Inner Classes visit Aliasing

Matthew Smith, Sophia Drossopoulou

Imperial College of Science Technology and Medicine

Abstract. Inner classes appear nested within class definitions. They may access any members of the classes in which they are contained. The interplay between inner classes, aliasing and subclasses can make resolution of such accesses intricate.
We offer a succinct model for member classes, in which we highlight these intricacies and we prove soundness.

1 Introduction

Inner classes were introduced into Java with version 1.1, and were motivated mainly by the AWT and later Swing[1]. Java's event model requires the use of many small listener and adaptor classes for which inner classes are perfectly suited. This continues to be the way inner classes are most often used. These applications usually only apply the basic features of inner classes. Thus, mamny developers may be surprised by the issues arising from inner classes. Inner classes (or their equivalents) also exist in other object-oriented languages, notably Beta [2]. They can be useful and powerful.

An inner class in Java can access any of the members of any class in which it is contained[1]. To allow this, an instance of an inner class has to be bound to (have a host) an instance of the containing class. Through inheritance an object may require *several* hosts; these may be of the same class. Moreover, through inheritance, an object may have *one* object fulfilling the role of *many* hosts. Such issues cannot be reflected in a functional model of inner classes, e.g. [3]. They are reflected in the model presented here.

In Section 2 we present the basics as well as some more complex behaviours of inner classes, and show examples. In Section 3 we discuss $\mathcal{L}_{\text{jm}}$ design decisions. Section 4 presents $\mathcal{L}_{\text{jm}}$, a compact object oriented language with inner classes and aliasing. We present the operational semantics and type system with particular focus on inner classes. In Section 5 we state important properties of $\mathcal{L}_{\text{jm}}$, and esp. soundness. In section 6 we describe translation to top-level classes. In Section 7 we offer observations we have made about inner classes. In section 8 we draw conclusions and outline further work.

A version of this paper including the Appendices is available at http : //www.doc.ic.ac.uk/ mjs198/InnerClasses.

2 Introduction to Java Inner Classes [2]

2.1 Basic Concepts

A class, `A.B`, is *inner* to a class `A` if it is declared inside class `A`; class `A` is then called an *enclosing class* of `A.B`. An instance of an inner class has a link to an instance of the enclosing class which we call the *host* object. Thus, an object of class `A.B` has a host of class `A`, which it can access through the expression `A.this`. We say that `A.B` *requires* a host of class `A`.

Inner classes may extend and may be extended by other classes. A class extending an inner class 'inherits' the requirement for host objects. These inherited hosts ensure that inherited methods which access the (inherited) host can be executed. Methods defined in the subclass may not access

[1] particularly useful for AWT/Swing applications as it avoids explicit passing of call back objects
[2] Java supports four kinds of inner class: member, local, anonymous and static member classes. In this work, as in FJI, we consider member classes only.

the inherited host since the subclass is not inner to (not lexically contained in) the inherited enclosing class.

In Fig. 1[3] classes M.N and M.O are inner classes of M (and as such have names of the form <enclosing class name>.<simple name>) and M.O extends M.N. This is shown diagrammatically in a UML-like style on the RHS of the figure. Objects of class M.O require *two* host objects of class M: one because M.O extends M.N, and one because M.O is inner to M.

The methods f and g access host objects: In f the expression M.this returns the host object of type M required by class M.N, while in g the expression M.this returns the host object of type M required by class M.N. Also, in f the expression M.N.this is a synonym for this – view a class as trivially inner to itself.

```
class M {
        int i;
        class M.N {
                int f() {
                        return M.this.i;
                }
        }
        class M.O extends M.N {
                int g() {
                        return M.this.i;
                }
        }
}
```

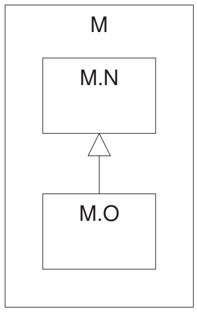

Fig. 1. Code and Diagram for Interesting Inner Classes

Figure 2 shows a possible object configuration. in a UML-like style: 1 is an object of class M, with field i containing the value 1. The arrows indicate links to hosts. Some objects have two hosts, e.g. 4 has 1 and 2 as hosts. In order to distinguish between the host required by the object's class and the inherited hosts, we annotate the links with the name of the class requiring the host. For example, for object 4, the M.N host is object 1 and the M.O host (inherited) is object 2.[4]

Table 3 shows the results of calling methods f and g on these objects. Notice, that although the method body for f and g are identical, the call of f on 4 returns a different result than the call of g on 4. This is so, because M.this in f indicates the host required by M.N, whereas M.this in g indicates the host required by class M.O. Therefore, evaluation of access to hosts depends on which class defined the method being executed.

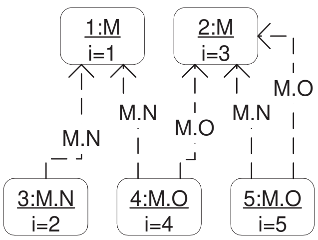

Fig. 2. Possible object configurations for Fig. 1

Receiver	Method	Result
3	f	1
4	f	1
5	f	3
4	g	3
5	g	3

Fig. 3. Method call results

[3] In Java qualified names, e.g. M.N, are optional in some contexts and illegal in class declarations. We use qualified names here for consistency with Sec. 4. We purposefully omit constructors.

[4] Annotating the link with the class of the hosts (rather than with the class requiring the host) would not work because several hosts may have the same class, as for example the hosts of class M.O.

