First-Class Object Sets

Erik Ernst

University of Aarhus, Denmark
eernst@cs.au.dk

Abstract. Typically, objects are monolithic entities with a fixed interface. To increase the flexibility in this area, this paper presents first-class object sets as a language construct. An object set offers an interface which is a disjoint union of the interfaces of its member objects. It may also be used for a special kind of method invocation involving multiple objects in a dynamic lookup process. With support for feature access and late-bound method calls object sets are similar to ordinary objects, only more flexible. The approach is made precise by means of a small calculus, and the soundness of its type system is shown by a mechanically checked proof in Coq.

Key words: Object sets, composition, multi-object method calls, types.

1 Introduction

In an object-oriented setting the main concept is the object. It is typically a monolithic entity with a fixed interface, such as a fixed set of messages that the object accepts. This paper presents a language design where sets of objects are first-class entities, equipped with operations that enable such object sets to work in a similar way as monolithic objects. An object set offers an interface to the environment which is a disjoint union of the interfaces of its members, and it supports cross-member operations, known as *object set method calls*, which are similar in nature to late-bound method calls on monolithic objects. The object set thus behaves in a way which resembles the behavior of a monolithic instance of a 'large' class that combines all the classes of the members of the object set, e.g., by mixin composition or multiple inheritance.

Object sets are useful because they are more flexible than such a monolithic instance of a large class: There are no restrictions on which classes may be put together in the creation of an object set, and there is no need to declare a large, composite class and refer to that class by name everywhere. In fact, object set types are structural with member class granularity—the type of an object set is a set of classes, and every subset is a supertype. Moreover, object sets could be modified during their life time, which would correspond to a change of class in the monolithic case.

On the other hand, access to a feature of an object set requires explicit selection of the member class which provides this feature, and the object set method call mechanism is quite simple rather than convenient. This is because

the emphasis in this language design has been put on expressing the required *primitives*, rather than giving the design of a fragment of a convenient and pragmatic programming language.

In fact, work on the implementation of the language gbeta [1–3] served as our starting point for the design of object sets. Objects in gbeta have a semantics which may be represented by collections of instances of mixins, i.e., as multi-entity phenomena rather than monolithic entities. Like object sets, they provide an interface which is a disjoint union of the interfaces of the included mixins, but unlike object sets there is no need to specify explicitly which mixin to use when accessing a feature. Like object sets, gbeta objects can have cross-entity features (such as methods or inner classes, which are then known as virtual), but unlike object sets these features are accessed in exactly the same way as single-mixin features. Similarly, since all gbeta objects are conceptually object sets there is no distinction (syntactically or otherwise) between the usage of object sets and "ordinary objects".

The language gbeta also supports dynamic change of class for existing objects, and this corresponds to the replacement of the contents of the object by a larger object set. In context of the features included in this paper this operation is simple and safe, though of course it would require addition of mutable references to object sets to make it work as a dynamic change of class. In context of gbeta it is considerably more complex because dynamic specialization of an object may have effects that correspond to a dynamic replacement of actual type arguments of the class of the (multi-entity) object by some subtypes, which may cause a run-time error, e.g., because the value of an instance variable may thus become type incorrect. Because of this, dynamic object specialization in gbeta has been extended with restricted versions that are safe, but it is beyond the scope of this paper to model these refinements of the concept. Nevertheless, it is worth noting that it is possible to embody the primitives presented in this paper into a full-fledged programming language in such a way that they are convenient to use.

The contributions of this paper is the notion of object sets, the precise definition of their semantics and typing in a formal calculus, $\mathrm{FJ}_{\mathrm{set}}$, and the mechanically checked proof of soundness [4] for this calculus, using the Coq [5] proof assistant.

The rest of the paper is organized as follows: Section 2 presents the calculus informally and discusses the design. Next, Sect. 3 gives the formal definitions, and Sect. 4 describes the soundness result. Finally, Sect. 5 describes related work, and Sect. 6 concludes.

2 An Informal Look at the $\mathbf{FJ}_{\mathbf{set}}$ Calculus

The $\mathrm{FJ}_{\mathrm{set}}$ calculus is derived from the Featherweight Java calculus [6] by adding the object set related operations, allowing covariant method return types, and removing casts. The covariant method return types are included because they

are useful and standard today, and the casts are left out because they do not provide extra benefits in this context.

The crux of this calculus is of course the ability to express and use object sets. An object set is a set of objects collected into a single, typed entity. An object set may be decomposed in order to use individual members of the set, and used as a whole in a special kind of method call, the *object set method call*. Each object set is associated with a set of classes and each object in the set is uniquely associated with one particular class in the set of classes. Another way to describe this would be to say that each object in the set is *labeled* by a class. The object is an instance of that class or a subclass thereof. This makes it possible to access the object set members and to use each one of them according to an interface that it is known to support.

The correspondence between objects and classes in an object set is maintained by considering the set of objects and the set of classes as lists, and pairing up the lists element by element. This is possible for an object set creation expression (a variant of the well-known `new` expression for monolithic objects) because such an expression contains the two lists syntactically, and this ensures that every object set from its creation has a *built-in definition of the mapping* from classes to member objects. It also equips the members of the object set with an ordering. This ordering is insignificant with respect to typing, but it is significant with respect to the dynamic semantics, because it determines which method implementation is most specific during an object set method call. In other words, the ordering of the members of an object set is an implementation detail—crucial in the definition of its internal structure and behavior, but encapsulated and invisible at the level of types.

