Object Inheritance without Classes

Timothy Jones[1], Michael Homer[1], James Noble[1], and Kim Bruce[2]

1 **Victoria University of Wellington**
2 **Pomona College**

Abstract

Which comes first: the object or the class? Language designers enjoy the conceptual simplicity of object-based languages (such as Emerald or Self) while many programmers prefer the pragmatic utility of classical inheritance (such as Simula and Java). Programmers in object-based languages have a tendency to build libraries to support traditional inheritance, and language implementations are often contorted to the same end. In this paper, we revisit the relationship between classes and objects. We model various kinds of inheritance in the context of an object-oriented language whose objects are not defined by classes, and explain why class inheritance and initialisation cannot be easily modelled purely by delegation.

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.1

1 Introduction

Class-based object-oriented languages have a simple story about the relationships between objects and the classes that create them: an object is an instance of a class [2]. A specialised, 'one-off' object is just an instance of a specialised, one-off, anonymous class [15]. Inheritance is between classes, and new objects are constructed and initialised by their classes.

This simple story comes at the expense of a more complicated story about classes – especially so if classes are themselves objects. Thirty years ago, Alan Borning identified eight separate roles that classes can play in most class-based object-oriented languages [4], each of these roles adding to the complexity of the whole language, which typically leads inexorably to various kinds of infinite regress in meta-object systems [21, 28, 43].

To address this problem, prototype-based object-oriented languages, beginning with Lieberman's work inspired by LOGO [32] and popularised by Self [50], adopted a conceptually simpler model in which objects were the primary concept, defined individually, without any classes. Inheritance-like sharing of state and behaviour was handled by delegation between objects, rather than inheritance between their defining classes. Special-purpose objects could be defined directly, while new objects could be created in programs by cloning existing objects. Emerald [3] went one step further and aimed to eschew all implicit sharing mechanisms, supporting neither inheritance nor delegation.

Programmers using object-based languages have found the need to reintroduce classes — several times over in many of these languages. The Emerald compiler, and later the Self IDE, added explicit support for class-style inheritance. Languages with object inheritance such as Lua [34], JavaScript [45], and Tcl [51] have a variety of libraries implementing classes in terms of objects. Most recently classes have been added explicitly to the recent ECMAScript standard [52], to bring some order to the profusion of libraries already offering classes.

The problem we address in this paper is precisely the tension between the conceptual simplicity of objects and the practical utility of classes: can a language be based conceptually on 'objects first' and include a relatively familiar notion of inheritance? In this paper we present models of eight inheritance mechanisms for a language without an inherent class construct: forwarding, delegation, concatenation, merged identity, uniform identity,

multiple uniform identity, method transformations, and positional inheritance. The first three correspond to the foundational object-based models, while merged identity and uniform identity introduce more classical behaviour that parallels C++ and Java, and the remaining models introduce multiple object inheritance with different conflict resolution techniques.

We evaluate the tradeoffs between power and complexity of these models, particularly in their object initialisation semantics, and compare them to the behaviour of other languages, demonstrating that the typical class initialisation semantics are fundamentally at odds with prototypical object inheritance. We have also implemented all of the models in PLT Redex, making the models executable and allowing direct comparison of the differences in execution for any program.

2 Inheritance Without Classes

The term 'inheritance' typically refers to reuse relationships between classes, but there are also a number of 'objects-first' languages that eschew classes and permit reuse relationships directly between objects. Consider the following example of an object constructor:

```
method graphic {
    object {
        method image { abstract }
        method draw { canvas.render(image) }
        var name := "A graphic"; displayList.register(self); draw
    }
}
```

If we can inherit from this object, what is its behaviour when we do so? Expectations vary for different interpretations of inheritance. We can interpret this method as a factory, and inherit from the fresh object it creates, or assign special semantics to constructor methods and inherit from the method itself. Because of the presence of initialisation code in the class, these interpretations have different — and potentially unexpected — behaviours.

In its most basic form, inheritance permits reuse by providing a mechanism for an object to defer part of its implementation to another, already implemented object, but the reality is that there is much more to consider: the value of 'self' in method bodies and during initialisation, intended and accidental method overriding, whether objects are composed of several object identities in an inheritance chain or a single unified identity, the meaning of method requests which are not qualified with a receiver, and so on.

Suppose we create an object which inherits from graphic.

```
def amelia = object {
    inherits graphic
    def image = images.amelia; self.name := "Amelia"
}
```

We can draw different conclusions about the state of our program after the creation of this object depending on which inheritance semantics is in play. We can group these into our relevant concerns:

- **Down-calls**. Can a method in a super-object call down into a method in a lower object? Can it do so during initialisation? The implementation of the draw method relies on a down-call to the image method.

- **Registration**. Is the identity of a super-object stored during initialisation, either explicitly or through lexical capture, the same as the final object? This is clearly the intention of the call to register in graphic's initialisation.
- **Action at a Distance**. Can operations on an object implicitly affect another object? If the registered graphic object is different to amelia, what is the value of its name field after amelia is initialised?
- **Freshness**. Can an object inherit from any other object it has a reference to? Does amelia have to inherit a call to the constructor, or will a preëxisting graphic object suffice?
- **Stability**. Is the implementation of methods in an object the same throughout its lifetime? Which image method will be invoked by the request to draw at the end of graphic? Can the structural type of an object change after it has been constructed?
- **Multiplicity**. Can an object inherit from multiple other objects? If amelia also wished to have the behavior of another object, can a second inherits clause be added? If so, how are multiple methods with the same name resolved, and where are fields located?

We are also interested in the general semanticses of the inheritance systems, such as what order the parts of initialisation execute in, what visibility and security concerns arise, and how local method requests are resolved.

These are the concerns that we will judge in the following object inheritance models. While our graphic example relies on some of these features to behave correctly, these concerns are not necessarily desirable, and each provides certain abilities or guarantees at some cost. Our intention is to use these concerns to provide an accurate description of the tradeoffs involved with each inheritance model. Some of our models of inheritance attempt to interpret graphic as a class, but only in the sense that it is a factory that guarantees fresh graphic objects. We also compare the models to existing languages which use each form of inheritance, particularly JavaScript, which is capable of implementing all of the models.

3 Graceless

In order to provide a formal semantics for the various object inheritance systems, we first present a base model of the Grace programming language without inheritance, and then proceed to extend this base model in different ways to construct the different behaviours. Grace is a useful language to model object inheritance features in, as it is not class-based, and it does not permit mutation of an existing object's structure, so we can consider pure object concerns without being overcome by 'open objects' with trivially mutable structure, as found in JavaScript.

Our core model resembles the existing Tinygrace language [27], but with more features, such that 'Tiny' is not an appropriate moniker. As the language is still not a complete implementation of Grace, we have opted to name it *Graceless*. We have modelled object references, mutable and immutable fields, and method requests unqualified by a receiver, in order to demonstrate the wide-ranging effects of changes to the behaviour of inheritance. Explicit types have been removed, in order to focus on the dynamic semantics.

The grammar for Graceless is provided in Figure 1. The boxed areas in the grammar represent forms which only exist in the typing judgment or at runtime, not in the user-side syntax of programs. As in Grace, we omit empty argument and parameter parentheses, making some method requests indistinguishable from local variables and field lookups per Meyer's uniform access principle [36].

Within an object are method definitions M and statements S. A statement is either an expression or a field declaration. Fields are either constant **def**initions or **var**iable, and **var**

Syntax

e ::= $o \mid e.m(\overline{e}) \mid m(\overline{e}) \mid \textbf{self} \mid v \boxed{\mid e;e \mid e \xrightarrow{x} \mid e \xleftarrow{x} e \mid \textbf{uninitialised}}$ *(Expression)*

S ::= $\textbf{def}\ x = e \mid \textbf{var}\ x \mid \textbf{var}\ x := e \mid e$ *(Statement)*

M ::= $\textbf{method}\ m(\overline{x})\ \{\,\overline{e};\ e\,\}$ *(Method)*

m ::= $x \mid x{:=}$ *(Method name)*

o ::= $\textbf{object}\,\{\,\overline{M}\ \overline{S}\,\}$ *(Object expression)*

r ::= e *(Receiver)*

v ::= $\textbf{done}\ \boxed{\mid \ell}$ *(Value)*

σ ::= $\boxed{\varnothing \mid \ell \mapsto \langle \overline{F}, \overline{M} \rangle, \sigma}$ *(Object store)*

F ::= $\boxed{x \mapsto v}$ *(Field)*

s ::= $v/x \mid \textbf{self}.m/m$ *(Substitution)*

Evaluation context

E ::= $[] \mid E.m(\overline{e}) \mid v.m(\overline{v}, E, \overline{e}) \mid m(\overline{v}, E, \overline{e}) \mid E;e \mid v \xleftarrow{x} E$

Figure 1 The grammar of the Graceless base model, with runtime-only components boxed

fields may be declared without an initial value.

Expressions e in the user-side syntax are object expressions o, method requests either qualified or unqualified by a receiver, or the sentinel value **done**. At runtime, expressions can also include references ℓ to locations in the heap, sequences $e;e$, field fetch $e \xrightarrow{x}$, and field assign $e \xleftarrow{x} e$. The field operations are distinguished from method calls $e.x$ and $e.x := e$. The receivers of the field operations are not holes in the context E, because we only ever create instances of the operations where the receiver is the term **self** (which will then be substituted for some actual reference ℓ).

Objects can be nested inside of each other (as method bodies) and unqualified requests can be made on the resulting local scope. We assume Barendregt's rule [1] for **self** references, such that each **self** variable introduced by nested object expressions is unique. Because method names appear in the local scope as well as the public interface of their directly surrounding object, we cannot assume Barendregt's rule that their names are unique — an object may need to shadow an outer object's method in order to conform to a given interface — so substitution cannot continue past a shadowing definition.

Graceless has two forms of substitution. The typical substitution $[v/x]e$ replaces the term x with the value v in the term e. A *qualifying substitution* $[\textbf{self}.m/m]e$ replaces any local request $m(\overline{e})$ with the qualified form $\textbf{self}.m(\overline{e})$ in the term e. Both forms of substitution are ended by a shadowing definition, which can be either an adjacent method name or a surrounding parameter definition: substitutions into an object expression $[v/x]o$ and $[\textbf{self}.m/m]o$ do not modify o if it contains a method x or m respectively, and the substitutions into a method $[v/x]M$ and $[\textbf{self}.x/x]M$ do not modify the method body if M has a parameter x. Because the local variable x and the local method request x are indistinguishable, the ordering of the substitution is important: $[\textbf{self}.x/x][v/x]x$ produces v, but $[v/x][\textbf{self}.x/x]x$ produces $\textbf{self}.x$.

