Dynamic Semantics of Sheep Cloning: Proving Cloning

Paley Li[1], Nicholas Cameron[2], and James Noble[1]

[1] Victoria University, Wellington, New Zealand
[2] Mozilla Research

Abstract. Cloning objects is a common procedure in object oriented programming. Sheep cloning has been proposed as an automated object cloning procedure driven by object ownership. Formalising sheep cloning, and proving that formalism is correct, will provide reassurance that it is sound and works as designed, as well as providing a blueprint for implementation. In this paper, we discuss and compare three different dynamic semantics for sheep cloning. We show how each semantics progressively improves on the previous version.

1 Introduction

Sheep cloning is an automated cloning operation that uses ownership types to create clones. The concept is intuitive and relatively straightforward to describe, once the fundamentals of deep ownership are understood. Formalising sheep cloning, however, has not been as easy as we expected. We have iterated through three different ways to formalise the dynamic semantics of sheep cloning. In this paper, we describe each formalisation, their strengths and weaknesses, and how our approach evolved over time. We hope our journey illustrates the trade-offs and design goals in designing each formalism.

A formalism is useful for specifying a language feature (or features), offering insight into the crucial aspects of that feature (which, hopefully feeds back into the design of the feature or similar features, and how they integrate into a real language), and showing that a feature is safe (when used to prove safety properties such as type soundness). A formalism is not an implementation and these motivations for formalisation may lead to conflicting requirements: we would like the formalism to be as close as possible to the implementation (if it is far from it, then it tells us nothing useful), we would like the formalism to be small and simple to understand, and we would like the formalism to be amenable to proof.

Given that sheep cloning is a relatively simple feature, we hoped we could have a formalism which was fairly close to the implementation whilst still being useful. Our first formalism is satisfactory in that it is close to an implementation, but not in other respects. We noticed in the first formalism that an intermediate data structure (the mapping from the objects being cloned to the new objects) was highly significant. For the second formalism, we teased out the operations

around this mapping and gained some further insight into the nature of sheep cloning and its implementation. The second formalism was also easier to comprehend, however, it still missed the final requirements, as its proofs were complex and unwieldy. For our final attempt, we moved significantly towards the abstract. This third formalism is much easier to use in proofs, and is simpler to read and understand, however, this simplicity comes at the expense in the details of how a sheep clone could be computed.

The rest of this paper is organized as follows: in Sect. 2 we introduce ownership types, and sheep cloning; in Sect. 3 we describe three formalisms of sheep cloning, and discuss the quality of each; in Sect. 4 we discuss related work; and in Sect. 5 we conclude and discuss ideas for possible future work.

2 Ownership Types

Ownership types were introduced in 1998 by Clarke et al [6]. This was followed by a variety of ownership systems, such as Ownership Domains [1], Universe Types [13][14], Ownership with Disjointness of Types and Effects [5], External Uniqueness [4], and Ownership Generic Java [16]. More recently, ownership types have been used in Deterministic Parallel Java [2], and Scala Actors [10]. The descriptive and/or prescriptive properties of these systems differ, but in all these systems the type system enforces an hierarchical structure on the heap.

An object consists of more than just its in-memory representation. Some fields of an object point to other objects which are logically part of that object (owned objects), and some fields point to logically separate objects. In an ownership type system the type of an object indicates its owner, making the difference between owning and non-owning references explicit. An object's logical representation can be found by computing all objects which it transitively or directly owns.

Owners-as-dominators, or deep ownership, ensures an objects representation can never be exposed outside that object [6][3]. In practice, this means that references from an object must either go to the objects it directly owns, its siblings, or up the ownership hierarchy. References are allowed up the ownership hierarchy because these references are pointing into representations that they are a part of.

2.1 Sheep Cloning

Two objects are considered shallow copies of each other if they are bit-wise equal. Two objects are deep copies of each other if they are of the same type, each field of value type is bit-wise equal, and each field of reference type is a deep copy of the corresponding field in the other object.