However, an expression denoting an object set may by subsumption be typed with an arbitrary subset of the associated classes, and they may be listed in an arbitrary order in the type. In other words, it is possible to forget some of the objects and also to ignore the ordering of the objects. However, the dynamic semantics only operates on an object set when it has been evaluated to such an extent that it is an object set creation at top level. This ensures that the object set operations are consistent because they are based on the built-in mapping.

Two operations are provided to decompose an object set. They both rely on addressing a specific member of the set via its associated class. One operation provides access to the object associated with the given class, and the other operation deletes that object and class from the object set, thus producing a smaller object set.

The object set method call operation is provided in order to gather contributions from all suitable objects in an object set, in a process that resembles a fold operation on a list. The call is based on an ordinary method whose signature must follow a particular pattern, namely that it takes a positive number of arguments and the type of the first argument is identical to the return type of the method. This makes it possible for the method to accept an arbitrary list of "ordinary" arguments—the arguments number two and up—and also to accept and return a value which plays the role as an accumulator of the final result. In this

```
1  class Printable extends Object {
2      String print(String s) { return "Plain Printable"; }
3      String separator() { return ", "; }
4  }
5  class Agent extends Printable {
6      String id;
7      String print(String s) { return s+" "+id; }
8  }
9  class Person extends Printable {
10     String name;
11     String print(String s) { return name+" "+s; }
12     String separator() { return " -- "; }
13 }
14 class Main extends {
15     { Printable } p;
16     String doPrint() {
17         return p.print@Agent("The name is") +
18                p@Printable.separator() +
19                p.print@Printable("");
20     }
21 }
22
23 // the following yields "The name is Bond -- James Bond"
24 new Main(new {Agent, Printable} (new Agent("Bond"),
25                                  new Person("James")))
26 .doPrint()
```

Fig. 1. An small example program in $\mathrm{FJ}_{\mathrm{set}}$.

sense the object set method call supports iteration over the selected members of the object set and collection of contributions to the final result, not unlike a folding operator applied to a list. Note that an object set method call does not require static knowledge about the type of any of the objects in the set.

The design of the object set method call mechanism was chosen to enable iteration over a subset of the members of the object set supporting a specific interface, without adding extra language mechanisms. Pragmatically, it might be more natural to use actual iteration in an imperative setting, or to return a data structure like an array containing the eligible object set members. But in this context we prefer a minimal design, and hence we ended up chosing the programmer convention driven approach based on ordinary nested method calls.

Figure 1 shows a small example program in $\mathrm{FJ}_{\mathrm{set}}$. This program shows how to create objects and object sets, how to perform an object set method call, how to decompose an object set in order to use a feature of one of its members, and it indicates the result of the computation. In order to make the example compact and readable it uses an extension of the calculus that includes a `String` type, string literals, and concatenation of strings with the '`+`' operator.

Lines 1–13 define three classes to support modeling a human being from two different points of view in an object set; the only difference from standard Java code is that there are no constructors, but the constructors in FJ style calculi are trivial and somewhat of an anomaly so we left them out. Note that the signature of the `print` method is such that it can be used for object set method calls: Its return type is also the type of the first (and only) argument.

The class `Main` has an instance variable (line 15) whose type is an object set, { `Printable` }, which means that it is guaranteed that there is an object labeled `Printable` in this object set, but there may be other objects as well. The `doPrint` method (line 16–20) makes two object set method calls (line 17 and 19) and one ordinary method call (line 18), and returns the concatenation of the results. The object set method call on line 17 involves only one member of `p`, because only the first one is labeled by `Agent` or a subclass thereof. The call on line 19 involves both objects in `p`. The expression `p@Printable` on line 18 extracts the object labeled as `Printable` in `p`, which is the `Person` object, and calls its `separator` method, which by ordinary late binding returns `" -- "`.

Finally, note that subsumption makes it possible for the instance variable `p` to refer to an object set of type { `Agent, Printable` }, and also that the usage of different classes in the object set method call can be used to filter the contributors to such a call in various ways.

3 The $\mathrm{FJ}_{\mathbf{set}}$ Calculus

We now proceed to present the syntax, the dynamic semantics, and the type system of the $\mathrm{FJ}_{\mathrm{set}}$ calculus, interspersed with short discussions about why the calculus is designed the way it is. We also give some remarks on how the presentation in this paper and the accompanying Coq proof fit together, based on the assumption that this kind of knowledge is useful for the development of a strong culture of using proof assistant software.

3.1 Syntax and Notation

A program is a class table and a main expression, and the semantics of a program is to execute the main expression in context of the given classes. As is common, we assume the existence of a fixed, globally accessible class table, `CT`, which lists all the class definitions in the program.

The syntax of the calculus from the level of classes and down is shown in Fig. 2. Notationally, we use overbars to denote lists of terms, so $\overline{\mathtt{C}}$ stands for the list $\mathtt{C}_1\ \mathtt{C}_2 \ldots \mathtt{C}_n$ for some natural number n; n=0 is allowed and yields the empty list, '•'. There may be separators such as commas or semicolons between the elements of such a list, but they are implicit and implied by the context.