2.2 A Note on Terminology

The terminology of this paper is a little different to that of the JLS [4]. The JLS terminology is more descriptive but consequently more complex. The JLS talks of *enclosing instances* rather than hosts. In order to accurately describe the semantics of the language the JLS further qualifies terminology, for example the *nth lexically enclosing instance* is the host relating to the class in which the current class is nested n levels deep. We choose to describe demonstrate the semantics informally through example and formally through operational semantics and type rules.

2.3 Advanced Concepts: Aliasing and Deep Nesting of Inner Classes

The interplay of deep inner classes (those which are nested within other inner classes) with aliasing cause some more complex properties, which we demonstrate through the classes from Fig. 4 and objects in Fig. 5. The properties are:

P1 Inner classes may be nested to any level.
Here, `A.B.C.D` is inner to `A.B.C` which is inner to `A.B` while `A.B` is inner to `A`.

P2 An object may have more than one host.
Here, objects `2` and `4` have two hosts, object `7` has three hosts.

P3 The host of an object may be of a subclass of the corresponding enclosing class.
Here, the host of `6` is `5`, which has class `F`, a subclass of `A.B.C`.

P4 An object may access through the host mechanism several - but not all - objects in the transitive closure of the host relation. Namely, an object of class $\mathtt{A}_1 \ldots \mathtt{A}_\mathtt{n}$ may access any host $\mathtt{A}_1 \ldots \mathtt{A}_\mathtt{k}$`.this` for $k \leq \mathtt{n}$; also, if $\mathtt{A}_1 \ldots \mathtt{A}_\mathtt{n}$ is a subclass of $\mathtt{B}_1 \ldots \mathtt{B}_\mathtt{m}$, when executing an inherited method it may access any host $\mathtt{B}_1 \ldots \mathtt{B}_\mathtt{j}$`.this`.
In our example, the methods of `A.B.C.D` can access host objects `A.this`, `A.B.this` and `A.B.C.this`. For object `6` these hosts are 1, 4 and 5 respectively. A method of `A.B.C` inherited in class `F` can access hosts `A.this` and `A.B.this`, so, object `5` could reach objects `1` and `4` through the host mechanism.

P5 All other objects in the transitive closure of the host relationship cannot be accessed through the host mechanism.
Here, because `F` is not inner to `A.B`, methods in `F` cannot contain the expression `A.B.this`. Therefore, object `5` will never access object `3` through the host mechanism.

3 Main Design Decisions for $\mathcal{L}_{\mathrm{jm}}$

The issues discussed in the previous section motivated our design choices for our formalism, $\mathcal{L}_{\mathrm{jm}}$.

Object Representation An object may have many hosts which can be distinguished by the name of the class requiring them. Consequently we represent objects in the form:

$$[\![\ \mathsf{C} \mid\mid \mathsf{C}_1 : \iota_1, \ldots, \mathsf{C}_\mathsf{n} : \iota_\mathsf{n} \mid\mid \mathsf{f}_1 : \mathsf{v}_1, \ldots, \mathsf{f}_\mathsf{r} : \mathsf{v}_\mathsf{r}\ ]\!]$$

where C is the class of the object. $\mathsf{C}_1 : \iota_1, \ldots, \mathsf{C}_\mathsf{n} : \iota_\mathsf{n}$ are the hosts required by $\mathsf{C}_1, \ldots, \mathsf{C}_\mathsf{n}$ respectively, and $\mathsf{f}_1 : \mathsf{v}_1, \ldots, \mathsf{f}_\mathsf{r} : \mathsf{v}_\mathsf{r}$ are the names and values of fields.

The Store Stores capture both heaps and stacks:

$$\begin{array}{ll}
\mathit{store} & = (\ \{\mathsf{this}\ \} \rightarrow \mathit{addr}) \cup (\ \{x\} \rightarrow \mathit{val}) \cup (\mathit{addr} \rightarrow \mathit{object})) \\
\mathit{val} & = \{\mathit{null}\ \} \cup \mathit{addr} \\
\mathit{dev} & = \{\mathit{nullPntrExc}, \mathit{stuckErr}\ \} \\
\mathit{object} & = \{\ [\![\ \mathsf{C} \mid\mid \mathsf{C}_1 : \iota_1, \ldots, \mathsf{C}_\mathsf{k} : \iota_\mathsf{k} \mid\mid \mathsf{f}_1 : \mathsf{v}_1, \ldots, \mathsf{f}_\mathsf{r} : \mathsf{v}_\mathsf{r}\ ]\!]\ \} \\
\mathit{addr} & = \mathbb{N}
\end{array}$$

where $\mathsf{v}, \mathsf{v}', \mathsf{v}_1, \ldots$ range over *val*, $\iota, \iota', \iota_1, \ldots$ range over *addr*, $\sigma, \sigma', \sigma_1, \ldots$ range over *store* and $\mathsf{C}, \mathsf{C}', \mathsf{C}_1, \ldots$ range