The reduction judgment $\langle \sigma, e \rangle \rightsquigarrow \langle \sigma', e' \rangle$ is defined in Figure 2, indicating an expression e with store σ is reduced to an expression e' with a potentially modified store σ'. Rule E-Context uses the evaluation context to perform congruence reduction, and Rule E-Next evaluates to the next expression in a sequence when the current one has finished evaluating. Rules E-Fetch, E-Uninitialised, and E-Assign handle operations on a field store, with

$\boxed{\langle \sigma, e\rangle \rightsquigarrow \langle \sigma, e\rangle}$

$$\frac{\langle \sigma, e\rangle \rightsquigarrow \langle \sigma', e'\rangle}{\langle \sigma, E[e]\rangle \rightsquigarrow \langle \sigma', E[e']\rangle}\ (\text{E-Context}) \qquad \frac{}{\langle \sigma, v; e\rangle \rightsquigarrow \langle \sigma, e\rangle}\ (\text{E-Next}) \qquad \frac{}{\langle \sigma, \ell \xleftarrow{x} v\rangle \rightsquigarrow \langle \sigma(\ell)(x \mapsto v), \textsf{done}\rangle}\ (\text{E-Assign})$$

$$\frac{}{\langle \sigma, \ell \xrightarrow{x}\rangle \rightsquigarrow \langle \sigma, \sigma(\ell)(x)\rangle}\ (\text{E-Fetch}) \qquad \frac{(x \mapsto v) \notin \sigma(\ell)}{\langle \sigma, E[\ell \xrightarrow{x}]\rangle \rightsquigarrow \langle \sigma, \textsf{uninitialised}\rangle}\ (\text{E-Uninitialised}) \qquad \frac{\textsf{method}\ m(\overline{x})\ \{\, e\,\} \in \sigma(\ell)}{\langle \sigma, \ell.m(\overline{v})\rangle \rightsquigarrow \langle \sigma, [\ell/\textsf{self}]\overline{[v/x]}e\rangle}\ (\text{E-Request})$$

$$\frac{\ell \text{ fresh} \qquad \overline{m} = \text{names}(\overline{M}, \overline{S}) \qquad \langle \overline{M_f}, \overline{e}\rangle = \text{body}(\overline{S})}{\langle \sigma, \textsf{object}\,\{\, \overline{M}\ \overline{S}\,\}\rangle \rightsquigarrow \langle \sigma(\ell \mapsto \langle \varnothing, \overline{[\textsf{self}.m/m]M}\ \overline{M_f}\rangle), [\ell/\textsf{self}]\overline{[\textsf{self}.m/m]e};\ \ell\rangle}\ \overline{m}\ \text{unique}\ (\text{E-Object})$$

Auxiliary Definitions

$$\text{names}(\overline{\textsf{method}\ m(\overline{x})\ \{\, e\,\}}, \overline{S}) = \overline{m} \cup \overline{m_f} \qquad \textbf{where}\ \langle \overline{\textsf{method}\ m_f(\overline{y})\ \{\, e_f\,\}}, e\rangle = \text{body}(\overline{S})$$

$$\begin{aligned}
\text{body}(\varnothing) &= \langle \varnothing, \varnothing\rangle \\
\text{body}(\textsf{def}\ x = e, \overline{S}) &= \langle \text{accessors}(\textsf{def}, x, y)\ \overline{M}, \textsf{self} \xleftarrow{y} e\ \overline{e}\rangle && \textbf{where}\ y \text{ fresh and } \langle \overline{M}, \overline{e}\rangle' = \text{body}(\overline{S}) \\
\text{body}(\textsf{var}\ x, \overline{S}) &= \langle \text{accessors}(\textsf{var}, x, y)\ \overline{M}, \overline{e}\rangle && \textbf{where}\ y \text{ fresh and } \langle \overline{M}, \overline{e}\rangle' = \text{body}(\overline{S}) \\
\text{body}(\textsf{var}\ x := e, \overline{S}) &= \langle \text{accessors}(\textsf{var}, x, y)\ \overline{M}, \textsf{self} \xleftarrow{y} e\ \overline{e}\rangle && \textbf{where}\ y \text{ fresh and } \langle \overline{M}, \overline{e}\rangle' = \text{body}(\overline{S}) \\
\text{body}(e, \overline{S}) &= \langle \overline{M}, e\ \overline{e}\rangle && \textbf{where}\ \langle \overline{M}, \overline{e}\rangle' = \text{body}(\overline{S})
\end{aligned}$$

$$\text{accessors}(\textsf{def}, x, y) = \textsf{method}\ x\ \{\, \textsf{self} \xrightarrow{y}\,\}$$

$$\text{accessors}(\textsf{var}, x, y) = \textsf{method}\ x\ \{\, \textsf{self} \xrightarrow{y}\,\}\ \textsf{method}\ x{:=}(y)\ \{\, \textsf{self} \xleftarrow{y} y\,\}$$

Figure 2 Term reduction

uninitialised crashing the program in any context. The store access $\sigma(\ell)(x)$ looks up the field x in the object at location ℓ, and $\sigma(\ell)(x \mapsto v)$ sets the field x to the value v in the object at location ℓ, introducing a new field if one was not already present.

Rule E-Request process requests by looking up the corresponding method in the receiver. In the method body, it substitutes both the arguments for parameter names, and the receiver for the name **self**. Rule E-Object takes an object expression and builds a corresponding object in the store, with no fields, the methods in the object expression, and the generated getter and setter methods for the fields. The rule also converts the fields in the object expression into a series of assignments, which ultimately result in the new reference. The internal field names are fresh to avoid having to worry about overridden names under inheritance. Both the methods and the assignments have the relevant qualifying substitutions applied, and the body of the object has **self** bound to the new reference.

Note that we could bind **self** in the bodies of an object's methods either in the Rule E-Object (early binding, when the object is allocated) or in Rule E-Request (late binding, when a method is requested). In this model, either choice produces the same behaviour. As we add inheritance to Graceless, our choice to use late binding in the base model will matter: it is trivial to overwrite by also using early binding, but not the other way around.

Extended Syntax

I ::= **inherits** e $\boxed{\overline{s}}$ *(Inherits clause)*

o ::= $\cdots$ | **object** { I $\overline{M}$ $\overline{S}$ } *(Object expression)*

s ::= $\cdots$ | (ℓ **as self**)/**super** *(Substitution)*

v ::= $\cdots$ $\boxed{\mid (\ell \textbf{ as } v)}$ *(Value)*

r ::= $\cdots$ | **super** $\boxed{\mid (\ell \textbf{ as } e)}$ *(Receiver)*

Extended Evaluation Context

E ::= $\cdots$ | E_I $\quad$ E_I ::= **object** { **inherits** E $\overline{s}$ $\overline{M}$ $\overline{S}$ }

$\boxed{\langle \sigma, e\rangle \rightsquigarrow \langle \sigma, e\rangle}$

(E-Request/Super)

$$\frac{\textbf{method } m(\overline{x})\ \{\, e \,\} \in \sigma(\ell_\uparrow)}{\langle \sigma, (\ell_\uparrow \textbf{ as } \ell_\downarrow).m(\overline{v})\rangle \rightsquigarrow \langle \sigma, [\ell_\downarrow/\textbf{self}][\overline{v/x}]e\rangle}$$

(E-Inherits)

$$\frac{\langle \overline{F}, \overline{M_\uparrow}\rangle = \sigma(\ell) \qquad \overline{M'_\uparrow} = \text{override}(\overline{M_\uparrow}, \text{names}(\overline{M}, \overline{S}))}{\langle \sigma, \textbf{object}\ \{\, \textbf{inherits } \ell\ \overline{s}\ \overline{M}\ \overline{S}\ \}\rangle \rightsquigarrow \langle \sigma, \textbf{object}\ \{\, \overline{M'_\uparrow}\ \overline{[s]}[(\ell \textbf{ as self})/\textbf{super}](\overline{M}\ \overline{S})\ \rangle\ \}}$$

Auxiliary Definitions

$$\text{override}(\textbf{method } m(\overline{x})\ \{\, e \,\}\ \overline{M}, \overline{m}) = \begin{cases} m \in \overline{m} & \text{override}(\overline{M}, \overline{m}) \\ m \notin \overline{m} & \textbf{method } m(\overline{x})\ \{\, e \,\}\ \text{override}(\overline{M}, \overline{m}) \end{cases}$$

Figure 3 Object inheritance extension

4 Object Inheritance

We now extend Graceless with various implementations of object inheritance, and consider the complexity and impacts of the changes. The extensions are presented in a rough ordering of implementation complexity. These represent the three foundational strands of object inheritance: forwarding, as used in E; delegation, as found in JavaScript [52], Lua [25], and Self [50, 10]; and concatenation, as in Kevo [47, 48] and numerous libraries and idioms for languages with open objects.

The extended syntax for object inheritance is given in Figure 3. The extension introduces two new components of user-facing syntax: the bodies of object expressions may now begin with an **inherits** e clause, and requests can now be qualified by the special variable **super**. These two components each result in a runtime complication.

Inheritance introduces the methods from the super-object into the local scope of the inheriting object using an 'up, then out' rule: inherited definitions take precedence over those introduced in a surrounding scope. Because the inherits clause contains an arbitrary expression, we might not know what the names are that the clause will introduce. To counter this, substitutions are delayed by an inherits clause in an object expression: while the substitution will transform the expression in the clause itself, it *will not* proceed into the body of the object expression, and gets 'stuck' on the clause instead. Once the expression in an inherits clause is resolved to an object reference, the substitution can proceed into the body of the surrounding object expression, where it may be removed by shadowing.

Although there is an explicit super-object in this model of inheritance, making a request to **super** is not the same as making a direct request to the super-object, as the value of **self** will be bound to the inheriting object. At runtime, the variable **super** is substituted for a

(E-Object/Forwarding)

$$\frac{\ell \text{ fresh} \qquad \overline{m} = \text{names}(\overline{M}, \overline{S}) \qquad \langle \overline{M_f}, \overline{e} \rangle = \text{body}(\overline{S})}{\langle \sigma, \textbf{object}\ \{\ \overline{M}\ \ \overline{S}\ \}\rangle \rightsquigarrow \langle \sigma(\ell \mapsto \langle \varnothing, [\ell/\textbf{self}]([\textbf{self}.m/m]\overline{M}\ \ \overline{M_f})\rangle), [\ell/\textbf{self}][\textbf{self}.m/m]\overline{e};\ \ell\rangle}\ \overline{m}\text{ unique}$$

Figure 4 Forwarding modifications

special 'up-call' receiver (ℓ **as self**), which indicates the method to call should be sourced from the object at location ℓ, but **self** in the body of that method should be bound to the eventual value of **self** at the site of the request.