Several constructs in the syntax are identical to the ones known from Featherweight Java. Class and method definitions are standard, using the variant of Featherweight Java that omits explicit constructors. The standard expressions are variables, field lookups, method calls, and `new` expressions.

`Q`	::=	`class C extends D {` $\overline{\texttt{T f}}$`;` $\overline{\texttt{M}}$ `}`	*class declarations*
`M`	::=	`T m(`$\overline{\texttt{T x}}$`) { return e; }`	*method declarations*
`e`	::=	`x` | `e.f` | `e.m(`$\overline{\texttt{e}}$`)` | `e@C` | `e\C` | `new C(`$\overline{\texttt{e}}$`)` | `new {`$\overline{\texttt{C}}$`} (`$\overline{\texttt{e}}$`)` | `e.m@C(`$\overline{\texttt{e}}$`)`	*expressions*
`v`	::=	`new C(`$\overline{\texttt{v}}$`)` | `new {`$\overline{\texttt{C}}$`} (`$\overline{\texttt{v}}$`)`	*values*
`T,U`	::=	`C` | `{`$\overline{\texttt{C}}$`}`	*types*
`Object,this`			*predefined names*
`C,D`			*class names*
`f,g`			*field names*
`x`			*variable names*
`m`			*method names*
`N`			*any kind of name*

Fig. 2. Syntax of FJ_{set}.

The remaining expressions are concerned with object sets. A class selection expression, `e@C`, provides the object labeled with the class `C` from the object set `e`. A class exclusion expression, `e\C`, provides an object set from which the object labeled with `C` as well as `C` itself has been deleted. The expression `new {`$\overline{\texttt{C}}$`} (`$\overline{\texttt{e}}$`)` denotes creation of an object set which contains each of the objects denoted by the expression list $\overline{\texttt{e}}$, labeled by the list of classes $\overline{\texttt{C}}$.

Finally, the expression `e.m@C(`$\overline{\texttt{e}}$`)` denotes an object set method call, which selects all objects from the object set `e` which are labeled with the class `C` or a subclass thereof, and calls a method `m` on each of them in the order they appear in the class list of the built-in mapping of the object set `e`. The method `m` must be defined in or inherited by the class `C`, and it must take a non-zero number of arguments where the first argument has the same type as the method return type, in order to enable the nested method call process mentioned in Sect. 1.

3.2 Auxiliary methods, Subtyping, and Wellformedness

Figure 3 defines the auxiliary functions used for field lookup and similar tasks. They are standard except for the function *distinct* which simply expresses that a given list of names (of any kind such as class names, method names, etc.) are distinct. As is common, quoting a class definition as a premise of a rule indicates the requirement that `CT` must contain that class definition.

The rules in Fig. 4 show subclassing ($\vdash$ `C` $\sqsubset$: `D`), which is standard, subtyping for object sets ($\vdash$ `T` $\subset$: `U`), which corresponds to the superset relation among the sets, and subtyping, which combines the two. Furthermore the judgement `C` $\in$: `{`$\overline{\texttt{C}}$`}` holds whenever `C` is a superclass of one of the classes $\overline{\texttt{C}}$; this is used in the dynamic semantics of object set method calls.

$$fields(\texttt{Object}) = \bullet \qquad \frac{\texttt{class C extends D } \{\overline{\texttt{T f; }}\ \overline{\texttt{M}}\} \quad fields(\texttt{D}) = \overline{\texttt{U g}}}{fields(\texttt{C}) = \overline{\texttt{U g}}, \overline{\texttt{T f}}}$$

$$\frac{\texttt{class C extends D } \{\overline{\texttt{U f; }}\ \overline{\texttt{M}}\} \quad \texttt{T m(}\overline{\texttt{T x}}\texttt{) } \{\texttt{return e;}\} \in \overline{\texttt{M}}}{mBody(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}.\texttt{e}} \qquad \frac{\texttt{class C extends D } \{\overline{\texttt{U f; }}\ \overline{\texttt{M}}\} \quad \texttt{m} \notin \overline{\texttt{M}} \quad mBody(\texttt{m}, \texttt{D}) = \overline{\texttt{x}}.\texttt{e}}{mBody(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}.\texttt{e}}$$

$$\frac{\texttt{class C extends D } \{\overline{\texttt{U f; }}\ \overline{\texttt{M}}\} \quad \texttt{m} \notin \overline{\texttt{M}} \quad mType(\texttt{m}, \texttt{D}) = (\overline{\texttt{T}} \to \texttt{T})}{mType(\texttt{m}, \texttt{C}) = (\overline{\texttt{T}} \to \texttt{T})} \qquad \frac{\texttt{class C extends D } \{\overline{\texttt{U f; }}\ \overline{\texttt{M}}\} \quad \texttt{T m(}\overline{\texttt{T x}}\texttt{) } \{\texttt{return e;}\} \in \overline{\texttt{M}}}{mType(\texttt{m}, \texttt{C}) = (\overline{\texttt{T}} \to \texttt{T})}$$

$$distinct(\bullet) \qquad \frac{\texttt{N} \notin \overline{\texttt{N}} \quad distinct(\overline{\texttt{N}})}{distinct(\texttt{N}\ \overline{\texttt{N}})}$$

Fig. 3. Auxiliary functions for FJ_{set}.

