Tribe: More Types for Virtual Classes*

Dave Clarke[1], Sophia Drossopoulou[2], James Noble[3], and Tobias Wrigstad[4]

[1] CWI, Amsterdam, The Netherlands
[2] Imperial College London, UK
[3] Victoria University of Wellington, Wellington, NZ
[4] Stockholm University, Stockholm, Sweden

Abstract. Beginning with BETA, a range of programming language mechanisms have been developed to allow inheritance in the presence of mutually dependent classes. This paper presents Tribe, a type system which generalises and simplifies other formalisms of such mechanisms, by treating issues which are inessential for soundness, such as the precise details of dispatch and path initialisation, as orthogonal to the core formalism. Tribe can support path types dependent simultaneously on both classes and objects, which is useful for writing library code, and ubquitous access to an object's family (= `owner`), which offers family polymorphism without the need to drag around family arguments. Languages based on Tribe will be both simpler and more expressive than existing designs, while having a simpler type system, serving as a useful basis for future language designs.

1 Introduction

Languages and formalisms such as BETA [13], gBeta [5], Caesar [15], Scala [8], Jx [16], .FJ [11], Concord [12] and *vc* [7] offer advanced notions of inheritance which overcome many of the weaknesses of standard single and multiple inheritance. In particular, they feature a notion of *family polymorphism* in which a group of mutually dependent classes can be inherited together in such a way that the relationship between the classes is preserved through inheritance. *Virtual classes* are one means for achieving family polymorphism. A virtual class is a nested class which can be overridden like a method. A key advantage of overriding a class definition, as opposed to extending it, is that the class name remains the same. This enables any code which operates on one family to also work for extensions of that family.

In this paper, we introduce Tribe, a general yet simple system which incorporates these mechanisms into a single seamless framework, extends the language of types, the notion of subtypes, offers powerful inheritance, and tackles method overriding.

In this section we illustrate the basics of family polymorphism, and outline our extensions to existing systems. In the following example, nested classes are

* Work partially supported by a gift from Microsoft Research, , by the Royal Society of New Zealand Marsden Fund, and the EU grant MOBIUS.

used to express that the family `Graph` has member (virtual) classes `Node` and `Edge`.

```
class Graph {
    class Node {
        Edge connect(Node other) { return new Edge(this, other); }
    }
    class Edge {
        Node from, to;
        Edge(Node f, Node t) { from = f; to = t; }
    }
}
```

Subclassing enables the construction of new families from existing ones. In the example below, the family `ColouredGraph` inherits member classes `Node` and `Edge`, and can extend them using a mechanism known as *further binding*. Here, a new field `nodeColour` is added to `Node` inherited from `Graph`.

```
class ColouredGraph extends Graph {
    class Node {
        Colour nodeColour;
    }
}
```

Distinguishing the Families Grouping classes into families raises the question as to whether mixing objects from different families should be allowed. In our example the intention is that coloured nodes should be connected to coloured nodes only. However:

```
Graph.Node n   = new Graph.Node();
Graph.Node cn  = new ColouredGraph.Node();  // dubious subsumption

n.connect(cn);                              // mixes two kinds of nodes
```

Through the—*prima facie* obvious—subtype relation `ColouredGraph.Node` $\leq$ `Graph.Node`, we can create an edge between a node and a coloured node. If the coloured node defines additional methods not found in node, this dubious use of subsumption could lead to "message not understood" errors.

A number of approaches have been devised to address this problem, by keeping better track of the family through the type system. The first approach, used in Concord, .FJ, and Jx, [12, 11, 16, 15] uses types (e.g. `Graph.Node`) based on a static notion of a type family, typically a *class family*, and achieves its goal by eliminating relationships such as `ColouredGraph.Node` $\leq$ `Graph.Node`. The second approach, used in *vc* [7] and *vObj*/Scala [17, 8], keeps close track of which *object family* the nested classes belong to by using paths in types. An *object family type* such as `x.f.C` refers to the class `C` which is nested within the object at the end of path `x.f`. Paths have to be invariant, and so consist of chains of final fields or variables and some class. To illustrate how the error in the above example is caught, consider the following:

```
Graph g    = new Graph();
g.Node n   = new g.Node();
ColouredGraph cg = new ColouredGraph();
cg.Node cn = new cg.Node();

n.connect(cn);   // Type Error !!!
```

The type error occurs because the type system is unable to derive a relationship between `g.Node` and `cg.Node`, as it (correctly) cannot derive that `g` and `cg` are the same family.

Note, that in contrast to Jx, .FJ, and Concord, our approach, *vc* and *vObj*/Scala can distinguish nodes coming from different coloured graphs, *e.g.,*

```
ColouredGraph cg1, cg2;
cg1.Node cn1, cn3; cg2.Node cn2;

cn1.connect(cn3);   // Type Correct
cn2.connect(cg3);   // Type Error!!!
```

Family Parameters When writing library code which operates on objects of nested classes, it is sometimes necessary to pass around an object to represent the family. For example, in a *vc*-like setting, in a library outside the `Graph` family, to copy edges, one would have:

```
class Library {
  g.Edge copyEdge(Graph g, g.Edge e) {
    g.Node from = e.from;
    g.Node to   = e.to;
    return new g.Edge(from, to);   }
}
```

where the parameter `g` is in some sense superfluous, as it serves no other purpose, than to express the type of the parameter `e`, and the result.

Some languages have types to cater for cases where the family object is not needed. For example, Scala [8] has projection types `Graph#Node`, and Jx [16] has types such as `Graph.Node`. The natural subtype relation `g.Node` $\leq$ `Graph.Node` is valid and has a natural reading: while `g.Node` can be read as saying a `Node` from family `g`, `Graph.Node` can be read as a node from *some* graph family.

Both *vc* and Scala (and Java) have types which are used to refer to the surrounding instance of a given class (from within that class). *vc* uses types of the form `this.out.out.out.C` to refer the surrounding `C` instance (Scala's syntax is different, and closer to Java's). Types are further generalised to have the form `this.out.out.out.f.g.h.C`, which access some path of some surrounding class.

Tribe extends this approach by supporting types that cater for the case where the family object is not needed using the `owner` keyword, and also supports path types to talk about types depending on it. Thus, it can avoid passing superfluous family parameters:

```
class Library{
  int distance(Graph.Node n1, n1.owner.Node n2) { ... }

  e.owner.Edge copyEdge(Graph.Edge e) {
    e.owner.Node from = e.from;
    e.owner.Node to   = e.to;
    new e.owner.Edge(from, to);
  }
}
```

This kind of code gives the benefits of family polymorphism without having to pass the family around. It is also quite robust to change. For example, a method may originally be designed to only require a `Graph.Node`. Subsequent changes may require it to use other features of `Graph`. Without types such as `n1.owner.Node` this possible only by adding an extra argument to pass in a family object.

Tribe Types We have shown a number of different kinds of types that appear (and ought to appear) in different systems. All are based on some restriction of the following (where *ff* are final fields including "`owner`"):

$$T ::= (x \mid \texttt{this} \mid \mathsf{C}).(\mathsf{C} \mid \mathit{ff})^*$$

In this paper we eliminate any restrictions, and make the type system both simpler and more expressive. Following this grammar, Tribe Types have a very natural reading, and subtyping is general and natural, as the following example illustrates. Consider the following code:

```
class Musician {
  class Instrument { }
}
class Guitarist extends Musician {
  final Guitar axe;
  class Guitar extends Instrument {
    final String first, second, third, ...;
    class String { }
  }
}
final Guitarist slash;
```

The following are some of the possible types our system can describe:

Type	Interpretation
`slash`	ex-Guns 'n' Roses' guitarist Slash
`Guitarist`	Some guitarist
`slash.axe`	Slash's main guitar
`slash.Guitar`	one of Slash's guitars
`Guitarist.axe`	the main guitar of some guitarist
`Guitarist.Guitar`	some guitarist's guitar
`slash.axe.String`	some string of Slash's main guitar
`slash.Guitar.first`	the first string of one of Slash's guitars
`Guitarist.Guitar.String`	some string of some guitarist's guitar

The natural reading of types in Tribe also extends in an obvious manner to subtyping. For the above code snippet, the following subtyping relations hold:

$$\begin{array}{ccc} \texttt{slash.axe} & \leq & \texttt{Guitarist.axe} \\ \wedge\mid & & \wedge\mid \\ \texttt{slash.Guitar} & \leq & \texttt{Guitarist.Guitar} \\ \wedge\mid & & \wedge\mid \\ \texttt{slash.Instrument} & \leq & \texttt{Guitarist.Instrument} \leq \texttt{Musician.Instrument} \end{array}$$

Formalising Tribe Developing sound static type systems for the mechanisms described thus far is a challenge to which many researchers have risen (see citations above). Extending such languages and type systems often requires considerable work due to their subtle nature. For instance, Ernst, Ostermann and Cook's formalisation of virtual classes was the first account establishing their soundness [7].

This paper presents a formalisation of virtual classes which is both simpler and more general than *vc*. The changes we have made are as follows:

- The notions of type and hence subtyping have been generalised.
- We are non-committal as to exactly which method is dispatched in the case of ambiguity. Requiring that all candidates satisfy the desired typing constraint is sufficient for soundness. Any more precise choice is a *refinement* of our approach, and thus also sound.
- Correct final field initialisation and use is crucial for soundness of path-based types, but dealing with it can significantly increase the complexity of a type system. Rather than cluttering the type system with tests for handling this properly, we simply treat both uninitialised final fields and attempts at reassigning final fields as errors. Dealing with finals properly is known and orthogonal to the rest of the system.
- We present both classes and methods, rather than unify them as in BETA.
- Class tables are statically resolved, whereas in *vc* they are dynamically computed.
- We adopt a small-step semantics rather than a large-step semantics.

