Sheep Cloning with Ownership Types

Paley Li
Victoria University of Wellington
New Zealand
lipale@ecs.vuw.ac.nz

Nicholas Cameron
Mozilla Corporation
ncameron@mozilla.com

James Noble
Victoria University of Wellington
New Zealand
kjx@ecs.vuw.ac.nz

ABSTRACT

Object-oriented programmers often need to clone objects. Mainstream languages, such as C# and Java, typically default to shallow cloning, which copies just one object and aliases references from that object. Other languages, such as Eiffel, provide deep cloning. A deep clone is a copy of the entire object graph reachable from the cloned object, which could include many unnecessary objects. Alternatively, programmers can implement their own object cloning functions, however, this is often difficult and error prone.

Sheep Cloning is an automated cloning technique which uses ownership information to provide the benefits of both shallow and deep cloning without the costs. We describe, formalise, and prove soundness of Sheep Cloning in a minimal object-oriented calculus with ownership types.

Categories and Subject Descriptors

D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features*

General Terms

Languages

Keywords

Ownership types, object cloning, type system

1. INTRODUCTION

Traditional object cloning techniques produce clones using either shallow cloning or deep cloning. In Java, an object can be shallow cloned by calling its `clone()` method, provided its class implements the `Cloneable` interface. A similar approach is taken in C# where the object's class is required to implement the `ICloneable` interface. To create deep clones in Java, programmers would need to overwrite the object's `clone()` method with an implementation of deep cloning themselves. This task is often daunting and challenging. There are cases when it is not obvious which cloning technique would produce the more suitable clone, and there are even cases when neither technique is suitable. In all of these cases, languages tend to offer little support, forcing programmers to design and implement a custom cloning implementation themselves.

Ownership types enforce the heap into a hierarchically structured, by introducing an owner object for every object [10]. The term context is used to mean the formal set of objects owned by an object, and the term representation means the set of objects which are conceptually part of an object, therefore ownership types help describe an object's representation. Prescriptive ownership policies, like owners-as-dominators, can be incorporated on top of descriptive ownership types. The owners-as-dominators policy [7] ensures an object's representation can never be exposed outside of its enclosing owner, by forcing all reference paths to an object to pass through that object's owner.

Sheep Cloning [21] is an intuitive and automated ownership-based cloning technique. Sheep cloning clones the object's representation by copying an object's context and aliases the reachable objects not owned the object. The owners-as-dominators policy and the hierarchical structure of the heap are key in constructing Sheep clones. This is because the decision to copy or alias an object is determined by the object's owner. A Sheep clone preserves owners-as-dominators and is structurally equivalent to the object it is a clone of. In this paper, we describe and formalise Sheep Cloning, prove our formalism is sound, and present its correctness properties.

The rest of this paper is organized as follows: in Sect. 2 we introduce object cloning and ownership types; in Sect. 3 we introduce Sheep Cloning; in Sect. 4 we describe Sheep Cloning formally, and show type soundness; in Sect. 5 we discuss possible future work; in Sect. 6 we discuss related work; and, in Sect. 7 we conclude.

2. BACKGROUND

Programs today are regularly required to be written defensively under the assumption they will interact with malicious code. Defensive copying and ownership types [10] are two mechanisms to reduce the possible harm caused by malicious code. Defensive copying is a programming discipline that aims to stop malicious code from unexpectedly retaining and mutating objects. This is achieved by requiring all method calls to pass in clones and for all method returns to

return clones [16]. Ownership types can statically restrict access to an object's context, encapsulating the object's behaviour and preventing unexpected mutation of the object.

2.1 Object Cloning

Object cloning is the process of copying objects [2, 22, 1]. Traditionally, there are two object cloning techniques. One is shallow cloning, which copies the single object to be cloned and aliases the fields of that object. The other is deep cloning, which copies the object to be cloned and every object reachable from it.

```
class Window {
 Document document;
 Database database;

 Window(Database database) {
  document = new Document(database);
  this.database = database;
 }
 ...
}
class Document {
 Database database;
 ...
}
```

Figure 1: Code of the display window.

In Fig. 1 and Fig. 2, we present an example of a display `window`, as code and a diagram respectively. The display `window` contains a `document` and a reference to a `database`. The `window` simply displays the items it retrieves from the `database`. The `document` has a reference to the same `database` as the `window`. This allows the `document` to reference items from the `database`, independent of `window`.

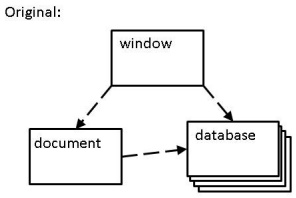

Figure 2: Diagram of the display window.

In Fig. 3 and Fig. 4, we present `window`'s shallow and deep clone respectively. We intend the clones ($\texttt{window}_s$ and $\texttt{window}_d$) to have the same structure and behaviour as the original object (`window`). Which means the clones should reference `window`'s `database` and they each should have their own `document` which also reference that `database`.

Shallow cloning `window` creates a new window ($\texttt{window}_s$), which aliases the `document` and `database` of `window`. $\texttt{window}_d$ is produced by deep cloning `window`, creating an entirely new document ($\texttt{document}_d$) and database ($\texttt{database}_d$). A reference from $\texttt{document}_d$ to $\texttt{database}_d$ is also created.

The shallow clone, $\texttt{window}_s$, does not have the structure of `window`. As $\texttt{window}_s$ references the same `document` as `window`, any changes to the `document` would affect both `window` and $\texttt{window}_s$. Meanwhile, deep cloning copies `database`, which can be costly. The deep clone, $\texttt{window}_d$, also presents the problem that any changes to `database` will not be reflected in $\texttt{database}_d$, and therefore can not be displayed in $\texttt{window}_d$.

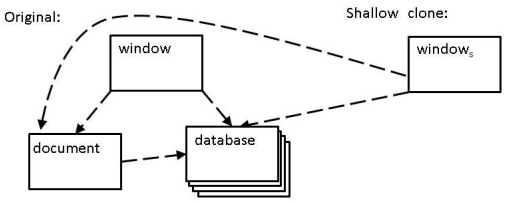

Figure 3: Shallow clone of the display window.

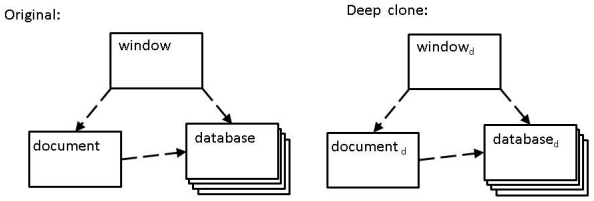

Figure 4: Deep clone of the display window.

2.2 Ownership Types

Ownership types were introduced in 1998 by Clarke et al [10]. This was followed by a variety of ownership systems, such as Ownership Domains [3], Universe Types [17, 18], Ownership with Disjointness of Types and Effects [9], External Uniqueness [8], and Ownership Generic Java [23]. The descriptive and/or prescriptive properties of these systems may differ, but the heap of all of these systems is structured hierarchically.

An ownership type is constructed from a class name and a list of owner parameters. In Fig. 5, we present the display `window` with ownership types. The ownership type for a `document` is `this:Document<dbowner>`. `Document` is the class name of the type, while `this` and `dbowner` are the owner parameters. `this` being the owner of the `document`. The owner `this` denotes a special owner, the current "this" object. The `window` object owns the `document` object, hence the owner of the `document` is this instance of the `Window` class. Declaring `dbowner` as the owner of the `database` permits the `window` to refer to the `database`. `dbowner` is also passed to the `document`, allowing the `document` to reference the same `database` as `window`. The workings of the owner parameters will be explained in greater detail in our formalism.

In Fig. 6, we present a diagram of the ownership typed display `window`. In this diagram, the boxes denote objects, and since objects are owners, the boxes are also owners. Objects

```
class Window<dbowner> {
 this:Document<dbowner> document;
 dbowner:Database<> database;
 ...
}
class Document<dbowner> {
 dbowner:Database<> database;
 ...
}
```

Figure 5: Code of ownership typed display window.

inside a box make up the context of the object that box represents. The `document` is owned by `window`, and therefore is inside `window`, while `database` is not part of `window`'s representation, and therefore is outside `window`. The dotted black arrows represents valid references.

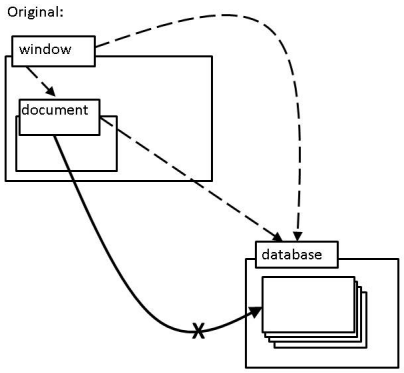

Figure 6: Diagram of ownership typed display window.

Owners-as-dominators, or deep ownership, ensures an object's representation can never be exposed outside its enclosing context [10, 7]. In practice this means all the references from a context must either go to its direct descendant, i.e., the objects it owns, its siblings, or up its ownership hierarchy. References are allowed up the ownership hierarchy because these references are pointing to representations that they are part of. In terms of the boxes, references can always go out of a box but never into a box. This is shown in Fig. 6. The `document` is permitted to reference the `database`. The `document`, however, is not permitted to reference the objects in the `database`'s context as shown by the cross on the solid black arrow. It is important to note that although ownership is transitive. An object can only reference the objects it owns directly, and never the objects that those objects own.

3. SHEEP CLONING

An ownership-based cloning operation was first proposed by Noble et al [21] who called this operation Sheep Cloning. Noble describes how Sheep Cloning clones an ownership-typed object by copying the objects it owns, while aliasing the references to external objects it doesn't own. They then discuss the need to maintain a map, to prevent objects from being copied more than once. Finally, they present an example where they Sheep clone a `Course`, which is represented by a linked list of `Students`. Sheep Cloning the linked list copies the nodes of the linked list, while aliasing the `Students`. While Sheep Cloning the `Course` creates a replica of the entire linked list, with new copies of the `Students`.

Sheep Cloning incorporates aspects of both shallow and deep cloning. Like deep cloning, Sheep Cloning clones an object's representation by traversing references and copying every reachable object inside the context. Like shallow cloning, Sheep Cloning creates aliases after the essential object(s) is copied. Unlike deep cloning, however, Sheep Cloning uses the inside relation of ownership types to determine when it needs to stop copying, so no unnecessary objects are copied. Unlike shallow cloning, Sheep Cloning can recreate the entire representation of an object, instead of just copying a single object.