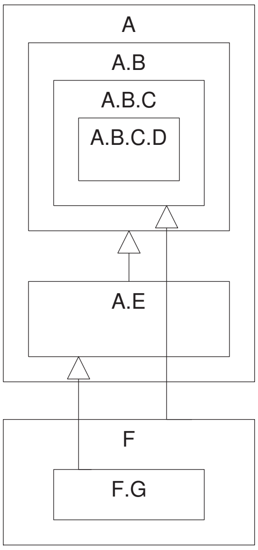

Fig. 4. A complex inner class hierarchy

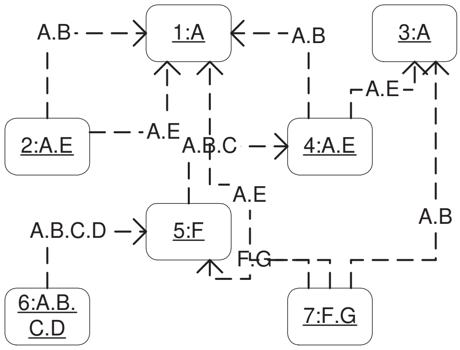

Fig. 5. Possible object configuration

over class names and f,f',f_1,... range over field names. The identifier this refers to the current receiver and x to the current method argument. Store lookup, $\sigma(\iota)(\mathsf{f})$, and update, $\sigma\ [\iota \mapsto \sigma(\iota)[\mathsf{f} \mapsto \mathsf{v}]]$, have the obvious meaning.

Assuming no fields in the classes, the objects of Fig. 5 would be represented in σ_0 as:

$$
\begin{aligned}
\sigma_0(1) &= [\![\ \mathsf{A}\ ||\ ||\ \ ]\!] \\
\sigma_0(2) &= [\![\ \mathsf{A.E}\ ||\ \mathsf{A.E}:1, \mathsf{A.B}:1||\ \ ]\!] \\
\sigma_0(3) &= [\![\ \mathsf{A}\ ||\ ||\ \ ]\!] \\
\sigma_0(4) &= [\![\ \mathsf{A.E}\ ||\ \mathsf{A.E}:3, \mathsf{A.B}:1||\ \ ]\!] \\
\sigma_0(5) &= [\![\ \mathsf{F}\ ||\ \mathsf{A.B.C}:4||\ \ ]\!] \\
\sigma_0(6) &= [\![\ \mathsf{A.B.C.D}\ ||\ \mathsf{A.B.C.D}:5||\ \ ]\!] \\
\sigma_0(7) &= [\![\ \mathsf{F.G}\ ||\ \mathsf{F.G}:5, \mathsf{A.E}:1, \mathsf{A.B}:3||\ \ ]\!]
\end{aligned}
$$

Operational Semantics Format Execution needs to keep track of the class defining the method currently being executed. Therefore, the operational semantics has the format:

$$\mathsf{P}, \mathsf{C} \vdash \mathsf{e}\,,\,\sigma \ \rightsquigarrow\ \mathsf{r}\,,\,\sigma'$$

where P is the program, C is the class definting the method currently being executed, e is the expression to be evaluated, σ is the store before evaluation, r is the result of evaluation (a value or a deviation) and σ' the store after evaluation.

Host Object Lookup Host lookup depends on the store, the address ι whose host is looked up, the class C containing the expression being executed, and the class C' of the host being looked up. For store σ as above, we have: The function is defined as follows;

$$
\mathcal{H}(\sigma, \iota, \mathsf{C}, \mathsf{C}') = \begin{cases}
\iota & if\ \ \mathsf{C} = \mathsf{C}' \\
\mathcal{H}(\sigma, \iota', \mathsf{C}'', \mathsf{C}') & if\ \ \mathsf{C} \neq \mathsf{C}' \ \ \wedge\ \ \mathsf{C} = \mathsf{C}''.\mathsf{c} \\
 & \wedge\ \sigma(\iota) = [\![\ \mathsf{C}'''\ ||\ \ldots\ \mathsf{C} : \iota'\ \ldots ||\ \ldots\ ]\!] \\
udf & otherwise
\end{cases}
$$

In our example:

$$\begin{aligned}\mathcal{H}(\sigma_0, 6, \mathsf{A.B.C.D}, \mathsf{A}) &= \mathcal{H}(\sigma_0, 5, \mathsf{A.B.C}, \mathsf{A}) \\ &= \mathcal{H}(\sigma_0, 4, \mathsf{A.B}, \mathsf{A}) \\ &= \mathcal{H}(\sigma_0, 1, \mathsf{A}, \mathsf{A}) \\ &= 1\end{aligned}$$

4 $\mathcal{L}_{\text{jm}}$, our formal model of inner classes

In this section we present $\mathcal{L}_{\text{jm}}$, a compact, class-based, object-oriented language with aliasing and member classes. $\mathcal{L}_{\text{jm}}$ is an extension of earlier work on the formalization of Java subsets, e.g. [5, 6]. $\mathcal{L}_{\text{jm}}$ is similar to Java. We ignore features of Java which are not immediately relevant. e.g. threads, casting etc.

For the sake of simplicity, we impose the following restrictions 1-8. Also, the deviations 9-12 differ slightly from Java, and allow for a simpler model:

1. The receiver of field access and method calls must be made explicit (no implicit this).
2. In instantiation of inner classes, the host object must be made explicit (no implicit this, we require this.new A.B()).
3. Full names must be used for inner class types in field and method declarations.
4. Fully qualified names must be used for all host object accesses e.g. always `A.B.this`, even when Java would allow `B.this` as a synonym.
5. No unqualified this expression, `<class name>.this` is a synonym.
6. No sequences of expressions.
7. No constructors - objects are instantiated with default values.
8. Methods have precisely one parameter.
9. Methods implicitly return the result of evaluating their body.
10. Inner classes are declared with their full name, e.g. `A.B` as opposed to just `B` in class `A`.
11. Inner classes are instantiated with their full name, so `a.new A.B()` rather than `a.new B()`.
12. Inherited hosts are provided to `new` through a sequence of expressions.