In the Coq formalization of the calculus, transitivity for subclassing includes the requirement that the two pairs of classes are distinct, i.e., that $\texttt{C} \neq \texttt{C}''$ and $\texttt{C}'' \neq \texttt{C}'$. An easy induction shows that each of the two definitions of subclassing is able to derive all the subclass judgements of the other. However, in order to show in Coq that subclassing is decidable, the addition of these requirements solves a problem because it is hard to specify in Coq that the derivation tree for a subtyping judgement is finite. As we shall see later on, the required finiteness is a consequence of the rule for class typing.

The requirement in the rule for object set subtyping that the classes $\overline{\texttt{D}}$ are distinct is necessary in order to prevent occurrences of class lists with duplicate elements. It is only required for the supertype, $\{\overline{\texttt{D}}\}$, because distinctness for the subtype is ensured by other rules, in particular in the typing of object set creation expressions shown below in Fig. 7.

The type wellformedness requirements are shown in Fig. 5. They state that a class name is well-formed if there is a class of that name in the class table, and that an object set type, $\{\overline{\texttt{C}}\}$, is well-formed if it consists of distinct class names. Finally CT must satisfy $\texttt{Object} \notin \texttt{CT}$. Note that the class names in an object set type are not explicitly required to be defined in CT, because this requirement is a consequence of other rules. In general, the wellformedness requirements in this calculus are sufficient to enable the proof of soundness, but they are also minimal in the sense that removing any of them invalidates the proof. We believe that the use of proof assistent software may tighten the specification of well-formedness requirements in calculi, which is an area that otherwise easily gets a slightly imprecise treatment.

$$\frac{\texttt{class C extends D } \{\ \overline{\texttt{T f}};\ \overline{\texttt{M}}\ \}}{\vdash \texttt{C} \sqsubset: \texttt{D}} \qquad \frac{\vdash \texttt{C} \sqsubset: \texttt{C}'' \qquad \vdash \texttt{C}'' \sqsubset: \texttt{C}'}{\vdash \texttt{C} \sqsubset: \texttt{C}'}$$

$$\vdash \texttt{C} \sqsubset: \texttt{C} \qquad \frac{\overline{\texttt{D}} \subseteq \overline{\texttt{C}} \qquad \mathit{distinct}(\overline{\texttt{D}})}{\vdash \{\overline{\texttt{C}}\} \subset: \{\overline{\texttt{D}}\}}$$

$$\frac{\vdash \texttt{T} \sqsubset: \texttt{U}}{\vdash \texttt{T} <: \texttt{U}} \qquad \frac{\vdash \texttt{T} \subset: \texttt{U}}{\vdash \texttt{T} <: \texttt{U}} \qquad \frac{\vdash \texttt{C}_i \sqsubset: \texttt{C}}{\texttt{C} \in: \{\overline{\texttt{C}}\}}$$

Fig. 4. Subclassing and subtyping for $\mathrm{FJ}_{\mathrm{set}}$.

$$\vdash \texttt{Object}\ \textsc{ok} \qquad \frac{\texttt{C} \in \texttt{CT}}{\vdash \texttt{C}\ \textsc{ok}} \qquad \frac{\mathit{distinct}(\overline{\texttt{C}})}{\vdash \{\overline{\texttt{C}}\}\ \textsc{ok}}$$

Fig. 5. Wellformedness rules for $\mathrm{FJ}_{\mathrm{set}}$.

3.3 Expression Evaluation

The dynamic semantics of $\mathrm{FJ}_{\mathrm{set}}$ is presented in Fig. 6. Selection of a redex in a larger expression is defined in terms of evaluation contexts rather than congruence rules; they are listed at the bottom of the figure, where $\texttt{E}$ denotes an expression with exactly one hole and $\texttt{E}^*$ denotes a (non-empty) list of expressions with exactly one hole. With respect to the evaluation order, this calculus follows the tradition from FJ whereby the evaluation order is restricted as little as possible, and particular strategies like call-by-value are available as one of the possible choices.

The rules for field lookup and method invocation are standard. The rule for class selection, (R-Select), selects the member of the given object set labeled with the specified class. This rule serves as an example of the evaluation order issue mentioned above: evaluation has to proceed until the top level expression is an object set creation expression ($\texttt{new}\ \{\overline{\texttt{C}}\}\ \texttt{(...)}$) in order to reveal $\{\overline{\texttt{C}}\}$ and thus the built-in mapping of the object set, but the arguments need not be fully evaluated.