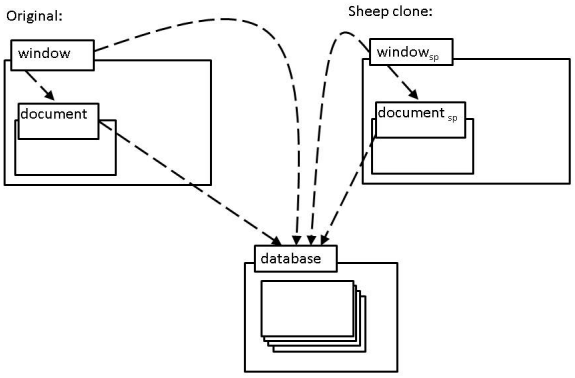

Figure 7: Sheep clone of the ownership typed display window.

In Fig. 7, we present the Sheep clone of the ownership-typed display `window`. Sheep Cloning `window` creates a new `window`, `window`$_{sp}$. In `window`$_{sp}$, a copy of `document` is created, `document`$_{sp}$, and `document`$_{sp}$ contains a reference to the `database`. Finally, an alias of the reference to the `database` is created for `window`$_{sp}$.

The inside relation defines the ownership relation between two contexts. Sheep Cloning requires two variations of the same inside relation: a compile-time inside relation that ensures the owners-as-dominator property at compile-time, and a run-time inside relation that Sheep Cloning uses to determine whether to copy or alias an object. If an object has already been copied, then instead of copying this object twice, Sheep Cloning uses a map to refer to the copy that already exists.

Currently, Sheep Cloning requires owners-as-dominators. Systems with only descriptive ownership do not restrict access to an object's representation, which means there are objects that can not be reachable (directly or transitivity) by their owner. This is problematic for Sheep Cloning as it is expensive to locate every object in an object's representation, as a traversal over the entire heap is required. Owners-as-dominators guarantees every object inside an object's representation is reachable (directly or transitivity) from the owner.

4. FORMALISATION

In this section, we present our formalisation of Sheep Cloning, a calculus in the tradition of Featherweight Java [15]. The aim for our formalism is to present Sheep Cloning as a language feature that can be implemented in any ownership system that has owners-as-dominators. By formalising Sheep Cloning with the minimal amount of features it requires, we have lost some aspect of realism in our system, as our formalism is not Turing complete, since we do not support inheritance, and therefore cannot formalise conditionals. We have, however, maintained our aim, as most languages features required for Turing completeness are orthogonal to Sheep Cloning. Turing completeness is not required to show type safety, and we will present soundness of our formalism in a later section.

4.1 Static System

$$
\begin{array}{lcll}
Q & ::= & \texttt{class C<}\overline{\texttt{o}_l \preceq \texttt{x} \preceq \texttt{o}_u}\texttt{>}\ \texttt{\{}\overline{N\ \texttt{f;}}\ \overline{M}\texttt{\}} & \textit{class declarations} \\
M & ::= & N\ \texttt{m(}N\ \texttt{x)}\ \texttt{\{return}\ e\texttt{;\}} & \textit{method declarations} \\
 & & & \\
T & ::= & N \mid \top & \textit{type} \\
N & ::= & \texttt{o:C<}\overline{\texttt{o}}\texttt{>} & \textit{class type} \\
\texttt{o} & ::= & \gamma \mid \texttt{world} \mid \texttt{owner} & \textit{owners} \\
 & & & \\
e & ::= & \texttt{null} \mid \gamma \mid \gamma\texttt{.f} \mid \gamma\texttt{.f} = e \mid \gamma\texttt{.m(}e\texttt{)} & \textit{expressions} \\
 & & \mid \texttt{new o:C<}\overline{\texttt{o}}\texttt{>} \mid \texttt{sheep(}e\texttt{)} \mid v & \\
v & ::= & \texttt{null} \mid \iota & \textit{values} \\
 & & & \\
\gamma & ::= & \texttt{x} \mid \texttt{this} \mid \iota & \textit{expression variables and addresses} \\
\Gamma & ::= & \overline{\gamma\texttt{:}T}, \overline{\texttt{o:}\top} & \textit{variable environments} \\
\mathcal{E} & ::= & \overline{\texttt{o} \preceq \texttt{o}} & \textit{owners environments} \\
\mathcal{H} & ::= & \overline{\iota \rightarrow \{N, \overline{\texttt{f} \rightarrow v}\}} & \textit{heaps} \\
\texttt{map} & ::= & \{\iota \rightarrow \iota\} & \textit{map} \\
 & & & \\
\texttt{x} \preceq \texttt{o} & & & \textit{owners relation} \\
\texttt{x} & & & \textit{variables} \\
\iota & & & \textit{object address} \\
\texttt{err} & & & \textit{errors} \\
\texttt{null} & & & \textit{null expression} \\
\texttt{f} & & & \textit{field names} \\
\texttt{m} & & & \textit{method names} \\
\texttt{C} & & & \textit{class names}
\end{array}
$$

Figure 8: Syntax.

In Fig. 8, we present the syntax for our formalism. The syntax in grey is for our run-time model. Classes are parameterised with owner parameters. The formal owner parameters ($\overline{\texttt{x}}$) in the class declaration are bounded by a lower bound, $\texttt{o}_l$, and an upper bound, $\texttt{o}_u$. The valid owners of the system are: `world`, `owner`, `this`, and variables (`x`). `world` represents the top owner in the ownership hierarchy. Objects with `world` as their owner can be referenced from anywhere in the system. The `world` owner continues to exist at runtime. The owner parameter `owner` represents the owner of the current object, **this**. `owner` is only used statically, and at run-time it is substituted by the actual object it represents. The owner `this` represents the current object, **this**. At runtime, `this` is substituted by the instance of **this**. `x` is a variable representing a formal owner parameter within the class declaration.

Classes contain fields and methods. Fields are initialised to `null` when an object is created. Sub-classing is orthogonal to Sheep Cloning and is therefore omitted. Our method declaration is equivalent to those in Java. Method bodies consist of a return statement with an expression. Class types contain the class name, a single owner parameter (`o`), denoting

Well-formed owner: $\boxed{\mathcal{E};\Gamma \vdash \texttt{o}\ \text{OK}}$

$$\frac{\Gamma(\gamma) = \top}{\mathcal{E};\Gamma \vdash \gamma\ \text{OK}}\ \text{(F-VAR)} \qquad \frac{}{\mathcal{E};\Gamma \vdash \texttt{world}\ \text{OK}}\ \text{(F-WORLD)}$$

$$\frac{}{\mathcal{E};\Gamma \vdash \texttt{owner}\ \text{OK}}\ \text{(F-OWNER)} \qquad \frac{}{\mathcal{E};\Gamma \vdash \texttt{this}\ \text{OK}}\ \text{(F-THIS)}$$

Well-formed types: $\boxed{\mathcal{E};\Gamma \vdash N\ \text{OK}}$

$$\frac{\texttt{class C<}\overline{\texttt{o}_l \preceq \texttt{x} \preceq \texttt{o}_u}\texttt{>...} \qquad \mathcal{E};\Gamma \vdash \texttt{o}, \overline{\texttt{o}}\ \text{OK} \qquad \mathcal{E};\Gamma \vdash \texttt{[o/x]}(\overline{\texttt{o}_l \preceq \texttt{o}}) \qquad \mathcal{E};\Gamma \vdash \texttt{[o/x]}(\overline{\texttt{o} \preceq \texttt{o}_u})}{\mathcal{E};\Gamma \vdash \texttt{o:C<}\overline{\texttt{o}}\texttt{>}\ \text{OK}}\ \text{(F-CLASS)}$$

Well-formed heap: $\boxed{\vdash \mathcal{H}\ \text{OK}}$

$$\frac{\forall \iota \rightarrow \{N; \overline{\texttt{f} \rightarrow v}\} \in \mathcal{H} : \quad \frac{\mathcal{H} \vdash N\ \text{OK}}{fType(\texttt{f}, N) = N' \qquad \mathcal{H} \vdash v : \texttt{[}\iota\texttt{/this]}N'} \qquad \forall v \in \overline{v} : v \neq \texttt{null} \Rightarrow v \in dom(\mathcal{H})}{\vdash \mathcal{H}\ \text{OK}}\ \text{(F-HEAP)}$$

Figure 9: Well-formed judgements.

$$\frac{\Gamma = \texttt{this : owner:C<}\overline{\texttt{x}}\texttt{>,}\ \overline{\texttt{x:}\top} \qquad \mathcal{E} = \overline{\texttt{o}_l \preceq \texttt{x}},\ \overline{\texttt{x} \preceq \texttt{o}_u},\ \texttt{owner} \preceq \texttt{world} \qquad \mathcal{E};\Gamma \vdash \overline{\texttt{owner} \preceq \texttt{o}_l} \qquad \mathcal{E};\Gamma \vdash \overline{N}, \overline{M}\ \text{OK}}{\vdash \texttt{class C<}\overline{\texttt{o}_l \preceq \texttt{x} \preceq \texttt{o}_u}\texttt{>}\ \texttt{\{}\overline{N\ \texttt{f;}}\ \overline{M}\texttt{\}}\ \text{OK}}\ \text{(T-CLASS)}$$

$$\frac{\Gamma' = \Gamma, \texttt{x:}N' \qquad \mathcal{E};\Gamma \vdash N, N'\ \text{OK} \qquad \mathcal{E};\Gamma' \vdash e : N}{\mathcal{E};\Gamma \vdash N\ \texttt{m(}N'\ \texttt{x)}\ \texttt{\{return}\ e\texttt{;\}}\ \text{OK}}\ \text{(T-METHOD)}$$

Figure 10: Classes and methods typing.