4.1 $\mathcal{L}_{\text{jm}}$ Syntax

$$\begin{array}{lcl}
\mathsf{progr} & ::= & \mathsf{class}^* \\
\mathsf{class} & ::= & \textit{class}\ \mathsf{C}\ \textit{extends}\ \mathsf{C}\ \{\ \mathsf{field}^*\ \mathsf{meth}^*\ \mathsf{class}^*\} \\
\mathsf{field} & ::= & \mathsf{C}\ \ \mathsf{f} \\
\mathsf{meth} & ::= & \mathsf{C}\ \ \mathsf{m}\,(\mathsf{C}\,\mathsf{x})\{\mathsf{e}\} \\
\mathsf{C} & ::= & \mathsf{c}\ \mid\ \mathsf{c.C} \\
\mathsf{e} & ::= & \mathsf{e.f}\ \mid\ \mathsf{e.f}\ :=\ \mathsf{e}\mid\ \mathsf{e.m(e)}\ \mid\ \textit{new}\ \mathsf{c}(\mathsf{e}^*)\ \mid\ \mathsf{e}.\textit{new}\ \mathsf{C}(\mathsf{e}^*)\ \mid\ \mathsf{x}\mid\ \mathsf{C}.\textit{this}\ \mid\ \textit{null}
\end{array}$$

The expression *new* c(e_1,...,e_n) instantiates non-inner classes (hence the restriction to simple class names). The expression e_0.*new* C(e_1,...,e_n) instantiates inner classes, where e_0 is the host object for C, as in Java, and e_1,...e_n supply inherited host objects - if any. The inherited hosts are supplied starting from the most specific superclass. For example, instantiation of class `F.G` from Fig. 4 would have the form $<$ F.G host $>$ .new F.G($<$ A.E host $>$, $<$ A.B host $>$).

4.2 Lookup Functions and Judgements on Programs

Full definitions of the following judgements are available in [7]

The judgement $\mathsf{P}\vdash\mathsf{C}\diamond_c$ states that C is a valid class name in the program. The judgement $\mathsf{P}\vdash\mathsf{C} \leq \mathsf{C}'$ expresses that C *is a subclass of* C′ and the relation is reflexive and transitive. Similarly, $\mathsf{P}\vdash\mathsf{C} \preceq \mathsf{C}'$ expresses that C is inner to C′ and again the relation is reflexive and transitive. The judgement $\vdash \mathsf{P}\diamond_\mathsf{n}$ expresses that all class names are unique, that field and method names are unique per class, that the name of an inner class is the name of the class that contains it extended with a simple class name and that the inheritance hierarchy is acyclic. We implicitly require $\vdash \mathsf{P}\diamond_{\mathsf{n}}$.

The function $\mathcal{C}(\mathsf{P},\mathsf{C})$ returns the definition of class C in program P. The function $\mathcal{MD}(\mathsf{P},\mathsf{C},\mathsf{m})$ returns the definition of the method m in the class C of program P. The function $\mathcal{M}(\mathsf{P},\ \mathsf{C},\ \mathsf{m})$ returns the pair consisting of the name of the class containing the method definition and the definition of the method. The function $\mathcal{F}(\mathsf{P},\mathsf{C},\mathsf{f})$ returns the type of field f in class C. $\mathcal{F}s(\mathsf{P},\mathsf{C})$ returns the set of all fields names declared in C and its superclasses. All above classes return *udf* if C is not defined in P.

$\mathcal{SI}s$ produces the list of the super classes of a given class which are also inner classes, ordered on the inheritance relation. In other words, $\mathcal{SI}s$ is the list of superclasses requiring hosts.

$$\mathcal{S}(\mathsf{P},\mathsf{C}) \ = \begin{cases} \mathsf{C}' & \text{if } \mathcal{C}(\mathsf{P},\mathsf{C}) = \mathsf{class\ C\ extends\ C'}\ \ldots \\ \mathit{udf} & \text{if } \mathcal{C}(\mathsf{P},\mathsf{C}) = \mathit{udf} \text{ or } \mathsf{C} = \mathsf{Object} \end{cases}$$

$$\mathcal{SI}_{\mathcal{S}}(\mathsf{P},\mathsf{C}) = \begin{cases} \mathsf{C} :: \mathcal{SI}_{\mathcal{S}}(\mathsf{P},\mathcal{S}(\mathsf{P},\mathsf{C})) & \text{if } \mathsf{P} \vdash \mathsf{C} \diamond_{\mathsf{c}} \text{ and } \mathsf{C} = \mathsf{C}'.\mathsf{c} \\ \mathcal{SI}_{\mathcal{S}}(\mathsf{P},\mathcal{S}(\mathsf{P},\mathsf{C})) & \text{if } \mathsf{P} \vdash \mathsf{C} \diamond_{\mathsf{c}} \text{ and } \mathsf{C} = \mathsf{c} \\ \epsilon & \text{otherwise} \end{cases}$$

