \\
\langle\!\langle \Lambda X <: S.t \rangle\!\rangle & = & \text{new}\ [\ \mathit{val} = [\ X : \{\mathit{Arg} <: \langle\!\langle S\rangle\!\rangle\}|\ \mathit{fun} = [\mathit{res} = \langle\!\langle t\rangle\!\rangle]\ ]\ ] \\
\langle\!\langle t[T] \rangle\!\rangle & = & \nu\, x \leftarrow \langle\!\langle t\rangle\!\rangle.\mathit{val}\ \&\ [\mathit{Arg} = \langle\!\langle T\rangle\!\rangle]\ ;\ \nu\, y \leftarrow x.\mathit{fun}\ ;\ y.\mathit{res} \\
\langle\!\langle x \rangle\!\rangle & = & x.\mathit{arg}
\end{array}$$

Finally, here is the translation of $F_{<:}$ environments into νObj environments.

$$\begin{array}{lcl}
\langle\!\langle x : T\rangle\!\rangle & = & x : \{\mathrm{arg} : \langle\!\langle T\rangle\!\rangle\} \\
\langle\!\langle X <: T\rangle\!\rangle & = & X : \{Arg <: \langle\!\langle T\rangle\!\rangle\} \\
\langle\!\langle \epsilon\rangle\!\rangle & = & \epsilon \\
\langle\!\langle \Gamma, \Sigma\rangle\!\rangle & = & \langle\!\langle \Gamma\rangle\!\rangle, \langle\!\langle \Sigma\rangle\!\rangle
\end{array}$$

In the translation, we use letters x and X for names, words consisting of lower-case letters for value labels, and words consisting of upper-case letters for type labels. Specifically, *arg* labels a value parameter, *Arg* labels a type parameter, *res* labels a function result, and *val* labels a class value.

Given this translation, here is how $F_{<:}$'s polymorphic identity function $\Lambda X <: \top.\lambda x : X.x$ is expressed in our calculus.

$$\begin{array}{l}
new\ [val = [X\colon \{Arg <: \{\}\} \mid \\
\quad fun = [res = new\ [val = [x\colon \{arg\colon X.Arg\}\ \mid fun = [res = x.arg]]]]]]
\end{array}$$

To give some sense to our encoding we can easily show the following properties.

Lemma 1 *For any environment Γ, types T and U, term t in $F_{<:}$:*

1. $\Gamma \vdash_{F_{<:}} T <: U$ *implies* $\langle\!\langle \Gamma\rangle\!\rangle \vdash \langle\!\langle T\rangle\!\rangle \leq \langle\!\langle U\rangle\!\rangle$.
2. $\Gamma \vdash_{F_{<:}} t : T$ *implies* $\langle\!\langle \Gamma\rangle\!\rangle \vdash \langle\!\langle t\rangle\!\rangle : \langle\!\langle T\rangle\!\rangle$.

Lemma 2 $\vdash_{F_{<:}} t : T$ *and* $t \rightarrow u$ *implies* $\langle\!\langle t\rangle\!\rangle \rightarrow^{+} e_G\langle\langle\!\langle u\rangle\!\rangle\rangle$*, where* e_G *is a "garbage context" of the form* $\nu x_1 \leftarrow u_1\ ;\ \ldots\ ;\nu x_n \leftarrow u_n\ ;\langle\rangle$ *such that no name* x_i *is free in* $\langle\!\langle u\rangle\!\rangle$.

The introduction of the garbage context e_G in the previous lemma is necessary because translation of λ-abstraction and λ-application involves the creation of objects, which are persistent, contrary to the λs that disappear during the lambda reduction rule.

Lemma 3 $\langle\!\langle t\rangle\!\rangle \rightarrow$ *implies* $t \rightarrow$.

The reduction relation $\rightarrow$ that we use for $F_{<:}$ in 3 is the call-by-value small-step semantics, i.e. we never reduce under the λs and an argument has to be reduced to a value before being passed to a function. Together with the previous lemma, this lemma has as corollary that if a well-typed term reduces to an irreducible term then its translation reduces to the translation of this term, which is also irreducible.