Static inside relation: $\boxed{\mathcal{E};\Gamma \vdash \texttt{o} \preceq \texttt{o}}$

$$\frac{\texttt{o} \preceq \texttt{o}' \in \mathcal{E}}{\mathcal{E};\Gamma \vdash \texttt{o} \preceq \texttt{o}'}\ \text{(IC-ENV)} \qquad \frac{\Gamma\texttt{(this)} = N}{\mathcal{E};\Gamma \vdash \texttt{this} \preceq \texttt{owner}}\ \text{(IC-THIS)}$$

$$\frac{\mathcal{E};\Gamma \vdash \texttt{o}\ \text{OK}}{\mathcal{E};\Gamma \vdash \texttt{o} \preceq \texttt{o}}\ \text{(IC-REFL)} \qquad \frac{\mathcal{E};\Gamma \vdash \texttt{o} \preceq \texttt{o}'' \qquad \mathcal{E};\Gamma \vdash \texttt{o}'' \preceq \texttt{o}'}{\mathcal{E};\Gamma \vdash \texttt{o} \preceq \texttt{o}'}\ \text{(IC-TRANS)}$$

$$\frac{\mathcal{E};\Gamma \vdash \texttt{o}\ \text{OK}}{\mathcal{E};\Gamma \vdash \texttt{o} \preceq \texttt{world}}\ \text{(IC-WORLD)}$$

Figure 11: Inside relation.

$$\frac{}{own_{\mathcal{H}}(\texttt{o:C<}\overline{\texttt{o}}\texttt{>}) = \texttt{o}}$$

$$\frac{\mathcal{H}(\iota) = \{\iota' : \texttt{C<}\overline{\texttt{o}}\texttt{>}, \ ...\}}{own_{\mathcal{H}}(\iota) = \iota'}$$

$$\frac{\texttt{class C<}\overline{\texttt{o}_l \preceq \texttt{x} \preceq \texttt{o}_u}\texttt{> \{}\overline{N\ \texttt{f;}}\ \overline{M}\texttt{\}}}{fields(\texttt{C}) = \overline{\texttt{f}}}$$

$$\frac{\texttt{class C<}\overline{\texttt{o}_l \preceq \texttt{x} \preceq \texttt{o}_u}\texttt{> \{}\overline{N\ \texttt{f;}}\ \overline{M}\texttt{\}}}{fType(\texttt{f}_i, \texttt{o:C<}\overline{\texttt{o}}\texttt{>}) = \texttt{[o/owner, }\overline{\texttt{o/x}}\texttt{]}N_i}$$

$$\frac{\texttt{class C<}\overline{\texttt{o}_l \preceq \texttt{x} \preceq \texttt{o}_u}\texttt{> \{}\overline{N\ \texttt{f;}}\ \overline{M}\texttt{\}} \quad N\ \texttt{m(}N'\ \texttt{x')\ \{return}\ e\texttt{;\}} \in \overline{M}}{mBody(\texttt{m,o:C<}\overline{\texttt{o}}\texttt{>}) = (x'; \texttt{[o/owner, }\overline{\texttt{o/x}}\texttt{]}e)}$$

$$\frac{\texttt{class C<}\overline{\texttt{o}_l \preceq \texttt{x} \preceq \texttt{o}_u}\texttt{> \{}\overline{N\ \texttt{f;}}\ \overline{M}\texttt{\}} \quad N\ \texttt{m(}N'\ \texttt{x')\ \{return}\ e\texttt{;\}} \in \overline{M}}{mType(\texttt{m,o:C<}\overline{\texttt{o}}\texttt{>}) = \texttt{[o/owner, }\overline{\texttt{o/x}}\texttt{]}(N' \to N)}$$

Figure 12: Look up functions.

Expression typing: $\boxed{\mathcal{E}; \Gamma \vdash e : N}$

$$\frac{\mathcal{E}; \Gamma \vdash e : N}{\mathcal{E}; \Gamma \vdash \texttt{sheep(}e\texttt{)} : N}\ \text{(T-Sheep)} \qquad \frac{\mathcal{E}; \Gamma \vdash e : N' \quad \mathcal{E}; \Gamma \vdash N' <: N \quad \mathcal{E}; \Gamma \vdash N\ \text{ok}}{\mathcal{E}; \Gamma \vdash e : N}\ \text{(T-Subs)}$$

$$\frac{}{\mathcal{E}; \Gamma \vdash \gamma : \Gamma(\gamma)}\ \text{(T-Var)} \qquad \frac{\mathcal{E}; \Gamma \vdash \gamma : \texttt{o:C<}\overline{\texttt{o}}\texttt{>} \quad fType(\texttt{f}, \texttt{o:C<}\overline{\texttt{o}}\texttt{>}) = N' \quad \mathcal{E}; \Gamma \vdash e : N \quad \mathcal{E}; \Gamma \vdash N <: \texttt{[}\gamma\texttt{/this]}N'}{\mathcal{E}; \Gamma \vdash \gamma\texttt{.f = }e : N}\ \text{(T-Assign)}$$

$$\frac{\mathcal{E}; \Gamma \vdash N\ \text{ok}}{\mathcal{E}; \Gamma \vdash \texttt{null} : N}\ \text{(T-Null)} \qquad \frac{\mathcal{E}; \Gamma \vdash \gamma : \texttt{o:C<}\overline{\texttt{o}}\texttt{>} \quad \mathcal{E}; \Gamma \vdash e : N'' \quad mType(\texttt{m}, \texttt{o:C<}\overline{\texttt{o}}\texttt{>}) = N' \to N \quad \mathcal{E}; \Gamma \vdash N'' <: \texttt{[}\gamma\texttt{/this]}N'}{\mathcal{E}; \Gamma \vdash \gamma\texttt{.m(}e\texttt{)} : \texttt{[}\gamma\texttt{/this]}N}\ \text{(T-Invk)}$$

$$\frac{\mathcal{E}; \Gamma \vdash \gamma : \texttt{o:C<}\overline{\texttt{o}}\texttt{>} \quad fType(\texttt{f}, \texttt{o:C<}\overline{\texttt{o}}\texttt{>}) = N}{\mathcal{E}; \Gamma \vdash \gamma\texttt{.f} : \texttt{[}\gamma\texttt{/this]}N}\ \text{(T-Field)} \qquad \frac{\mathcal{E}; \Gamma \vdash \texttt{o:C<}\overline{\texttt{o}}\texttt{>}\ \text{ok}}{\mathcal{E}; \Gamma \vdash \texttt{new o:C<}\overline{\texttt{o}}\texttt{> : o:C<}\overline{\texttt{o}}\texttt{>}}\ \text{(T-New)}$$

Figure 13: Expression typing.

the owner of the type, and a set of owner parameters ($\overline{\texttt{o}}$), denoting the actual owner parameters for the class declaration.

The owners-as-dominators policy is enforced by the bounds on the formal owner parameters ($\overline{\texttt{x}}$) in the class declaration, and the premise in T-Class that states the lower bound ($\texttt{o}_l$) is always outside `owner`. This ensures the owner parameters of a class must refer to classes that are outside the owner of this class.

Our judgments are decided under two environments: Γ and $\mathcal{E}$. Γ maps variables to their type, and $\mathcal{E}$ stores the static inside relations of the system. The variables in Γ are either expression variables or owner parameters. Owner parameters are distinguished by always having the top type ($\top$). At run-time, judgments are decided under the heap ($\mathcal{H}$). $\mathcal{H}$ is a set of mappings from address (ι) to object ($\{N; \overline{\texttt{f} \to v}\}$).

We elide presenting our sub-typing rules as they are trivially defined on reflexivity, transitivity, and top.

Well-formed owners, types, and heaps are defined in Fig. 9. An owner parameter is well-formed if it is in Γ. The owners `world`, `owner`, and `this` are variables and therefore are always well-formed. A class type is well-formed if a class declaration for that class exists, if its owner and actual owner parameters are well-formed, and if the upper and lower bounds are valid inside relations when the actual owners are substituted for the formal owner parameters. A heap is well-formed if every non-null object in the heap is well-formed.

In Fig. 10, we define class well-formedness (T-Class) and method well-formedness (T-Method). T-Class initialises Γ and $\mathcal{E}$, ensures the methods and types declared in the class are well-formed, and preserves the owner-as-dominator policy. T-Method ensures the method's return type and argument type are well-formed, and that the type of the expression in the method and the method's return type are the same.

In Fig. 11, we define the static inside relation. The inside relation defines the ordering of the owners, i.e., when a owner (`o`) is inside another (`o'`). Most valid inside relations are deduced from the relations in $\mathcal{E}$, this is reflected in IC-Env, where a relation is valid if it is in $\mathcal{E}$. The owner `this` is only valid if it exists in Γ and is inside `owner`, as stated in IC-This. IC-Refl and IC-Trans respectively define reflexivity and transitivity relations on owners. Finally, IC-World denotes that all owners are inside `world`.

In Fig. 12, we present the look up functions in our formalism. The function $own_{\mathcal{H}}$ can either take a type or an address. If $own_{\mathcal{H}}$ is given a type, it returns the owner of that type. Otherwise if $own_{\mathcal{H}}$ is given an address (ι), it returns the owner of the object at ι in $\mathcal{H}$. The function $fields$ takes a class name and returns the names of the fields in that class. The function $fType$ takes a field ($\texttt{f}_i$) and a type ($\texttt{o:C<}\overline{\texttt{o}}\texttt{>}$) and returns the type of $\texttt{f}_i$ in the class `C`. The function $mBody$ takes a method (`m`) and a type ($\texttt{o:C<}\overline{\texttt{o}}\texttt{>}$) and returns the argument and expression of `m` in the class `C`. Finally, the function $mType$ takes the same arguments as $mBody$ but it returns the type of the method, which is the argument type and return type of `m`.

Finally in Fig. 13, we define expression typing. T-Var ensures variables have the type as defined in Γ. T-Null allows `null` to take any well-formed type. T-Field, T-Assign, T-Invk, and T-New describe standard typing rules for field look up, field assignment, method invocation, and object creation. T-Subs is our subsumption rule. The expression typing rule for Sheep Cloning (T-Sheep) gives the new clone the same type as the expression being cloned.

4.2 Dynamic System

Our small step operational semantics for expressions are defined in Fig. 14. They are mostly standard: R-Field reduces a field look up expression to the value in that field. R-Assign reduces a field assignment expression to the assigning value and updates the heap. R-New reduces an object creation expression to the address of the newly created object in the heap. R-Invk reduces a method invocation expression to the expression returned in the body of the method. Finally, R-Sheep performs Sheep Cloning, which we describe in detail in the next section. We elide the congruence and error reduction rules.