4.3 $\mathcal{L}_{\mathbf{jm}}$ Operational Semantics

$$\frac{}{\mathsf{P},\mathsf{C} \vdash \mathsf{v}, \sigma \rightsquigarrow \mathsf{v}, \sigma}\ \mathit{val} \qquad \frac{}{\mathsf{P},\mathsf{C} \vdash \mathsf{x}, \sigma \rightsquigarrow \sigma(\mathsf{x}), \sigma}\ \mathit{var} \qquad \frac{\mathcal{H}(\sigma, \sigma(\mathsf{this}), \mathsf{C}, \mathsf{D}) = \iota}{\mathsf{P},\mathsf{C} \vdash \mathsf{D.this}, \sigma \rightsquigarrow \iota, \sigma}\ \mathit{qThis}$$

$$\frac{\begin{array}{l}\mathsf{P},\mathsf{C} \vdash \mathsf{e}, \sigma \rightsquigarrow \iota, \sigma' \\ \sigma'(\iota)(\mathsf{f}) \neq \mathit{udf}\end{array}}{\mathsf{P},\mathsf{C} \vdash \mathsf{e.f}, \sigma \rightsquigarrow \sigma'(\iota)(\mathsf{f}), \sigma'}\ \mathit{fld} \qquad \frac{\begin{array}{l}\mathsf{P},\mathsf{C} \vdash \mathsf{e}, \sigma \rightsquigarrow \iota, \sigma'' \\ \mathsf{P},\mathsf{C} \vdash \mathsf{e}', \sigma'' \rightsquigarrow \mathsf{v}, \sigma''' \\ \sigma'''(\iota)(\mathsf{f}) \neq \mathit{udf} \\ \sigma' = \sigma'''[\iota \mapsto \sigma'''(\iota)[\mathsf{f} \mapsto \mathsf{v}]]\end{array}}{\mathsf{P},\mathsf{C} \vdash \mathsf{e.f} := \mathsf{e}', \sigma \rightsquigarrow \mathsf{v}, \sigma'}\ \mathit{fldass}$$

$$\frac{\begin{array}{l}\mathsf{P},\mathsf{C} \vdash \mathsf{e}, \sigma \rightsquigarrow \iota, \sigma_0 \\ \mathsf{P},\mathsf{C} \vdash \mathsf{e}', \sigma_0 \rightsquigarrow \mathsf{v}', \sigma_1 \\ \sigma_1(\iota) = [\![\ \mathsf{C}' \,||\, \ldots \,||\, \ldots\ ]\!] \\ \mathcal{M}(\mathsf{P},\mathsf{C}',\mathsf{m}) = (\mathsf{t\ m(t_1\ x)}\ \{\mathsf{e}''\}, \mathsf{C}'') \\ \sigma' = \sigma_1[\mathsf{this} \mapsto \iota][\mathsf{x} \mapsto \mathsf{v}'] \\ \mathsf{P},\mathsf{C}'' \vdash \mathsf{e}'', \sigma' \rightsquigarrow \mathsf{v}, \sigma''\end{array}}{\mathsf{P},\mathsf{C} \vdash \mathsf{e.m(e')}, \sigma \rightsquigarrow \mathsf{v}, \sigma''[\mathsf{this} \mapsto \sigma(\mathsf{this})][\mathsf{x} \mapsto \sigma(\mathsf{x})]}\ \mathit{methcall}$$

$$\frac{\begin{array}{l}\mathsf{P} \vdash \mathsf{c}\diamond_{\mathsf{c}} \\ \mathsf{P},\mathsf{C} \vdash \mathsf{e_i}, \sigma_{\mathsf{i}-1} \rightsquigarrow \iota_{\mathsf{i}}, \sigma_{\mathsf{i}} \quad 1 \leq \mathsf{i} \leq \mathsf{n} \\ \mathcal{SI}_{\mathcal{S}}(\mathsf{P},\mathsf{c}) = \mathsf{C}_1 :: \ldots :: \mathsf{C}_{\mathsf{n}} \\ \mathcal{F}s(\mathsf{P},\mathsf{c}) = \{\mathsf{f}_1, \ldots, \mathsf{f}_{\mathsf{r}}\} \\ \mathsf{v} = \mathit{null},\ \ \iota \text{ new in } \sigma_{\mathsf{n}} \\ \mathsf{o} = [\![\ \mathsf{c} \,||\, \mathsf{C}_1 : \iota_1, \ldots, \mathsf{C}_{\mathsf{n}} : \iota_{\mathsf{n}} \,||\, \mathsf{f}_1 : \mathsf{v}, \ldots, \mathsf{f}_{\mathsf{r}} : \mathsf{v}\ ]\!]\end{array}}{\mathsf{P},\mathsf{C} \vdash \mathsf{new\ c}(\mathsf{e}_1, \ldots, \mathsf{e}_{\mathsf{n}}), \sigma_0 \rightsquigarrow \iota, \sigma_{\mathsf{n}}[\iota \mapsto \mathsf{o}]}\ \mathit{newT}$$

$$\frac{\begin{array}{l}\mathsf{P} \vdash \mathsf{C}'\diamond_{\mathsf{c}} \\ \mathsf{P},\mathsf{C} \vdash \mathsf{e}_0, \sigma \rightsquigarrow \iota_0, \sigma_0 \\ \mathsf{P},\mathsf{C} \vdash \mathsf{e_i}, \sigma_{\mathsf{i}-1} \rightsquigarrow \iota_{\mathsf{i}}, \sigma_{\mathsf{i}} \quad 1 \leq \mathsf{i} \leq \mathsf{n} \\ \mathcal{SI}_{\mathcal{S}}(\mathsf{P},\mathsf{C}') = \mathsf{C}_0 :: \ldots :: \mathsf{C}_{\mathsf{n}} \\ \mathcal{F}s(\mathsf{P},\mathsf{C}') = \{\mathsf{f}_1, \ldots, \mathsf{f}_{\mathsf{r}}\} \\ \mathsf{v} = \mathit{null},\ \ \iota \text{ new in } \sigma_{\mathsf{n}} \\ \mathsf{o} = [\![\ \mathsf{C}' \,||\, \mathsf{C}_0 : \iota_0, \ldots, \mathsf{C}_{\mathsf{n}} : \iota_{\mathsf{n}} \,||\, \mathsf{f}_1 : \mathsf{v}, \ldots, \mathsf{f}_{\mathsf{r}} : \mathsf{v}\ ]\!]\end{array}}{\mathsf{P},\mathsf{C} \vdash \mathsf{e}_0.\mathsf{new\ C}'(\mathsf{e}_1, \ldots, \mathsf{e}_{\mathsf{n}}), \sigma \rightsquigarrow \iota, \sigma_{\mathsf{n}}[\iota \mapsto \mathsf{o}]}\ \mathit{newI}$$