The culmination of all of these changes is a simpler formalism and simpler proofs. As a result, we hope that our approach gives a simpler and more general model of the main issues underlying virtual classes, which will serve as the basis for future developments.

We also present a way of dealing with method invocation in the presence of overriding and multiple inheritance, and outline two novel constructs—over-the-top types and adoption—that allow cross-family subclassing while retaining the advantages of virtual classes.

2 The Tribe Programming Language

The syntax of Tribe is given in Figure 1, where x ranges over variables, f over all field names, ff over final field names, m over method names, C over class

names, and ι over addresses. For convenience, all variables are final. At times we use the syntactic category *ff* to indicate finality, other times we use the `final` keyword, and sometimes both.

class	::=	`class` C `extends` $\overline{\mathsf{C}}$ `{` *cnstr fld** *class** *mthd** `}`	CLASSES
cnstr	::=	$\mathsf{C}(\overline{T\ x})$ `{` $\overline{\texttt{this.}ff := e;}$ `}`	CONSTRUCTORS
fld	::=	T f; | `final` T *ff*;	FIELDS
mthd	::=	T *md* $(\overline{T\ x})$ `{` *e* `}`	METHODS
var	::=	`this` | *x* | $\circ$ | ι	TARGETS
T	::=	$var.(\mathsf{C} \mid ff)^*$	TYPES
p	::=	$var.ff^*$	PATHS
e	::=	`null` | `error` | p | $e.f$ | $p.f := e$	EXPRESSIONS
	|	$p.m(\overline{p})$ | `new` $p.\mathsf{C}(\overline{p})$ | `final` $T\ x := e;\ e$ | $e; e$	

Fig. 1. Syntax of Tribe, with run-time entities in grey.

A class consists of a collection of fields, methods and nested classes. Classes inherit from multiple other classes. $\circ$ denotes the top-level collection of classes.

Fields can either be final or not. Final fields can be used to form paths in order to refer to specific families. Each object has a special field called `owner` (part of the *ff* syntax) which refers to the object's surrounding object (or $\circ$ for instances of top level classes).

Types in Tribe are formed out of paths, and may, but need not, include class names. Types which do not include a class name are singleton types, referring to a single object. For example, the type of `this` is singleton type `this`, and the type of `owner` is singleton type `this.owner` (effectively, `owner`). A type $T.\mathsf{C}$ describes all the objects of class C nested inside an object of type T (a class family). A type $T.ff$ describes all the objects which may be referred to through the field *ff* of an object of type T (an object family). Run-time types may contain addresses.

Expressions are as usual. Following the syntax from Jx [16], the "let" expression `final` $T\ x := e_1;\ e_2$ introduces the final variable x with value e_1 which can used within e_2. Run-time expressions may contain addresses. Constructors support field initialisation, which is required because of the existence of final fields. In order to keep the language description simple, we do not support calls to the superclass constructor; instead, we allow a constructor to initialise all fields, including inherited fields. For the same reason, we do not statically check that *all* fields have been initialised. As a consequence, uninitialised fields may be accessed in a running program, or final fields may be assigned twice; in both cases we raise errors. As for null pointer exceptions, such errors do not affect type safety. We consider these simplifications to be legitimate, since techniques exist for ensuring that final fields' initialisation and superclass constructor calls take place exactly once [14, 10, 9, 20].

$$\frac{P(\mathsf{A})\ \text{defined}}{\vdash \mathsf{A.Object}\ \mathit{cls}}\ (\textsc{Def-Obj}) \qquad \frac{\texttt{class}\ \mathsf{C}\ \cdots \in P(\mathsf{A})}{\vdash \mathsf{A.C}\ \mathit{cls}}\ (\textsc{Def-Prog}) \qquad \frac{\vdash \mathsf{A}' \sqsubseteq_i \mathsf{A} \quad \vdash \mathsf{A.C}\ \mathit{cls}}{\vdash \mathsf{A}'.\mathsf{C}\ \mathit{cls}}\ (\textsc{Def-Inh})$$

$$\frac{\texttt{class}\ \mathsf{C}'\ \texttt{extends}\ \overline{\mathsf{C}}\ \cdots \in P(\mathsf{A})}{\vdash \mathsf{A.C}' \sqsubseteq_s \mathsf{A.C}_i}\ (\textsc{Subcl-Prog}) \qquad \frac{\vdash \mathsf{A}' \sqsubseteq_i \mathsf{A} \quad \vdash \mathsf{A.C'} \sqsubseteq_s \mathsf{A.C}}{\vdash \mathsf{A}'.\mathsf{C}' \sqsubseteq_s \mathsf{A}'.\mathsf{C}}\ (\textsc{Subcl-Inh})$$

$$\frac{\vdash \mathsf{A} \sqsubseteq_i \mathsf{A}' \quad \vdash \mathsf{A}'.\mathsf{C}\ \mathit{cls}}{\vdash \mathsf{A.C} \sqsubseteq_f \mathsf{A}'.\mathsf{C}}\ (\textsc{Further-Bind}) \qquad \frac{\vdash \mathsf{A} \sqsubseteq_s \mathsf{A}' \ \vee\ \vdash \mathsf{A} \sqsubseteq_f \mathsf{A}'}{\vdash \mathsf{A} \sqsubseteq_i \mathsf{A}'}\ (\textsc{Inh})$$

Fig. 2. Classes and Subclassing and Further Binding Resolution

3 Class Tables

The semantics of Tribe is defined in terms of class tables, CT, which map class identifiers to the members (*i.e.,* fields, methods and nested class constructors) available in the class. In contrast to calculi such as .FJ [11], class tables represent more than the contents of the program. Following approaches used for the semantics of mixins and traits [2, 19], a class table represent a "flattened" version of the program, where a class not only contains the members directly declared in the class itself, but also all the members inherited by the class.

3.1 Subclasses and Further Binding

Classes appear within other classes, and the same identifier may be used to describe classes within several classes, *e.g.,* `Edge` is defined both within `Graph` and within `ColourGraph`. We therefore distinguish between *absolute classes*, which are absolute paths starting from the program's root, *e.g.,* ∘.`ColourGraph`.`Node`, and classes, which consist of a class identifier, *e.g.,* `Node`.

The syntax of absolute class names, where ∘ corresponds to the root of the program, is:

$$\mathsf{A} \ ::= \ \circ \ \mid \ \mathsf{A.C}$$

We use P to denote the program text. $P(\mathsf{A})$ correspond to the code which is inside absolute class A. $P(\circ)$ corresponds to the entire program. To reduce clutter, we assume that P is global across the type system, and can be used wherever required.

We introduce the following judgements, defined by the rules in Figure 2:

$\vdash \mathsf{A}\ \mathit{cls}$	A is a class in the program	$\vdash \mathsf{A} \sqsubseteq_s \mathsf{A}'$	A is a direct subclass of A'
$\vdash \mathsf{A} \sqsubseteq_f \mathsf{A}'$	A directly further binds A'	$\vdash \mathsf{A} \sqsubseteq_i \mathsf{A}'$	A directly inherits from A'

We demonstrate the above judgements in terms of the following nest of classes, where, for simplicity, we show class nesting but drop class contents:

```
class A
   class B
       class C
       class C' extends C
   class B' extends B
class A' extends A
```

Rule (Def-Obj) defines Object as nested in every class, *e.g.*, $\vdash \circ.\mathtt{A.B.Object}\ cls$. Facts expressed directly in the program are reflected through rules (Def-Prog) and (Subcl-Prog), *e.g.*, $\vdash \circ.\mathtt{A.B}\ cls$, and $\vdash \circ.\mathtt{A.B}'\ cls$, and $\vdash \circ.\mathtt{A}' \sqsubseteq_s \circ.\mathtt{A}$. Subclassing implies inheritance (Inh), therefore $\vdash \circ.\mathtt{A}' \sqsubseteq_i \circ.\mathtt{A}$. Rules (Subcl-Inh) and (Def-Inh) express that inheritance implies "copying" nested classes and their relationships; thus, $\vdash \circ.\mathtt{A}'.\mathtt{B}'\ cls$, and $\vdash \circ.\mathtt{A}'.\mathtt{B}' \sqsubseteq_s \circ.\mathtt{A}'.\mathtt{B}$. By repeated application of above rules we obtain, *e.g.*, that $\vdash \circ.\mathtt{A}'.\mathtt{B}'.\mathtt{C}' \sqsubseteq_s \circ.\mathtt{A}'.\mathtt{B}'.\mathtt{C}$, even though classes $\circ.\mathtt{A}'.\mathtt{B}'.\mathtt{C}'$ and $\circ.\mathtt{A}'.\mathtt{B}'$ do *not* appear in the program.

The following lemma guarantees that 1) the subclassing relationship holds only between classes which are (derived to be) nested within the same class; 2) further binding implies that the two types are identical up to subclassing at some point in their path; and 3) that inheritance of nested classes implies inheritance of their surrounding classes.

Lemma 1.

1. *If* $\vdash \mathsf{A}' \sqsubseteq_s \mathsf{A}$, *then* $\exists \mathsf{A}''$, C, C' *such that* $\mathsf{A}' = \mathsf{A}''.\mathsf{C}'$ *and* $\mathsf{A} = \mathsf{A}''.\mathsf{C}$.
2. *If* $\vdash \mathsf{A}' \sqsubseteq_f \mathsf{A}$, *then* $\exists \mathsf{A}'', \mathsf{C}, \mathsf{C}', \overline{\mathsf{C}} \neq \epsilon$ *such that* $\mathsf{A}' = \mathsf{A}''.\mathsf{C}'.\overline{\mathsf{C}}$, $\mathsf{A} = \mathsf{A}''.\mathsf{C}.\overline{\mathsf{C}}$, *and* $\vdash \mathsf{A}''.\mathsf{C}' \sqsubseteq_s \mathsf{A}''.\mathsf{C}$.
3. $\vdash \mathsf{A}.\mathsf{C} \sqsubseteq_i^* \mathsf{A}'.\mathsf{C}$ *if and only if* $\vdash \mathsf{A} \sqsubseteq_i^* \mathsf{A}'$ *and* $\vdash \mathsf{A}'.\mathsf{C}\ cls$.