We stated earlier that an object set offers an interface which is a disjoint union of the interfaces of its members. The class selection operation fulfills this promise as follows: For a given object set, the classes used to label some members of the object set are made explicit in its type (others may have been lost by subsumption). The interface of the object set is the union of the interfaces of these classes, and thus the object set supports access to all these features of

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}\ \mathtt{f}}}{\mathtt{new\ C}(\overline{\mathtt{e}})\mathtt{.f}_i \rightsquigarrow \mathtt{e}_i}$$

(R-Field)

$$\mathtt{(new\ \{}\overline{\mathtt{C}}\mathtt{\}\ (}\overline{\mathtt{e}}\mathtt{))\backslash C}_i \rightsquigarrow \mathtt{new\ \{}\overline{\mathtt{C}}\mathtt{\backslash\#i\}\ (}\overline{\mathtt{e}}\mathtt{\backslash\#i)}$$

(R-Drop)

$$\frac{\mathit{mBody}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}_0}{\mathtt{(new\ C(}\overline{\mathtt{e}}\mathtt{))\ .m(}\overline{\mathtt{e'}}\mathtt{)} \rightsquigarrow \mathtt{[this/new\ C(}\overline{\mathtt{e}}\mathtt{),}\overline{\mathtt{x/e'}}\mathtt{]e}_0}$$

(R-Invk)

$$\frac{\begin{array}{c}\vdash \mathtt{C}_i \sqsubset\colon \mathtt{C} \qquad i = \mathit{min}\{\ j \mid \vdash \mathtt{C}_j \sqsubset\colon \mathtt{C}\ \} \\ \mathtt{v}_i = \mathtt{new\ D(}\overline{\mathtt{v'}}\mathtt{)} \qquad \mathit{mBody}(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{x}}.\mathtt{e}_0 \\ \mathtt{e}'_0 = \mathtt{[this/new\ D(}\overline{\mathtt{v'}}\mathtt{),}\overline{\mathtt{x}}\mathtt{/}\overline{\mathtt{e}}\mathtt{]e}_0\end{array}}{\begin{array}{c}\mathtt{(new\ \{}\overline{\mathtt{C}}\mathtt{\}\ (}\overline{\mathtt{v}}\mathtt{))\ .m@C(}\overline{\mathtt{e}}\mathtt{)} \rightsquigarrow \\ \mathtt{(new\ \{}\overline{\mathtt{C}}\mathtt{\backslash\#i\}\ (}\overline{\mathtt{v}}\mathtt{\backslash\#i))\ .m@C(e'}_0\overline{\mathtt{e}}\mathtt{)}\end{array}}$$

(R-SInvk)

$$\mathtt{(new\ \{}\overline{\mathtt{C}}\mathtt{\}\ (}\overline{\mathtt{e}}\mathtt{))@C}_i \rightsquigarrow \mathtt{e}_i$$

(R-Select)

$$\frac{\mathtt{C} \not\sqsubset\colon \{\overline{\mathtt{C}}\}}{\mathtt{(new\ \{}\overline{\mathtt{C}}\mathtt{\}\ (}\overline{\mathtt{v}}\mathtt{))\ .m@C(e,}\overline{\mathtt{e}}\mathtt{)} \rightsquigarrow \mathtt{e}}$$

(R-SInvk-Done)

$$\begin{array}{lcl}\mathtt{E} & ::= & \mathtt{[\_]} \mid \mathtt{E.f} \mid \mathtt{E.m(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{e.m(E^*)} \mid \mathtt{new\ C(E^*)} \mid \mathtt{new\ \{}\overline{\mathtt{C}}\mathtt{\}\ (E^*)} \mid \mathtt{E@C} \mid \\ & & \mathtt{E\backslash C} \mid \mathtt{E.m@C(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{e.m@C(E^*)} \\ \mathtt{E^*} & ::= & \overline{\mathtt{e}}\ \mathtt{E}\ \overline{\mathtt{e'}}\end{array}$$

Fig. 6. Evaluation rules and evaluation contexts for $\mathrm{FJ}_{\mathrm{set}}$.

all those members. There are no naming conflicts because the choice of class is made explicit, i.e., it is a disjoint union. In a full-fledged language it is much more convenient to avoid the explicit class selection, but this is trivial in the cases where there is no naming conflict, and it should be handled explicitly when a conflict exists; the language gbeta uses this approach.

The rule for class exclusion, (R-Drop), deletes the requested class and the corresponding member from the object set. This rule introduces notation for a simple function that deletes the i'th element from a list, namely $\overline{\mathtt{t}}\mathtt{\backslash\#i}$, where $\overline{\mathtt{t}}$ are terms of any kind, e.g., class names or expressions. Usage of this notation implies that the list is long enough to contain the position to delete.

In the Coq formalization of this calculus the (R-Drop) rule zips the list of classes and the list of expressions together to a list of pairs, then deletes the pair which contains the specified class, and then unzips the shortened list of pairs to get the resulting list of classes and list of expressions. This is a relatively typical situation where the convenient formalization in Coq does not correspond exactly to a well-known or convenient notation for presentation in a paper, but the slightly awkward notation $\overline{\mathtt{t}}\mathtt{\backslash\#i}$ used to express deletion-by-position seems to be the most readable way to bridge the gap.

The object set method call semantics is specified by two rules, (R-SInvk) and (R-SInvk-Done). As mentioned, an object set method call amounts to a composite operation which includes a method call on each of the members of the set labeled by a class supporting that method. The rule (R-SInvk) specifies what to do when the object set contains a member supporting the requested method `m`, and the rule (R-SInvk-Done) specifies what to do in the end when all such objects have been processed.