In rule *qThis* the search for a host object starts from the current receiver $\sigma(\mathsf{this})$, with C the class containing the expression $\mathsf{D.this}$, and D the type of the host object. For method calls, *methCall*, the expressions for the receiver and the argument are evaluated and used to construct a context for the evaluation of the method body. Note that in the rewrite for the method body, the method environment is updated to $\mathsf{C''}$, the class returned by $\mathcal{M}$. For object creation, *newI* and *newT* build suitable objects and place them in a new address in the store. The inherited hosts are the results of evaluating $\mathsf{e}_1,\ldots,\mathsf{e}_n$, while $\mathcal{SI_S}$ finds all superclasses requiring hosts.

4.4 $\mathcal{L}_{\mathbf{jm}}$ Types

The only types in $\mathcal{L}_{\mathrm{jm}}$ are classes. Since $\mathsf{P}\vdash\mathsf{C}\diamond_c$ guarantees that C is a class it also guarantees that C is a type. Typing judgements in $\mathcal{L}_{\mathrm{jm}}$ take the form:

$$\mathsf{P},\Gamma \vdash \mathsf{e} \,:\, \mathsf{C}$$

where P is the program, Γ is a type environment mapping $\{\mathsf{this},\mathsf{x}\ \}$ to types, e is the expression being typed and C is the type assigned to the expression.

$$\frac{\mathsf{P} \vdash \mathsf{C}\diamond_{\mathsf{c}}}{\mathsf{P},\Gamma \vdash \mathit{null} \,:\, \mathsf{C}}\ \mathit{null} \qquad \frac{}{\mathsf{P},\Gamma \vdash \mathsf{x} \,:\, \Gamma(\mathsf{x})}\ \mathit{var}$$

$$\frac{\mathsf{P} \vdash \Gamma(\mathsf{this}) \preceq \mathsf{C}}{\mathsf{P},\Gamma \vdash \mathsf{C.this} \,:\, \mathsf{C}}\ \mathit{qThis} \qquad \frac{\begin{array}{l}\mathsf{P},\Gamma \vdash \mathsf{e} \,:\, \mathsf{C}' \\ \mathcal{F}(\mathsf{P},\mathsf{C}',\mathsf{f}) = \mathsf{C}\end{array}}{\mathsf{P},\Gamma \vdash \mathsf{e.f} \,:\, \mathsf{C}}\ \mathit{fld}$$

$$\frac{\begin{array}{l}\mathsf{P},\Gamma \vdash \mathsf{e} \,:\, \mathsf{C} \\ \mathsf{P},\Gamma \vdash \mathsf{e}' \,:\, \mathsf{C}' \\ \mathcal{F}(\mathsf{P},\mathsf{C},\mathsf{f}) = \mathsf{C}'' \\ \mathsf{P} \vdash \mathsf{C}' \leq \mathsf{C}''\end{array}}{\mathsf{P},\Gamma \vdash \mathsf{e.f} := \mathsf{e}' \,:\, \mathsf{C}'}\ \mathit{fldAss} \qquad \frac{\begin{array}{l}\mathsf{P} \vdash \mathsf{c}\diamond_{\mathsf{c}} \\ \mathcal{SI_S}(\mathsf{P},\mathsf{c}) = \mathsf{C}_1 :: \ldots :: \mathsf{C}_n \\ \mathsf{C}_i = \mathsf{C}'_i.\mathsf{c}_i \quad 1 \leq i \leq n \\ \mathsf{P},\Gamma \vdash \mathsf{e}_i \,:\, \mathsf{C}''_i \\ \mathsf{P} \vdash \mathsf{C}''_i \leq \mathsf{C}'_i \quad 1 \leq i \leq n\end{array}}{\mathsf{P},\Gamma \vdash \mathit{new}\ \mathsf{c}(\mathsf{e}_1,\ldots,\mathsf{e}_n) \,:\, \mathsf{c}}\ \mathit{newT}$$

$$\frac{\begin{array}{l}\mathsf{P} \vdash \mathsf{C}\diamond_{\mathsf{m}} \\ \mathcal{SI_S}(\mathsf{P},\mathsf{C}) = \mathsf{C}_0 :: \ldots :: \mathsf{C}_n \\ \mathsf{C}_i = \mathsf{C}'_i.\mathsf{c}_i \quad 0 \leq i \leq n \\ \mathsf{P},\Gamma \vdash \mathsf{e}_i \,:\, \mathsf{C}''_i \\ \mathsf{P} \vdash \mathsf{C}''_i \leq \mathsf{C}'_i \quad 0 \leq i \leq n\end{array}}{\mathsf{P},\Gamma \vdash \mathsf{e}_0.\mathit{new}\ \mathsf{C}(\mathsf{e}_1,\ldots,\mathsf{e}_n) \,:\, \mathsf{C}}\ \mathit{newI} \qquad \frac{\begin{array}{l}\mathsf{P},\Gamma \vdash \mathsf{e} \,:\, \mathsf{C} \\ \mathsf{P},\Gamma \vdash \mathsf{e}' \,:\, \mathsf{C}' \\ \mathcal{M}(\mathsf{P},\mathsf{C},\mathsf{m}) = (\mathsf{C}''\ \mathsf{m}\ (\mathsf{C}'''\ \mathsf{x})\ \{\mathsf{e}''\},\ldots) \\ \mathsf{C}' \leq \mathsf{C}'''\end{array}}{\mathsf{P},\Gamma \vdash \mathsf{e.m}(\mathsf{e}') \,:\, \mathsf{C}''}\ \mathit{meth}$$