The evaluation context has been extended as expected, though we have split the context of an inherits clause into its own form E_I, as future models will restrict the evaluation of expressions in these clauses. Rule E-Inherits transforms an object expression with an inherits clause into one without, by copying the methods from the super-object which are not overridden by a method with the same name into the body of the inheriting object (directly copying the methods has the same behaviour as creating new methods which delegate to the super-object with the same arguments, or searching through a chain of objects for the method, which is representative of typical object inheritance implementations). It also applies the delayed substitutions to the body, after substituting **super** for an up-call receiver to the inherited object. Rule E-Request/Super simply applies an up-call as described.

By making subtle modifications to the existing rules in this extended form of Graceless, we can produce models for various implementations of object inheritance.[1]

4.1 Forwarding

Under forwarding, inherited methods are simply redirected to the super-object. The super-method receives the same arguments and **self** binding. In the example from earlier, if amelia receives a request for the draw method, which is not implemented directly inside of amelia, the request is passed on to the graphic super-object instead. The value of **self** in the resulting invocation is the identity of the super-object: in this example, the draw method crashes, complaining the graphic has not implemented its image method, because the local request to image has been resolved to the graphic object and not passed back down to amelia.

The modification to the existing Graceless dynamic semantics to implement forwarding is provided in Figure 4. The modification makes one subtle change (highlighted) to the Rule E-Object, by early-binding the value of **self** when an object is created in both its methods and field accessors. The result of this change is that the late-binding of **self** in requests (both normal and to super) no longer achieves anything, because **self** has already been bound to the object that the method or field originally appeared in. Any forwarded request behaves as though the original object had received the request directly.

The value of **self** is always the object a method was defined in, so down-calls are not possible; similarly, the value of **self** during initialisation is the distinct identity of the super-object, so registration cannot occur then. In this model, an object can only inherit from another before it has run any of its own initialisation, so every object has a stable structure, but one object may be inherited many times. While this model does not permit

[1] This existing extension of Graceless already implements a form of object inheritance similar to concatenation with all fields reset to **uninitialised**, but we do not know of any language with this behaviour, nor can we see why it would be desirable.

$$\text{(E-Object/Delegation)}\quad \frac{\ell \text{ fresh} \qquad \overline{m} = \text{names}(\overline{M}, \overline{S}) \qquad \langle \overline{M_f}, \overline{e} \rangle = \text{body}(\overline{S})}{\langle \sigma, \textbf{object}\,\{\,\overline{M}\ \overline{S}\,\}\rangle \rightsquigarrow \langle \sigma(\ell \mapsto \langle \varnothing, \overline{[\textbf{self}.m/m]M}\ \overline{[\ell/\textbf{self}]\,M_f}\rangle), [\ell/\textbf{self}]\overline{[\textbf{self}.m/m]e};\ \ell\rangle}\ \overline{m}\text{ unique}$$

Figure 5 Delegation modifications

multiple super-objects, it would only require updating Rule E-Inherits to include methods from multiple inherited objects, with some arbitrary mechanism to resolve multiply-defined methods (such as placing significance on the order of the inherits clauses, or requiring that a multiply-defined method be overridden in the inheriting object).

There is no concept of confidential access of methods between implementations, as a forwarded request is sent as a regular request, and so will only be handled by the public interface of the super-object. Forwarding also does not permit downcalls: an inherited object cannot invoke an inheriting object's method. On the other hand, an object can forward messages to a preëxisting object, and many forwarding objects can share a single target. Inherited fields are shared between all inheriting objects, and the mutation of an inherited field will implicitly affect the super-object and all of its heirs.

In the E language, there is no explicit self reference, as all object definitions are explicitly named. The authors of E refer to the language's inheritance mechanism as "delegation", but in the absence of self references the behaviour aligns with what we have called forwarding.

```
def graphic { to draw() { canvas.render(graphic.image()) } }
def amelia extends graphic { to image() { images.cat() } }
```

Even though amelia defines a method image, the call in draw clearly looks for the method in the graphic object. Methods cannot be requested on the local scope, so the receiver must always be explicit. In order to achieve down-calls, the inheriting object must explicitly be passed to the super-object, which is a standard pattern for simulating class behaviour in E.

4.2 Delegation

Delegation is an implementation of object inheritance that aimed to be at least as powerful as inheritance, if not more so [32, 50, 33]. Delegation has a subtle distinction from forwarding: a **self** request in a method called under delegation goes *back to the original object*, while under forwarding a **self** request to a delegatee will be handled only by that delegatee. This allows delegation to support down-calls, where forwarding cannot.

In the previous example, it was not safe to request the draw method on amelia under forwarding, as the implementation of draw expects to be able to see an overridden implementation of the image method. Under delegation, this now works as expected: draw executes with **self** as amelia, so the local request to image calls down into amelia and successfully retrieves the image of Amelia.

The modification to the existing Graceless dynamic semantics to implement delegation is provided in Figure 5. Like forwarding, the modification early-binds the value of **self** in the Rule E-Object, but this time *only in the field accessors*. The value of self remains late-bound in the bodies of methods to the receiver of a method request, allowing super-objects to perform down-calls to methods implemented in a sub-object. Fields are shared between the original object and any inheriting objects, and, as under forwarding, mutation of any field which originated in a super-object is reflected in all of its heirs.

(E-Inherits/Concatenation)

$$\frac{\langle \overline{x \mapsto v}, \overline{M_\uparrow} \rangle = \sigma(\ell) \qquad \overline{M'_\uparrow} = \mathrm{override}(\overline{M_\uparrow}, \mathrm{names}(\overline{M}, \overline{S}))}{\langle \sigma, \textbf{object}\,\{\,\textbf{inherits}\ \ell\ \overline{s}\ \overline{M}\ \overline{S}\,\}\rangle \rightsquigarrow \langle \sigma, \textbf{object}\,\{\,\overline{M'_\uparrow}\ \overline{[s]}[(\ell\ \textbf{as self})/\textbf{super}](\overline{M}\ \overline{\textbf{self} \xleftarrow{x} v}\ \overline{S})\ \rangle\,\}}$$

Figure 6 Concatenation modification

Delegation makes no requirement of freshness, which combined with down-calls produces a further complication to information hiding in a language with confidential methods that can only be called on **self**. These methods may provide access to secret data or capabilities, and the ability to access them from arbitrary code could be a security concern. Delegation to preëxisting objects opens the way for the 'vampire' problem: any object to which a reference has been obtained can be taken over and fully controlled from the outside, merely by defining a new child object. If access to confidential methods is instead *not* provided to delegators, simulating common classical patterns becomes difficult or impossible, and an odd asymmetry is introduced: the delegator can override an inherited method, changing its behaviour, but has no access to use the overridden method in its own implementation.

Unlike forwarding, delegation permits down-calls after the object has been constructed, but *not during initialisation*. Objects under delegation cannot perform registration during initialisation, as a captured **self** reference in a super-object refers to that super-object, which may have no other references and will not have access to any overriding definitions from the child. These two limitations of object initialisation are the major barriers to simulating the typical behaviour of class-based inheritance under delegation.

Object structure and behaviour is not stable during construction, as new methods may appear on **self** and existing methods may have different implementations depending on the stage of inheritance. Like forwarding, delegation permits inheritance from a preëxisting object; if this is allowed, stability does not exist after construction either. Delegation, like forwarding, can be easily extended to multiple inheritance by introducing multiple inherits clauses and some resolution of multiply-defined methods.

Including the inherits clause in the object constructor ensures that objects can delegate either to a preëxisting prototype object or construct a new object with custom arguments and delegate to that, as an equivalent to calling a super-constructor in a class system. This distinguishes it from the prototypical inheritance in JavaScript, which requires that the prototype property of the constructor be set before any inheriting object is constructed. In Self, object expressions can set their parents as a fresh object constructor call:

```
( | parent* = factory new. | )
```

This has the same semantics in terms of **self** binding in the super-constructor as presented in our model of delegation.

4.3 Concatenation

Concatenation is an alternative approach to both forwarding and delegation that aimed to have the power of inheritance without the drawbacks of delegation [47, 48]. Under concatenation, one object inherits from another by (conceptually) taking a shallow copy of the methods and fields of its parent into itself, and then appending local overriding definitions. Concatenation supports down-calls, but unlike delegation does not allow subsequent changes in either the parent or the child to affect each other.

The modification to the existing Graceless dynamic semantics to implement concatenation is provided in Figure 6. Unlike the previous two models, this modification instead changes the Rule E-Inherits, to copy the fields from the super-object into the inheriting object as assignments. The existing late-binding of **self** in methods is sufficient to provide the desired behaviour: any inherited method executes in the context of the inheriting object, and any request to an inherited field accessor will access the copied field in the inheriting object as well. The only relationship the inheriting object has with its super-object is explicit up-calls, but it is impossible to access or modify the state of the super-object without explicitly referring to it through an existing reference. The resulting behaviour is equivalent to delegating to a clone of the super-object.

Concatenation also makes no requirement of freshness. Concatenation with preëxisting objects does not quite permit the 'vampirism' of delegation, but does allow 'mind reading': any confidential state in an object can be read simply by inheriting from it, but the existing object cannot be manipulated by the child. Unlike the two previous models, mutations to inherited fields do not cause action at a distance, as the mutation will always affect a field in the actual receiver of the request (even for super-calls). With lexical scoping, the two objects are also not as independent as they seem: methods exist in the same scope in both objects, and any lexically captured state is shared between the two.

Like delegation, concatenation permits down-calls after the object has been constructed, but still not during construction. Concatenation does not allow registration, as a captured **self** reference in a super-object refers to the parent. Object structure and behaviour is not stable during construction; as for delegation, if inheritance from preëxisting objects is allowed then stability does not exist afterwards either. Concatenation can be straightforwardly extended to multiple inheritance by inserting the contents of each parent into the child, with some resolution of multiply-defined methods.

Objects with mutable structure can trivially implement concatenation, by manually copying the structure of the inherited object into the inheriting one. JavaScript complicates this story with unenumerable properties, implicit field accessors, and existing delegation relationships, but for the most part this is a valid implementation of concatenation:

```
for (var name in inherited) { inheriting[name] = inherited[name]; }
```

It is also possible to use mutable object structure to implement either forwarding or delegation, by assigning methods (or field accessors) to the inheriting object that directly forward or delegate to the inherited object, as in our models. In a JavaScript constructor:

```
var self = this; this.foo = function () { self.bar(); };
```

If the foo method is called on a sub-object, the call to bar is guaranteed to not perform a down-call, because self is bound directly to the original object.

JavaScript's built-in object inheritance otherwise works the same way as Graceless delegation, but field assignments are directly available in the language instead of only through accessor methods. The result is that an object shares each field of its inherited object, but when a field is assigned to it, the field is unique to that object, shadowing the inherited one. In any language where the objects have mutable structure it is necessary to retain a parent reference in order to implement delegation, in order to accurately defer to the current implementation of a parent object. Implementing more complicated forms of inheritance in JavaScript, such as with traits or mixins, tends to involve combining the built-in delegation alongside manual concatenation.