6 Meta-Theory

In this chapter, we establish three results for νObj. First, that the reduction relation is confluent. Second, that the typing rules are sound with respect to the operational semantics. Third, that the subtyping relation (and with it type checking) is undecidable. For reasons of space we refer to an accompanying technical report [34] for proofs.

6.1 Confluence

Theorem 6.1 The $\twoheadrightarrow$ relation is confluent: If $t \twoheadrightarrow t_1$ and $t \twoheadrightarrow t_2$ then there exists a term t' such that $t_1 \twoheadrightarrow t'$ and $t_2 \twoheadrightarrow t'$.

6.2 Type Soundness

We establish soundness of the νObj type system using the syntactic technique of Wright and Felleisen [43]. We first show a subject reduction result which states that typings are preserved under reduction. We then characterize a notion of evaluation result called an *answer* and show that every well-typed, non-diverging term reduces to an answer that has the same type as the original term.

Theorem 6.2 [Subject Reduction] Let Γ be an environment. Let t, t' be terms such that $bn(t, t') \cap dom(\Gamma) = \emptyset$ and let T be a type. If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

To establish type soundness from subject reduction, we still need to show that well-typed non-diverging terms reduce to answers. These notions are defined as follows.

Definition. A term t *diverges*, written $t \Uparrow$ if there exists an infinite reduction sequence $t \rightarrow t_1 \rightarrow \ldots \rightarrow t_n \rightarrow \ldots$ starting in t.

Definition. An *answer* is a value, possibly nested in ν-binders from classes all of whose definitions are fully evaluated. Thus, the syntax of answers a is:

$$\begin{aligned} a &::= v \mid \nu x \leftarrow [x : S |\ \overline{f}]\ ; a \\ f &::= l = v \mid L \preceq T\ . \end{aligned}$$

Theorem 6.3 [Type Soundness] If $\epsilon \vdash t : T$ then either $t \Uparrow$ or $t \twoheadrightarrow a$, for some answer a such that $\epsilon \vdash a : T$.

6.3 Undecidability of Type Checking

Theorem 6.4 There exists no algorithm that can decide if a judgment $\Gamma \vdash t : T$ is derivable or not.

7 Conclusion

This paper develops a calculus for reasoning about classes and objects with type members. We define a confluent notion of reduction, as well as a sound type system based on dependent types.

There are at least three areas where future work seems worthwhile. First, there is the problem of undecidablility of νObj. We need to develop decidable subsystems, or describe type reconstruction algorithms that are incomplete but can be shown to work reasonably well in practice. Second, we would like to explore extensions of the calculus, such as with imperative side effects or with richer notions of information hiding. Third, we would like to study in more detail the relationships between νObj and existing object-oriented languages and language proposals. We hope that the work presented here can be used as a foundation for these research directions.

Acknowledgments We thank Luca Cardelli, Erik Ernst, Benjamin Pierce, Mads Torgersen, Philip Wadler, and Christoph Zenger for discussions on the subject of this paper. We thank Philippe Altherr and Stéphane Micheloud for comments on previous versions of it.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer Verlag, 1996.
2. D. Ancona and E. Zucca. A primitive calculus for module systems. In *Principles and Practice of Declarative Programming*, LNCS 1702, 1999.
3. D. Ancona and E. Zucca. A calculus of module systems. *Journal of Functional Programming*, 2002.
4. H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.
5. V. Bono, A. Patel, and V. Shmatikov. A core calculus of classes and mixins. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 43–66, Lisbon, Portugal, 1999.
6. G. Boudol. The recursive record semantics of objects revisited. Technical Report 4199, INRIA, jun 2001. to appear in Journal of Functional Programming.
7. G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, University of Utah, 1992.
8. G. Bracha and D. Griswold. Extending Smalltalk with mixins. In *OOPSLA '96 Workshop on Extending the Smalltalk Language*, April 1996.
9. G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 282–290, Washington, DC, 1992. IEEE Computer Society.
10. K. B. Bruce. *Foundations of Object-Oriented Programming Languages: Types and Semantics.* MIT Press, Cambridge, Massachusetts, February 2002. ISBN 0-201-17888-5.
11. K. B. Bruce, A. Fiech, and L. Petersen. Subtyping is not a good "Match" for object-oriented languages. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 104–127, 1997.
12. K. B. Bruce, M. Odersky, and P. Wadler. A statical safe alternative to virtual types. In *Proceedings of the 5th International Workshop on Foundations of Object-Oriented Languages*, San Diego, USA, 1998.