Expression reduction:

$$\frac{\mathcal{H}(\iota) = \{N; \overline{\texttt{f}\rightarrow v}\}}{\iota.\texttt{f}_i; \mathcal{H} \rightsquigarrow v_i; \mathcal{H}} \text{(R-Field)}$$

$$\frac{\mathcal{H}(\iota) = \{N; \overline{\texttt{f}\rightarrow \texttt{v}}\} \quad \mathcal{H}' = \mathcal{H}[\iota \mapsto \{N; \overline{\texttt{f}\rightarrow v[\texttt{f}_i \mapsto v]}\}]}{\iota.\texttt{f}_i \texttt{ = } v; \mathcal{H} \rightsquigarrow v; \mathcal{H}'} \text{(R-Assign)}$$

$$\frac{\mathcal{H}(\iota)\ undefined \qquad fields(\texttt{C}) = \overline{\texttt{f}} \quad \mathcal{H}' = \mathcal{H}, \iota \rightarrow \{\texttt{o:C<}\overline{\texttt{o}}\texttt{>}; \overline{\texttt{f}\rightarrow\texttt{null}}\}}{\texttt{new o:C<}\overline{\texttt{o}}\texttt{>}; \mathcal{H} \rightsquigarrow \iota; \mathcal{H}'} \text{(R-New)}$$

$$\frac{\mathcal{H}(\iota) = \{\texttt{o:C<}\overline{\texttt{o}}\texttt{>}; ...\} \quad mBody(\texttt{m}, \texttt{o:C<}\overline{\texttt{o}}\texttt{>}) = (\texttt{x}; e)}{\iota.\texttt{m}(v); \mathcal{H} \rightsquigarrow \texttt{[}v\texttt{/x,}\iota\texttt{/this,o/owner]}e; \mathcal{H}} \text{(R-Invk)}$$

$$\frac{\texttt{SheepAux}(v,\ v,\ \mathcal{H},\ \emptyset) = v';\ \mathcal{H}';\ \{\overline{\iota \rightarrow \iota'}\}}{\texttt{sheep}(v);\ \mathcal{H} \rightsquigarrow v';\ \mathcal{H}'} \text{(R-Sheep)}$$

Figure 14: Expression reduction rules.

4.3 Sheep Cloning Semantics

The reduction for Sheep Cloning, R-Sheep, reduces an expression passed to a Sheep clone by using the `SheepAux` function, given in Fig. 17. `SheepAux` Sheep clones a value by performing a graph traversal on the heap. The `SheepAux` function takes two values (v, v'), the heap ($\mathcal{H}$), and a `map` function. v is the object being cloned, and remains the same throughout the traversal. v' is the current object `SheepAux` has reached. The map is a mapping from objects to their clone, ($\iota \rightarrow \iota'$), and is used to ensure objects are copied at most once. The `SheepAux` function returns the Sheep clone of v' (v''), a heap ($\mathcal{H}'$), and a `map` function.

In Fig. 15, we present the dynamic variant of the inside relation. Both variants are reflexive, transitive, and contain a `world` case. The dynamic relation, however, uses the heap to derive the relation between an object and its owner (I-Rec), while the static relation (IC-Env) is deduced from the bounds in the class declaration and $\mathcal{E}$.

In Fig. 16, we present our well-formed map judgment and define `map`. `map` is a function that maps the object address (ι) of an original object to the address of its clone (ι'). A `map` is well-formed when it is either empty (F-EmptyMap) or when every clone in the `map` is in the heap it is judged under (F-Map). The mapping over types is crucial in defining the type of the Sheep clones. A mapping over a type (N) is when the owner parameters of N are applied with the mappings in `map` (M-Type). Applying an empty `map` over N will simply return N.

Next we discuss the cases of the `SheepAux` function in Fig. 17.

Dynamic inside relation:

$$\frac{}{\mathcal{H} \vdash \iota \preceq \iota} \text{(I-Ref)} \qquad \frac{}{\mathcal{H} \vdash \iota \preceq \texttt{World}} \text{(I-World)}$$

$$\frac{\mathcal{H} \vdash \iota \preceq \iota'' \quad \mathcal{H} \vdash \iota'' \preceq \iota'}{\mathcal{H} \vdash \iota \preceq \iota'} \text{(I-Trans)} \qquad \frac{\mathcal{H}(\iota) \texttt{ = } \{\iota' \texttt{:C<}\overline{\texttt{o}}\texttt{>, ...}\}}{\mathcal{H} \vdash \iota \preceq \iota'} \text{(I-Rec)}$$

Figure 15: Dynamic inside relation.

Well-formed map and use of map:

$$\frac{}{\mathcal{H} \vdash \emptyset\ \text{OK}} \text{(F-EmptyMap)} \qquad \frac{\forall\, \iota : \iota \in \texttt{range(map)} \Rightarrow \iota \in \texttt{dom}(\mathcal{H})}{\mathcal{H} \vdash \texttt{map OK}} \text{(F-Map)}$$

Mapping of type:

$$\frac{\texttt{map = } \{\overline{\iota \mapsto \iota'}\}}{\texttt{map}(N) \texttt{ = } [\overline{\iota'/\iota}]N} \text{(M-Type)}$$

Figure 16: Map and mapping.

Auxiliary Sheep Clone Functions:

$$\frac{\begin{array}{c}\mathcal{H}(\iota') \texttt{ = } \{N;\ \overline{\texttt{f}\rightarrow v}\} \quad \mathcal{H} \vdash \iota' \preceq \iota \quad \texttt{map}(\iota')\ \texttt{undefined} \\ \texttt{map}_1 \texttt{ = map, } \iota' \mapsto \iota'' \quad \mathcal{H}(\iota'')\ \texttt{undefined} \\ \mathcal{H}_1 \texttt{ = } \mathcal{H},\ \iota'' \mapsto \{\texttt{map}(N);\ \overline{\texttt{f}\rightarrow\texttt{null}}\} \quad \texttt{n = } |\{\overline{\texttt{f}\rightarrow v}\}| \\ \forall_j : 1 \leq \texttt{j} \leq \texttt{n} : \{\texttt{SheepAux}(\iota,\ v_j,\ \mathcal{H}_j,\ \texttt{map}_j) \texttt{ = } v'_j;\ \mathcal{H}_{j+1};\ \texttt{map}_{j+1}\} \\ \mathcal{H}' \texttt{ = } \mathcal{H}_{n+1}\texttt{[}\iota'' \mapsto \{\texttt{map}(N);\ \overline{\texttt{f}\rightarrow v'}\}\texttt{]}\end{array}}{\texttt{SheepAux}(\iota,\ \iota',\ \mathcal{H},\ \texttt{map}) \texttt{ = } \iota'';\ \mathcal{H}';\ \texttt{map}_{n+1}} \text{(R-SheepInside)}$$

$$\frac{\mathcal{H} \not\vdash \iota' \preceq \iota \quad \texttt{map}(\iota')\ \texttt{undefined} \quad \texttt{map}' \texttt{ = map, } \iota' \mapsto \iota'}{\texttt{SheepAux}(\iota,\ \iota',\ \mathcal{H},\ \texttt{map}) \texttt{ = } \iota';\ \mathcal{H};\ \texttt{map}'} \text{(R-SheepOutside)}$$

$$\frac{\texttt{map}(\iota') \texttt{ = } \iota''}{\texttt{SheepAux}(\iota,\ \iota',\ \mathcal{H},\ \texttt{map}) \texttt{ = } \iota'';\ \mathcal{H};\ \texttt{map}} \text{(R-SheepRef)}$$

$$\frac{}{\texttt{SheepAux}(v,\ \texttt{null},\ \mathcal{H},\ \texttt{map}) \texttt{ = null; } \mathcal{H};\ \texttt{map}} \text{(R-SheepNull)}$$

Figure 17: Auxiliary sheep functions.

The inductive case, R-SheepInside, constructs the Sheep clone (ι'') of the object in ι', if ι' exists in the heap ($\mathcal{H}$) and if ι' is inside ι, as defined by the dynamic inside relation. The clone is created with a fresh address ι'', where all its fields are initially set to `null`. A recursive call is made to `SheepAux` for each field in ι'. The returned values are assigned into the fields of ι'' once all the recursive calls have finished. A new heap ($\mathcal{H}'$) is constructed from the old heap ($\mathcal{H}$) with the addition of ι'' and any changes to the heap from the recursive calls on `SheepAux`. Similarly, the `map` is updated with the mapping from ι' to ι'' and any changes to the `map` from the recursive calls on `SheepAux`.

The case R-SheepOutside occurs when ι' is outside ι. In this case, `SheepAux` returns ι' as it would be aliased. The `map` is updated with a mapping from ι' to ι', this shows that ι' is its own Sheep Clone. Owners-as-dominators ensures that ι' will not be encountered later in the context of an object that needs to be copied.

The case R-SheepRef occurs when ι' already exists in the `map`. This indicates that ι' has already been cloned. `SheepAux` returns the Sheep clone (ι'') in the `map`, with no changes to the heap or the `map`.

Finally, the case R-SheepNull occurs when `SheepAux` has to Sheep clone `null`. In this case `SheepAux` returns `null`, with no changes to the heap or the `map`.

4.4 Subject Reduction

In this subsection, we present subject reduction along with proofs for other properties of our formalism.

> **Theorem 1:** Subject Reduction.
>
> *For all* $\mathcal{H}$, $\mathcal{H}'$, e, v, *and* N, **if** $\mathcal{H} \vdash e : N$ *and* $e; \mathcal{H} \rightsquigarrow v; \mathcal{H}'$ *and* $\vdash \mathcal{H}$ OK **then** $\mathcal{H}' \vdash v : N$ *and* $\vdash \mathcal{H}'$ OK.

Subject reduction requires the system to show preservation of expression typing, heap well-formedness, and owners-as-dominators, for every expression reduction. We decided to state owners-as-dominators as a separate theorem away from subject reduction. The proof of subject reduction is by structural induction over the derivation of the expression reduction in Fig. 14. We have a large number of associated lemmas, mostly the standard weakening, inversion, well-formedness, and substitution lemmas. We state and prove some of the more interesting lemmas below.

The most interesting case of subject reduction is R-Sheep, where `e = Sheep(`v'`)`, for some v'. Intuitively, the proof for this case is to show that the Sheep clone has the same type as the cloned object. We use 'cloned object' to mean the object being Sheep cloned and 'clone' to refer to the newly created object which is a copy of the cloned object, that is ι and ι' respectively in the reduction `sheep(`ι`)`; $\mathcal{H} \rightsquigarrow \iota'; \mathcal{H}'$. We use the terms 'inside' and 'outside' to refer to our inside relation, which, defines the relation between two objects in the ownership hierarchy.