Whether an object set member supports `m` is determined by requiring that the member is associated with a subclass of the class `C` specified in the object set call. This means that each selected object will be an instance of `C` or one of its subclasses, and the method `m` will be defined for that object, with a signature which is identical to the signature of `m` in `C`, except for possible covariance in the return type. Objects supporting unrelated methods with the same name `m` are ignored.

The rule (R-SInvk) specifies how to call one method `m` and provide the results produced by this method call to the next method call. It requires that the list of classes $\overline{\texttt{C}}$ associated with the object set contains a subclass $\texttt{C}_i$ of the class requested in the call, `C`, and selects the 'first' one (the one with the smallest index i). It then removes the selected object from the receiver object set and repeats the object set method call with the result of the invocation of `m` on the selected object as its first argument. Note that the minimality of i is not needed for soundness, it is needed in order to ensure that object set method calls have a predictable semantics: it should accumulate the results from its members according to their built-in ordering.

However, the first argument does not look like a method call, it is actually given as $\texttt{[this/new D(}\overline{\texttt{v}'}\texttt{),}\overline{\texttt{x}}\texttt{/}\overline{\texttt{e}}\texttt{]e}_0$, but inspection of the rule for method call, (R-Invk), reveals that this is the result of taking one evaluation step after the method invocation $\texttt{new D(}\overline{\texttt{v}'}\texttt{).m(}\overline{\texttt{e}}\texttt{)}$. It is necessary to express the rule in this form in order to maintain the property that all rules are compositional.

A similar investigation shows that the receiver of the object set method call after the step in (R-SInvk) is the result of taking one step after excluding the selected class $\texttt{C}_i$ from the receiver object set before the step. Compositionality again forces the rule to take that step rather than expressing the result in terms of an explicit class exclusion operation.

The semantics of an object set method call may thus seem to be expressible in terms of other operations, but this is not the case because there is no way to select the class $\texttt{C}_i$ appropriately without this operation. A primitive could be provided in order to make such a selection, but we have not found any which enables the same functionality without requiring strictly more static knowledge about the contents of object sets.

Finally, the rule (R-SInvk-Done) yields the first argument of the object set method call in the situation where no object in the object set can be selected.

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash \mathtt{x} : \mathtt{T}}$$

(T-Var)

$$\frac{\Gamma \vdash \mathtt{e} : \{\overline{\mathtt{C}}\} \qquad \vdash \mathtt{C}_i \text{ OK}}{\Gamma \vdash \mathtt{e@C}_i : \mathtt{C}_i}$$

(T-Select)

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C} \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}\ \mathtt{f}}}{\Gamma \vdash \mathtt{e.f}_i : \mathtt{T}_i}$$

(T-Field)

$$\frac{\Gamma \vdash \mathtt{e} : \{\overline{\mathtt{C}}\}}{\Gamma \vdash \mathtt{e} \backslash \mathtt{C}_i : \{\overline{\mathtt{C}} \backslash \mathtt{\#i}\}}$$

(T-Drop)

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C} \qquad \mathit{mType}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{T}} \to \mathtt{T}) \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \vdash \overline{\mathtt{U}} <: \overline{\mathtt{T}}}{\Gamma \vdash \mathtt{e.m(\overline{e})} : \mathtt{T}}$$

(T-Invk)

$$\frac{\mathit{distinct}(\overline{\mathtt{C}}) \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \vdash \overline{\mathtt{U}} <: \overline{\mathtt{C}}}{\Gamma \vdash \mathtt{new}\ \{\overline{\mathtt{C}}\}\ \mathtt{(\overline{e})} : \{\overline{\mathtt{C}}\}}$$

(T-SNew)

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}\ \mathtt{f}} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \vdash \overline{\mathtt{U}} <: \overline{\mathtt{T}}}{\Gamma \vdash \mathtt{new\ C(\overline{e})} : \mathtt{C}}$$

(T-New)

$$\frac{\Gamma \vdash \mathtt{e} : \{\overline{\mathtt{C}}\} \qquad \Gamma \vdash \overline{\mathtt{e}} : \mathtt{T}\ \overline{\mathtt{T}} \qquad \mathit{mType}(\mathtt{m}, \mathtt{C}) = (\mathtt{T}\ \overline{\mathtt{T}} \to \mathtt{T})}{\Gamma \vdash \mathtt{e.m@C(\overline{e})} : \mathtt{T}}$$

(T-SInvk)

Fig. 7. Type rules for $\mathrm{FJ}_{\mathrm{set}}$.

3.4 Typing

The type rules for $\mathrm{FJ}_{\mathrm{set}}$ are shown in Fig. 7. The rules for the typing of variables, field lookups, ordinary method invocations, and ordinary object creation are standard.

The rule (T-Select) specifies that the target must be typable as an object set containing the requested class, and the resulting type is then that class. It would be easy to change this rule and (R-Select) to select a subclass, i.e., to allow for the selection of a class $\mathtt{C}$ as long as $\mathtt{C} \in: \{\overline{\mathtt{C}}\}$, but this could prevent the selection of a class $\mathtt{C}'$ from an object set that is also associated with some subclass $\mathtt{C}''$ of $\mathtt{C}'$ or make the operation ambiguous, and since there is no depth subtyping for object set types it would not enhance the expressive power or the flexibility of the language.