A sheep clone [15][12] is a compromise between shallow and deep cloning. Some notion of an object's logical representation, such as that given by ownership types, is required. We say that two objects are sheep clones of one another if they have the same type, fields of value type are equal, fields of reference type

which point outside the object's representation are bit-wise equal, and fields of reference type which point into the object's representation are sheep clones. This last caveat means that when we consider the recursive cases of a sheep clone, we look at whether the current object is inside or outside the original object's representation, not the current object's own representation.

It should be clear from the definition above that sheep cloning is a refinement of deep cloning. It is also instructive to think of sheep cloning as a refinement of shallow cloning. A shallow copy is a literal copy and thus all references from the object are aliased and all the objects internal data is copied. In a sheep clone, all aliases from the logical object (the object and its representation) are aliased and all internal data (including owned objects) is copied. Put more succinctly, a sheep clone is a shallow clone from the perspective of an object's representation, rather than the perspective of a single object.

Sheep cloning is particularly suited for implementation in a language which enforces the owners-as-dominators property. In such a language, if an object `b` is outside an object `a`'s representation, then `b` cannot hold a reference into `a`'s representation. That means that two objects which are sheep clones are interchangeable — no other object can have references into either clone's internals. Furthermore, when testing two objects for the sheep clone property (or, equivalently, creating a sheep clone of an object), once we are outside an object's representation, we no longer have to worry about references back into that representation.

Based on this understanding of sheep cloning, we set out to formalise a sheep clone operation which takes an object and creates a sheep clone of that object. We imagine such an operation added to a Java-like language with deep ownership. To start with, we chose a very operational approach, hoping to remain close to a hypothetical implementation, and making explicit the traversal of an object and its references.

3 Sheep Cloning Semantics

In this section, we present the syntax, ownership features, auxiliary functions, and well formedness judgments required to formalise sheep cloning. We present three unique semantics for sheep cloning, the motivations behind each semantic, and discuss the validity of each.

The syntax for formalising sheep cloning is presented in Fig. 1. The heap, $\mathcal{H}$, is a set of mappings from object address, ι, to actual objects, $\{N, \overline{\mathtt{f}{\rightarrow}v}\}$. An object consists of its type, N, and a mapping from the names of it's fields, `f`, to the value of those fields, v. A type contains the owner of the object, `o`, a set of context variables, $\overline{\mathtt{o}}$, and a class name, `C`. Values are either object addresses, ι, or `null`. Similarly, context variables are either object addresses, ι, or `world`. A `map` is a set of mappings between object addresses, ι.

In this ownership system, the inside relationship describes the hierarchical structure of the objects in the heap. The judgments for the inside relation of ownership are presented in Fig. 2. The judgment $\mathcal{H} \vdash \iota \preceq \iota'$ is interpreted as ι

Dynamic syntax:

$$
\begin{array}{lll}
\mathcal{H} & ::= & \overline{\iota \rightarrow \{N,\ \overline{\mathtt{f} \rightarrow v}\}} \quad \textit{heaps} \\
N & ::= & \mathtt{o{:}C\langle\overline{o}\rangle} \quad \textit{class type} \\
\mathtt{map} & ::= & \{\overline{\iota \mapsto \iota'}\} \quad \textit{map} \\
v & ::= & \iota \mid \mathtt{null} \quad \textit{value} \\
\mathtt{o} & ::= & \iota \mid \mathtt{world} \quad \textit{context variable} \\
\iota & & \textit{object address} \\
\mathtt{C} & & \textit{class names} \\
\mathtt{f} & & \textit{field names}
\end{array}
$$

Fig. 1. Dynamic syntax.

Dynamic inside relation:

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

$$\frac{\mathcal{H} \vdash \iota \preceq \iota'' \quad \mathcal{H} \vdash \iota'' \preceq \iota'}{\mathcal{H} \vdash \iota \preceq \iota'} \text{(I-Trans)} \qquad \frac{}{\mathcal{H} \vdash \iota \preceq \mathtt{own}_{\mathcal{H}}(\iota)} \text{(I-Owner)}$$

Fig. 2. Dynamic inside relation.

is inside ι' under $\mathcal{H}$. I-Ref and I-Trans describe the reflexive and transitive properties of the inside relation. I-World states all objects are inside `world`. Finally, I-Owner states that every object is inside its owner.