Extended Evaluation Context

$E_I ::=$ **object** { **inherits** $E\ \overline{s}\ \overline{M}\ \overline{S}$ } **where** $E \neq \ell.m(\overline{v})$

$$\frac{\langle \sigma, v.m(\overline{v}) \rangle \rightsquigarrow \langle \sigma', \overline{e}; o \rangle}{\langle \sigma, \textbf{object}\ \{\ \textbf{inherits}\ v.m(\overline{v})\ \overline{s}\ \overline{M}\ \overline{S}\ \}\rangle \rightsquigarrow \langle \sigma', \textbf{object}\ \{\ \textbf{inherits}\ \overline{e}; o\ \overline{s}\ \overline{M}\ \overline{S}\ \}\rangle} \quad \text{(E-Inherits/Fresh)}$$

Figure 7 Fresh inheritance modification

5 Emulating Classes

The inheritance models in the previous section represent the three foundational strands of purely object-based inheritance. Class-based languages tend to provide different semantics, and programmers and language designers may wish to use those behaviours, or may unconsciously expect object-based languages to behave similarly.

It is possible to construct object-based models that parallel many of these classical behaviours. In some languages with very flexible semantics, such as JavaScript and Lua, libraries exist to provide "classes" as a second-class construct by mutating objects or leveraging specially-constructed objects with the existing inheritance systems [45, 34]. The two models in this section approximate the inheritance behaviour of C++ (Section 5.1) and Java (Section 5.2) in an object-based system. They remain purely object-based, but trade off some of the flexibilities of the object inheritance models for the classical functionalities they provide, such as registration, down-calls, and stability.

The new models use as their base a further extension to Graceless inheritance, provided in Figure 7, which enforces that the inherited object is a 'fresh' reference. This extension modifies the environment context E_I so that expressions directly in an inherits clause may only be reduced if they are not a request ready to be computed. These requests are manually processed by the Rule E-Inherits/Fresh, which applies the request as normal but ensures that the body of the resulting method call returns a fresh object constructor. These rules permit the object constructor to then be resolved to a reference before handling the inherits clause.

The requirement of strict syntactic freshness is not required for merged identity, but it prevents potentially dangerous manipulation of object identity from outside of an object's creator. Merged identity builds directly on the new extension, while uniform identity further restricts E_I to also prevent an object expression from being computed in an inherits clause.

5.1 Merged Identity

In this model, an inheriting object takes over the identity of its parent, 'becoming' that object but putting in place all of its own method definitions, in effect the reverse of concatenation. The parent object is constructed and initialised before the child, and then mutated at the point of inheritance. Rather than mutating preëxisting objects, inheritance must now occur from a fresh object, one newly created and immediately returned from a method call. Without the requirement of freshness, objects can directly steal the identity of any existing object: the resulting 'body-snatchers' problem is substantially worse than delegation's vampirism, which can only control objects internally. Overridden methods from the parent remain accessible through **super**, and will always execute with the final identity of the object.

Merged identity provides essentially the C++ inheritance behaviour: the apparent type

(E-Inherits/Merged)

$$\frac{\langle \overline{F}, \overline{M_\uparrow} \rangle = \sigma(\ell) \qquad \ell_\uparrow \text{ fresh} \qquad \overline{m_\downarrow} = \text{names}(\overline{M}, \overline{S}) \qquad \overline{M'_\uparrow} = \text{override}(\overline{M_\uparrow}, \overline{m_\downarrow}) \qquad \overline{m} = \overline{m_\downarrow} \cup \text{names}(\overline{M'_\uparrow}, \varnothing) \qquad \langle \overline{M_f}, \overline{e} \rangle = \text{body}(\overline{S}) \quad \overline{m} \text{ unique}}{\begin{array}{c} \langle \sigma, \textbf{object}\ \{\ \textbf{inherits}\ \ell\ \overline{s}\ \overline{M}\ \overline{S}\ \}\rangle \rightsquigarrow \\ \langle \sigma(\ell_\uparrow \mapsto \langle \varnothing, \overline{M_\uparrow} \rangle)(\ell \mapsto \langle \overline{F}, \overline{M'_\uparrow}\ \overline{[s][\textbf{self}.m/m]}[(\ell_\uparrow\ \textbf{as self})/\textbf{super}]\overline{M}\ \overline{M_f} \rangle), \\ \overline{[s]}[\ell/\textbf{self}]\overline{[\textbf{self}.m/m]}[(\ell_\uparrow\ \textbf{as self})/\textbf{super}]\overline{e};\ \ell \rangle \end{array}}$$

Figure 8 Merged identity modification

of the object changes during construction, as each layer of inheritance is processed. After object construction is complete, down-calls resolve to their final overridden method, but until then they obtain the most recent definition from ((great-)grand)parent to child.

The use of **self** for registration in graphic will now correctly store the value of amelia when initialising the super-object. Although the object is *not* amelia when it is registered, amelia *becomes* the object that was registered once it has finished initialising, merging the two identities together. The initialisation of amelia will still fail at the local request of draw, because amelia's overriding of the abstract method has not yet been merged into the identity of the object.

The modification to Graceless with fresh inheritance to implement merged identity is provided in Figure 8. The Rule E-Inherits/Merged overwrites the existing Rule E-Inherits. Where the old rule resulted in a new object expression with some of the methods from the super-object, this new rule skips straight to returning the resulting reference. This is necessary because the resulting reference is not fresh: it is the reference of the inherited object. The inherited object is updated with the new methods in the store, including keeping all of the old field values, but removing overridden methods.

The new rule is a combination of the behaviour of the old E-Inherits and Rule E-Object. We have highlighted the changes to their combined behaviour. There is no longer an object which corresponds to **super** to perform an up-call substitution on the body of the object as in Rule E-Inherits, so rather than creating a fresh location for the resulting object, we create a fresh location for the super-object instead. The super-object has all of the methods of the inherited object before it was modified, while the merged object retains its own fields instead of beginning with an empty field store. All of the models which emulate classes with super-references create these 'part-objects', which exist purely to store the super-methods for super calls, never have any fields of their own, and are only ever referenced under an alias for the 'real' object that is bottom-most in the hierarchy.

Like delegation, merged identity enables down-calls after, but not during, initialisation. Unlike delegation, a single identity is preserved throughout the construction process, so registration is possible. It does not provide classical stability during initialisation, but once objects are complete they cannot change again. The meta-mutation of merging objects means that it unsafe to permit inheritance from arbitrary objects, so only freshly-constructed objects are valid parents. Merged identity is really the only model presented in this paper that does not lend itself well to multiple inheritance, because only a single parent can have its identity preserved.

As a reverse form of concatenation, merged identity is just as easy to implement for objects with mutable structure. A constructor can call into the super-constructor, then concatenate its own definitions into the resulting object. Copying the original object beforehand ensures that a super-reference is still available.

Extended Evaluation Context

$E_I ::=$ **object** $\{$ **inherits** $E\ \overline{s}\ \overline{M}\ \overline{S}\ \}$ **where** $E \neq \ell.m(\overline{v})$ **and** $E \neq$ **object** $\{\ \overline{M}\ \overline{S}\ \}$

(E-Inherits/Uniform)

$$\frac{\ell \text{ fresh} \qquad \overline{m} = \text{names}(\overline{M}, \overline{S}) \qquad \overline{M_\uparrow} = [\textbf{self}.m/m]\overline{M} \qquad \langle \overline{M_f}, \overline{e} \rangle = \text{body}(\overline{S}) \qquad \overline{m_\downarrow} = \text{names}(\overline{M_\downarrow}, \overline{S_\downarrow}) \qquad \overline{M'_\uparrow} = \text{override}(\overline{M_\uparrow}\ \overline{M_f}, \overline{m_\downarrow})}{\begin{array}{c}\langle \sigma, \textbf{object}\ \{\ \textbf{inherits}\ \textbf{object}\ \{\ \overline{M}\ \overline{S}\ \}\ \overline{s}\ \overline{M_\downarrow}\ \overline{S_\downarrow}\ \}\rangle \rightsquigarrow \langle \sigma(\ell \mapsto \langle \varnothing, \overline{M_\uparrow}\ \overline{M_f}\rangle),\\ \textbf{object}\ \{\ \overline{M'_\uparrow}\ \overline{[s]}[(\ell\ \textbf{as self})/\textbf{super}]\overline{M_\downarrow}\ \ [\textbf{self}.m/m]\overline{e}\ \ \overline{[s]}[(\ell\ \textbf{as self})/\textbf{super}]\overline{S_\downarrow}\ \}\rangle\end{array}}\ \overline{m}\text{ unique}$$

Figure 9 Uniform identity inheritance modification

5.2 Uniform Identity

The uniform identity design is a direct attempt to match as closely as possible the behaviour of a typical class-based inheritance system, but based on objects rather than on classes. In this design, the *first* observable action is to create a new object identity in the bottom-most 'child' object constructor; this exactly mirrors the merged identity design, which creates a single identity of the *topmost* parent. All inherited declarations are assembled in a single object associated with this identity, but no fields are initialised and no inline code run until the direct inheritance completes. Inheritance occurs with the identity "passed along": declarations are attached to the original object, without initialisation, until the topmost object with no parent is reached. Finally, the initialisation code runs from top to bottom.

All initialisation occurs in the context of the final object, including its behaviour. No new methods or overrides are added visibly during construction. While initialisation code always sees objects as a consistent type, it may observe uninitialised fields, including constant fields. We model uninitialised field access as a fatal error, and leave static detection to a higher layer in the language. This model essentially aligns with the semantics of Java-like languages.

The behaviour of amelia is the same as in the merged identity semantics, except that the local request to draw in the initialisation of graphic now successfully down-calls into amelia's implementation, as the object *is* amelia during all initialisation in the hierarchy. The result is still an error, as amelia's initialisation of the image field has not yet been executed. Super-constructors may still encounter uninitialised fields that will be assigned to in some sub-constructor, particularly when methods can be overridden by fields.

The modification to Graceless with fresh inheritance to implement uniform identity is provided in Figure 9. As with merged identity, the Rule E-Inherits/Uniform overwrites the existing Rule E-Inherits. The evaluation context E_I has been further refined to also prevent the evaluation of object expressions directly inside an inherits clause.

Where Rule E-Inherits/Merged was a modified application of E-Inherits and then E-Object, the uniform identity modification applies this in reverse. As the body of the inherits clause is an object expression, we have to apply many of the same processes for reducing a regular object expression, but then move the sequence of expressions resulting from the body of the super-object down into the body of the inheriting object. The evaluation *does* create a new object reference for the super-object, but only for up-calls to **super**. All fields will end up in the ultimate inheriting object.