The rule (T-Drop) specifies that the target must be typable as an object set that includes the class to exclude, which is then removed from the type of the object set to produce the result type. For the same reasons as above it would not be useful to allow the requested class to be a superclass of the excluded class. The rule (T-SNew) specifies the typing of object set creations. It simply requires that the classes used as labels are distinct and that each member has a subtype of its associated class. The rule (T-SInvk) specifies how to type object set method calls. It requires that the receiver is typable as an object set, but

$$\frac{\mathit{override}(\mathtt{m},\mathtt{D},\mathtt{T},\overline{\mathtt{T}}) \quad \Gamma;\ \mathtt{this}{:}\mathtt{C} \vdash \mathtt{e} : \mathtt{U} \quad \vdash \mathtt{U} <: \mathtt{T} \quad \vdash \mathtt{T},\overline{\mathtt{T}}\ \text{OK} \quad \mathit{distinct}(\overline{\mathtt{x}})}{\vdash \mathtt{T}\ \ \mathtt{m}(\overline{\mathtt{T}\,\mathtt{x}})\{\ \mathtt{return\ e;}\ \}\ \text{OK in } \mathtt{C},\mathtt{D}}$$

(T-Method)

$$\frac{\vdash \mathtt{D}\ \text{OK} \quad \vdash \overline{\mathtt{T}}\ \text{OK} \quad \vdash \overline{\mathtt{M}}\ \text{OK in } \mathtt{C},\mathtt{D} \quad \vdash \mathtt{D} \not<: \mathtt{C} \quad \vdash \mathtt{D} <: \mathtt{Object} \quad \mathit{distinct}(\mathit{fields}(\mathtt{D})\,\overline{\mathtt{f}}) \quad \mathit{distinct}(\mathit{names}(\overline{\mathtt{M}}))}{\vdash \mathtt{class\ C\ extends\ D}\ \{\ \overline{\mathtt{T}\,\mathtt{f}};\ \overline{\mathtt{M}}\ \}\ \text{OK}}$$

(T-Class)

$$\frac{\mathit{mType}(\mathtt{m},\mathtt{D})\ \mathit{is\ undefined}}{\mathit{override}(\mathtt{m},\mathtt{D},\mathtt{T},\overline{\mathtt{T}})} \qquad \frac{\mathit{mType}(\mathtt{m},\mathtt{D}) = (\overline{\mathtt{T}} \to \mathtt{T}') \quad \vdash \mathtt{T} <: \mathtt{T}'}{\mathit{override}(\mathtt{m},\mathtt{D},\mathtt{T},\overline{\mathtt{T}})}$$

Fig. 8. Class and method typing for $\mathrm{FJ}_{\mathrm{set}}$.

does not require anything about the set of classes associated with this object set. On the other hand, the method `m` must be defined or inherited in the class `C`, it must take at least one argument, and the type of the first argument must be identical to the return type, which is also the type of the entire object set method call.

It would be very easy to change the (T-SInvk) rule to require $\mathtt{C} \in: \{\overline{\mathtt{C}}\}$ and adapt the soundness proof accordingly, which would guarantee that the object set method call would include at least one actual method call, but this is not required for soundness. Similarly, it would be easy to relax the rule such that the return type only has to be a subtype of the type of the first argument rather than being identical to it, but it would be hard to exploit this information unless the rule were modified to enforce that there is at least one actual method call. Even then, the accumulation of contributions from several members of the object set would have to start "from scratch" at each member, because the type of the first argument is fixed. Hence, these variations do not seem to be worthwhile.

Finally, Fig. 8 shows the rules for class and method typing, i.e., rules that apply type checking to the entire program. As opposed to the traditional treatment, these rules include all the requirements needed for programs to be well-formed—for instance in order to avoid cyclic inheritance graphs.

The rule (T-Method) specifies that a method `m` defined in a class `C` with superclass `D` must correctly override any definitions of `m` available in the superclass, it must have a body whose type is a subtype of the declared return type, it must have distinct argument names, and the specified types must be well-formed. The only non-standard element here is the requirement that argument names must be distinct.

The rules for *override* are given at the bottom of the figure; they are used to specify when a definition of a method `m` with argument types $\overline{\mathtt{T}}$ and return type

`T` is correct in relation to definitions available in the superclass `D`. It is standard except that it allows for covariance in the return type, just like the Java language of today.

The rule (T-Class) specifies the standard requirements that the superclass `D`, all field types, and all methods must be well-formed. Moreover, the superclass cannot be a subclass of `C` itself, which prevents cycles in the inheritance graph; and the superclass must be a subclass of `Object`, which ensures that all inheritance chains are finite. This finiteness ensures that subclassing is decidable, which is used in the progress proof. Finally, there are distinctness requirements for field and method names.

All in all, this is not much more involved or verbose than the usual class and method typing rules, but it is complete in the sense that there are no additional (informal and maybe even implicit) well-formedness rules about programs to worry about. We think that it would be useful to make program well-formedness fully explicit, as we have done it here; it is, of course, a consequence of using proof assistant software, because the proofs cannot be completed unless such things are made precise and included in the specification.