3.2 Class Table Construction

Class tables, CT, map absolute class names As to tuples (fs, ms, cn) representing the fields, methods, and nested class constructors that are either directly present in the class or inherited. Class tables are constructed using the operation $\oplus$ on tuples, which gives priority to the first argument.

Definition 1 ($\oplus$). *For functions g and g', define function $g \oplus g'$ as:*

$$(g \oplus g')(x) = \begin{cases} g(x), & \text{if } g(x) \text{ is defined;} \\ g'(x) & \text{otherwise.} \end{cases}$$

Define $\oplus$ for tuples as:

$$(fs, ms, cn) \oplus (fs', ms', cn') = (fs \oplus fs', ms \oplus ms', cn \oplus cn')$$

The construction of class tables is defined as follows.

Definition 2. *For a program P, we define the class table* CT *as*

$$
\begin{array}{ll}
\mathsf{BT}(\circ) & = (\emptyset, \emptyset, \{\mathit{cnstr} \mid \texttt{class}\ \ldots \texttt{\{}\ \mathit{cnstr}\ \ldots\ \ \texttt{\}} \in P(\circ)\}) \\
\mathsf{BT}(\mathsf{A.C}) & = (\mathit{flds}, \mathit{mthds}, \mathit{cnstrs}) \\
 & \quad \text{where } \texttt{class C extends } \overline{\mathsf{C}'} \texttt{ \{} \ldots \mathit{classes}\ \mathit{flds}\ \mathit{mthds}\ \texttt{\}} \in P(\mathsf{A}) \\
 & \quad \text{and } \mathit{cnstrs} = \{\mathit{cnstr} \mid \texttt{class}\ \ldots \texttt{\{}\ \mathit{cnstr}\ \ldots\ \ \texttt{\}} \in \mathit{classes}\}
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{CT}(\mathsf{A.Object}) & = (\{\texttt{final owner} : \texttt{this.owner}\}, \emptyset, \emptyset) \\
\mathsf{CT}(\mathsf{A}) & = \mathsf{BT}(\mathsf{A}) \oplus \mathsf{CT}(\overline{\mathsf{A}}), \text{ where } \overline{\mathsf{A}} = \{\mathsf{A}' \mid \vdash \mathsf{A} \sqsubseteq_i \mathsf{A}'\}
\end{array}
$$

The auxiliary function BT collects the fields and methods directly present in a class, as well as the constructors defined in directly enclosed classes. Note, that the treatment of constructors differs from that of methods and fields. This is so, because for a class C defined within A, objects of class $\mathsf{A.C}$ can only be constructed within A objects; therefore, the C constructor is in some sense a special method of A.

The definition of $\mathsf{CT}(\mathsf{A.Object})$ has the effect that every object has a field called `owner` which refers to an object of its surrounding class (even when the class is at the top-level). The definition of $\mathsf{CT}(\mathsf{A})$ collects all the inherited fields, methods and constructors, using those defined in BT as overriding definitions.

Class tables offer a *flattened view* of the program which directly classes with the members inherited from superclasses or further bound classes. This approach, first used in the study of mixins [2], separates the mechanisms that *produce* inheritance (in our cases subclasses and further binding), from the *effects* of inheritance (*i.e.,* the copying and potential overriding of members. This simplifies the model considerably, simplifies the proofs, and demonstrates easily how Tribe could be implemented.

Remark 1. Observe that we are not too concerned with the order in which methods are inherited. Our system ensures that *all* inherited methods work soundly; the decision about which one to use is an orthogonal issue to soundness.

The following shorthands will be used throughout the remainder of the paper.

Definition 3. *Given a class table* CT*, define the following shorthands:*

- $\mathsf{fields}(\mathsf{A}) = \mathit{fst}(\mathsf{CT}(\mathsf{A}))$.
- $\mathsf{finals}(\mathsf{A}) = \{\mathit{ff} : T \mid \texttt{final}\ \mathit{ff} : T \in \mathit{fst}(\mathsf{CT}(\mathsf{A}))\}$.
- $\mathsf{methods}(\mathsf{A}) = \mathit{snd}(\mathsf{CT}(\mathsf{A}))$.
- $\mathsf{constructors}(\mathsf{A}) = \mathit{thd}(\mathsf{CT}(\mathsf{A}))$.
- $(\mathit{fs}, \mathit{ms}, \mathit{cn}) \sqsubseteq (\mathit{fs}', \mathit{ms}', \mathit{cn}')$ *iff there exist* fs'', ms'', *and* cn'' *such that* $\mathit{fs} = \mathit{fs}'' \oplus \mathit{fs}'$, $\mathit{ms} = \mathit{ms}'' \oplus \mathit{ms}'$, *and* $\mathit{cn} = \mathit{cn}'' \oplus \mathit{cn}'$.

Finally, define $\sqsubseteq_i^*$ as the reflexive transitive closure of $\sqsubseteq_i$. The following lemma guarantees that 1) class table entries exist only for well formed absolute classes, 2) each class table entry has an `owner` field of appropriate type, inheritance implies that the class table entry expands that of the inherited class.

$$
\begin{array}{lcll}
\mathbf{v} & ::= & \circ \mid \iota & \textsc{Non-Null values} \\
v & ::= & \mathbf{v} \mid \texttt{null} & \textsc{Values} \\
E & ::= & [-] \mid E.f \mid E.f := e \mid \mathbf{v}.f := E & \textsc{Reduction Context} \\
 & & \mid E.m(\overline{e}) \mid \mathbf{v}.m(\overline{v}, E, \overline{e}) \mid \texttt{new}\ E.\mathsf{C}(\overline{e}) & \\
 & & \mid \texttt{new}\ \mathbf{v}.\mathsf{C}(\overline{v}, E, \overline{e}) \mid E; e & \\
err & ::= & \texttt{null} \mid \texttt{error} & \textsc{Error Values} \\
N & ::= & err.f \mid err.f := e \mid err.\mathit{ff} := e & \textsc{Error Contexts} \\
 & & \mid v.\mathit{ff} := \texttt{error} \mid err.m(\overline{e}) \mid \texttt{new}\ err.\mathsf{C}(\overline{e}) &
\end{array}
$$

Fig. 3. Dynamic Expressions

Lemma 2. *The following properties hold for a class table* CT*:*

1. $\mathsf{CT}(\mathsf{A}) \neq \bot$ *if and only if* $\vdash \mathsf{A}$ *cls.*
2. *If* $\mathsf{CT}(\mathsf{A}) \neq \bot$*, then* $\mathsf{Object} \in \mathit{classes}(\mathsf{A})$ *and* $\texttt{owner} : \texttt{this.owner} \in \mathsf{finals}(\mathsf{A})$.
3. *If* $\vdash \mathsf{A} \sqsubseteq_i^* \mathsf{A}'$*, then* $\mathsf{CT}(\mathsf{A}) \sqsubseteq \mathsf{CT}(\mathsf{A}')$.

4 Dynamic Semantics

The semantics of Tribe is presented as a small-step reduction relation of the form $H, e \rightsquigarrow H', e'$, which states that configuration H, e reduces (in one step) to H', e', where H and H' are heaps and e and e' are expressions. The syntax of heaps H, and objects o:

$$H ::= \emptyset \mid \iota \mapsto o, H \qquad\qquad o ::= [\, \bar{f} \mapsto \bar{v} \,]_{\mathsf{A}}$$

An object consists of the name of its absolute class, A, and values for all of its fields, denoted $\bar{f} \mapsto \bar{v}$. The fields also contain a value for the `owner` field.

Define $H(\iota)$, $H + \iota \mapsto o$ and $H[\iota \mapsto o]$ as look-up, extend, and update of a heap, and analogously for o. Also let $H(\iota).f$ denote $H(\iota)(f)$ and $H(\iota).f := v$ denote $H[\iota \mapsto (H(\iota)[f \mapsto v])]$, as shorthands for accessing and updating the field of some object in the heap.

Figure 3 contains additional syntax used in the operational semantics. A reduction context, $E[-]$, is defined to be an expression with a hole in it, in the standard manner. We also have *error contexts*, N, (called null contexts in Jx's semantics [16]) to gracefully handle dereferencing of `null`. For convenience a `null`-dereference evaluates immediately to `error`. We apply the same treatment to errors arising from assigning to already initialised final fields and to errors resulting from accessing a final field which has not yet been initialised. Such expressions are not stuck, and the value they reduce to can have any type, thus their presence does not interfere with subject reduction and progress lemmas.

The rules for the operational semantics are given in Figure 4.