> **Subject reduction case:** R-Sheep.
>
> *For all* $\mathcal{H}$, $\mathcal{H}'$, v, v', *and* N, **if** $\mathcal{H} \vdash$ `sheep(`v`)` : N *and* $\vdash \mathcal{H}$ OK *and* `sheep(`v`)`$; \mathcal{H} \rightsquigarrow v'; \mathcal{H}'$ **then** $\mathcal{H}' \vdash v' : N$ *and* $\vdash \mathcal{H}'$ OK.

Proof outline: The reduction of `sheep(`v`)` invokes the function `SheepAux(`v`, `v`, `$\mathcal{H}$`, `$\emptyset$`)` by the premise of R-Sheep. We then apply **Lemma 1** on this function, which returns $\vdash \mathcal{H}'$ OK and $\mathcal{H}'(v')\downarrow_1$ `= map(`$\mathcal{H}$ $(v)\downarrow_1$`)`. We can deduce that $\mathcal{H}'(v')\downarrow_1$ `= map(`$\mathcal{H}(v)$ $\downarrow_1$`)` as `map` is initially empty and that $v = v'$. Then by T-Var, we get $\mathcal{H}' \vdash v' : N$.

The key to subject reduction is **Lemma 1**.

> **Lemma 1:** Sheep Cloning preserves heap and map well-formedness, and type of the cloned object.
>
> *For all* $\mathcal{H}$, $\mathcal{H}'$, v, v', v'', `map`, *and* `map`$'$, **if** $\vdash \mathcal{H}$ OK *and* $\mathcal{H} \vdash$ `map` OK *and* `SheepAux(`$v, v', \mathcal{H},$ `map`$) = v''; \mathcal{H}'$; `map`$'$ **then** $\vdash \mathcal{H}'$ OK *and* $\mathcal{H}' \vdash$ `map`$'$ OK *and* $\mathcal{H}'(v'')\downarrow_1$ `= map(`$\mathcal{H}(v')\downarrow_1$`)`.

Proof outline: There are four cases to consider in the proof of **Lemma 1**, they are: R-SheepInside; R-SheepOutside; R-SheepRef; R-SheepNull. The proof for R-SheepNull is trivial as the heap and the map are unchanged, and by T-Null the `null` expression can take any well-formed type.

For the case R-SheepOutside, we have $\mathcal{H} = \mathcal{H}'$ and $v' = v'' = \iota'$, while `map`$'$ contains an identity mapping of ι in addition to the mappings in `map`. To show $\mathcal{H}' \vdash$ `map`$'$ OK, we start by stating that $\iota' \in \mathcal{H}$, therefore $\iota' \in \mathcal{H}'$ as $\mathcal{H} = \mathcal{H}'$. Now with $\iota' \in \mathcal{H}'$, $\vdash \mathcal{H}'$ OK, the definition of `map`, and F-Map we can state that $\mathcal{H}' \vdash$ `map`$'$ OK. To show $\mathcal{H}'(\iota')\downarrow_1$ `= map(`$\mathcal{H}(\iota')\downarrow_1$`)` we must show that the mappings in `map` does not apply through the type in $\mathcal{H}(\iota')\downarrow_1$. If we let $\mathcal{H}(\iota')\downarrow_1 = N'$, for some N', then for all $\iota'' \in$ `dom(map)` either $\mathcal{H} \nvdash \iota' \preceq \iota''$ or $\iota'' \mapsto \iota''$. As either ι' is outside of ι'', which means that ι'' is an object that has already been cloned, or that ι'' is an object that is also outside of ι. By **Theorem 2**, we can guarantee that N' does not have any owner parameters inside ι'' because the inside relation is transitive. This means that even if N' has owner parameters where ι'' is outside of ι, we still can ensure that N' remains unchanged as ι'' maps to itself in `map`. Therefore `map(`$\mathcal{H}(\iota')\downarrow_1$`) = `$N'$, and $\mathcal{H}'(\iota')\downarrow_1$ `= map(`$\mathcal{H}(\iota')\downarrow_1$`)`.

In the case R-SheepRef, $\mathcal{H} = \mathcal{H}'$ and `map` $=$ `map`$'$, which trivially proves $\vdash \mathcal{H}'$ OK and $\mathcal{H}' \vdash$ `map`$'$ OK respectively. To show $\mathcal{H}(\iota'')\downarrow_1$ `= map` $(\mathcal{H}(\iota')\downarrow_1)$ we must consider how ι' was stored in `map`. If it is by R-SheepOutside then we can use the same reasoning as the case above, where $\iota' = \iota''$ and $\mathcal{H}(\iota')\downarrow_1$ `= map(`$\mathcal{H}(\iota')\downarrow_1$`)`. This is because either the mappings in `map` apply the owner parameters of $\mathcal{H}(v')\downarrow_1$ with an identity mapping or none at all. If ι' was added into `map` by R-SheepInside then by definition ι'' is a Sheep clone. Therefore for some $\mathcal{H}_1$ and $\mathcal{H}_2$, and some `map`$''$, we have $\mathcal{H}_2(\iota'')\downarrow_1$ `= map`$''(\mathcal{H}_1(\iota')\downarrow_1)$. Then by the recursive nature of R-SheepInside we know that $\mathcal{H}_1 \subseteq \mathcal{H}$ and $\mathcal{H}_2 \subseteq \mathcal{H}$. Therefore $\mathcal{H}(\iota'')\downarrow_1$ `= map`$''(\mathcal{H}(\iota')\downarrow_1)$, so now we must show `map`$''(\mathcal{H}(\iota')\downarrow_1) =$ `map` $(\mathcal{H}(\iota')\downarrow_1)$. Again by the recursive nature of R-SheepInside we know that `map`$''\subseteq$ `map`, hence `map` $=$ `map`$''$, `map`*, for some `map`*. By **Theorem 2**, we can deduce that $\mathcal{H}(\iota')\downarrow_1$ has no formal owner parameters in `dom(map`*`)` because the formal owner parameters of ι' have

to be outside of ι''s `owner`. Therefore $\texttt{map}''(\mathcal{H}(\iota')\downarrow_1) = \texttt{map}(\mathcal{H}(\iota')\downarrow_1)$, and finally, $\mathcal{H}(\iota'')\downarrow_1 = \texttt{map}(\mathcal{H}(\iota')\downarrow_1)$.

The proof for the case R-SheepInside is far more complicated than the previous three cases. We will only outline the proof here. Please contact the authors for the full proof. The first step is to show $\vdash \mathcal{H}_1$ OK, $\mathcal{H}_1 \vdash \texttt{map}_1$ OK and $\mathcal{H}_1(\iota'')\downarrow_1 = \texttt{map}(\mathcal{H}(\iota')\downarrow_1)$. This can achieved by the premises of R-SheepInside, along with **Lemma 2** and **Lemma 3**. Then we introduce a **sublemma** to show the recursive calls of `SheepAux` give $\vdash \mathcal{H}_j$ OK, $\mathcal{H}_j \vdash \texttt{map}_j$ OK and $\mathcal{H}_j(v'_j)\downarrow_1 = \texttt{map}_{j-1}(\mathcal{H}(v_{j-1})\downarrow_1)$. This **sublemma** is proved by numerical induction over j, where the base case is when $j = 1$. To prove the inductive case we invoke the induction hypothesis of the main lemma. Next we need to show $\vdash \mathcal{H}'$ OK when $\mathcal{H}' = \mathcal{H}_{n+1}[\iota'' \mapsto \{\texttt{map}(N);\ \overline{\texttt{f}\rightarrow v'}\}]$. For all the v' produced by the **sublemma** to be assigned into $\mathcal{H}_{n+1}$, we must first show that each v' has the same type as the `null` when the clone (ι'') was created. This is achieved by the correctness property that `SheepAux` does not change objects in $\mathcal{H}$, **Theorem 2**, and substitution principles. Then with the **Lemma:** *heap preserves expression typing on field assignment*, we show that the v's have the correct types under $\mathcal{H}'$. This gives $\vdash \mathcal{H}'$ OK. Next is to show $\mathcal{H}' \vdash \texttt{map}_{n+1}$ OK. From the **sublemma** we have $\mathcal{H}_{n+1} \vdash \texttt{map}_{n+1}$ OK and because `null`s are not in $\texttt{dom}(\texttt{map}_{n+1})$ or $\texttt{range}(\texttt{map}_{n+1})$, by the definition of `map`, and F-Map, and the definition of $\mathcal{H}'$, we can deduce that $\forall\ \iota : \iota \in \texttt{range}(\texttt{map}_{n+1}) \Rightarrow \iota \in \texttt{dom}(\mathcal{H}')$. Therefore $\mathcal{H}' \vdash \texttt{map}_{n+1}$ OK. Since we have already shown that $\mathcal{H}'(\iota'')\downarrow_1 = \texttt{map}(\mathcal{H}(\iota')\downarrow_1)$, we are done.

> **Lemma 2:** Mapped types preserves well-formedness.
>
> *For all* $\mathcal{H}$, `map`, *and* N, **if** $\vdash \mathcal{H}$ OK, $\mathcal{H} \vdash$ `map` OK, *and* $\mathcal{H} \vdash N$ OK **then** $\mathcal{H} \vdash \texttt{map}(N)$ OK.