Owner function:

$$\frac{\mathcal{H}(\iota) = \{\mathtt{o : C\langle\overline{o}\rangle},\ ...\}}{own_{\mathcal{H}}(\iota) = \mathtt{o}}$$

Look up function for field type:

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

Fig. 3. Auxiliary functions.

Two auxiliary functions, $own_{\mathcal{H}}$ and $fType$, are defined in Fig. 3. The $own_{\mathcal{H}}(\iota)$ function gives the owner of ι. The $fType(\texttt{f}, N)$ function gives the type of the field `f`. The static variables in the type of `f` are substituted with the actual context variables presented at runtime.

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

$$\frac{\mathcal{H} \vdash \mathcal{H} \text{ OK}}{\vdash \mathcal{H} \text{ OK}} \text{ (F-HeapE)}$$

$$\frac{\begin{array}{c}\mathcal{H}' \subseteq \mathcal{H} \qquad \forall\, \iota \to \{N; \overline{\texttt{f}\to v}\} \in \mathcal{H}' \colon \{\ \mathcal{H} \vdash N \text{ OK} \\ fType(\texttt{f}, N) = N' \quad \mathcal{H} \vdash v : [\iota/\texttt{this}]N' \quad \mathcal{H} \vdash \texttt{own}_{\mathcal{H}}(\iota) \not\preceq \iota\} \\ \forall\, v \in \overline{v} : \{v \neq \texttt{null} \Rightarrow v \in dom(\mathcal{H}) \ \wedge\ \mathcal{H} \vdash \iota \preceq own_{\mathcal{H}}(v)\}\end{array}}{\mathcal{H} \vdash \mathcal{H}' \text{ OK}} \text{ (F-Heap)}$$

Fig. 4. Well-formed heap.

Fig. 4 presents the judgments, F-HeapE and F-Heap, for heap well formedness. Heap well formedness can be judged under a larger heap. F-HeapE states heaps are well formed under themselves, while F-Heap states that a heap, $\mathcal{H}'$, is well formed under a larger heap, $\mathcal{H}$, if for all objects in $\mathcal{H}'$: their type is well formed; the values of their fields are well typed; they are not inside their owner; and for all non-`null` values inside $\mathcal{H}$, those objects are inside the owner of these non-`null` values. The remaining semantics, typing, and well formedness judgments of the formalism are standard, and previously presented [12].

3.1 Recursive Traversal

In this subsection, we present our first attempt at formalising sheep cloning. This approach is inspired by the concept that sheep cloning is a variant of shallow and deep cloning as described in section 2.1. The structure of an object's sheep clone can be the same as that object's shallow clone or deep clone. Consider an object that does not own any other objects, the sheep clone of this object is identical to its shallow clone. Next consider an object that does not refer to any object outside of its representation, the sheep clone of this object is identical to its deep clone.

First sheep cloning semantics:

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

Fig. 5. First sheep cloning reduction.

Sheep Auxiliary function:

$$\frac{\begin{array}{c}\mathcal{H}(\iota') = \{N;\ \overline{\mathtt{f}\to v}\} \\ \mathcal{H} \vdash \iota' \preceq \iota \\ \mathtt{map}(\iota')\ \mathtt{undefined} \\ \mathcal{H}(\iota'')\ \mathtt{undefined} \\ \mathtt{map}_1 = \mathtt{map},\ \iota' \mapsto \iota'' \\ \mathcal{H}_1 = \mathcal{H},\ \iota'' \mapsto \{\mathtt{mapT}(N, \mathtt{map});\ \overline{\mathtt{f}\to\mathtt{null}}\} \\ \mathtt{n} = |\{\overline{\mathtt{f}\to v}\}| \\ \forall_j : 1 \le \mathtt{j} \le \mathtt{n} : \{\mathtt{SheepAux}(\iota,\ v_j,\ \mathcal{H}_j,\ \mathtt{map}_j) = v'_j;\ \mathcal{H}_{j+1};\ \mathtt{map}_{j+1}\} \\ \mathcal{H}' = \mathcal{H}_{n+1}[\iota'' \mapsto \{\mathtt{mapT}(N, \mathtt{map});\ \overline{\mathtt{f}\to\mathtt{null}}[\mathtt{f}_j \mapsto v'_j]\}]\end{array}}{\mathtt{SheepAux}(\iota,\ \iota',\ \mathcal{H},\ \mathtt{map}) = \iota'';\ \mathcal{H}';\ \mathtt{map}_{n+1}}$$