(Eval-Field) returns the value of an object's field. Both (Eval-FieldAsgn) and (Eval-FinalFieldAsgn) update fields. (Eval-FinalFieldAsgn) is only applicable if the field contains `error`, indicating that it is uninitialised. (Eval-FinalFieldAsgn-Error) traps the case when an attempt to initialise an already

$$\frac{H(\iota) = [\,\ldots]_{\mathsf{A}} \quad f \in \mathsf{nonfinals}(\mathsf{A}) \quad H' = H(\iota).f := v}{H, \iota.f := v \rightsquigarrow H', v} \text{ (Eval-FieldAsgn)}$$

$$\frac{H(\iota) = [\,\ldots]_{\mathsf{A}} \quad \mathit{ff} \in \mathsf{finals}(\mathsf{A}) \quad H(\iota).\mathit{ff} = \mathtt{error} \quad H' = H(\iota).f := v}{H, \iota.\mathit{ff} := v \rightsquigarrow H', v} \text{ (Eval-FinalFieldAsgn)}$$

$$\frac{H(\iota) = [\,\ldots]_{\mathsf{A}} \quad \mathit{ff} \in \mathsf{finals}(\mathsf{A}) \quad H(\iota).\mathit{ff} \neq \mathtt{error}}{H, \iota.\mathit{ff} := v \rightsquigarrow H, \mathtt{error}} \text{ (Eval-FinalFieldAsgn-Error)}$$

$$\frac{\begin{array}{c}\mathbf{v} = \circ = \mathsf{A} \;\vee\; \mathbf{v} = \iota' \wedge H(\iota') = [\,\ldots]_{\mathsf{A}} \\ \mathsf{C}(\overline{T\ x})\{\ e\ \} \in \mathsf{constructors}(\mathsf{A}) \\ \mathrm{dom}(\mathsf{nonfinals}(\mathsf{A.C})) = \overline{f} \quad \mathrm{dom}(\mathsf{finals}(\mathsf{A.C})) = \overline{\mathit{ff}} \\ \iota \notin \mathrm{dom}(H) \quad H' = H + \iota \mapsto [\,\mathtt{owner} \mapsto \mathbf{v}, \overline{f} \mapsto \overline{\mathtt{null}}, \overline{\mathit{ff}} \mapsto \overline{\mathtt{error}}\,]_{\mathsf{A.C}}\end{array}}{H, \mathtt{new}\ \mathbf{v}.\mathsf{C}(\overline{v}) \rightsquigarrow H', e[\iota/\mathtt{this}, \overline{v}/\overline{x}]; \iota} \text{ (Eval-New)}$$

$$\frac{H(\iota) = [\,\ldots]_{\mathsf{A}} \quad T\ m(\overline{T\ z})\{\ e\ \} \in \mathsf{methods}(\mathsf{A})}{H, \iota.m(\overline{v}) \rightsquigarrow H, e[\iota/\mathtt{this}, \overline{v}/\overline{z}]} \text{ (Eval-Meth)}$$

$$\frac{}{H, \mathtt{final}\ T\ x := v;\ e \rightsquigarrow H, e[v/x]} \text{ (Eval-Let)}$$

$$\frac{H(\iota).f = v}{H, \iota.f \rightsquigarrow H', v} \text{ (Eval-Field)} \qquad \frac{}{H, v; e \rightsquigarrow H, e} \text{ (Eval-Seq)}$$

$$\frac{H, e \rightsquigarrow H', e'}{H, E[e] \rightsquigarrow H', E[e']} \text{ (Eval-Context)} \qquad \frac{}{H, E[N] \rightsquigarrow H, \mathtt{error}} \text{ (Eval-Error)}$$

Fig. 4. Reduction Rules

initialised final field is made. (Eval-New) determines firstly which class table to look-up to find the constructors. This depends upon the path for which the new class is being created—the new owner. A new object is created, with the owner field set appropriately, all non-final fields set to `null`, and all final fields set to `error` to indicate that they are not initialised. The result is an expression which will evaluate the constructor and return the new location. (Eval-Meth) finds the code for the method in the class table, and reduces to the body with the targets and arguments substituted. (Eval-Let) is standard let statement. (Eval-Seq) is standard sequential composition, which discards the result of the first expression. (Eval-Context) is also standard, stating that the evaluation of any expression proceeds by evaluating one of its redexes. (Eval-Error) detects an error condition and immediately reduces to an error.

5 Type System

The type system is defined in terms of the following judgements:

$\Gamma \vdash \diamond$	Γ is a good typing environment	$\Gamma \vdash T$	T is a good type
$\Gamma \vdash T \Uparrow \mathsf{A}$	Class A corresponds to type T	$\Gamma \vdash T \leq T'$	T is a subtype of T'
$\Gamma \vdash e : T$	expression e has type T		

Typing environments have the following syntax:

$$\Gamma \;::=\; \emptyset \mid \mathsf{A} \mid \Gamma, x : T \mid \Gamma, \iota : T \mid \Gamma, \iota.\mathit{ff} = v$$

$$\frac{}{\emptyset \vdash \diamond}\ (\textsc{Env-Empty}) \qquad \frac{\vdash \mathsf{A}\ \mathit{cls}}{\mathsf{A} \vdash \diamond}\ (\textsc{Env-AbsClass}) \qquad \frac{\Gamma \vdash T \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : T \vdash \diamond}\ (\textsc{Env-Decl})$$

$$\frac{\iota.\mathit{ff} = v' \notin \Gamma \quad \Gamma \vdash \iota \Uparrow \mathsf{A} \quad \mathit{ff} : T \in \mathsf{finals}(\mathsf{A}) \quad \Gamma \vdash v : T[\iota/\mathtt{this}]}{\Gamma, \iota.\mathit{ff} = v \vdash \diamond}\ (\textsc{Env-Eq})$$

Fig. 5. Good Environments—(Env-Decl) applies to both variables and addresses

A typing environment maps variables, including the receiver `this`, and addresses to types. The absolute class name A in a typing environment indicates the context in which type checking is performed. Equations of the form $\iota.\mathit{ff} = v$ in a typing environment keep track of paths; they are used to track paths when proving soundness, and are *not* used in the static semantics.

Figure 5 defines well-formed environments, $\Gamma \vdash \diamond$. The rules are mostly straightforward. The non-obvious rule, (Env-Eq), is applicable for run-time environments, and enables an equation to be added to the typing environment, if no equation is already present for the given object field. It simply requires that v is a value which can be stored in final field ff of object ι.

5.1 Well-formed Types

A well-formed type T corresponds to an absolute class, A, as expressed through the judgement $\Gamma \vdash T \Uparrow \mathsf{A}$ in Figure 6. Only types which correspond to absolute classes are well formed, from rule (Type).

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \circ \Uparrow \circ}\ (\textsc{Type-Start}) \qquad \frac{x : T \in \Gamma \quad x \neq \mathtt{this} \quad \Gamma \vdash T \Uparrow \mathsf{A}}{\Gamma \vdash x \Uparrow \mathsf{A}}\ (\textsc{Type-Decl}) \qquad \frac{\Gamma \vdash \diamond \quad \mathsf{A}, \mathtt{this} : \mathtt{this} \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \mathtt{this} \Uparrow \mathsf{A}}\ (\textsc{Type-This})$$

$$\frac{\Gamma \vdash T \Uparrow \mathsf{A} \quad \mathit{ff} : T' \in \mathsf{finals}(\mathsf{A}) \quad \mathsf{A}, \mathtt{this} : \mathtt{this} \vdash T' \Uparrow \mathsf{A}'}{\Gamma \vdash T.\mathit{ff} \Uparrow \mathsf{A}'}\ (\textsc{Type-Final-Field})$$

$$\frac{\Gamma \vdash T \Uparrow \mathsf{A} \quad \vdash \mathsf{A.C}\ \mathit{cls}}{\Gamma \vdash T.\mathsf{C} \Uparrow \mathsf{A.C}}\ (\textsc{Type-Class}) \qquad \frac{\Gamma \vdash T \Uparrow \mathsf{A.C}}{\Gamma \vdash T.\mathtt{owner} \Uparrow \mathsf{A}}\ (\textsc{Type-Owner}) \qquad \frac{\Gamma \vdash T \Uparrow \mathsf{A}}{\Gamma \vdash T}\ (\textsc{Type})$$

Fig. 6. Good Types

Rule (Type-Start) allows $\circ$ as the starting point for absolute types. Rule (Type-Decl) states that variables and addresses can be used as singleton types.

$$\frac{}{\Gamma \vdash T \leq T}\ (\textsc{Sub-Refl}) \qquad \frac{\Gamma \vdash T \leq T' \quad \Gamma \vdash T' \leq T''}{\Gamma \vdash T \leq T''}\ (\textsc{Sub-Trans}) \qquad \frac{\Gamma \vdash T \Uparrow \mathsf{A} \quad \vdash \mathsf{A.C} \sqsubseteq_s \mathsf{A.C'}}{\Gamma \vdash T.\mathsf{C} \leq T.\mathsf{C'}}\ (\textsc{Sub-Subclass})$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x \leq T}\ (\textsc{Sub-Decl}) \qquad \frac{\Gamma \vdash T \Uparrow \mathsf{A} \quad \mathit{ff} : T' \in \mathsf{finals(A)}}{\Gamma \vdash T.\mathit{ff} \leq T'[T/\texttt{this}]}\ (\textsc{Sub-Final-Field}) \qquad \frac{\Gamma \vdash T \Uparrow \mathsf{A}}{\Gamma \vdash T \leq \mathsf{A}}\ (\textsc{Sub-Abs})$$

$$\frac{\Gamma \vdash T.\mathsf{C}.\texttt{owner}}{\Gamma \vdash T.\mathsf{C}.\texttt{owner} = T}\ (\textsc{Sub-Owner-1}) \qquad \frac{\Gamma \vdash T \Uparrow \mathsf{A.C}}{\Gamma \vdash T \leq T.\texttt{owner}.\mathsf{C}}\ (\textsc{Sub-Owner-2}) \qquad \frac{\Gamma \vdash T \leq T'.\texttt{owner}.\mathsf{C}}{\Gamma \vdash T.\texttt{owner} = T'.\texttt{owner}}\ (\textsc{Sub-Owner-3})$$

$$\frac{\Gamma \vdash T \leq T' \quad \Gamma \vdash T'.\mathit{ff}}{\Gamma \vdash T.\mathit{ff} \leq T'.\mathit{ff}}\ (\textsc{Sub-Nest-Final}) \qquad \frac{\Gamma \vdash T \leq T' \quad \Gamma \vdash T'.\mathsf{C}}{\Gamma \vdash T.\mathsf{C} \leq T'.\mathsf{C}}\ (\textsc{Sub-Nest-Class})$$

Fig. 7. Good Subtyping—$T = T'$ is interpreted as $T \leq T'$ and $T' \leq T$.