13. L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, August 1992.
14. L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1-2):4–56, 1994 1994.
15. L. Cardelli and J. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–38, 1991.
16. K. Crary, R. Harper, and S. Puri. What is a recursive module? In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–63, 1999.
17. O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Simula: Common base language. Technical report, Norwegian Computing Center, October 1970.
18. D. Duggan and C. Sourelis. Mixin modules. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 262–273, Philadelphia, Pennsylvania, June 1996.
19. E. Ernst. *gBeta: A language with virtual attributes, block structure and propagating, dynamic inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.
20. E. Ernst. Family polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 303–326, Budapest, Hungary, 2001.
21. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 171–183, San Diego, California, 1998.
22. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Java Series, Sun Microsystems, second edition, 2000. ISBN 0-201-31008-2.
23. R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, January 1994.
24. A. Igarashi. On inner classes. In *Proceedings of the European Conference on Object-Oriented Programming*, Cannes, France, June 2000.
25. A. Igarashi and B. C. Pierce. Foundations for virtual types. *Proc. ECOOP'99, Lecture Notes in Computer Science*, 1628, 1999.
26. A. Igarashi and B. C. Pierce. Foundations for virtual types. *Information and Computation*, 175(1):34–49, 2002.
27. A. Igarishi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proc. OOPSLA*, Nov. 1999.
28. X. Leroy. A syntactic theory of type generativity and sharing. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, 1994.
29. D. MacQueen. Modules for Standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 198–207, New York, August 1984.
30. O. L. Madsen and B. Møller-Pedersen. Virtual Classes: A powerful mechanism for object-oriented programming. In *Proceedings OOPSLA'89*, pages 397–406, October 1989.
31. O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993. ISBN 0-201-62430-3.
32. T. Nipkow and D. von Oheimb. Java-light is type-safe — definitely. In L. Cardelli, editor, *Conference Record of the 25th Symposium on Principles of Programming Languages (POPL'98)*, pages 161–170, San Diego, California, 1998. ACM Press.
33. M. Odersky. Report on the programming language Scala, 2002. École Polytechnique Fédérale de Lausanne, Switzerland. `http://lamp.epfl.ch/~odersky/scala`.

34. M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. Technical report IC/2002/70, EPFL, Switzerland, September 2002. `http://lamp.epfl.ch/papers/technto.pdf`.
35. M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *Proc. FOOL 10*, Jan. 2003. `http://www.cis.upenn.edu/~bcpierce/FOOL/FOOL10.html`.
36. K. Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, Málaga, Spain, 2002.
37. C. Russo. First-class structures for Standard ML. In *Proceedings of the 9th European Symposium on Programming*, pages 336–350, Berlin, Germany, 2000.
38. Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. *Lecture Notes in Computer Science*, 1445, 1998.
39. K. K. Thorup. Genericity in Java with virtual types. In *Proceedings of the European Conference on Object-Oriented Programming*, LNCS 1241, pages 444–471, June 1997.
40. K. K. Thorup and M. Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. *Lecture Notes in Computer Science*, 1628, 1999.
41. M. Torgersen. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages*, San Diego, CA, USA, January 1998.
42. M. Torgersen. Inheritance is specialization. In *The Inheritance Workshop, with ECOOP 2002*, June 2002. `http://www.cs.auc.dk/~eernst/inhws/`.
43. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115, 1994.
44. M. Zenger. Type-safe prototype-based component evolution. In *Proceedings of the European Conference on Object-Oriented Programming*, Málaga, Spain, June 2002.