(R-SheepInside)

$$\frac{\begin{array}{c}\mathcal{H} \vdash \iota' \not\preceq \iota \\ \mathtt{map}(\iota')\ \mathtt{undefined} \\ \mathtt{map}' = \mathtt{map},\ \iota' \mapsto \iota'\end{array}}{\mathtt{SheepAux}(\iota,\ \iota',\ \mathcal{H},\ \mathtt{map}) = \iota';\ \mathcal{H};\ \mathtt{map}'}$$

(R-SheepOutside)

$$\frac{\mathtt{map}(\iota') = \iota''}{\mathtt{SheepAux}(\iota,\ \iota',\ \mathcal{H},\ \mathtt{map}) = \iota'';\ \mathcal{H};\ \mathtt{map}}$$

(R-SheepRef)

$$\frac{}{\mathtt{SheepAux}(v,\ \mathtt{null},\ \mathcal{H},\ \mathtt{map}) = \mathtt{null};\ \mathcal{H};\ \mathtt{map}}$$

(R-SheepNull)

Fig. 6. First sheep cloning semantics

Mapped type:

$$\frac{\mathtt{map} = \{\overline{\iota \mapsto \iota'}\}}{\mathtt{mapT}(N, \mathtt{map}) = [\overline{\iota'/\iota}]N}$$

(T-Map)

Fig. 7. Map function.

Our first formalism of the sheep cloning reduction is presented in Fig. 5. The sheep cloning expression is reduced by invoking the `SheepAux` function. The `SheepAux` function, as presented in Fig. 6, takes four arguments and returns a 3-tuple. The first argument is the object being cloned, which we have defined as the target object. The second argument is the object being traversed: this object determines which case of the `SheepAux` function will be applied by its relationship with the target object. The third argument is the heap at the current state of

sheep cloning, and the final argument is a `map`. The 3-tuple returned by the `SheepAux` function contains the sheep clone of the second argument, the heap, and `map`. The heap and `map` are updated with the newly created sheep clones. The `SheepAux` function has four cases. The R-SheepNull case describes when the function traversed a `null`. In R-SheepNull, the `SheepAux` function returns a `null`, and no updates are made to the heap or `map`. The R-SheepRef case occurs when the traversed object already exists in the domain of the `map`. This indicates the object has previously been traversed, and the sheep clone of this object already exists in the `map`. The `SheepAux` function returns the mapping of this object, no updates are made to the heap or `map`. The R-SheepOutside case describes when the traversed object is outside the target object. In R-SheepOutside, the reference leading to the traversed object needs to be copied. The `SheepAux` function returns this object, as this object is not part of the sheep clone. The heap is not updated as no sheep clone is created. The `map` is updated with a mapping from this object to itself. To ensure the reference to this object is copied as well as to prevent infinite traversals of a reference loop. The R-SheepInside case describes when the traversed object is inside the target object. In R-SheepInside, the object needs to be copied. A new object is created with the mapped type of the traversed object. The fields of the new object are initialised to `null`. The `SheepAux` function is recursively called on every field of the traversed object, with the values produced assigned to the corresponding field in the new object. The `map` returned is then updated with the new object, and every new object from the recursive calls over the fields of the traversed object. Similarly, the heap is updated with the new object, and every new object created from the recursive call of `SheepAux` over the fields of the traversed object.

In Fig. 7, we present the `mapT` function. `mapT` uses the `map` to create a mapped type (`mapT(`N`)`) of a type, N, by substituting over N with the elements in the domain of the `map` for the corresponding element in the range.