The type rule *qThis* allows qualification of this, only if the qualification is a prefix (i.e. an enclosing class) of the type of this i.e. $\Gamma(\mathsf{this})$. The type rules for object creation, *newT* and *newI*, check the type of the host object expressions and ensure that the correct number of expressions are supplied, which is the number of class names in the list returned by $\mathcal{SI_S}$.

4.5 Agreement, and well-formedness of program

Judgements pertaining to agreement can be found in [7]: $\mathsf{P},\sigma \vdash\iota\lhd \mathsf{C}'$, expresses that the object at address ι conforms to class C' i..e. its fields and hosts agree with their expected types. The judgement $\mathsf{P},\Gamma\vdash\sigma \diamond$ expresses that all objects agree with their types.

Judgements pertaining to well-formedness can also be found in [7]: A class is well-formed, $\mathsf{P}\vdash\mathsf{C}\diamond$, if it satisfies the requirements on naming of classes and members, and if the body of each of its methods has a subtype of the declared return type given an appropriate environment. Additionally, field names must not be used in superclasses (no field shadowing), and should a method name have been defined in a super class, then the argument and return types must be the same (method overriding but not overloading). A program is well formed, $\mathsf{P}\diamond$, if all its classes are well-formed.

5 Properties of $\mathcal{L}_{\mathbf{jm}}$

In this section we first show determinism of the operational semantics. For this, we define ?? equivalence of stores, $\sigma =_\alpha \sigma'$, up to renaming of addresses, where α is a bijection across addresses, definitions are available in the full version [7]. We then combine progress and soundness of typing into one result.

Determinism of the Operational Semantics. *If*

$\mathsf{P},\mathsf{C} \vdash \mathsf{e}\,,\sigma \rightsquigarrow \mathsf{r}\,,\sigma'$, *and* $\mathsf{P},\mathsf{C} \vdash \mathsf{e}\,,\sigma'' \rightsquigarrow \mathsf{r}'\,,\sigma'''$, *and* $\sigma =_\alpha \sigma''$ *for some* α,

then

$\mathsf{r} =_{\alpha'} \mathsf{r}'$, *and* $\sigma' =_{\alpha'} \sigma'''$, *for some* α' *extending* α.

Subject Reduction and Progress. *If*

$\vdash \mathsf{P}\diamond\,,\ \mathsf{P},\Gamma \vdash \mathsf{e} : \mathsf{C}$, *and* $\mathsf{P},\Gamma \vdash \sigma\diamond$, *and* $\mathsf{P},\mathsf{D} \vdash \mathsf{e}\,,\sigma \rightsquigarrow \mathsf{r}\,,\sigma'$

then

($\mathsf{r} = \mathsf{v}$ *and* $\mathsf{P},\sigma' \vdash \mathsf{v} \lhd \mathsf{C}$ *and* $\mathsf{P},\Gamma \vdash \sigma'\diamond$) *or* ($\mathsf{r} = \mathsf{nullPntrExc}$ *and* $\mathsf{P},\Gamma \vdash \sigma'\diamond$)

6 Translation from $\mathcal{L}_{\mathbf{jm}}$ to $\mathcal{L}_{\mathbf{j}}$

SUN implements inner classes in Java is via translation to equivalent top level classes [8]. We offer a (nearly) equivalent translation from $\mathcal{L}_{\mathrm{jm}}$ to $\mathcal{L}_{\mathrm{j}}$ ($\mathcal{L}_{\mathrm{j}}$ is $\mathcal{L}_{\mathrm{jm}}$ without member classes). We follow the rules which Sun uses for conversion of qualified names and other inner class features. To simplify matters we assume sequences of expressions and multi-argument methods in $\mathcal{L}_{\mathrm{j}}$. Additionally, we assume the existence of a method in the Object class which returns the receiver. Our translation is not quite correct due to the simplicity of the language. Our lack of access modification makes the host fields globally accessible, though the use of the protected \$ symbol affords some control at the source level.

In [7] we define the following functions, where $\mathcal{P}$, $\mathcal{C}$, $\mathcal{T}$, $\mathcal{M}$ and $\mathcal{E}$ denote sets of programs, class definitions, types, methods and expressions, and the subscripts denote $\mathcal{L}_{\mathrm{jm}}$, and respectively for $\mathcal{L}_{\mathrm{j}}$.