The absolute class to which a variable or address corresponds is determined using its declared type. Rule (Type-This) declares `this` to be a singleton type, which corresponds to the absolute class given by the present context. By rule (Type-Class), any valid type for which it makes sense to have a nested class C can be extended with suffix C. Similarly, rule (Type-Final-Field) enables any type which contains a final field *ff* to be extended with suffix *ff*. (Type-Owner) states that any type can be extended with suffix `owner`, as long as the type corresponds to an actual absolute class, *not* $\circ$.

Note that we not only have more types that *vc*, as described in the introduction. We even permit types to go outside of a hierarchy. For example, in a topmost class `A` the type `this.owner.B` refers to an instance of the topmost class `B`, even though the two classes are not enclosed in another class. This is useful when using adoption (Section 7).

We can prove that 1) a type corresponds to at most one absolute class; 2) if a type corresponds to an absolute class, then this absolute class is a class (possibly inferred through inheritance); and 3) types correspond to more specialised absolute classes in more specialised contexts.

Lemma 3. *For program P, environment Γ, type T, absolute classes A_0, A, $\mathsf{A'}$:*

1. *If $\Gamma \vdash T \Uparrow \mathsf{A}$ and $\Gamma \vdash T \Uparrow \mathsf{A'}$, then $\mathsf{A} = \mathsf{A'}$.*
2. *If $\Gamma \vdash T \Uparrow \mathsf{A}$, then $\vdash \mathsf{A}$ cls.*
3. *If $\vdash \mathsf{A}_1 \sqsubseteq_i^* \mathsf{A}_2$, $\mathsf{A}_1, \Gamma \vdash T \Uparrow \mathsf{A}_3$, and $\mathsf{A}_2, \Gamma \vdash T \Uparrow \mathsf{A}_4$, then, $\vdash \mathsf{A}_3 \sqsubseteq_i^* \mathsf{A}_4$.*

5.2 Subtyping

Subtyping in Tribe is very rich. The rules are presented in Figure 7.

As usual, the type system supports reflexivity (Sub-Refl) and transitivity (Sub-Trans) of the subtype relation. (Sub-Subclass) converts subclassing into

subtyping. (Sub-Decl) states that a singleton type which corresponds to either a variable or an address is a subtype of its declared type. (Sub-Abs) establishes a subtype relation between any type and its absolute class.

(Sub-Final-Field) establishes a relationship between a type with a final field suffix and the type of the field in the given context. This is achieved by substituting the prefix into the type of the final field. For example, if $\Gamma \vdash T \Uparrow \circ.\texttt{A.B}$ and class $\circ.\texttt{A.B}$ has final fields $\mathit{ff}_1 : \circ.\texttt{C.D}$ and $\mathit{ff}_2 : \texttt{this.E}$, then (Sub-Final-Field) gives that $\Gamma \vdash T.\mathit{ff}_1 \leq \circ.\texttt{C.D}$ and $\Gamma \vdash T.\mathit{ff}_2 \leq T.\texttt{E}$.

Rule (Sub-Owner-1) is two rules combined into one statement. The equality is to be read both left to right and right to left as a subtype rule. A type can have $\mathsf{C}.\texttt{owner}$ added or removed from the end (whenever it produces a sensible type). This works since C is a nested type, and owner goes back to the enclosing instance. Rule (Sub-Owner-2) allows any type to be given in terms of a type in the surrounding class. This is essential for obtaining that $\texttt{this} : \texttt{this.owner.}\mathsf{C}$, for the appropriate C. Rule (Sub-Owner-3) is again really two rules. This enables one to determine whether two types reside in the same family, in which case, their owners are the same.

Both rules (Sub-Nest-Final) and (Sub-Nest-Class) enable a prefix of a type to be replaced by any subtype. Combined with transitivity, these two rules enable any internal part of a type path to be replaced by its subtype. Rule (Sub-Nest-Class) resembles the rule ($\leq$-Nest) from Jx [16], with an additional well-formedness check.

The following lemma states that 1) subtyping of absolute classes implies inheritance; 2) the absolute class corresponding to a type is its most specific absolute supertype; 3) the absolute classes corresponding to subtypes are in the inheritance relationship, meaning that everything expected of some type is available in all subtypes (crucial for proving progress); 4) subtype relations are preserved in more specialised contexts. The last property means that subtype relations remain true in all *inherited* code. A similar property applies to typing, as we will see in the next section. Such preservation properties are crucial for proving soundness.

Lemma 4.

1. *If* $\Gamma \vdash \mathsf{A} \leq \mathsf{A}'$, *then* $\vdash \mathsf{A} \sqsubseteq_i^* \mathsf{A}'$.
2. *If* $\Gamma \vdash T \Uparrow \mathsf{A}$, *and* $\Gamma \vdash T \leq \mathsf{A}'$, *then* $\Gamma \vdash \mathsf{A} \leq \mathsf{A}'$.
3. *If* $\Gamma \vdash T \leq T'$, $\Gamma \vdash T \Uparrow \mathsf{A}$, *and* $\Gamma \vdash T' \Uparrow \mathsf{A}'$, *then* $\vdash \mathsf{A} \sqsubseteq_i^* \mathsf{A}'$.
4. *If* $\vdash \mathsf{A}_1 \sqsubseteq_i^* \mathsf{A}_0$ *and* $\mathsf{A}_0, \Gamma \vdash T \leq T'$, *then* $\mathsf{A}_1, \Gamma \vdash T \leq T'$.

5.3 Expression Typing

Expression typing is given in Figure 8.

(Expr-Bullet) gives to $\circ$ the type $\circ$ so that it $\circ$ be used in expressions which create absolute types. (Expr-Decl) gives a singleton type to any variable or address (as they are final). Through a combination of (Sub-Decl) and (Expr-Subsumption) we obtain, for example, that if $x : T \in \Gamma$, then $\Gamma \vdash x : T$,

$$\frac{}{\Gamma \vdash \diamond}\ (\textsc{Expr-Bullet}) \qquad \frac{\Gamma \vdash \diamond \quad x : T \in \Gamma}{\Gamma \vdash x : x}\ (\textsc{Expr-Decl}) \qquad \frac{\Gamma \vdash T}{\Gamma \vdash \texttt{null} : T}\ (\textsc{Expr-Null}) \qquad \frac{\Gamma \vdash T}{\Gamma \vdash \texttt{error} : T}\ (\textsc{Expr-Error})$$

$$\frac{\Gamma \vdash e : T \quad \Gamma \vdash T \Uparrow \mathsf{A} \quad \mathit{ff} : T' \in \mathsf{finals}(\mathsf{A})}{\Gamma \vdash e.\mathit{ff} : T.\mathit{ff}}\ (\textsc{Expr-Final-Field}) \qquad \frac{\Gamma \vdash e : T \quad \Gamma \vdash T \Uparrow \mathsf{A} \quad f : T' \in \mathsf{nonfinals}(\mathsf{A})}{\Gamma \vdash e.f : T'[T/\texttt{this}]}\ (\textsc{Expr-Field})$$

$$\frac{\Gamma \vdash p : T \quad \Gamma \vdash T \Uparrow \mathsf{A} \quad [\texttt{final}]\ f : T' \in \mathsf{fields}(\mathsf{A}) \quad \Gamma \vdash e : T'[p/\texttt{this}]}{\Gamma \vdash p.f := e : T'[p/\texttt{this}]}\ (\textsc{Expr-FieldAsgn})$$

$$\frac{\Gamma \vdash p_0 \Uparrow \mathsf{A} \quad \mathsf{C}(\overline{T\ x})\{\ \ldots\ \} \in \mathsf{constructors}(\mathsf{A}) \quad \Gamma \vdash \overline{p} : \overline{T}[\overline{p}/\overline{x}]}{\Gamma \vdash \texttt{new}\ p_0.\mathsf{C}(\overline{p}) : p_0.\mathsf{C}}\ (\textsc{Expr-New})$$

$$\frac{\Gamma \vdash p_0 \Uparrow \mathsf{A} \quad T\ m(\overline{T\ x})\{\ \ldots\ \} \in \mathsf{methods}(\mathsf{A}) \quad \Gamma \vdash \overline{p} : \overline{T[p_0/\texttt{this}][\overline{p}/\overline{x}]}}{\Gamma \vdash p_0.m(\overline{p}) : T[p_0/\texttt{this}][\overline{p}/\overline{x}]}\ (\textsc{Expr-Call})$$

$$\frac{\Gamma \vdash e : T \quad \Gamma, x : T \vdash e' : T' \quad \Gamma \vdash T'}{\Gamma \vdash \texttt{final}\ T\ x := e;\ e' : T'}\ (\textsc{Expr-Let}) \qquad \frac{\Gamma \vdash e : T \quad \Gamma \vdash e' : T'}{\Gamma \vdash e; e' : T'}\ (\textsc{Expr-Seq}) \qquad \frac{\Gamma \vdash e : T \quad \Gamma \vdash T \leq T'}{\Gamma \vdash e : T'}\ (\textsc{Expr-Subsumption})$$

Fig. 8. Expression Typing

as expected. (Expr-Null) is standard, giving any type to `null`. Rule (Expr-Error) is similar. It is used to enable the subject reduction result to go through in the presence of errors.

(Expr-Field) is as expected, giving the type of the field relative to the type of the object whose field it is (since the field type may contain `this`). (Expr-Final-Field) instead extends the type with a singleton type, though whether the resulting type is singleton depends upon the type T. For example, from $\Gamma \vdash x : x$ we might obtain that $\Gamma \vdash x.\mathit{ff} : x.\mathit{ff}$, which is a singleton type, but from $\Gamma \vdash x : y.\mathtt{D}$ we obtain $\Gamma \vdash x.\mathit{ff} : y.\mathtt{D}.\mathit{ff}$, which is not a singleton type.