Reflecting on Recursive Traversal

We found the `SheepAux` function to be an inadequate way of formalising sheep cloning. The inductive case, R-SheepInside, is monolithic and complicated. This is basically the classic software engineering mistake of making a function that does too much. In the following formalism we improve this by spitting `SheepAux` into two functions, one that creates the `map` and the other that uses the `map` to create the sheep clone.

A formalism can only be considered sound once it has been proven. To prove soundness we require the heap produced by the `SheepAux` function to be well formed. This is proved by structural induction over the derivation of the `SheepAux` function, with case analysis over each of its cases. The difficulty with this proof lies in the large amount of interleaving inductive cases required for the recursive case, R-SheepInside. A numerical induction is required for each field of every object that is being copied. The `map` is critical in creating sheep clones, it is used in the base case, R-SheepOutside, and the recursive

case, R-SheepInside, as a mechanism to prevent looping during the traversal of the target object's representation. R-SheepInside also uses the `map` via the `mapT` function in Fig. 7, to construct the type for the objects in the sheep clone's representation. The `map` describes an abstract representation of the sheep clone. The domain of the `map` contains the target object's representation, while the range of the `map` contains the sheep clone's representation. When the `map` is mapped over the target object, the structure of the target object is imprinted with the structure of its sheep clone, creating an abstract representation of the sheep clone. The `map` is also used in the construction of the objects in the sheep clone's representation, as shown in the `mapT` function. A closer examination of `SheepAux`, reveals several possible simplifications to formalising sheep cloning. `SheepAux` constructs the representation of the sheep clone while the target object is being traversed. A better alternative is to create the representation of the sheep clone using the `map` as a blueprint. The `map` becomes a representation of the clone as well as a way to create the sheep clone. This will allow for a much cleaner and more concise sheep cloning procedure.

3.2 Using makeMap and makeHeap

In this section, we present our `map` inspired formalism of sheep cloning. The fundamental idea behind this semantics is to formalise sheep cloning through the abstract representation of the sheep clone presented in the `map`. This is achieved with two functions, `makeMap` and `makeHeap`. The `makeMap` function creates a `map`, while the `makeHeap` function takes the `map` and constructs the objects that comprise the sheep clone's aggregate.

Second sheep cloning semantics:

$$\frac{\texttt{makeMap}(\iota,\ \emptyset)_{\mathcal{H},\iota} = \texttt{map} \quad \texttt{makeHeap}(\texttt{map})_{\mathcal{H},\texttt{map}} = \mathcal{H}' \quad \texttt{map}(\iota) = \iota'}{\texttt{sheep}(\iota);\ \mathcal{H} \rightsquigarrow \iota';\ \mathcal{H}'}$$

(R-Sheep)

Fig. 8. Second sheep cloning reduction.

Fig. 8 presents the sheep cloning reduction using the `makeMap` and `makeHeap` functions. The `makeMap` function is a recursive function that traverses through the object graph of the target object. For each object inside the target object, as defined by the inside relation in Fig. 2, a mapping is created and added to the `map`. `makeMap` takes an object address (ι) and a `map`, and produces a `map`. There are also two subscripts on the `makeMap` function. $\mathcal{H}$ is the heap before the reduction of the sheep cloning expression, and ι is the address of the target object. The `map` for the target object is created by invoking `makeMap` with the

Dynamic Map Well-formedness:

$$\frac{\mathcal{H} \vdash \texttt{map}\ \text{OK} \quad \iota \notin \texttt{dom(map)} \quad \iota' \notin \texttt{range(map)}}{\mathcal{H} \vdash \iota \mapsto \iota', \texttt{map}\ \text{OK}}\ (\text{F-MAP}) \qquad \frac{}{\mathcal{H} \vdash \emptyset\ \text{OK}}\ (\text{F-EMPTY})$$

Fig. 9. Dynamic Map Well-formedness:

Type Mapping:

$$\frac{\texttt{map} = \{\overline{\iota \mapsto \iota'}\}}{\texttt{map}(N) = [\overline{\iota'/\iota}]N}\ (\text{T-MAP})$$

Fig. 10. Type Maping

address of the target object and an empty map. Once again the `map` prevents objects being cloned more than once. The sheep clone and its representation is created by `makeHeap` taking a `map`, and recursively traversing it, creating a new object for each mapping. The result contains the sheep clone as well as the heap before the reduction of the sheep cloning expression. The two subscripts of the `makeHeap` function are the heap before the sheep cloning reduction ($\mathcal{H}$), and the `map` of the target object. Finally, the sheep clone of an object (ι) is the mapping of that object (`map(`ι`)`).