Because the inherited object expression has not been processed into a runtime object, the rule needs to manually build its field methods and body. The substitutions to local definitions $\overline{m}$ need to be applied to the inherited methods (both those in the store, and the non-overridden methods in the inheriting object). The same substitution into the body of the

inheriting object occurs as before, but now **self** is not bound in the body: **self** will be bound by the ultimate application of Rule E-Object. The substitutions applied simultaneously to $M_{\downarrow}$ and $S_{\downarrow}$ in the inheriting object by Rule E-Inherits now have to be split between the two, as the super-body e appears in between, but it amounts to the same behaviour.

Uniform identity permits down-calls both after and during construction, as well as registration, as the identity and structure of the object are both constant, also guaranteeing stability. It does not allow inheriting from preëxisting objects, instead requiring that parents be fresh. It does not support multiple inheritance, but there is a logical extension to do so.

Constructors have a special role in JavaScript, and must have their prototype property set before they are used to construct a new object. As a result, inheriting from a preëxisting prototype object is simple, but inheriting from another constructor, particularly one that requires arguments specific to a particular inheriting object, is more difficult. Object.create allows the creation of an object from a prototype without actually invoking the constructor:

```
function Bar(arg) { Foo.call(this, arg); }
Bar.prototype = Object.create(Foo.prototype);
```

The ability to bind the value of **this** in a function call allows JavaScript to implement uniform identity, ensuring that the initialisation code in a super-constructor can be executed in the context of the inheriting object instead of creating a fresh object and inheriting from that. JavaScript's class syntax, introduced by ECMAScript 2015, is just sugar for this behaviour.

The most natural forms of inheritance in JavaScript are the built-in single delegation and, by extension, single uniform inheritance, now codified directly in the language with the class syntax. Our primary concern is that without the particular dynamism of the necessary JavaScript features, simulating class initialisation is impossible without reinterpreting factory methods as constructors with special semantics, at which point the language has arguably just implemented classes.

6 Multiple Inheritance

Reusing behaviour from multiple parents is a widespread desire, but is less commonly supported in language designs in practice. In this section we show three logical extensions enabling multiple object inheritance. The first extends the uniform identity model with multiple **inherits** statements. The second is a separate model following the tradition of trait systems, where method names are unique in any object. The third extends any of the base models with the ability to include multiple **inherits** statements in an object, processed imperatively. All of these extensions process inherits clauses as under uniform identity, but they could all be simplified to perform inheritance from preëxisting objects as per forwarding, delegation, or concatenation. We have demonstrated this by constructing all possible combinations in our Redex implementation.

All systems enable code reuse from arbitrarily-many parents. All parents are treated symmetrically in the first two models, but further restrictions or privileges could be accorded to some parents without substantially affecting the models. In order to handle conflicts in symmetric inheritance, methods can be **abstract** in their body, which causes an error if that method is called. Alternatively, the implementations could ban the construction of an object with abstract methods, but this would only be valid for uniform identity, as the overriding of a concrete implementation would not take effect until the object was inherited under any of the other interpretations.

Extended Syntax

$I ::= \textbf{inherits}\ e\ \textbf{as}\ x$ *(Inherits clause)* $\quad o ::= \textbf{object}\,\{\,\overline{I}\ \boxed{\overline{s}}\ \overline{M}\ \overline{S}\,\}$ *(Object expression)*

$e ::= \cdots \mid \textbf{abstract}$ *(Expression)* $\quad s ::= \cdots \mid (\ell\ \textbf{as self})/x$ *(Substitution)*

$I^o ::= \textbf{inherits object}\,\{\,\overline{M}\ \overline{S}\,\}\ \textbf{as}\ x$ *(Evaluated inherits clause)*

Extended Evaluation Context

$E ::= \cdots \mid \textbf{object}\,\{\,\overline{I^o}\ \textbf{inherits}\ E\ \textbf{as}\ x\ \overline{s}\ \overline{M}\ \overline{S}\,\}\ \textbf{where}\ E \neq \ell.m(\overline{v})\ \textbf{and}\ E \neq \textbf{object}\,\{\,\overline{M}\ \overline{S}\,\}$

(E-Inherits/Multiple-Fresh)

$$\frac{\langle \sigma, \ell.m(\overline{v})\rangle \rightsquigarrow \langle \sigma', \overline{e};\, o\rangle}{\langle \sigma, \textbf{object}\,\{\,\overline{I^o}\ \textbf{inherits}\ \ell.m(\overline{v})\ \textbf{as}\ x\ \overline{I}\ \overline{s}\ \overline{M}\ \overline{S}\,\}\rangle \rightsquigarrow \langle \sigma, \textbf{object}\,\{\,\overline{I^o}\ \textbf{inherits}\ \overline{e};\, o\ \textbf{as}\ x\ \overline{I}\ \overline{s}\ \overline{M}\ \overline{S}\,\}\rangle}$$

(E-Inherits/Multiple)

$$\frac{\overline{\ell\ \text{fresh}} \quad \overline{m} = \text{names}(\overline{M}, \overline{S}) \quad \overline{\overline{M'} = [\textbf{self}.m/m]M} \quad \overline{\langle \overline{M_f}, \overline{e}\rangle = \text{body}(\overline{S})} \quad \overline{M_\uparrow} = \text{join}(\overline{\overline{M'}\ \overline{M_f}}) \quad \overline{m_\downarrow} = \text{names}(\overline{M_\downarrow}, \overline{S_\downarrow}) \quad \overline{M'_\uparrow} = \text{override}(\overline{M_\uparrow}, \overline{m_\downarrow}) \quad \overline{\overline{m}\ \text{unique}}}{\begin{array}{l}\langle \sigma, \textbf{object}\,\{\,\overline{\textbf{inherits object}\,\{\,\overline{M}\ \overline{S}\,\}\ \textbf{as}\ x\ \overline{s}\ \overline{M_\downarrow}\ \overline{S_\downarrow}}\,\}\rangle \rightsquigarrow \langle \sigma(\overline{\ell \mapsto \langle \varnothing, \overline{M'}\ \overline{M_f}\rangle}),\\ \quad \textbf{object}\,\{\,\overline{M'_\uparrow}\ \overline{[s][(\ell\ \textbf{as self})/x]M_\downarrow}\ \overline{[\textbf{self}.m/m]\overline{e}}\ \overline{[s][(\ell\ \textbf{as self})/x]S_\downarrow}\,\}\rangle\end{array}}$$

Auxiliary Definitions

$$\text{join}(\varnothing) = \varnothing$$

$$\text{join}(\textbf{method}\ m(\overline{x})\ \{\,e\,\}\ \overline{M}) = \begin{cases} m \notin \text{names}(\overline{M}, \varnothing) & \textbf{method}\ m(\overline{x})\ \{\,e\,\}\ \text{join}(\overline{M}) \\ e \equiv \textbf{abstract} & \text{join}(\overline{M}) \\ \textbf{method}\ m(\overline{y})\ \{\,\textbf{abstract}\,\} \in \overline{M} & \text{join}(\overline{M}\ \textbf{method}\ m(\overline{x})\ \{\,e\,\}) \\ \text{otherwise} & \textbf{method}\ m\ \{\,\textbf{abstract}\,\}\ \text{join}(\text{override}(\overline{M}, m)) \end{cases}$$

Figure 10 Multiple uniform identity modification

6.1 Multiple Uniform

The multiple uniform model allows a sequence of **inherits** statements to appear at the start of an object body. Each statement includes an **as** name clause, which binds name locally to have **super** semantics with regard to that inheritance tree. The single-inheritance uniform identity model is recovered with **inherits** parent **as super**. This model approximates Java classes with multiple inheritance.

When the same method name is inherited from multiple parents, none has priority and an abstract method of that name is inserted instead. The programmer must provide a local override calling the version from a particular named parent if desired. All methods are collected and installed before any initialisation code from the object bodies executes, so a consistent set of method implementations is seen throughout the initialisation.

The implementation of multiple uniform identity is presented in Figure 10, as an extension to Graceless inheritance with the caveat that **super** is no longer a special form of receiver. The evaluation context and Rule E-Inherits/Multiple-Fresh implements fresh inheritance in objects with potentially multiple inherits clauses, ensuring each clause is evaluated in order.

Rule E-Inherits/Multiple is essentially the same as Rule E-Inherits/Uniform, but processing multiple inherits clauses at once (hence the extra multiplicities for many of the bindings). Once all of the super-methods are collected, conflicting methods are resolved with the join auxiliary function that, for each unique method name in the collection of methods, accepts exactly one concrete implementation of a method with that name and removes all

of the abstract implementations, or provides a single abstract method with that name and removes all other implementations.

Multiple uniform supports registration and downcalls exactly as in uniform identity. It supports multiple inheritance from freshly-created parents, and is stable because methods are collected first. These are all properties of uniform inheritance: applying this modification to the simpler object inheritance models each retains their own particular properties as well.

6.2 Method Transformations

Multiple inheritance under method transformations resembles trait-like composition of objects, representing object values as a single mapping of method names to methods, with no equivalent to **super** in the previous designs. Instead, an **inherits** statement can have any number of **alias** or **exclude** clauses associated, which respectively create an alternative name for an inherited method and exclude its implementation. If a method is overridden locally and still needs to be accessed, the inherited method can be aliased to a different name and accessed through that name within the object. An object contains at most one method with any given name, and there are no part-objects.

Multiple **inherits** statements can appear in an object, treated symmetrically. If the same name is inherited from multiple parents, all but one must be **exclude**d, or a local overriding method declared, if a concrete implementation of that method is to appear in the object. All inheritance expressions are evaluated and the final method set of the object assembled before any initialisation code executes. Initialisation occurs from top to bottom (depth-first search) within each branch of the inheritance hierarchy. Methods are decoupled from their names because aliases may provide multiple equivalent names all reaching the method, while exclusion means that local definitions may not be implemented in the final object.

If amelia wished to simultaneously be a graphic and a gunslinger, then two inherits clauses can be included under the transformation model. If the gunslinger class also contained a draw method for drawing her gun, amelia would be required to chose a single implementation by excluding one of the two. The excluded method can still be accessed if it is aliased:

```
def amelia = object {
    inherits gunslinger
    inherits graphic alias draw as render exclude draw
    def image = images.amelia; var name := "Amelia"
}
```

Even if the gunslinger class also has a name method, amelia has successfully combined the two by overriding both. Note that method resolution is consistent throughout the entire object: the local request to draw in the graphic initialisation will unholster instead of rendering.

The implementation of method transformation multiple uniform inheritance is given in Figure 11 as a modification of the previous uniform multiple inheritance extension. Inherits clauses no longer have super-names, using the method aliasing and excluding syntax instead. The obvious modifications to the evaluation context and fresh rule to account for method transformations instead of super-names are omitted.