(Expr-FieldAsgn) handles field assignment. For precise typing, the target containing the field must be a path. This path is then used to give a precise type to the value assigned to the field. This rule permits both assignment of both final and non-final fields. This is for uniformity, since final fields need to be initialised in the constructor. Attempts at assigning an initialised final field or attempts at accessing non-initialised fields result in errors, as previously discussed.

(Expr-New) describes the type of object creation The arguments $\overline{p}$ must have the types required by the constructor, where any appearance of the formal parameter has been updated by the actual paths. Note that rule (Expr-Call) is similar to (Expr-New) in the treatment of the arguments. In fact, *vc* unifies the two (following Beta and gBeta), eliminating methods.

The rules (Expr-New) and (Expr-Method) may seem somewhat odd in that they don't give a type to the receiver. Observe the following lemma:

Lemma 5. *If* $\Gamma \vdash p \Uparrow \mathsf{A}$*, then* $\Gamma \vdash p : p$.

$$\frac{\forall \mathit{class} \in P(\circ).\ \circ \vdash \mathit{class} \quad \sqsubseteq_i^+ \text{ is acyclic}}{\vdash P}\ (\textsc{Good-Program})$$

$$\frac{\Gamma, \overline{x} : \overline{T} \vdash e : T}{\Gamma \vdash T\ md(\overline{T\ x})\{\ e\ \}}\ (\textsc{Good-Method})$$

$$\frac{\begin{array}{c}\Gamma = \mathsf{A.C}, \mathtt{this} : \mathtt{this} \qquad \mathit{sup} = \{\mathsf{A.C} \mid \mathsf{C} \in \overline{\mathsf{C}'}\} \qquad \overline{\mathsf{A}} = \mathit{sup} \cup \{\mathsf{A}' \mid \vdash \mathsf{A.C} \sqsubseteq_f \mathsf{A}'\} \\ \vdash \mathsf{A}\ \mathit{cls} \text{ for each } \mathsf{A} \in \mathit{sup} \qquad \forall f : T \in \mathit{flds}.\ \Gamma \vdash T \qquad \mathsf{A.C} \vdash \mathit{classes} \qquad \Gamma \vdash \mathit{mthds} \\ \forall m, m' \in (\mathit{mthds} \cup \mathsf{methods}(\overline{\mathsf{A}}))\ .\ \mathit{name}(m) = \mathit{name}(m') \implies \mathit{type}(m) = \mathit{type}(m') \\ \forall f.\ f : T, f : T' \in (\mathit{flds} \cup \mathsf{fields}(\overline{\mathsf{A}})) \implies T = T' \qquad \Gamma, \overline{x : T} \vdash e : T' \text{ for some } T' \\ \forall \mathsf{A}' : \vdash \mathsf{A} \sqsubseteq_i \mathsf{A}' \text{ and } \mathsf{C}(\overline{T'\ x'})\{\ \ldots\ \} \in \mathsf{constructors}(\mathsf{A}') \implies \overline{T'} = \overline{T}\end{array}}{\mathsf{A} \vdash \mathtt{class}\ \mathsf{C}\ \mathtt{extends}\ \overline{\mathsf{C}'}\ \{\ \mathsf{C}(\overline{T\ x})\{\ e\ \}\ \mathit{flds}\ \mathit{classes}\ \mathit{mthds}\ \}}\ (\textsc{Good-Class})$$

Fig. 9. Program, Class and Method Typing.

With the above lemma, the additional premise $\Gamma \vdash p_0 : p_0$ could be redundantly added to the premises of (Expr-New), thus giving the target a type.

(Expr-Let) is for "let" expressions. It is standard, including the last premise to ensure that variable x, which could occur in the resulting type, does not escape its scope. This is standard in dependent typing [1]. (Expr-Seq) handles sequential composition in a straightforward fashion. Finally, (Expr-Subsumption) is the standard subsumption rule.

5.4 Well-formed Programs and Classes

Figure 9 defines well-formed programs and classes.

A program is well-formed if all its top level classes are well-formed, and there are no cyclic dependencies in the inheritance hierarchy.

A class is well-formed if its super classes are all valid classes *in the present context*, the fields have types that are well-formed in the current context, all methods and nested classes are well formed. In addition, all methods of the same name which are either declared in the present class or inherited *must* have the same type (where *name* gets a method's name and *type* gets a method's signature). Any inherited fields or local fields with the same name must have the same type; this last constraint simplifies matters.[5] Finally, the constructor must have the same type as all inherited constructors.

A method is well-defined in the usual sense, though the types of arguments may depend upon the path of other arguments and on the path of the receiver.

Finally, we will give some further results about well-formed programs. Define the the relation $(\mathit{fs}, \mathit{ms}, \mathit{cn}) \leq (\mathit{fs}', \mathit{ms}', \mathit{cn}')$ to express that the members $(\mathit{fs}, \mathit{ms}, \mathit{cn})$ enhance those from $(\mathit{fs}', \mathit{ms}', \mathit{cn}')$, while preserving types.

[5] Another approach would be to have a compiler phase which annotated field use with the class containing their definition, as is done by Java and C++ compilers. The description of such an approach is not difficult, but we consider this matter orthogonal to soundness.

Definition 4. *Define* $(fs, ms, cn) \leq (fs', ms', cn')$ *iff the following four conditions hold:*

- $(fs, ms, cn) \sqsubseteq (fs', ms', cn')$;
- *If* $f : T \in fs'$ *and* $f : T \in fs$, *then* $T' = T$;
- *If* $m \in names(ms) \cap names(ms')$, *then* $type(ms(m)) = type(ms'(m))$; *and*
- *If* $\mathsf{C}(\overline{Tx})\{\ ...\ \} \in cn$ *and* $\mathsf{C}(\overline{T'x})\{\ ...\ \} \in cn'$, *then* $\overline{T} = \overline{T'}$.

The following lemma guarantees that in well formed programs 1) expressions preserve their type in more specialised contexts; 2) class table entries of inheriting classes enhance those of the inherited class while preserving types, and 3) method defined in or inherited by an absolute class is well formed in that class context. The last guarantee is stronger than what is required by rule (Good-Class), because the rule only guarantees well-formedness for methods directly defined in the class.

Lemma 6. *For any well-formed program and its associated class table* CT*:*

- *If* $\vdash \mathsf{A}_1 \sqsubseteq_i^* \mathsf{A}_0$ *and* $\mathsf{A}_0, \Gamma \vdash e : T$, *then* $\mathsf{A}_1, \Gamma \vdash e : T$.
- *If* $\vdash \mathsf{A} \sqsubseteq_i^* \mathsf{A}'$, *then* $\mathsf{CT}(\mathsf{A}) \leq \mathsf{CT}(\mathsf{A}')$.
- *If* $T\ md(\overline{T\ x})\{\ e\ \} \in \mathsf{methods}(\mathsf{A})$, *then* $\mathsf{A}, \mathtt{this} : \mathtt{this}, \overline{x : T} \vdash e : T$.

6 Soundness of the Type System

The soundness of the Tribe type system is established using the standard technique of providing subject reduction and progress theorems due to Wright and Felleisen [21]. We now present a number of auxiliary notions and the rules for well-defined heaps.

In order to type run-time expressions, we need to extract a typing environment from a heap. This information includes the types of addresses and equations involving the values of final fields. This is achieved using the following definition:

Definition 5. *Given a heap* H, *we define operation* H^γ *as follows:*

$$\begin{aligned} \emptyset^\gamma &= \emptyset \\ (H, \iota \mapsto [\mathit{fds}\,]_{\mathsf{A.C}})^\gamma &= H^\gamma, \iota : \mathbf{v}.\mathsf{C}, \iota.\mathit{ff}_1 = v_1, \ldots, \iota.\mathit{ff}_n = v_n \\ &\qquad \textit{where } H(\iota).\mathtt{owner} = \mathbf{v} \textit{ and} \\ &\qquad \mathit{fds} = \{\overline{\mathit{ff} \mapsto v}, \overline{\mathit{ff}' \mapsto \mathtt{error}}, \overline{f \mapsto \_}\}, v_i \neq \mathtt{error} \end{aligned}$$

Note that only equations on *defined* final fields get recorded in H^γ. Uninitialised final fields, those containing `error`, do not contribute to the typing environment.

H^γ has the same shape as an environment Γ, so the judgements from the static semantics can be reused to type dynamic expressions. Rather than *explicitly* inserting the extraction operation, we use it *implicitly*. Thus, whenever we write, for example, $H \vdash T \Uparrow \mathsf{A}$, we really mean $H^\gamma \vdash T \Uparrow \mathsf{A}$. This applies to all rule shapes.

$$\frac{\mathsf{fields}(\mathsf{A}) = \bar{f} : \overline{T} \quad fds = \bar{f} \mapsto \bar{v} \quad H \vdash \bar{v} : \overline{T[\iota/\mathtt{this}]} \quad fds(\mathtt{owner}) \neq \mathtt{error}}{H \vdash \iota \mapsto [\, fds \,]_{\mathsf{A}}} \text{(Object)}$$

$$\frac{\Gamma \vdash \iota' : T \quad \iota'.f = v \in \Gamma}{\Gamma \vdash v : T.f} \text{(Expr-Eq)} \qquad \frac{\forall \iota \mapsto o \in H : \; H \vdash \iota \mapsto o}{\vdash H} \text{(Heap)} \qquad \frac{\vdash H \quad H \vdash e : T}{\vdash H, e : T} \text{(Config)}$$

Fig. 10. Heap Typing

6.1 Well-formed heaps

The type rules for heaps, objects and configurations are given Figure 10.