$T : \mathcal{P}_{jm} \rightarrow \mathcal{P}_j$ $\quad T_C : \mathcal{P}_{jm} \times \mathcal{C}_{jm} \rightarrow \mathcal{C}_j$ $\quad T_T : \mathcal{T}_{jm} \rightarrow \mathcal{T}_j$

$T_M : \mathcal{T}_{jm} \times \mathcal{M}_{jm} \rightarrow \mathcal{M}_j$ $\quad T_E : \mathcal{T}_{jm} \times \mathcal{E}_{jm} \rightarrow \mathcal{E}_j$

We can then prove correctness of the translation as follows:

Theorem 1 *If* $\vdash \mathsf{P}\diamond$ *then* $\vdash T(P)\diamond$

Theorem 2 *If*

$\Gamma_{\mathsf{jm}}(\mathsf{this}) = \mathsf{C}_1$, $\Gamma_{\mathsf{jm}}(\mathsf{x}) = \mathsf{C}_2$, $\Gamma_{\mathsf{j}}(\mathsf{this}) = T_T(\mathsf{C}_1)$, $\Gamma_{\mathsf{j}}(\mathsf{x}) = T_T(\mathsf{C}_2)$, $\mathsf{P}_{\mathsf{jm}},\Gamma_{\mathsf{jm}} \vdash \mathsf{e} : \mathsf{C}$,

then

$T(\mathsf{P}_{\mathsf{jm}}),\Gamma_{\mathsf{j}} \vdash T_E(\mathsf{P},\mathsf{e}) : T_T(\mathsf{C})$

We now show equivalence of the translation under execution. For this, in Appendix ??, we define equivalent stores with respect to α, a bijection between addresses, and the relation $=^T_\alpha$which express equivalence of $\mathcal{L}_{\mathrm{jm}}$ and $\mathcal{L}_{\mathrm{j}}$ stores.

Theorem 3 *If*

$T(\mathsf{P}_{\mathsf{jm}}) = \mathsf{P}_{\mathsf{j}}$, $T_E(\mathsf{P}_{\mathsf{jm}},\mathsf{e}_{\mathsf{jm}}) = \mathsf{e}_{\mathsf{j}}$, $\sigma_{\mathsf{jm}} =^{\mathsf{T}}_{\alpha} \sigma_{\mathsf{j}}$, $\mathsf{P}_{\mathsf{jm}},\mathsf{C} \vdash \mathsf{e}_{\mathsf{jm}}\,,\sigma_{\mathsf{jm}} \rightsquigarrow \mathsf{v}_{\mathsf{jm}}\,,{\sigma_{\mathsf{jm}}}'$,

then

$T(\mathsf{P}_{\mathsf{jm}}) \vdash \mathsf{e}_{\mathsf{j}} \rightsquigarrow \mathsf{v}_{\mathsf{j}},{\sigma_{\mathsf{j}}}'$, *and there exists* α' *extending* α *such that:* ${\sigma_{jm}}' =^{T}_{\alpha'} {\sigma_j}'$ $\quad \mathsf{v}_{\mathsf{jm}} =^{\mathsf{T}}_{\alpha'} \mathsf{v}_{\mathsf{j}}$

7 Observations

In the process of our work on the formalization of inner classes we made the following observations with respect to their design, and the reasons for various restrictions imposed by the language:

O1 Any class which was part of a cycle in the the union of the subclass and inner class relation, cannot be instantiated. This is so, because instantiation of such a class would require an instance of that class (as a host or inherited host) since this must pre-exist the new instance such a class can never be instantiated by code.

O2 Although Java forbids classes as in **O1**, a language that allowed them, without allowing cycles in the inheritance hierarchy, would still be sound. Namely, such classes do not interact adversely with other code, they simply cannot be instantiated, and thus are redundant. Should an instance of such a class somehow "magically" exist, execution of well-typed expressions would return values of appropriate classes. [5]

O3 The host relation in object graphs generated by code forms a DAG. Namely, cycles cannot be generated by code since host objects must pre-exist the objects they are host to. We can easily show that such cycles cannot be created though execution of $\mathcal{L}_{\mathrm{jm}}$ expressions.

O4 However, soundness does not depend on **O3**. Namely, our proof of subject reduction holds even for stores where there are cycles in the host relation.

8 Conclusions and further work

This paper describes the basic and advanced behaviours of Java inner classes in the context of aliasing in a simple manner. We have shown that inner classes are sound with respect to aliasing. This is, we believe, an important result especially since, in the past, features developed in functional calculi have not remained sound when converted to a setting allowing aliasing.

$\mathcal{L}_{\mathrm{jm}}$ should be extensible to include the other styles of inner classes, local and anonymous, since these types of inner classes do not change the host relationship, but they do have interesting behaviour with respect to other Java features, e.g. access modifiers.

We note certain similarities between the host relation and relations which arise between objects in work on ownership types. In fact [9] utilises inner classes for ownership types, and further work would be the extension of ownership types with inner classes. In that work inner classes features are not heavily exploited, though the system suggests they could be.

References

1. Walrath, K., Campione, M.: The swing tutorial (1999)
2. et al., L.M.: Object-Oriented Programming in the Beta Programming Langugae. First edn. (1993)
3. Igarashi, A., Pierce, B.C.: On Inner Classes. In: European Conference in Object Oieted Programming. Volume LNCS 1850. (2000)
4. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification Second Edition. Addison-Wesley, Boston, Mass. (2000)
5. Drossopoulou, S., Eisenbach, S., Khurshid, S.: Is the java type system sound? Theory and Practice of Object Systems **5** (1999) 3–24
6. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and mixins. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (1998) 171–183
7. Smith, M., Drossopoulou, S.: Inner classes visit aliasing - technical report (2003)
8. Sun Microsystems Inc.: Specification for inner classes. http://java.sun.com/products/jdk/1.1/docs/guide /innerclasses/spec/innerclasses.doc.html (1999)
9. Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (2003) 213–223

[5] Igarashi & Pierce's informal statement of this requirement is not strong enough, though the Java compiler is.