Proof outline: This lemma is proved by natural deduction on $\mathcal{H} \vdash N$ OK. Let $\texttt{map} = \{\overline{\iota \mapsto \iota'}\}$, then by definition $\texttt{map}(N) = [\overline{\iota'/\iota}]N$. $\mathcal{H} \vdash [\overline{\iota'/\iota}]N$ OK is proved by structural induction on the derivation of $\mathcal{H} \vdash N$ OK when N= `o:C<`$\overline{\texttt{o}}$`>`. By the premises of F-Class $\mathcal{H} \vdash \texttt{o}, \overline{\texttt{o}}$ OK, $\mathcal{H} \vdash [\overline{\texttt{o}}/\overline{\texttt{x}}](\overline{\texttt{o}_l \preceq \texttt{o}})$ OK, $\mathcal{H} \vdash [\overline{\texttt{o}}/\overline{\texttt{x}}](\overline{\texttt{o} \preceq \texttt{o}_u})$ OK, and there exists a class `class C<`$\overline{\texttt{o}_l \preceq \texttt{x} \preceq \texttt{o}_u}$`>...`. Then we invoke the **Lemma:** *owner variable substitution preserves owner well-formedness* on $\mathcal{H} \vdash \texttt{o}, \overline{\texttt{o}}$ OK with $\mathcal{H} \vdash$ `map` OK and $\vdash \mathcal{H}$ OK to get $\mathcal{H} \vdash [\overline{\iota'/\iota}]\texttt{o}, [\overline{\iota'/\iota}]\overline{\texttt{o}}$ OK. Similarly, we invoke the **Lemma:** *owner variable substitution preserves inside relation* on $\mathcal{H} \vdash [\overline{\texttt{o}}/\overline{\texttt{x}}](\overline{\texttt{o}_l \preceq \texttt{o}})$ OK and $\mathcal{H} \vdash [\overline{\texttt{o}}/\overline{\texttt{x}}](\overline{\texttt{o} \preceq \texttt{o}_u})$ OK with $\mathcal{H} \vdash$ `map` OK, and $\vdash \mathcal{H}$ OK. This gives $\mathcal{H} \vdash [\overline{\iota'/\iota}]([\overline{\texttt{o}}/\overline{\texttt{x}}](\overline{\texttt{o}_l \preceq \texttt{o}}))$ OK and $\mathcal{H} \vdash [\overline{\iota'/\iota}]([\overline{\texttt{o}}/\overline{\texttt{x}}](\overline{\texttt{o} \preceq \texttt{o}_u}))$ OK. Finally we apply F-Class on $\mathcal{H} \vdash [\overline{\iota'/\iota}]\texttt{o}, [\overline{\iota'/\iota}]\overline{\texttt{o}}$ OK, $\mathcal{H} \vdash [\overline{\iota'/\iota}]([\overline{\texttt{o}}/\overline{\texttt{x}}](\overline{\texttt{o}_l \preceq \texttt{o}}))$ OK, and $\mathcal{H} \vdash [\overline{\iota'/\iota}]([\overline{\texttt{o}}/\overline{\texttt{x}}](\overline{\texttt{o} \preceq \texttt{o}_u}))$ OK to get $\mathcal{H} \vdash [\overline{\iota'/\iota}]N$ OK.

> **Lemma 3:** Map preserves field type.
>
> *For all* $\mathcal{H}$, `map`, $\texttt{f}_i$, *and* N, **if** $\vdash \mathcal{H}$ OK, $\mathcal{H} \vdash$ `map` OK, $\mathcal{H} \vdash \texttt{map}(N)$ OK, *and* $\texttt{fType}(\texttt{f}_i, N) = N'$ **then** $\texttt{fType}(\texttt{f}_i, \texttt{map}(N)) = \texttt{map}(N')$.

Proof outline: This lemma is proved by natural deduction on $\texttt{fType}(\texttt{f}_i, N) = N'$. Let $\texttt{map}=\{\overline{\iota \mapsto \iota'}\}$, then $\texttt{map}(N) = [\overline{\iota'/\iota}]N$, $\texttt{map}(N') = [\overline{\iota'/\iota}]N'$, and $\texttt{fType}(\texttt{f}_i, [\overline{\iota'/\iota}]N) = [\overline{\iota'/\iota}]N'$. Let N= `o:C<`$\overline{\texttt{o}}$`>`, then with $\mathcal{H} \vdash [\overline{\iota'/\iota}]$`o:C<`$\overline{\texttt{o}}$`>` OK and **Lemma 2**, we can state that $\mathcal{H} \vdash [\overline{\iota'/\iota}]\texttt{o}, [\overline{\iota'/\iota}]\overline{\texttt{o}}$ OK, therefore $\mathcal{H} \vdash ([\overline{\iota'/\iota}]\texttt{o})$`:C<`$[\overline{\iota'/\iota}]\overline{\texttt{o}}$`>` OK. Next, by applying the definition of `fType` on $\texttt{fType}(\texttt{f}_i, ([\overline{\iota'/\iota}]\texttt{o})$`:C<`$[\overline{\iota'/\iota}]\overline{\texttt{o}}$`>`) and $\texttt{fType}(\texttt{f}_i, N) = N'$, along with substitution principles, we get $\texttt{fType}(\texttt{f}_i, ([\overline{\iota'/\iota}]\texttt{o})$`:C<`$[\overline{\iota'/\iota}]\overline{\texttt{o}}$`>`$) = [\overline{\iota'/\iota}]N'$. This proof follows from the proof for the **Lemma:** *owner variable substitution preserves type well-formedness.*

Below we present the owners-as-dominators theorem. We are required to prove this theorem as part of subject reduction. This theorem states that for all well-formed heaps, all references to an object come from inside the owner (as defined by $own_{\mathcal{H}}$) of that object [4]. Intuitively this means all references to an object can only come from the object's owner, siblings of the object, or from inside the object's context.

> **Theorem 2:** Owners-as-dominators.
>
> *For any* $\mathcal{H}$, **if** $\vdash \mathcal{H}$ OK **then** $\forall \iota \mapsto \{N; \{\overline{\texttt{f} \mapsto v}\}\} \in \mathcal{H}$ *where* $\forall \iota' \in \overline{v}\ :\ \mathcal{H} \vdash \iota \preceq own_{\mathcal{H}}(\iota')$.

This theorem is proved by showing every expression reduction preserves this property on the heap it produces. For the reduction R-Sheep, the only interesting case is R-SheepInside. The proof for owners-as-dominators on the heap ($\mathcal{H}'$) produced by R-SheepInside is achieved in two parts. The first part shows the heap with the newly created clone (ι') preserves owners-as-dominators. This holds by the fact that the owner of the clone is inside the owner of the original cloned object, i.e., the owner of the object that initiated the Sheep Clone. The second part is to show all the values of ι' satisfy the owners-as-dominators property. This is achieved by the transitivity of two inside relations. The first inside relation is that the owner of the values is outside the owner of the fields they are assigned to. The second relation is that the owner of the field is outside ι', the object which the fields belong to. The transitivity of these two inside relations gives owners-as-dominators for $\mathcal{H}'$.

Finally, we present our progress theorem. The proof for our progress theorem is standard and has been omitted.

> **Theorem 3:** Progress.
>
> *For all* $\mathcal{H}$, $\mathcal{H}'$, e, e', *and* N, **if** $\mathcal{H} \vdash e : N$ *and* $\vdash \mathcal{H}$ OK **then** $e; \mathcal{H} \rightsquigarrow e'; \mathcal{H}'$ **or** $\exists v : e = v$.

Progress is proved by structural induction on the derivation of expression typing. A case analysis is required for T-Assign, T-Invk, and T-Sheep, as the reduction for these expressions does not always reduce down to a value in a single step.

4.5 Correctness of Sheep Cloning

Below, we present seven correctness properties and their proofs to show correctness for Sheep Cloning.

The first property states that a new object must be created when Sheep Cloning an object and the newly created object must not be the same as the cloned object:

Correctness property 1: Sheep Cloning creates a new object.

For all $\mathcal{H}$, $\mathcal{H}'$, ι, *and* ι', **if** $\vdash \mathcal{H}$ OK *and* $\texttt{sheep}(\iota)$; $\mathcal{H} \rightsquigarrow \iota'$; $\mathcal{H}'$ **then** $\iota' \notin \texttt{dom}(\mathcal{H})$ *and* $\iota \neq \iota'$.

Proof outline: This property is proved by case analysis on the premise of $\texttt{sheep}(\iota)$, which is the function $\texttt{SheepAux}(\iota, \iota, \mathcal{H}, \emptyset)$. The cases R-SheepNull, R-SheepRef, and R-SheepOutside are all not applicable for this particular SheepAux function. For the case R-SheepInside, the premise states that $\mathcal{H}(\iota')$ undefined, hence $\iota' \notin \texttt{dom}(\mathcal{H})$. We can then deduce $\iota \neq \iota'$ by $\iota' \notin \texttt{dom}(\mathcal{H})$ and the semantics of SheepAux, where it states that $\iota \in \texttt{dom}(\mathcal{H})$.

The second property states the clones must preserve owners-as-dominators, as stated in **Theorem 2**:

Correctness property 2: Sheep Cloning preserves owners-as-dominators.

For all $\mathcal{H}$, $\mathcal{H}'$, ι, *and* ι', **if** $\vdash \mathcal{H}$ OK *and* $\texttt{sheep}(\iota)$; $\mathcal{H} \rightsquigarrow \iota'$; $\mathcal{H}'$ *and* $\mathcal{H}$ *preserves* **owners-as-dominators** **then** $\mathcal{H}'$ *preserves* **owners-as-dominators**.

Proof outline: This property is proved by the proof of **Theorem 2:** *expression reduction preserves owners-as-dominators on heap*. The outline of the proof for **Theorem 2** is to show the heap preserves owners-as-dominators in each cases of the SheepAux function. This is trivial for R-SheepNull, R-SheepRef, and R-SheepOutside. For R-SheepInside, we have already discussed its proof outline.

The third property states that Sheep Cloning creates a sub-heap that contains the new object:

Correctness property 3: Sheep Cloning creates a sub-heap that contains the new object.

For all $\mathcal{H}$, $\mathcal{H}'$, $\mathcal{H}''$, ι, *and* ι', **if** $\vdash \mathcal{H}$ OK *and* $\texttt{sheep}(\iota)$; $\mathcal{H} \rightsquigarrow \iota'$; $\mathcal{H}'$ *and* $\iota' \neq \iota$ **then** $\exists\, \mathcal{H}''$ *where* $\mathcal{H}' = \mathcal{H}$, $\mathcal{H}''$ *and* $\iota' \in \texttt{dom}(\mathcal{H}'')$ *and* $\iota \in \texttt{dom}(\mathcal{H})$.

Proof outline: This property is proved by the same reasoning as the proof for **Lemma 1**. Once again R-SheepNull, R-SheepRef, and R-SheepOutside are not applicable. Making the only interesting case R-SheepInside, by the premise of the case and **Lemma 1**, we know that **if** $\texttt{SheepAux}(\iota, \iota', \mathcal{H}, \texttt{map}) = \iota'', \mathcal{H}', \texttt{map}'$ **then** $\mathcal{H} \subseteq \mathcal{H}'$. This give $\mathcal{H}_1 = \mathcal{H}, \iota' \mapsto \{...\}$, which implies $\mathcal{H} \subseteq \mathcal{H}_1$ and $\iota' \in \texttt{dom}(\mathcal{H}_1 \setminus \mathcal{H})$. Again by the premise of this case, we get $\mathcal{H}_1 \subseteq \mathcal{H}'$ and $\mathcal{H}_1 \setminus \mathcal{H} \subseteq \mathcal{H}''$. Which let us conclude that $\iota' \in \texttt{dom}(\mathcal{H}'')$.

The fourth property states that if a reference in a clone is pointing to an object (ι) in the original heap ($\mathcal{H}$), that does not contain the clone, then ι must be outside that clone.