(Object) requires that all the fields have the correct type with respect to some absolute class and the present object id. The fact that the object id appears in the types takes care of the family requirements. In particular, the owner is a final field, and therefore, the type rules require that it should have type $\mathtt{this.owner}[\iota/\mathtt{this}]$, *i.e.,* for $H(\iota).\mathtt{owner} = \mathbf{v}$, we require $H^\gamma \vdash \mathbf{v} : \iota.\mathtt{owner}$, which is satisfied from the definition of H^γ and rule (Expr-Eq).

(Expr-Eq) is an additional rule used to establish equivalences between a type involving a path and object to which the path refers. (Heap) states that a heap is well-formed whenever all of its objects are well-formed. (Config) states that a well-formed heap and expression typed against it form a configuration with the type of the expression.

We have the following properties of well-formed heaps: 1) a well-formed heap can act as a well-formed environment; 2) the class corresponding to an address (as a singleton type) is the class of that address; 3) the surrounding class of the object surrounding an object is the surrounding class of the object's class; and 4) the chains of owners forms a finite tree rooted at $\circ$.

Lemma 7. *Assume* $\vdash H$. *Then:*

1. $H^\gamma \vdash \diamond$.
2. $H^\gamma \vdash \iota \Uparrow \mathsf{A}$ *if and only if* $H(\iota) = [\ldots]_{\mathsf{A}}$.
3. *Assume* $H(\iota).\mathtt{owner} = \mathbf{v}$. *If* $H^\gamma \vdash \iota \Uparrow \mathsf{A.C}$, *then* $H^\gamma \vdash \mathbf{v} \Uparrow \mathsf{A}$.
4. *The graph* $G = (\mathrm{dom}(H) \cup \{\circ\}, \{(\iota, \mathbf{v}) \mid \iota \in \mathrm{dom}(H) \wedge H(\iota).\mathtt{owner} = \mathbf{v}\})$ *is a tree with root* $\circ$.

6.2 Soundness

We can prove subject reduction and progress properties. We first state our variants of the standard meta-theory required to establish these results.

Lemma 8. *If* $\mathsf{A}, \mathtt{this} : \mathtt{this}, \Gamma \vdash e : T$ *and* $H \vdash \iota \Uparrow \mathsf{A}$, *then* $H, \Gamma[\iota/\mathtt{this}] \vdash e[\iota/\mathtt{this}] : T[\iota/\mathtt{this}]$.

Lemma 9 (Substitution). *If* $\Gamma, x : T, \Gamma' \vdash e : T'$, *and* $\Gamma \vdash v : T$, *then* $\Gamma, \Gamma'[v/x] \vdash e[v/x] : T'[v/x]$.

Definition 6 (Environment Extension, Heap Extension).

- *Define* $\Gamma \subseteq \Gamma'$ *iff* $\exists \Gamma''$ *with* $\Gamma' = \Gamma, \Gamma''$.
- *Define* $H \subseteq H'$ *iff* $H^\gamma \subseteq H'^\gamma$.

Note, that $H \subseteq H'$ implies that $dom(H) \subseteq dom(H')$, $\forall \iota \in \mathrm{dom}(H).type(H(\iota)) = type(H'(\iota))$, and that final fields are more defined in H' than in H. The last point means that final fields containing `error` in H may become defined in H', but already defined final fields must preserve their contents.

Lemma 10 (Extension). *If* $\Gamma \vdash e : T$, $\Gamma \subseteq \Gamma'$ *and* $\Gamma' \vdash \diamond$, *then* $\Gamma' \vdash e : T$.

Lemma 11 (Retraction). *If* $\Gamma, x : T' \vdash e : T$ *and* $x \notin fv(e) \cup fv(T)$, *then* $\Gamma \vdash e : T$.

Lemma 12 (Subformula Property). *If* $\Gamma \vdash E[e] : T$, *then there exist a* Γ' *and a* T' *such that* $\Gamma \subseteq \Gamma'$ *and* $\Gamma' \vdash e : T'$.

Lemma 13 (Replacement). *If* $\Gamma' \vdash e : T'$ *is a sub-derivation of* $\Gamma \vdash E[e] : T$, *and* $\Gamma' \vdash e' : T'$, *then* $\Gamma \vdash E[e'] : T$.

Definition 7. *A* redex *is an expression which has the form of one of the left hands sides of our reduction rules (apart from the two context rules):* $\mathbf{v}.f$, $\mathbf{v}.f := v$, `new` $\mathbf{v}.\mathsf{C}(\overline{v})$, $\mathbf{v}.m(\overline{v})$, `final` $T\ x := v;\ e$, or $v; e$.

Lemma 14. *Any expression* e *factored uniquely as an evaluation context* $E[-]$ *and another expression* e', *such that* $e = E[e']$. *Either* e' *is a redex or an* N.

And now for subject reduction and progress, which together give soundness[6].

Theorem 1 (Subject Reduction). *If* $\vdash H, e : T$ *and* $H, e \rightsquigarrow H', e'$, *then* $H \subseteq H'$ *and* $\vdash H', e' : T$.

Proof. By induction on the structure of e.

Theorem 2 (Progress). *If* $H \vdash e : T$, *then either* e *is a value, or there exist an* H' *and an* e' *such that* $H, e \rightsquigarrow H', e$.

Proof. By case analysis on the structure of e. The result depends on Lemma 6 to do all the work.

7 Advanced Tribe

This section addresses a number of issues concerning the design of Tribe.

[6] Proof sketches are available at `http://dsv.su.se/~tobias/appendix.pdf`.

(Expr-Call-Overridden)

$$\frac{\Gamma \vdash p \Uparrow \mathsf{A}' \quad \Gamma \vdash p.qual \Uparrow \mathsf{A} \quad \vdash \mathsf{A}' \sqsubseteq_i^* \mathsf{A} \qquad T\ m(\overline{T\ x})\{\ \ldots\ \} \in \mathsf{methods}(\mathsf{A}) \quad \Gamma \vdash \overline{p} : \overline{T[p/\mathtt{this}][\overline{p}/\overline{x}]}}{\Gamma \vdash p :: qual.m(\overline{p}) : T[p/\mathtt{this}][\overline{p}/\overline{x}]}$$

(Eval-Call-Overridden)

$$\frac{H(\iota) = [\ \ldots\ ]_{\mathsf{A}'} \quad T\ m(\overline{T\ z})\{\ e\ \} \in \mathsf{methods}(\mathsf{A})}{H, \iota :: \mathsf{A}.m(\overline{v}) \rightsquigarrow H, e[\iota/\mathtt{this}, \overline{v}/\overline{z}]}$$

Fig. 11. Type and Reduction Rules for Overridden Method Call

7.1 Calling Overridden Methods

Name clashes and ambiguous super calls are problems faced by every language with multiple inheritance. In Tribe, we force the caller to explicitly state which class' implementation a method should be bound to, though we do so using a relative path to the desired class. This is more stable under changes to the inheritance hierarchy, so long as the targeted class is inherited where it is expected. Tribe provides uniform access for both methods appearing in super- and further bound classes. The syntax is:

$$e \ ::= \ \ldots \quad | \quad p :: qual.m(\overline{p}) \qquad\qquad qual \ ::= \ \mathtt{owner}^*.\mathsf{C}^* \quad | \quad \mathsf{A}$$

In an overridden method call, *qual* is a qualifier which is used to refer to some inherited class, relative to the present context. It uses `owner` to enter surrounding classes and class names to go select specific classes within the surrounding class. The relative path to a class is resolved to a static class name at compiler time.

The type rule for overridden method call is found in Figure 11. The rule checks that the path of the target, merged with the qualifier, corresponds to some class the present class inherits from that contains the named method. The rest is the same as for ordinary method call.

An overridden method call binds to the implementation in the class denoted statically by the qualifier. Thus our approach is the same as that taken in C++, Java and Smalltalk, except that our paths are relative rather than absolute. At compile time, all method calls that use qualifiers are rewritten as follows: Replace the call $p :: qual.m(\overline{p})$ by $p :: \mathsf{A}.m(\overline{p})$, where $\Gamma \vdash p.qual \Uparrow \mathsf{A}$.

The reduction rule, which uses the absolute class name, appears in Figure 11.

The following example illustrates how the qualifier can be used to resolve ambiguous method calls (comments from `*.Y.A`'s point-of-view).

```
class X {  // owner.owner.X
  class A { void m() { ... } }
}
class Y extends X {   // owner
  class C { void m() { ... } }
  class D { void m() { ... } }
```

```
  class A extends C, D { // inherits classes X.A, Y.C, Y.D
    this::owner.C.m();         // uses Y.C's impl. of m()
    this::owner.D.m();         // uses Y.D's impl. of m()
    this::owner.owner.X.A.m(); // uses A.B's impl. of m()
  }
}
```

As in C++, qualifiers can be used for paths not starting with `this`.

Finally, recall that class construction (§3) non-deterministally resolved method calls when no overriding was present. A simple super-call can break this non-determinism. For example, assume that a class `C` inherits `A` and `B`, which both contain method `m`. The following code in `C` chooses the `m` method from `A`.

```
void m() { this::owner.A.m(); }
```

7.2 Adoption and Over-the-top Types

In Tribe, every class contains an nested class `Object` as the root of the nested subclass hierarchy. Furthermore, subclassing occurs only between classes nested within the same class. While offering simplicity, this approach suffers from practical limitations. In this section, we describe the notions of *adoption* and of *over-the-top types* which overcome these limitations.

Consider the following top-level classes—*not* nested within another class:

```
class Observer {
  this.owner.Subject subject;  // relative type
}
class Subject { }
```

The `subject` field of `Observer` refers to a `Subject` type using a relative type rather than the absolute type ∘.`Subject`. This is okay as the absolute class of `this.owner` is ∘, which contains class `Subject`. Such relative types appearing in a top-level class—*over-the-top types*—are useful in combination with adoption.