Rule E-Inherits/Transform pre-processes the methods of each inherited object before passing them to join, with the aliases and excludes auxiliary functions. Both of these functions proceeds as expected: fold the transformations over the methods, applying the alias or exclude rules in order. Because these rules are ordered, it is possible to create an alias of an existing alias that occurs earlier in the list, and it is possible to exclude aliases. Exclusion is always processed after aliases, so a directly excluded method cannot be aliased.

Extended Syntax

$I \quad ::= \quad \textbf{inherits}\ e\ \textbf{alias}\ \overline{m\ \textbf{as}\ m}\ \textbf{exclude}\ \overline{m} \qquad \textit{(Inherits clause)}$

$$\frac{\begin{array}{c}\overline{\ell\ \text{fresh}} \qquad \overline{\overline{m} = \text{names}(\overline{M}, \overline{S})} \qquad \overline{\overline{M'} = \overline{[\textbf{self}.m/m]M}} \qquad \overline{\langle \overline{M_f}, \overline{e}\rangle = \text{body}(\overline{S})} \\ \overline{\overline{M_a} = \text{aliases}(\overline{m_a\ \textbf{as}\ m'_a}, \overline{M'\ M_f})} \qquad \overline{\overline{M_e} = \text{excludes}(\overline{m_e}, \overline{M_a})} \qquad \overline{M_\uparrow = \text{join}(\overline{\overline{M_e}})} \\ \overline{\overline{m_\downarrow} = \text{names}(\overline{M_\downarrow}, \overline{S_\downarrow})} \qquad \overline{M'_\uparrow = \text{override}(\overline{M_\uparrow}, \overline{m_\downarrow})}\end{array}}{\begin{array}{c}\langle \sigma, \textbf{object}\ \{\ \overline{\textbf{inherits object}\ \{\ \overline{M}\ \overline{S}\ \}\ \textbf{alias}\ \overline{m_a\ \textbf{as}\ m'_a}\ \textbf{exclude}\ \overline{m_e}}\ \overline{s}\ \overline{M_\downarrow}\ \overline{S_\downarrow}\ \}\rangle \rightsquigarrow \\ \langle \sigma, \textbf{object}\ \{\ \overline{M'_\uparrow}\ \overline{[s]M_\downarrow}\ \overline{\overline{[\textbf{self}.m/m]e}}\ \overline{[s]S_\downarrow}\ \}\rangle\end{array}}\ \overline{\overline{m}\ \text{unique}}$$

Auxiliary Definitions

$$\text{aliases}(\varnothing, \overline{M}) = \overline{M} \qquad \text{aliases}(m\ \textbf{as}\ m'\ \overline{m\ \textbf{as}\ m'}, \overline{M}) = \text{aliases}(\overline{m\ \textbf{as}\ m'}, \text{alias}(\overline{M}, m\ \textbf{as}\ m'))$$
$$\text{excludes}(\varnothing, \overline{M}) = \overline{M} \qquad \text{excludes}(m\ \overline{m}, \overline{M}) = \text{excludes}(\overline{m}, \text{exclude}(\overline{M}, m))$$

$$\text{alias}(\varnothing, m\ \textbf{as}\ m') = \varnothing$$
$$\text{alias}(\textbf{method}\ m(\overline{x})\ \{\, e\,\}\ \overline{M}, m\ \textbf{as}\ m') = \textbf{method}\ m(\overline{x})\ \{\, e\,\}\ \textbf{method}\ m'(\overline{x})\ \{\, e\,\}\ \text{alias}(\overline{M}, m\ \textbf{as}\ m')$$
$$\text{alias}(M\ \overline{M}, m\ \textbf{as}\ m') = M\ \text{alias}(\overline{M}, m\ \textbf{as}\ m')$$

$$\text{exclude}(\varnothing, m) = \varnothing$$
$$\text{exclude}(\textbf{method}\ m(\overline{x})\ \{\, e\,\}\ \overline{M}, m) = \textbf{method}\ m\ \{\, \textbf{abstract}\,\}\ \text{exclude}(\overline{M}, m)$$
$$\text{exclude}(M\ \overline{M}, m) = M\ \text{exclude}(\overline{M}, m)$$

Figure 11 Method transformation modification, with evaluation context and fresh rule omitted

Uniform method transformation permits both down-calls and registration. It provides stability through time, but not through local analysis of visible declarations. It imposes the same freshness requirements as the other models, and supports multiple inheritance. Applying method transformation to the simpler object inheritance models continues to maintain their own particular properties, as they were never stable to begin with.

6.3 Positional

The previous multiple inheritance models treated the inherits clauses as symmetric, such that no definition in any one was preferred in the case of conflicts, requiring resolution either by overriding or method transformations. Moreover, they did not permit any initialisation until all of the direct inheritance was completed. This can be a problem if, for instance, fields need to be initialised for down-calls in super-initialisation code, as with amelia's image field. Positional inheritance addresses these concerns, at the cost of visibly mutating the object during construction, similarly to what can be achieved with mutable object structure.

Under positional inheritance, multiple **inherits** clauses can appear in an object, amongst the typical statements. These inheritances are processed imperatively where they appear, in order, using the same semantics as the selected base model of inheritance. As under multiple uniform, each **inherits** statement can have a name associated. Positional could also be used for single inheritance, allowing some initialisation before the super-constructor is called, but the visible mutation is still present, and the distinct ordering of the inherits clauses in an object body implies a natural method conflict resolution.

In the most general version, an **inherits** clause can appear anywhere in the object body, and have other code before, after, and in between, with the semantics of the inheritance taking effect at the point of appearance and later parents having precedence over earlier. This allows some interesting programmer choices with some of the base models. Altering the order of parents affects which versions of same-named methods are accessed, and interleaving other code in between exposes both at different times. The availability and safety of upcalls and down-calls are affected by the placement of the inheritance, field initialisation, and other code. A more restrained approach could limit inheritance to appearing all at the top (or at the bottom) of the object body.

Positional delegation with named **super**s is essentially the behaviour of Self [10], where multiple parent pointers may exist in a single object; Self does not allow initialisation code to execute in the context of the object under construction or have any priority between parents, but does allow parent pointers to be mutable. The nature of concatenation fundamentally supports positional multiple inheritance, simply copying in the contents of the inherited object in place of the **inherits** statement, and the limitation in the base model was purely syntactic. Named **super**s (or a next-method functionality) are also necessary to access overridden methods. Multiple forwarding is quite straightforward, and strictly named **super**s are not required: because forwarding only accesses the public interface, an ordinary reference to each parent suffices. One of the primary use-cases of positional inheritance — initialising fields before invoking a super-constructor — is irrelevant without uniform identity, as the super-constructor cannot make a down-call into the inheriting object anyway.

Merged identity does not lend itself to the positional extension because it relies on taking over the identity of the parent object, which with multiple parents would result in repeated identity changes, some of which may even lose methods. A different extension could merge multiple identities together, or resolve the resulting issues in some other way, but we do not address this combination here and simply exclude it from consideration as confusing at best.

Positional inheritance is the only one of our multiple-inheritance models that permits inheriting from something obtained from a parent. A parent could define a number of specialised "inner classes", with the intention that its child would in turn inherit from one of those specialisations as well. It is not obvious to us that such an ability is useful, but nor is it obvious that it is not. We note this unique ability, but do not pursue it further.

The most interesting version of positional inheritance is based on uniform identity, so we will focus on that extension for the remainder of this section. Each **inherits** is processed in order of appearance, using the same structure-then-initialisation process as under uniform identity. Again, a single identity exists for all initialisation, and again down-calls are valid throughout.

Positional inheritance reintroduces mutation during construction to uniform identity, because each **inherits** adds new methods to the object. When multiple methods are inherited by the same name, the last-inherited method wins out. An unusual aspect is that while down-calls are always available, during construction 'side-calls' to co-parents of a common child can be made only to parents whose **inherits** preceded this one. An object can even define a fresh constructor directly inside of itself, and then inherit from it.

Each line of initialisation occurs after preceding inheritance statements and before subsequent inheritance statements. If **inherits** precedes a field initialisation or other expression, upcalls to that parent are available from that expression; if **inherits** follows a field initialisation, down-calls from that parent accessing that field are safe.

The implementation of positional uniform identity inheritance is provided in Figure 12 as an extension to the core Graceless language; the rule for fresh inheritance is omitted

Extended Syntax

$e ::= \cdots \boxed{\mid \textbf{super inherits}\ e\ \textbf{as}\ x\ \overline{s} \mid \overline{i}\ \textbf{inherits}\ e\ \textbf{as}\ x\ \overline{s}}$ *(Expression)*

$S ::= \cdots \mid \textbf{inherits}\ e\ \textbf{as}\ x$ *(Statement)* $\quad r ::= \cdots \boxed{\mid (\ell\ \textbf{as}\ \ell)}$ *(Receiver)*

$o ::= \textbf{object}\,\{\,\boxed{\overline{s}}\ \overline{M}\ \overline{S}\,\}$ *(Object expression)* $\quad i ::= \langle \ell, \overline{M}, \overline{s} \rangle$ *(Inherits context)*

$s ::= \cdots \mid (\ell\ \textbf{as}\ \ell)/x \mid \overline{i}/\textbf{super}$ *(Substitution)*

Extended Evaluation Context

$E ::= \overline{i}\ \textbf{inherits}\ E\ \textbf{as}\ x\ \overline{s}\ \textbf{where}\ E \neq \ell.m(\overline{v})\ \textbf{and}\ E \neq \textbf{object}\,\{\,\overline{s}\ \overline{M}\ \overline{S}\,\}$

(E-Object/Positional)

$$\frac{\ell \text{ fresh} \qquad \overline{m} = \text{names}(\overline{M}, \overline{S}) \qquad \langle \overline{M_f}, \overline{e} \rangle = \text{body}(\overline{S})}{\langle \sigma, \textbf{object}\,\{\,\overline{s}\ \overline{M}\ \overline{S}\,\}\rangle \rightsquigarrow \langle \sigma(\ell \mapsto \langle \varnothing, \overline{[s]}[\textbf{self}.m/m]\overline{M}\ \overline{M_f}\rangle), \overline{[s]}[\langle \ell, \overline{M}\ \overline{M_f}, \overline{s}\rangle/\textbf{super}][\ell/\textbf{self}]\overline{[\textbf{self}.m/m]e};\ \ell\rangle}\ \overline{m}\text{ unique}$$