Correctness property 4: Sheep Cloning does not introduce references to the cloned object's representation.

For all $\mathcal{H}$, $\mathcal{H}'$, ι, *and* ι', **if** $\vdash \mathcal{H}$ OK *and* $\texttt{sheep}(\iota)$; $\mathcal{H} \rightsquigarrow \iota'$; $\mathcal{H}'$ *where* $\mathcal{H}' = \mathcal{H}$, $\mathcal{H}''$ *and* $\iota' \neq \iota$ *and* $\forall\, \texttt{f} \mapsto \iota'' \in \texttt{range}{\downarrow_2}(\mathcal{H}'')$ *where* $\iota'' \in \texttt{dom}(\mathcal{H})$ **then** $\mathcal{H}' \vdash \iota \preceq \iota''$.

Proof outline: This property is proved by natural deduction on ways that ι'' can be added into the range of $\mathcal{H}'$. This is achieved by case analysis on the construction of the Sheep clone by the SheepAux function. The cases R-SheepNull and R-SheepInside are not applicable, as $\texttt{null} \notin \texttt{dom}(\mathcal{H})$ and $\iota'' \in \texttt{dom}(\mathcal{H})$ respectively. The case R-SheepRef does not offer any insight into the relation between ι'' and ι. We must determine how ι'' was added into the map. For the case R-SheepOutside we have $\iota'' \in \texttt{dom}(\mathcal{H})$ if $\texttt{f} \mapsto \iota'' \in \texttt{range}{\downarrow_2}(\mathcal{H}'')$ by the definition of R-SheepOutside. Then by the premise of R-SheepOutside, $\mathcal{H} \vdash \iota \preceq \iota''$, which then gives $\mathcal{H}' \vdash \iota \preceq \iota''$, since $\mathcal{H} \subseteq \mathcal{H}'$.

The fifth property states that a reference can point to objects in the cloned heap ($\mathcal{H}''$), if and only if those objects are inside the representation of a clone.

Correctness property 5: All new objects are in the representation of the clone, and all objects in that representation are new.

For all $\mathcal{H}$, $\mathcal{H}'$, ι, *and* ι', **if** $\vdash \mathcal{H}$ OK *and* $\texttt{sheep}(\iota)$; $\mathcal{H} \rightsquigarrow \iota'$; $\mathcal{H}'$ *where* $\mathcal{H}' = \mathcal{H}$, $\mathcal{H}''$ *and* $\iota' \neq \iota$ **then** $\iota'' \in \texttt{dom}(\mathcal{H}'')$ **if and only if** $\mathcal{H}' \vdash \iota'' \preceq \iota'$.

Proof outline: This property is proved in two parts by case analysis on the SheepAux function. The first part is to show $\mathcal{H}' \vdash \iota'' \preceq \iota'$ when $\iota'' \in \texttt{dom}(\mathcal{H}'')$. This is only possible by R-SheepInside. By the premises of R-SheepInside, $\mathcal{H} \vdash \iota^* \preceq \iota$, where $\texttt{map}(\iota^*) = \iota''$. By **lemma:** *address mapping preserves inside relation*, we have $\mathcal{H} \vdash \texttt{map}(\iota^*) \preceq \texttt{map}(\iota)$, which gives $\mathcal{H} \vdash \iota'' \preceq \iota'$, and finally gives $\mathcal{H}' \vdash \iota'' \preceq \iota'$, as $\mathcal{H} \subseteq \mathcal{H}'$. The second part is to show $\iota'' \in \texttt{dom}(\mathcal{H}'')$ when $\mathcal{H}' \vdash \iota'' \preceq \iota'$. This part is also only possible in R-SheepInside. By the same argument as the proof outlined for the fourth correctness property, we have $\mathcal{H} = \mathcal{H}, \iota' \mapsto \{N, \overline{\texttt{f} \mapsto v}\}$ and $\mathcal{H}_1 \subseteq \mathcal{H}'$. Hence $\iota' \mapsto \{N, \overline{\texttt{f} \mapsto v}\} \in \texttt{dom}(\mathcal{H}' \setminus \mathcal{H})$, which means $\iota' \mapsto \{N, \overline{\texttt{f} \mapsto v}\} \in \texttt{dom}(\mathcal{H}'')$.

The sixth property states that all objects outside the cloned object are also outside the clone.

Correctness property 6: All objects outside the cloned object are outside the clone.

For all $\mathcal{H}$, $\mathcal{H}'$, ι, *and* ι', **if** $\vdash \mathcal{H}$ OK *and* $\texttt{sheep}(\iota)$; $\mathcal{H} \rightsquigarrow \iota'$; $\mathcal{H}'$ *where* $\iota' \neq \iota$ *and* $\forall \iota'' \in \texttt{dom}(\mathcal{H})$ *and* $\mathcal{H}' \vdash \iota \preceq \iota''$ **then** $\mathcal{H}' \vdash \iota' \preceq \iota''$.

Proof outline: This property is proved by contradiction on the construction of the Sheep clone. Lets assume for some ι^*, where $\iota^* \in \texttt{dom}(\mathcal{H})$, $\iota^* \neq \iota$, and $\mathcal{H}' \vdash \iota \preceq \iota^*$, that $\mathcal{H}' \nvdash \iota' \preceq \iota^*$. If $\iota^* = \iota$, then this property trivially holds. $\mathcal{H}' \nvdash \iota' \preceq \iota^*$ could either mean $\mathcal{H}' \vdash \iota^* \preceq \iota'$ or there are no ownership relation between ι' and ι^*. The latter is not possible, because there must be an ownership relation between ι' and ι^*, as ι' and ι have the same owner and we know that ι is inside ι^*. By the definition of SheepAux, ι^* would have been applied by R-SheepInside as $\mathcal{H}' \vdash \iota^* \preceq \iota'$. By the premise of R-SheepInside we know that $\mathcal{H}' \vdash \iota^* \preceq \iota$, which contradicts $\mathcal{H}' \vdash \iota \preceq \iota^*$. Therefore $\mathcal{H}' \nvdash \iota' \preceq \iota^*$ is not possible for some ι^*, where $\iota^* \in \texttt{dom}(\mathcal{H})$ and $\mathcal{H}' \vdash \iota \preceq \iota^*$.

The seventh property states that for each reference inside a clone pointing to objects that are outside the cloned object, there exists a corresponding reference from the cloned object pointing to those objects.

> **Correctness property 7:** For all references from an object inside the clone to an object outside the clone, there is a reference to the same object from inside the cloned object.
>
> *For all* $\mathcal{H}$, $\mathcal{H}'$, ι, *and* ι', **if** $\vdash \mathcal{H}$ OK *and* $\texttt{sheep}(\iota)$; $\mathcal{H} \rightsquigarrow \iota'$; $\mathcal{H}'$ *where* $\mathcal{H}' = \mathcal{H}$, $\mathcal{H}''$ *and* $\iota' \neq \iota$ *and* $\forall\ \texttt{f} \mapsto \iota'' \in \texttt{range}{\downarrow_2}\ (\mathcal{H}'')$ *and* $\mathcal{H}' \vdash \iota' \preceq \iota''$ **then** $\exists\ \texttt{f}' \mapsto \iota'' \in \texttt{range}{\downarrow_2}\ (\mathcal{H})$.

Proof outline: This property is proved by natural deduction on the construction of Sheep clones by the `SheepAux` function. By $\mathcal{H}' \vdash \iota' \preceq \iota''$ we can deduce that $\iota'' \mapsto \iota'' \in \texttt{map}$ and $\mathcal{H}' \vdash \texttt{map}(\iota) \preceq \texttt{map}(\iota'')$, as $\texttt{map}(\iota') = \iota$. Hence by **lemma:** *address mapping preserves inside relation*, we have $\mathcal{H}' \vdash \iota \preceq \iota''$. Now we must consider how the field (`f`) was constructed. The only possible way is when R-SheepInside recursively traverse through the values of the object it is cloning. Hence, ι'' must be a value of a field of an object in the original heap.

5. FUTURE WORK

In this section, we explore some of our ideas for future work regarding Sheep Cloning.

5.1 Ownership Transfer with Sheep Cloning

Ownership transfer was first presented by Clarke and Wrigstad [8] using external uniqueness. External uniqueness ensures objects are only accessible via a single externally unique reference from outside the object's representation. No restrictions are placed on internal referencing from inside the object's representation. Ownership transfer is achieved by transferring the externally unique reference. Clarke and Wrigstad use movement bounds to designate the scope that each externally unique reference can be moved. A unique reference cannot be moved outside its movement bounds.

Müller et al. describe another implementation of ownership transfer with Universe Types [19]. Universe Types [18] enforce the owners-as-modifiers policy, where objects are freely aliased but only their owner is allowed to modify them. Ownership transfer in Universe Types is achieved by creating clusters of objects and moving a (externally) unique pointer between clusters. The external uniqueness property is achieved by allowing only one unique reference into each cluster, and similarly, internal referencing within a cluster is unrestricted.

Ownership transfer, with external uniqueness and Universe Types, illustrated the need for the owner of the object that would be transferred to enforce the external uniqueness property. By the semantics of Sheep Cloning in R-Sheep and Fig. 17, we can deduce that all Sheep Clones inherit the external uniqueness property. This is because Sheep Cloning guarantees that the only reference into a newly created Sheep clone is from the owner of the Sheep clone. This means the Sheep clones possess the externally uniqueness property as the reference from its owner behaves as an external unique reference. By transferring this reference to another owner, we can simulate the first part of ownership transfer. The second part is to disown and dereference the original object that has been cloned. Combining these two parts, we can simulate the behaviour of ownership transfer as shown in external uniqueness.

We believe there are advantages in supporting ownership transfer via Sheep Cloning. Any constraints that were on the object to allow its ownership to be transfer, would now be on its Sheep clone instead. For example, the movement bounds described by Clarke and Wrigstad [8] demand a trade-off between what an object can access and where it can be moved. An object with tight movement bounds has more restrictions on its movement but fewer restrictions on what it can access; whereas an object with loose movement bounds has fewer restrictions its movement but what it can access is severely limited. In a Sheep Cloning based ownership transfer system the movement bounds would only constrain the Sheep clones, and not the original object. The trade-off for movement bounds still exists, however, they would be generated when constructing the Sheep clones. We aim to formalise this style of ownership transfer in a formalism similar to the one presented in this paper.