Adoption occurs when a top-level class hierarchy is grafted into some other class, enabling better reuse of code which may have been written without family polymorphism in mind. The following syntax incorporates adoption into Tribe:

$$\mathit{class} ::= \texttt{class C extends } \overline{\texttt{C}} \texttt{ \{ } \mathit{cnstr}\ \mathit{fld}^*\ \mathit{class}^*\ \mathit{mthd}^*\ \mathit{adpt}^* \texttt{ \}}$$

$$\mathit{adpt} ::= \texttt{adopt } \circ\texttt{.C}$$

Note that adopting classes which are not at the top-level has no clear semantics.

The following code uses adoption to graft classes ∘.`Observer` and ∘.`Subject` into ∘.`Spy` to produce the classes ∘.`Spy.Observer` and ∘.`Spy.Subject`.

```
class Spy {
  adopt *.Observer;
  adopt *.Subject;
  class Observer { ... }  // further binds adopted Observer
  class MaleSubject extends Subject {  } // extension of adopted class
}
```

Class ∘.Spy.Observer inherits the field this.owner.Subject subject, whose type now corresponds to ∘.Spy.Subject. Without over-the-top types, this field would still have type ∘.Subject, and losing the advantages of virtual classes.

The following minor additions and modifications are required to incorporate adoption into the type system. Firstly, the following additional rules are added to subclassing and further binding resolution:

$$\frac{\texttt{class C} \cdots \texttt{\{} \ldots \overline{\texttt{adopt} \circ.\texttt{C}} \texttt{\}} \in P(\mathsf{A})}{\vdash \mathsf{A.C.C}_i \ \mathit{cls}} \ (\textsc{Adopt-Class}) \qquad \frac{\texttt{class C} \cdots \texttt{\{} \ldots \overline{\texttt{adopt} \circ.\texttt{C}} \texttt{\}} \in P(\mathsf{A})}{\vdash \mathsf{A.C.C}_i \sqsubseteq_s \circ.\mathsf{C}_i} \ (\textsc{Adopt-SubClass})$$

The only change that need be made is to (Class-Table) is to add $\vdash \circ.\mathsf{C}\ \mathit{cls}$ for each adopted class, and to check that the class C is not already inherited. The final change required is ensuring that every top-level class referred to in an adopted class through an over-the-top type is also adopted. If, for example, ∘.Subject above is not adopted, the type of field subject in ∘.Spy.Observer would refer to a non-existant class. Worse, a totally different class with the same name could be introduced into ∘.Spy, which would be unsound.

We have not proven the soundness of this extension, but believe it to be so.

Adoption sometimes requires the adoption of many classes, potentially *every* top-level class. We plan to investigate alternatives designs as future work.

7.3 Dynamic Type Casts

Extending Tribe to support dynamic casts is straightforward, as every object carries around sufficient information, an absolute class and an owner. The type rule is obvious. The reduction rules for type cast are as follows:

$$\frac{\Gamma \vdash e : T' \quad \Gamma \vdash T}{\Gamma \vdash (T)e : T} \ (\textsc{Expr-Cast}) \qquad \frac{H \vdash v : T}{H, (T)v \rightsquigarrow H, v} \ (\textsc{Eval-TypeCast-Success}) \qquad \frac{\neg(H \vdash v : T)}{H, (T)v \rightsquigarrow H, \texttt{error}} \ (\textsc{Eval-TypeCast-Fail})$$

7.4 Decidability of Type Checking

As our type system is not syntax directed, it is not clear whether type checking is decidable. At the time of writing, we have determined that $\Gamma \vdash T \Uparrow \mathsf{A}$ is decidable, and that decidability depends on the decidability of subtyping. Note that *vc*'s type system is decidable, and we believe that a fragment of our system corresponding to *vc* can easily be carved out (and would be decidable).

8 Related Work

There has been a range of research on family polymorphism since Ernst's original proposal [6], built on gBeta [5]. Systems that support family polymorphism can be divided into two categories: those that support *class families* and those that support *object families*. In class families, classes are nested in other classes,

whereas in object families, classes are nested in objects and types depend on paths. In terms of our initial graph example, class families can prevent non-coloured nodes from appearing in a coloured graph, but cannot prevent attempts at connecting nodes belonging to different graph instances. Object families can do both.

In this sense Concord [12] and CAESAR [15] groups are class families as well as the nested class system of .FJ [11]. The class families of Concord and .FJ are also "shallow" as they do not allow more than one level of nesting. Nystrom et al.'s Jx [16] presents a notion of nested inheritance that works much like class families with support for nestings of arbitrary depth. To enable family-polymorphic methods, Jx introduces the notion of *prefix types* of the form $\mathsf{C}[\mathsf{C}']$ that denote the innermost enclosing class of class C' that is a subtype of C. To solve a similar problem, .FJ uses a bounded type parameter that is passed in separately. We believe that neither of these solutions is as intuitive as using relative paths based on `owner` or `out`.

Scala [8] and *vObj* [17] have both object family and class family polymorphism, although *vObj* does not support virtual classes, only virtual types. They both lack an `out` or `outer` construct but can use the syntax `C.this` to refer to the innermost object enclosing this with type `C`. Note that Tribe allows the description of such a type using a path starting from any field or variable, not just `this`; and allows classes at any point in a part.

Our proposal is closest both in spirit and expressiveness to *vc* [7]. The main difference is that in *vc*, all paths are relative to the current `this`, whereas Tribe supports absolute paths; paths beginning with variables; and paths intermixing classes and objects in any order. One practical benefit is the way Tribe handles family parameters. Writing our example from page 3 in *vc* would not allow the use of `e.owner` to obtain the class enclosing the argument edge, requiring the "family" class to be passed in separately. Tribe also has richer types and subtypes. For example, *vc* only defines subtyping for class names in the same path, whereas Tribe defines subtyping for variables and absolute class names.

Table 1 summarises this brief comparison of related systems. Ernst et al.'s *vc* paper includes a more comprehensive account [7].

Table 1. Comparison. [1]) through `myGrp`, only one level and for class families only; [2]) through prefix types, for class families only; [3]) only from `this`; [4]) only from this through `C.this` that denotes the innermost class enclosing `this` with type `C`.

	Concord	Caesar	.FJ	Jx	*vc*	*vObj*	Scala	Tribe
Object families	no	no	no	no	yes	yes	yes	yes
Class family types ($\mathsf{C}.\overline{\mathsf{C}'}$)	yes	yes	yes	yes	no	yes	yes	yes
Relative paths	yes[1]	no	no	yes[2]	yes[3]	yes[4]	yes[4]	yes
Over-the-top types	no	no	no	no	no	no	no	yes
Adoption	no	no	no	no	no	no	no	yes

9 Conclusions and Future Work

We have presented Tribe, a type system for generalised class and family polymorphism. Tribe is simpler and more powerful than existing systems, with more flexible path-based types and extended subtype relations, resulting in a more expressive of the calculus with little additional conceptual overhead.

We are currently implementing a prototype programming system based upon Tribe, and more generally, we hope Tribe will serve as a suitable basis for research in a range of areas:

Ownership Types The savvy reader may have cottoned onto the fact that we have used the brand-name `owner` to refer to the surrounding instance. Our original motivation for doing this work was related to ownership types, but we found the formalism for *vc* to be too unwieldy to modify. Hence, we devised our own, incorporating a number of extensions that we found useful. The work on ownership types in this setting will be reported in future work.

Dynamic Nesting Structure The `owner` field must be final for soundness. But this prevents object aggregations (or nesting structures) from changing or evolving. It would be interesting to determine what would be required to change this. This can probably be achieved by linearising the objects whose type/ownership/structure will change, leveraging on DeLine and Fähndrich's work on type states [4], and Clarke and Wrigstad's External Uniqueness [3, 22].

Generics Adding generics and/or virtual types to our language would be, we expect, relatively simple to do (following perhaps Scala [8]). Type checking Scala is, unfortunately, undecidable. Venturing into this territory thus will require careful steps in order to remain decidable (assuming we can establish decidability for our system).

References

1. David Aspinall and Martin Hofmann. Dependent types. Chapter in [18].
2. Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.
3. David Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP)*, Darmstadt, Germany, 2003.
4. Robert DeLine and Manuel Fähndrich. The fugue protocol checker: Is your software baroque? Technical Report MSR-TR-2004-07, Microsoft Research, 2003.
5. Erik Ernst. *gBeta—A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritiance*. PhD thesis, University of Aarhus, Denmark, 1999.
6. Erik Ernst. Family polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, London, UK, 2001. Springer-Verlag.
7. Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Proceedings of Principles of Programming Languages (POPL)*, Charleston, South Carolina, USA, January 2006.

8. Martin Odersky et al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.
9. Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, 2003.
10. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. Addison-Wesley Professional, 3rd edition, 2005.
11. Atsushi Igarashi, Chieri Saito, and Mirko Viroli. Lightweight family polymorphism. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*, volume 3780 of *LNCS*, Tsukuba, Japan, 2005.
12. Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. Simple dependent types: Concord. In *6th ECOOP Workshop on Formal Techniques for Java-like Languages*, June 2004.
13. Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, MA, USA, 1993.
14. Jan-Willem Maessen and Xiaowei Shen. Improving the java memory model using crf. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2000.
15. Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In Mehmet Aksit, editor, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 90–100, Boston, USA, March 2003.
16. Nathaniel Nystrom, Sephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proceedings of Objects, Programming Langages, Systems and Applications (OOPSLA)*, Vancouver, Canada, October 2004.
17. Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, Springer LNCS, July 2003.
18. Benjamin Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2004.
19. Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In Luca Cardelli, editor, *ECOOP 2003 – Object-Oriented Programming: 17th European Conference*, volume 2473 of *Lecture Notes In Computer Science*, pages 248–274. Springer-Verlag, July 2003.
20. Bjarne Stroustroup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.
21. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
22. Tobias Wrigstad. *Ownership-Based Alias Management*. PhD thesis, Stockholm University, 2006. Forthcoming.