Definitions for `map` well-formedness and `map` substitution are presented in Fig. 9. A `map` is a set of mappings ($\iota \mapsto \iota'$) between objects, where objects (ι) inside the target object are mapped to their sheep clone (ι'). A `map` is considered well-formed if it is empty (F-EMPTYMAP), or if it is a bijective function (F-MAP). In Fig. 10, we present the definition for mapping over a type, T-MAP, where the `map` is applied to the context variables contained in the type. The ability to map over a type is crucial in defining the types for sheep clones.

Fig. 11 presents the three base cases, and the recursive case of `makeMap`. For the base cases the `map` remains unchanged, with `makeMap` returning the `map` that is passed in. The three base cases are: SC-MAPNULL, SC-MAPEXIST, and SC-MAPOUTSIDE. SC-MAPNULL occurs when `makeMap` traverses a `null`. SC-MAPEXIST occurs when `makeMap` reaches an object that is already in the domain of the `map`. To prevent circularity in the traversal, `makeMap` ignores any objects it has already seen. SC-MAPOUTSIDE occurs when `makeMap` reaches an object that is outside the target object. By the definition of sheep cloning, objects outside the target object are aliased, therefore they are not added to the `map`. The recursive case for `makeMap` is SC-MAPINSIDE. For this case, `makeMap` has reached an object (ι') that is inside the target object ($\mathcal{H} \vdash \iota' \preceq \iota$), and is not

MakeMap functions:

$$\frac{}{\texttt{makeMap(null, map)}_{\mathcal{H},\iota} = \texttt{map}}$$
(SC-MapNull)

$$\frac{\iota' \in \texttt{dom(map)}}{\texttt{makeMap}(\iota', \texttt{map})_{\mathcal{H},\iota} = \texttt{map}}$$
(SC-MapExist)

$$\frac{\mathcal{H} \vdash \iota' \not\preceq \iota}{\texttt{makeMap}(\iota', \texttt{map})_{\mathcal{H},\iota} = \texttt{map}}$$
(SC-MapOutisde)

$$\frac{\begin{array}{cc}\iota' \notin \texttt{dom(map)} & \mathcal{H} \vdash \iota' \preceq \iota \\ \iota'' \notin \texttt{dom}(\mathcal{H}) & \iota'' \notin \texttt{range(map)} \\ \texttt{map}_1 = \texttt{map},\ \iota' \mapsto \iota'' & \mathcal{H}(\iota') = \{N;\ \overline{\texttt{f}\rightarrow v}\} \\ \multicolumn{2}{c}{\forall\ v_i \in \overline{v} : \texttt{makeMap}(v_i, \texttt{map}_i)_{\mathcal{H},\iota} = \texttt{map}_{i+1}}\end{array}}{\texttt{makeMap}(\iota', \texttt{map})_{\mathcal{H},\iota} = \texttt{map}_{|\overline{v}|+1}}$$
(SC-MapInside)

Fig. 11. MakeMap functions

MakeHeap functions:

$$\frac{\begin{array}{c}\mathcal{H}(\iota) = \{N;\ \overline{\texttt{f}\rightarrow v}\} \\ \mathcal{H}' = \texttt{makeHeap}(\texttt{map}')_{\mathcal{H},\texttt{map}},\ \iota' \rightarrow \{\texttt{map}(N);\ \overline{\texttt{f}\rightarrow\texttt{map}(v)}\}\end{array}}{\texttt{makeHeap}(\iota \mapsto \iota', \texttt{map}')_{\mathcal{H},\texttt{map}} = \mathcal{H}'}$$
(SC-MakeHeap)

$$\frac{}{\texttt{makeHeap}(\emptyset)_{\mathcal{H},\texttt{map}} = \mathcal{H}}$$
(SC-MakeHeapE)

Fig. 12. MakeHeap functions

already in the domain of the `map` ($\iota' \notin$ `dom(map)`), ι' is added to the `map`, and `makeMap` is called on the fields of ι'.