5.2 Sheep Cloning without Owners as Dominators

Cheng and Drossopoulou present ideas to perform object cloning in an ownership system without owners-as-dominators [5]. Their system is build on top of the system developed by Drossopoulou and Noble [12]. Cheng and Drossopoulou identify a set of problematic cases. For example, when a reference path re-enters the representation of an object from outside the object's representation. It is also possible that this reference path is the only way to reach that particular part of the object's representation. Cheng and Drossopoulou offer two alternative solutions, either enforce owners-as-dominators and all possible problematic cases would cease to exist, or statically prevent cloning on these problematic cases.

It is possible for their system to determine these properties statically, however, Sheep Cloning operates at runtime. For a system without owners-as-dominators, Sheep Cloning must traverse the entire heap to determine which objects are in an object's representation. A simple solution is for Sheep Cloning to ignore all objects in the cloned object's representation that are not reachable from its owner without going out of its representation. However, this would mean Sheep Cloning would no longer clone every object in an object's representation.

Another issue is when a two-way reference exists between an object (`A`) inside a context and an object (`B`) outside that context, as shown in Fig. 18. Sheep Cloning the object (`A`) inside the context would create an object ($\texttt{A}'$) that also has a reference to `B`, however, `B` would not know the existence of $\texttt{A}'$. Consider if the system has an invariant property where the purpose of `A` is to pass messages to `B`, and that `B` has to reply to any message passed by `A`. Then a clone of `A` would expect this bidirectional relationship with `B`. However, if $\texttt{A}'$ passes a message to `B`, `B` would respond to `A` instead of $\texttt{A}'$, as `B` does not know the existence of $\texttt{A}'$, breaking the invariant of the system.

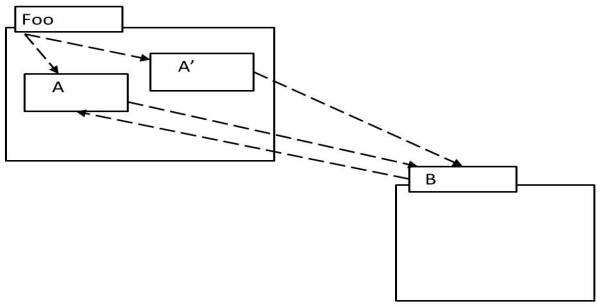

Figure 18: Sheep cloning with descriptive ownership types.

One of the benefits of Sheep Cloning is how it utilities ownership types and the structures provided by owners-as-dominators. Which means we may need to consider a different set of semantics for Sheep Cloning in systems without owners-as-dominators.

6. RELATED WORK

Drossopoulou and Noble [12] propose a static object cloning implementation, inspired by ownership types. Every object has a cloning domain and objects are cloned by cloning their domain. Just as ownership types enforce a topological structure upon the heap, the cloning domain provides a hierarchical structure for the objects in the program. This is achieved by placing cloning annotations on every field of every class. Using these annotations the cloning paths for each field of a class are created. Objects can have paths to other objects that are not in their cloning domain. The decision to clone an object is determined by the cloning domain of the initial cloning object or the *originator*. Each `clone()` method explicitly states, through `Boolean` parameters which fields are in its cloning domain. The `clone()` method then recursively calls the `clone()` method of each field, passing in `Boolean` arguments set by the originator.

In Drossopoulou and Noble's system, a parametric clone method is of the form `clone(Boolean` $\mathtt{s}_1$`, ..., Boolean` $\mathtt{s}_n$`, Map m)`. The variables $\mathtt{s}_1$, ..., $\mathtt{s}_n$ in the arguments of a class's `clone()` method are associated with the fields of that class. An object is cloned when that object's `clone()` method is called, and fields are cloned only if `true` is passed into the cloning parameter ($\mathtt{s}_i$). In contrast, the expression for Sheep Cloning is `sheep(`ι`)`, where ι is the object to be cloned.

Sheep Cloning is context free, which means the semantics of Sheep Cloning remain the same regardless of the class using it or the object (ι) passed in. Sheep Cloning uses a mapping between the original object and its clone, as does Drossopoulou and Nobles' `map` in their `clone()` method. The mapping in Sheep Cloning, however, operates on the heap at run-time, hiding the implementation of Sheep Cloning from its users.

There are also several papers that discuss the need for ownership-based cloning. In "Exceptions in ownership types systems" [11], Dietl and Müller outline several possible solutions to exception handling for Universe Types. One solution is to clone the exception object when it appears, then propagate the clone through the stack to the exception handler. They then explain that supporting exceptions by cloning requires no changes to their ownership system. In the end, they did not choose to handle exceptions with cloning. The reasons they cited were the need for every object in the system to be `cloneable` and the overhead cost of object cloning, especially if an exception is propagated multiple times before it is caught.

In "Minimal Ownership for Active Objects" [6], Clarke et al. develop active ownership, an ownership-based active object model for concurrency. An active object is an object that interacts with asynchronous methods while being controlled by a single thread. To guarantee safety and provide freedom from data races for the interaction between active objects, Clarke propose using unique references and immutable objects, and cloning the active object only when necessary. They then discuss three cases where they must clone the active objects, by using a "minimal clone operation". The minimal clone operation determines whether an object's fields are cloned or aliased based on their ownership annotation. This makes their operation very similar to Sheep Cloning, so much so that Clarke et al. mention how Sheep Cloning can be used in its place.

Aside from the work of Drossopoulou and Noble, we are aware of one other cloning implementation that is similar to Sheep Cloning. In Nienaltowski's PhD. thesis [20], he reiterates the excessiveness of copying an object's whole structure using deep_import (deep cloning) and the potential dangers introduced by shallow cloning. This inspired him to introduce a lightweight operation, object import, for Eiffel's SCOOP (Simple Concurrent Object-Oriented Programming). Object import copies the objects of non-separate references while the objects from a separate reference are left alone. When cloning objects in SCOOP all non-separate references must be followed and the objects reached, are copied, whereas the objects of separate references are considered harmless. The policy of copying objects by distinguishing between separate references and non-separate references is similar to the policy of cloning objects by distinguishing between objects inside the representation and objects outside the representation. Sheep Cloning and object import, however, still have their differences. Sheep Cloning uses ownership types, a method to control the topology of objects on the heap, while object import uses separate types, a method to identify objects for the SCOOP processor.

Jensen et al [16] propose placing static cloning annotations on classes and methods to aid users in constructing their cloning methods. The annotations define the copy policy for each class, where the policies ensure the maximum sharing possible between the original object and its clones. All cloning applications of a class must adhere to their copy policy. The copy policy is checked statically by a type and effect system. The copy policy does not perform cloning functions or generate the cloning method, it is just a set of specifications for clones produced. This differs from Sheep Cloning as our formalism includes an actual algorithm for object cloning, and our proofs guarantee the clones produced are structurally equivalent (as defined in Sect. 4.5) to the original object.

One of the first papers to identify the confusion between the semantics and the implementation of the copy function was Grogono and Chalin [13]. They discuss how it is more important if the objects being cloned are immutable or mutable than if the object is a value or a reference. They also touched on the idea of object representation, and the need to distinguish semantics from efficiency when copying objects. They concluded that effect-like systems need to play a greater role in object copying.

Grogono and Sakkinen [14] present a technique to generate a cloning function. They discuss the issues surrounding copying objects and the difficulty in comparing objects. Grogono and Sakkinen also present a set of detailed examples of various cloning operations and type equality. They explore the copying and comparing features in several programming languages.

7. CONCLUSION

In this paper we have presented a formalism of Sheep Cloning, and its soundness proof. We motivated the need for Sheep Cloning by comparing Sheep Cloning against existing form of object cloning and showing that Sheep Cloning is preferable.

8. REFERENCES

[1] Martín Abadi and Luca Cardelli. An imperative object calculus. In *Theory and Practice of Software Development (TAPSOFT)*. 1995.

[2] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An Interpretation of Objects and Object Types. In *Principles of Programming Languages (POPL)*, 1996.

[3] Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *European Conference on Object Oriented Programming (ECOOP)*, 2004.

[4] Nicholas Cameron. *Existential Types for Variance - Java Wildcards and Ownership Types*. PhD thesis, Department of Computing, Imperial College London, 2008.

[5] Ka Wai Cheng and Sophia Drossopoulou. Types for deep/shallow cloning. Technical report, Imperial College London, 2012.

[6] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Johnsen. Minimal ownership for active objects. In *Programming Languages and Systems*. 2008.

[7] David Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2001.

[8] David Clarke and Tobias Wrigstad. External Uniqueness is Unique Enough. In *European Conference on Object Oriented Programming (ECOOP)*, 2003.

[9] David G. Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[10] David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1998.

[11] Werner Dietl and Peter Müller. Exceptions in Ownership Type Systems. In *Formal Techniques for Java-like Programs (FTfJP)*, 2004.

[12] Sophia Drossopoulou and James Noble. Trust the clones . In *Formal Verification of Object-Oriented Software (FoVEOOS)*, 2011.

[13] Peter Grogono and Patrice Chalin. Copying, sharing, and aliasing. In *In Proceedings of the Colloquium on Object Orientation in Databases and Software Engineering (COODBSE'94)*, 1994.

[14] Peter Grogono and Markku Sakkinen. Copying and comparing: Problems and solutions. In *European Conference on Object Oriented Programming (ECOOP)*. 2000.

[15] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a Minimal Core Calculus For Java and GJ. *ACM Trans. Program. Lang. Syst.*, 2001.

[16] Thomas Jensen, Florent Kirchner, and David Pichardie. Secure the clones: Static enforcement of policies for secure object copying. In *European Symposium on Programming (ESOP)*, 2011.

[17] Peter Müller and Arnd Poetzsch-Heffter. Universes: A Type System for Controlling Representation Exposure. In *Programming Languages and Fundamentals of Programming*, 1999.

[18] Peter Müller and Arnd Poetzsch-Heffter. Universes: A Type System for Alias and Dependency Control. Technical Report 279, Fernuniversität Hagen, 2001.

[19] Peter Müller and Arsenii Rudich. Ownership transfer in universe types. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[20] Piotr Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, Department of Computer Science, ETH Zurich, 2007.

[21] James Noble, David Clarke, and John Potter. Object ownership for dynamic alias protection. In *Proceedings of the 32nd International Conference on Technology of Object-Oriented Languages (TOOL)*, 1999.

[22] John Plevyak and Andrew Chien. Type directed cloning for object-oriented programs. In *Languages and Compilers for Parallel Computing*. 1996.

[23] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic Ownership for Generic Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006.