(E-Inherits/Positional)

$$\frac{\begin{array}{c}\ell \text{ fresh} \qquad \overline{m} = \text{names}(\overline{M}, \overline{S}) \qquad \overline{M_\uparrow} = \overline{[s_\uparrow]}[\textbf{self}.m/m]\overline{M} \\ \langle \overline{M_f}, \overline{e_\uparrow} \rangle = \text{body}(\overline{S}) \qquad \ell_\downarrow = \text{first}(\text{last}(i)) \qquad i' = \text{add-subst}((\ell\ \textbf{as}\ \ell_\downarrow)/x, \overline{i})\end{array}}{\begin{array}{l}\langle \sigma, \overline{i}\ \textbf{inherits object}\,\{\,\overline{s_\uparrow}\ \overline{M}\ \overline{S}\,\}\ \textbf{as}\ x\ \overline{s}; e\rangle \rightsquigarrow \langle \text{update}(\sigma(\ell \mapsto \langle \varnothing, \overline{M_\uparrow}\ \overline{M_f}\rangle), \overline{M_\uparrow}\ \overline{M_f}, \overline{i'}),\\ \overline{[s_\uparrow]}[\langle \ell, \overline{M}\ \overline{M_f}, \overline{s_\uparrow}\rangle\ \overline{i'}/\textbf{super}][\ell_\downarrow/\textbf{self}]\overline{[\textbf{self}.m/m]e_\uparrow};\ \overline{[s]}[\textbf{self}.m/m][i'/\textbf{super}][(\ell\ \textbf{as}\ \ell_\downarrow)/x]e\rangle\end{array}}\ \overline{m}\text{ unique}$$

Extended Auxiliary Definitions

$$\text{body}(\textbf{inherits}\ e\ \textbf{as}\ x\ \overline{S}) = \langle \overline{M}, \textbf{super inherits}\ e\ \textbf{as}\ x\ \overline{e}\rangle \qquad \textbf{where}\ \langle \overline{M}, \overline{e}\rangle = \text{body}(\overline{S})$$

$$\text{add-subst}(s, \langle \ell, \overline{M}, \overline{s}\rangle\ \overline{i}) = \langle \ell, \overline{M}, \overline{s}\ s\rangle\ \overline{i}$$

$$\text{update}(\sigma, \overline{M_\uparrow}, \varnothing) = \sigma$$

$$\text{update}(\sigma, \overline{M_\uparrow}, \langle \ell, \overline{M}, \overline{s}\rangle\ \overline{i}) = \text{update}(\sigma(\ell \mapsto \langle \overline{F}, \overline{M'_\uparrow}\ \overline{M'}\ \overline{M'_\downarrow}\rangle), \overline{M'_\uparrow}\ \overline{M'}, \overline{i})$$

$$\textbf{where}\ \overline{m} = \text{names}(\overline{M}, \varnothing)\ \textbf{and}\ \overline{m_\uparrow} = \text{names}(\overline{M_\uparrow}, \varnothing)\ \textbf{and}\ \overline{M'_\uparrow} = \text{override}(\overline{M_\uparrow}, \overline{m})\ \textbf{and}\ \langle \overline{F}, \overline{M_\downarrow}\rangle = \sigma(\ell)$$

$$\textbf{and}\ \overline{m_\downarrow} = \text{names}(\overline{M_\downarrow}, \varnothing)\ \textbf{and}\ \overline{M'} = \overline{[s][\textbf{self}.m_\uparrow/m_\uparrow][\textbf{self}.m_\downarrow/m_\downarrow]M}\ \textbf{and}\ \overline{M'_\downarrow} = \text{override}(\overline{M_\downarrow}, \overline{m}\ \overline{m_\uparrow})$$

Figure 12 Positional uniform identity modification, with modified fresh rule omitted

again. The primary difficulty with implementing positional inheritance is that the meaning of local definitions can change imperatively: a local request to might refer to some definition in the surrounding scope, but after processing an inherits clause during the initialisation phase that request might now refer to an inherited definition instead. This presents a particular difficulty for substitution, which irreversibly binds an unqualified name to a particular definition. Substitutions are now delayed by all object expressions, as the inherits clauses are now nested in the object statements. Rule E-Object/Positional replaces the Rule E-Object, handling the new statement form with the updated body translation and applying the delayed substitution.

In order to implement the dynamic scoping, we introduce an *inherits context* i, which records the reference ℓ of the object that an inherits clause appeared inside, the source of the methods $\overline{M}$ that appeared directly in that object, and the delayed substitutions $\overline{s}$ that were on that object. Any processed inherits clause has an ordered stack of these contexts on it,

from the actual object the inherits clause appeared in, down to the bottom-most inheriting object. The **super** prefix is used as a placeholder on newly created inherits expressions, so that Rule E-Object/Positional can substitute is out for the initial inherits context. The **super** name has no other special meaning.

By retaining the source of the methods and the substitution scope they appeared in, the methods can be re-substituted in any new scope that appears. The update auxiliary function applies this for any newly inherited methods $\overline{M_{\uparrow}}$ to every object in the current stack of contexts. Rule E-Inherits/Positional handles any inherits expression, constructing a new part-object ℓ, and using update to include the new methods in the original object and every intervening part-object, after applying overrides from the existing methods. The value of **self** is bound in the inherited object body by the last object reference in the inherits context, as that is the location of the original object. The inherited body is processed in the same way as in Rule E-Object/Positional. One of the key distinctions here is that we already have the value of self to substitute in the inherited body, whereas under typical uniform identity the body is concatenated into an object expression, and self is substituted once that expression is evaluated.

Note that inherits expressions still delay substitutions, preventing them from applying to expressions later in any sequence. After each inherits expression is evaluated, the substitutions are then applied to the following expressions, after being shadowed by inherited definitions and the super-name defined by the **as** clause.

Positional uniform inheritance preserves the other traits of uniform identity from Section 5.2, with the exception of stability: during construction, an object's apparent type and behaviour can change. Applied to the simpler object inheritance models, it preserves each of their unique properties.

7 Discussion

Comparison Table 1 compares the models according to the criteria established in Section 2. Each model provides a different mix of the criteria, which may be appropriate for different circumstances or languages. The uniform identity design provides the closest match to Java semantics (given at the bottom of the table) while the other multiple-inheritance models following trade off one or more of these properties. No model provides every property; indeed, stability, downcalls, and inheriting from existing objects are fundamentally in conflict. As well, the complexity of each design and their implementation roughly increases down the table, which is a further trade-off for language designers to consider.

Planned Reuse. While delegation, forwarding, and concatenation can fundamentally support inheriting from arbitrary objects, the other models lean towards supporting planned reuse rather than ad-hoc reuse — that is, inheriting from objects that have been designed to be inherited from, rather than from any arbitrary object. Both planned and unplanned reuse have solid software-engineering motivations; indeed, language features exist both specifically to prevent inheritance (final or sealed classes) and to enable ad-hoc reuse (overlaid structural types).

We do not wish to present one or another choice as better, but to draw attention to a potentially-unintended side effect of various points in the solution space. Nonetheless, it is possible for any of the fresh-object–based systems to support delegation or forwarding semantics simply by exposing a method, accepting any object as an argument, that returns a fresh object whose methods provide the behaviour in question. Concatenation semantics can similarly be supported by inheriting from a standard clone.

	-first	Reg.	Down.	Dist.	Stable	Exist.	Mult.	Overl.	Par.
Forwarding	objects	no	no	yes	yes	yes	no	yes	no
Delegation	objects	no	no*	yes	no	yes	no	yes	no
Concatenation	objects	no	no*	no	no	yes	no	yes	no
Merged	objects	yes	no*	no	no*	fresh	no	yes	no
Uniform	objects	yes	yes	no	yes	fresh	no	yes	no
Mult. Uniform	objects	yes	yes	no	yes	fresh	yes	yes	no
Transform U.	objects	yes	yes	no	no	fresh	yes	no	no
Positional U.	objects	yes	yes	no	no	fresh	yes	yes	yes
Java	classes	yes	yes	no	yes	class	no	yes	no

Table 1 Comparison of models of object-first inheritance. * indicates answer holds during construction, but is reversed after. -first indicates the basis of the model. Overload indicates multiple definitions of the same method name in an object, accessible through super-references. Parent indicates ability to inherit from something obtained through another parent. All other columns relate to criteria from Section 2.

Diamonds. Diamond inheritance (repeatedly inheriting from the same class or trait two or more times) has long been recognised as a problem in object-oriented language design. Eiffel and C++ both offer the same essential solution to the problem: arranging that some classes can be replicated each time they are inherited, while other classes will be inherited only once. Malayeri and Aldrich [35] present a good discussion of the problems diamonds cause for inheritance, and then argue that diamond inheritance can be prevented in languages, partly by supporting a *requires* clause inspired by Scala which indicates that a trait depends upon the eventual final **self** object providing a set of methods, but not actually implementing those methods itself. The two multiple-inheritance systems we describe are open to this sort of collaboration, but do not require it. Because they are object-based, rather than class-based, some issues of diamond inheritance do not arise, as each instance of a parent is (unavoidably) separately obtained and constructed. In an object-based system, coalescing similar ancestors is a dubious activity, as side effects may occur on the path to construction and be semantically meaningful, which cannot happen in a static, declarative class system.

Errors In our formal model especially we have made a conscious effort to handle as many errors as possible in the operational semantics (i.e., at run time) rather than by defining erroneous programs as ill-formed (i.e., at compile time). There are several reasons for this, but most important is that we see further layers on top — such as type systems or checks for diamond inheritance — as an important, but separable part of the design process. Omitting such definitions highlights the inheritance designs, and enables the core language of the model to be smaller and more general. In fact, a language could provide the ability to impose additional families of static restrictions at the surface level [23].

Safe Initialisation. Similarly, we take a pragmatic approach to object initialisation: access to uninitialised variables raises a runtime error. This choice has helped simplify our inheritance designs, as we have not needed to contort the models to ensure that e.g. all variables are definitively initialised. Just as a range of static type systems can be layered over the language's semantics, so we expect that a safe initialisation scheme, such as Delayed Types [17], Masked Types [41], Hard Hats [20, 53], Freedom Before Committment [46], or the Billion Dollar Fix [42] should be able to to be layered on top of the initialisation semantics in the model. Highlighting where such runtime errors are liable to occur in the different designs helps language designers to choose where to trade off for or against additional safety.

8 Related Work

Class-based object-oriented languages begin with SIMULA [2], and much of the conceptual framework of object-orientation descends from the Scandanian school founded by Dahl and Nygaard, viewing programming as simulation, and consequently programs as models of phenomena in the real world [30]. Taivalsaari argues the class-based understanding of programming is also 'classical' in the sense of descending from the classical philosophy of Plato and Aristotle [48].