Fig. 12 presents the cases of `makeHeap`, SC-MakeHeapE and SC-MakeHeap. The base case SC-MakeHeapE occurs when there are no more mappings left in the `map`, in which case the heap in the subscript of `makeHeap` is returned. The recursive case SC-MakeHeap creates the clones with the mapped type of the type of the target object, and the mapped fields of the target object's fields.

Reflecting on makeMap and makeHeap

The `map` is a mapping of the object being cloned to the new object which is its clone. If the `map` is created correctly, substituting the representation of the object being cloned with its clones described in the `map` gives the representation of the sheep clone. This allows the properties describing the sheep clone to be translated and described on the `map` instead. Therefore, properties on the `map` can be proved instead of proving properties on the sheep clone directly. Forming the basis to our strategy of proving this formalism.

The proof of soundness for this sheep cloning semantics requires the heap, $\mathcal{H}'$, produced by the `makeHeap` function has to be well formed. To show $\mathcal{H}'$ is well formed we must show every object created by the `makeHeap` function is individually well formed, under the definition in Fig. 4, with respect to the resulting heap, $\mathcal{H}'$. To show an object is well formed, every field of that object must be well typed. These fields, however, may be pointing to objects that are not yet created by `makeHeap`. These objects will be eventually created, as they are in the `map`, and the recursive case of `makeHeap` creates a new object for every object in the range of the `map`. The possibility of an object with a reference to an object that is not yet created is the reason heaps must be judged under a larger heap for well formedness.

An important property required to show heap well formedness is owner consistency: when an object is not inside the target object, then the owner of that object is also not inside the target object. Properties describing the sheep clone can be described on the `map` instead of the heap, once these properties are established they can then be translated to described the sheep clone. The owner consistency property can be interpreted on the `map` as: if an object is not inside the domain of the `map`, then the owner of this object is also not in the domain of the `map`. Unfortunately, proving this property presents several issues. Intuitively, the `map` should have an inside property that describes how every object inside the target object's representation is inside the domain of the `map`. The SC-MapInside case of the `makeMap` function establishes this inside property for the `map`. SC-MapInside adds the objects inside the target object, as defined by the inside relation in Fig. 2, into the `map`. By contradiction on the inside property of `map`, along with the definitions of the inside relation, the owner consistency property over the `map` should fall out. The problem lies in that the inside property of the `map` is difficult to establish formally. We must show for every object inside the target object that is also reachable from the currently traversed object, they are inside the domain of the `map`. We call this *reachability inside*. To show reachability inside the recursive case, SC-MapInside, of `makeMap` we need to know which fields of the traversed object would not lead to a base case, as reachability inside is only possible for the objects that are inside the target object. Unfortunately, the `makeMap` function is simply not expressive enough to show that every object inside the target object is inside the domain of the `map`.

It is reasonable to expect the inside property where every object inside the target object's representation is inside the domain of the `map` to be part of `map` well formedness. Every `map` created by `makeMap` can be expected to have

the inside property, however, `map` well formedness (Fig. 7) only requires that the `map` be a bijective function. A stronger definition of `map` well formedness is required. A stronger well formedness definition could require the inside property over the `map`, however, `makeMap` must still show the `map`s it creates establishes the inside property. Resulting in `makeMap` having to show reachability inside, which we already know is not possible. The lack of expressiveness of `makeMap` has not changed.