4 Soundness

The $\mathrm{FJ}_{\mathrm{set}}$ calculus is sound, which is shown via the standard progress and preservation results:

Theorem 1 (Progress). *If* $\mathtt{e}$ *is an expression typeable by* $\emptyset \vdash \mathtt{e} : \mathtt{T}$ *then either* $\mathtt{e}$ *is a value or there exists an expression* $\mathtt{e}'$ *such that* $\mathtt{e} \rightsquigarrow \mathtt{e}'$.

Theorem 2 (Preservation). *If the expression* $\mathtt{e}$ *in the environment* Γ *is typable by* $\Gamma \vdash \mathtt{e} : \mathtt{T}$ *and it can take the step* $\mathtt{e} \rightsquigarrow \mathtt{e}'$, *then* $\Gamma \vdash \mathtt{e}' : \mathtt{U}$ *for some type* $\mathtt{U}$ *such that* $\vdash \mathtt{U} <: \mathtt{T}$.

A complete proof of these properties which has been mechanically checked by the proof assistant Coq is is available for download [4]. It consists of approximately 6500 lines of Gallina code, divided into approximately 3500 lines specifically on the calculus, and approximately 3000 lines of standard language metatheory facilities from the Coq tutorial given at POPL 2008 [7].

5 Related Work

Dynamic languages like Self [8, 9] support a very general and flexible style of composite objects by means of parent slots and genuine delegation. Object sets are less flexible, but in return they are statically typed.

Object sets are similar to extensible records in some ways. For instance, Gaster and Jones [10] define polymorphic, extensible records and unions based on row variables, i.e., mappings from labels to types. With object sets, the associated classes work as labels and types combined; this reduces the flexibility

because there cannot be two labels with the same type, but given that object sets are intended to model composite objects it would correspond to repeated inheritance to have more than one member associated with the same class, and this would preclude a surface level syntax where class selection is implicit due to the name clashes. Object sets as presented here do not support extension; this is because we consider a '`lacks C`' construct which promises that there is no class `C` in this object set to be unmanageable in real-world software development. On the other hand the extensible records do not have a late-bound operation that corresponds to our object set method calls, it only uses statically known components.

A well-established approach to extensible records is the Haskell `HList` library [11], where Kiselyov, Lämmel and Schupke use type level natural numbers and a number of layers on top of that to support type safe heterogeneous lists. Such lists are actually nested tuples, and the approach relies heavily on being able to use large type expressions which are inferred and never show up in the source code. If explicit typing is considered a valuable source of documentation then object sets are more manageable because they abstract away from the ordering of elements, and they may provide access to an arbitary set of members without depending on the internal structure, i.e., the order of known members and the presence of unknown members.

6 Conclusion

We have presented the concept of object sets as a first class language construct which is capable of emulating the main features of traditional, monolithic objects: access to the disjoint union of the features of all object set members in the type, and support for a kind of method calls whereby the choice of methods to call is made dynamically, corresponding to feature access and method calls for ordinary objects. However, object sets are more flexible than ordinary objects, because they combine the features of several classes (like mixins or multiple inheritance, but without the name clashes), and they provide the machinery needed in order to support dynamic metamorphosis of object sets. The mechanism is useful in its own right, but it is likely to benefit from a pragmatic layer on top of the operations shown in this paper, because this makes the syntax more compact and convenient. The mechanisms of this paper might then provide good service as primitives on main-stream platforms such as .Net or JVM, which would make these platforms better at handling flexible object models in a type safe manner.

Acknowledgments The design of the object set method call mechanism owes some very useful insights to Kim Birkelund. The Coq proof was developed from a starting point created by Bruno de Fraine which was a proof of soundness for Featherweight Java without casts, which again used a number of files from the POPL 2008 Coq tutorial. This was very valuable material in the process of getting up to speed in Coq.

References

1. Ernst, E.: gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance. PhD thesis, DEVISE, Department of Computer Science, University of Aarhus, Aarhus, Denmark (June 1999)
2. Ernst, E.: Higher-order hierarchies. In Cardelli, L., ed.: Proceedings ECOOP'03. LNCS 2743, Heidelberg, Germany, Springer-Verlag (July 2003) 303–329
3. Ernst, E., Ostermann, K., Cook, W.R.: A virtual class calculus. In: Proceedings POPL'06, Charleston, SC, USA, ACM (2006) 270–282
4. Ernst, E.: Coq proof of soundness for the FJ_{set} calculus (October 2008) `http://www.daimi.au.dk/~eernst/Sw65ab/objsetproof.tgz`.
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer Verlag (2004)
6. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. TOPLAS **23**(3) (May 2001) 396–459
7. Aydemir, B.: Using proof assistants for programming language research (January 2008) `http://www.cis.upenn.edu/~plclub/popl08-tutorial/`.
8. Ungar, D., Smith, R.B.: Self: The power of simplicity. In: Proceedings OOPSLA'87, Orlando, FL (October 1987) 227–242
9. Agesen, O., Bak, L., Chambers, C., , Chang, B.W., Hölzle, U., Maloney, J., Smith, R.B., Ungar, D., Wolczko, M.: The Self 4.0 Programmer's Reference Manual. Sun Microsystems, Inc., Mountain View, CA (1995)
10. Gaster, B.R., Jones, M.P.: A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham (November 1996)
11. Kiselyov, O., Lammel, R., Schupke, K.: Strongly typed heterogeneous collections. In: Haskell Workshop. (2004) 96–107