Simula's model of objects as instances of classes was greatly expanded on by Smalltalk [4]: unlike SIMULA, Smalltalk classes are also instances of other (meta-) classes themselves, importing all the power (and also all the complexity) of Lisp-style computational reflection into object-oriented languages [44]. Lisp then returned the favour with a series of object-oriented extensions culminating in the CLOS Meta-Object Protocol [28]. We also owe the notion of 'static' (per class) declarations, distinct from per-instance declarations, to Smalltalk.

The complexity of Smalltalk's meta-model clearly inspired Lieberman to propose languages based purely on objects, with delegation as the sharing mechanism [32], which then led to a general interest in 'prototype-based' programming languages, i.e. languages, following Self [50] that create new instances by copying existing instances, rather than by an apostrophe to a class. Emerald [3] also lacked classes, but objects were created by literal expressions.

The fates of Emerald and Self are instructive. By 1991 Emerald had a *"syntactic construct called a class that provides the functionality normally expected of classes"* [24]. At the same time, Self's programming style was based on separate objects corresponding to instance prototypes (defining all the instance variables in one object) and method suits (called "traits") with each conceptual class giving rise to both a trait, and a prototype delegating to that trait; traits were also linked by delegation corresponding to the conceptual class inheritance [49]. By 1995, Self objects were effectively given a single "copy down parent" attribute, and slots from that object would be copied into the object whenever it was edited. This is how e.g. if an extra 'z' slot was added into the graphic object, it would also be added into the gRectangle object. Eventually, a "Subclass me" button was added to the IDE, which copied both instance object and trait objects, recreating the class-like structure with delegation and copy-down parent links configured correctly.

Dony et al. modelled a range of different designs for class-, object-, and prototype- based languages, although by writing a software product line of definitional interpreters in Smalltalk, rather than modelling language features formally [13, 12]. Taivalsaari et al. [39] surveyed other contemporaneous research along these lines.

Relatively early on, Eiffel incorporated a sophisticated class-based multiple inheritance with deep renaming (rather than shallow aliasing) and exclusion ('undefine') and repeated inheritance [37]. Eiffel's design has many advantages, notably that an object implementing several different types can have multiple implementations of methods with the same name, depending on the type in which those definitions originate. Eiffel's development environment can generate the 'flat form' of the class: unfortunately because of Eiffel's type-aware semantics, the flattened form cannot represent all the possible behaviours of the class. Over the years, other languages incorporated many of the features of Eiffel's design, notably C++.

The other main stream of work is based on mixins, rather than classes [7]. Unlike CLOS or Eiffel multiple superclasses, mixins are applied one at a time, in a linear order specified by the programmer. Bracha's Jigsaw formalised mixin composition in a class-based style, along with a rich trait algebra including merge, restrict, select, project, overriding, and rename operators [8, 5]. Flatt et al. [18, 19] develop a semantics for classes and class-like mixins

(without composition operators) in a core language, and have incorporated mixins and traits into Scheme (now Racket) based on classes and macros. Lagorio et al. [29] modeled Jigsaw in a class-based formalism based on Featherweight Java [26], and then Corradi et al. [11] extended the formalism to handle family polymorphism [16]. More recently, class mixins have been incorporated into Newspeak (alongside family polymorphism) [9] and Dart [6].

Traits were revived for multiple inheritance in Smalltalk [14]: in this design, a class can inherit from one single superclass, and then incorporate multiple traits based with a trait algebra comprising sum, aliasing, exclusion operators. A key property of Smalltalk traits is that their conceptual model was based on flattening, rather than dynamic dispatch, although both semantics have been proven equivalent [38]. Scala [40] and Java 8 [22] incorporate traits, although relying on 'super' calls rather than aliasing or renaming.

A discussion of object-based inheritance systems would be incomplete without referring to OCaml [31], which is not dissimilar to the language of our base model. Both languages have object constructors, classes as sugar for methods returning fresh objects, symmetric multiple inheritance, and are structurally typed. We do not allow types to affect the operational semantics, however, and work on the theory of traditional object-style polymorphism rather than OCaml's row polymorphism.

More significantly, even if OCaml classes are described as syntatic sugar, objects (or other classes) can only inherit from classes, whereas in all of our models objects can inherit from (at least) any manifestly fresh objects whether or not defined by classes. OCaml also has a more complex, but in some ways less powerful, initialisation model than what we specify here. In OCaml, field initialisers are evaluated in the enclosing lexical scope, and initialisation code must be sequestered into initialisation blocks which are run late — neither of which reflect a straightforward reading of program's source code — while in our models initialisation is always in a straight line, within the context of the object being constructed.

9 Conclusion

Object-based inheritance is unexpectedly complicated, especially when commonplace desires for functionality available in classical models are involved, and programmers have resorted to increasingly complex workarounds in existing object-based languages. In this paper we showed that object inheritance without classes is both viable and desirable, avoiding the conceptual complexity of an additional kind of entity in an object-oriented language without losing functionality through careful feature selection, and set out a range of options with their various trade-offs made explicit.

We presented and discussed seven models of object inheritance, including the well-known approaches of delegation, forwarding, and concatenation. We presented a novel extended operational semantics for a base language incorporating advanced but standard features affected by inheritance, and formalised the models as extensions to that single base language, formally demonstrating the subtle behavioural differences of each model. In particular, we addressed the complex questions of downcalls, object registration, stability, inheriting preëxisting objects, action at a distance, and multiple inheritance, and their interactions, illustrating that object-based inheritance has the full range of possibilities of classical inheritance, and showed that many of these models can be used as effectively as purely declarative classes, but particular combinations — especially class initialisation semantics combined with inheritance from preëxisting objects — require a specific set of features usually reserved for very dynamic and reflective languages.

References

1 Henk Barendregt. *The Lambda Calculus*. North-Holland, revised edition, 1984.
2 G. M. Birtwistle, O. J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Studentlitteratur, 1979.
3 Andrew P. Black, Eric Jul, Norman Hutchinson, and Henry M. Levy. The development of the Emerald programming language. In *History of Programming Languages III*, 2007.
4 A. H. Borning. Classes versus prototypes in object-oriented languages. In *Proceedings of 1986 ACM Fall Joint Computer Conference*, 1986.
5 Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modules, and Multiple Inheritance*. PhD thesis, University of Utah, 1992.
6 Gilad Bracha. Mixins in Dart, October 2015.
7 Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA*, 1990.
8 Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *ICCL'92, Proceedings of the 1992 International Conference on Computer Languages*, 1992.
9 Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in Newspeak. In *ECOOP*, 2010.
10 Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are shared parts of objects: inheritance and encapsulation in Self. *Lisp and Symbolic Computation*, 4(3), 1991.
11 Andrea Corradi, Marco Servetto, and Elena Zucca. DeepFJig: modular composition of nested classes. In *PPPJ*, 2011.
12 Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based languages: From a new taxonomy to constructive proposals and their validation. In *OOPSLA*, October 1992.
13 Christophe Dony, Jacques Malenfant, and Pierre Cointe. Classifying prototype-based programming languages. In James Noble, Antero Taivalsaari, and Ivan Moore, editors, *Prototype-Based Programming: Concepts, Languages and Applications*, chapter 2. 1999.
14 Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *TOPLAS*, 2005.
15 Carlton Egremont III. *Mr. Bunny's Big Cup o' Java*. Addison-Wesley, 1999.
16 Erik Ernst. Family polymorphism. In *ECOOP*, 2001.
17 Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In *OOPSLA*, 2007.
18 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL*, 1998.
19 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, 1998.
20 Joseph(Yossi) Gil and Tali Shragai. Are we ready for a safer construction environment? In *ECOOP*, 2009.
21 Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
22 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Oracle, 2015.
23 Michael Homer, Timothy Jones, James Noble, Kim B. Bruce, and Andrew P. Black. Graceful dialects. In *ECOOP*, pages 131–156, 2014.
24 Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald programming language report. Computer Science, UBC, October 1991.
25 Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *HOPL-III*, 2007.
26 A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLaS*, 23(3):396–450, 2001.

27 Timothy Jones and James Noble. Tinygrace: A simple, safe and structurally typed language. In *FTFJP*. ACM, New York, NY, USA, 2014.

28 Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

29 Giovanni Lagorio, Marco Servetto, and Elena Zucca. Featherweight Jigsaw: Replacing inheritance by composition in Java-like languages. *Inf. Comput.*, 214:86–111, 2012.

30 Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

31 Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.00 documentation and user's manual, 2012.

32 Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *OOPSLA*, November 1986.

33 Henry Lieberman, Lynn Andrea Stein, and David Ungar. Treaty of Orlando. In *Addendum to OOPSLA Proceedings*, May 1988.

34 Lua-Users. Object oriented programming, 2014. [Online; accessed 30-November-2015].

35 Donna Malayeri and Jonathan Aldrich. Cz: Multiple inheritance without diamonds. In *OOPSLA*, 2009.

36 Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

37 Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

38 Oscar Nierstrasz, Stéphane Ducasse, and Nathanael Schärli. Flattening traits. *Journal of Object Technology*, 5:66–90, 2006.

39 James Noble, Antero Taivalsaari, and Ivan Moore, editors. *Prototype-Based Programming: Conecepts, Languages, Applications*. Springer-Verlag, 1997.

40 Martin Odersky, Lex Spoon, and Bill Venners. *Programming In Scala*. artima, 2011.

41 Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In *POPL*, 2009.

42 Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. The billion-dollar fix—safe modular circular initialisation with placeholders and placeholder types. In *ECOOP'13*.

43 Pat Shaughnessy. *Ruby Under A Microscope*. No Stach Press, 2013.

44 Brian C. Smith. Reflection and semantics in Lisp. In *Proceedings of the ACM Conference on Principles of Programming Languages*, 1984.

45 Stack Overflow. Which JavaScript library has the most comprehensive class inheritance support?, 2015. [Online; accessed 30-November-2015].

46 A. J. Summers and P. Müller. Freedom before commitment - a lightweight type system for object initialisation. In *OOPSLA*, 2011.

47 Antero Taivalsaari. Delegation versus concatenation or cloning is inheritance too. *OOPS Messenger*, 6(3), 1995.

48 Antero Taivalsaari. Classes vs. prototypes: Some philosophical and historical observations. In James Noble, Antero Taivalsaari, and Ivan Moore, editors, *Prototype-Based Programming: Concepts, Languages and Applications*, chapter 1. Springer-Verlag, 1999.

49 David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp and Symbolic Computation*, 4(3), June 1991.

50 David Ungar and Randall B. Smith. SELF: the Power of Simplicity. *Lisp and Symbolic Computation*, 4(3), June 1991.

51 Wikipedia. Snit, 2015. [Online; accessed 30-November-2015].

52 Allen Wirfs-Brock, editor. *ECMAScript 2015 Language Specification*. Ecma International, 6th edition, 2015.

53 Yoav Zibin, David Cunningham, Igor Peshansky, and Vijay Saraswat. Object initialization in X10. In *ECOOP*, 2012.