It is important to note that not being able to show the owner consistency property for this formalism does not mean the owner consistency property, or the other properties like reachability inside, do not hold for sheep cloning. Rather the inability to show these properties is a consequence of the design of the semantics, and not the theory of sheep cloning. Formalising sheep cloning with the `map` focuses too heavily on the implementation details of sheep cloning. The `map` should be viewed as a means to implement sheep cloning, instead of trying to describe the structural and ownership properties required by sheep cloning. The `SheepAux`, `makeMap`, and `makeHeap` functions are great for describing an implementation of sheep cloning. The difficulties of their proof, as well as the unsightliness of their formalism, especially with the `SheepAux` function, demonstrate that an algorithmic approach is unsuitable for formalising sheep cloning.

3.3 Declarative Semantics

In this subsection, we present our third and final formalism of sheep cloning: a more declarative semantics of sheep cloning with more focus placed on the properties of sheep cloning, instead of the methods which sheep clones are created.

Third sheep cloning semantics:

$$\frac{\overline{\iota' \to\{N;\ \overline{\mathtt{f}\to v}\}} = \{\iota' \to\{N;\ \overline{\mathtt{f}\to v}\} \in \mathcal{H} \mid \mathcal{H} \vdash \iota' \preceq \iota\ \} \quad \mathcal{H}(\overline{\iota''})\ \mathtt{undefined} \quad \mathcal{H}' = \overline{\iota'' \to[\overline{\iota''/\iota'}]\{N;\ \overline{\mathtt{f}\to v}\}}}{\mathtt{sheep}(\iota); \mathcal{H} \rightsquigarrow [\overline{\iota''/\iota'}]\iota; \mathcal{H}, \mathcal{H}'}\ \text{(R-Sheep)}$$

Fig. 13. Third sheep cloning reduction.

Fig. 13 presents our third declarative sheep cloning reduction. The set of objects, $\overline{\iota' \to\{N;\ \overline{\mathtt{f}\to v}\}}$, represents the sub-heap that is the representation of the target object, or as the set notion describes, all objects inside the target object ι. The objects $\overline{\iota''}$ are the objects that will compose the representation of the sheep clone. The sub-heap, $\mathcal{H}'$, represents the sheep clone and its representation. $\mathcal{H}'$ is created by mapping over the set of objects that contains the representation of the target object. The resulting heap from this sheep cloning

reduction is the original heap, $\mathcal{H}$, along with the sub-heap, $\mathcal{H}'$, that contains only the representation of the sheep clone.

Reflecting on Declarative Semantics

This formalism describes the representation of the target object being cloned as a collection of objects which are inside that target object. This formalism does not provide any detail to how this collection is constructed, unlike the previous formalism which algorithmically places the objects of the target object's representation into a `map`. The representation of the sheep clone is described by a collection of fresh variables that have the substituted type and fields of the corresponding object in the collection that contains the target object.

From the previous two formalisms we discovered that formalising sheep cloning by how sheep clones are constructed resulted in formalisms that are difficult to prove. The priority for this formalism is to focus on making the proof of the formalism as clean and concise as possible. During the onset our primary focus was on formalising sheep cloning in an ownership system with existing object cloning idioms and formalisms. From the initial formalism we realised the importance of the `map`, and how the sheep cloning is free once the `map` is created. The benefits of the `map`, however, does not extend to the proof of our second formalism being succinct. In this formalism the concept of sheep cloning is described in sub-heaps: there exists a sub-heap containing the representation of the target object, which is then renamed to become the representation of the sheep clone. We recognised the process of identifying the sub-heap that contains the representation of the target object, and the process of renaming that sub-heap as implementation details that does not affect the semantics of sheep cloning.

The soundness proof for this formalism requires the sub-heap, $\mathcal{H}'$, containing the sheep clone to be well formed, and the sheep clone preserves the type of the sheep cloning expression. $\mathcal{H}'$ is shown well formed by a weaker version of the owner consistency property we discussed in section 3.2. The new owner consistency property states the inside relation is structurally consistent, or in layman's terms: if an object is not inside the target object, then the objects outside of that object are not inside the target object. Consider three objects, `a`, `b`, and `c` that act as containers or boxes, items that can be placed within one another. The structurally consistent lemma describes a property where if object `a` is placed inside object `b`, and object `a` is not inside object `c`: then object `b` cannot be inside of object `c`.

> **Lemma:** Inside relation is structurally consistent.
> *For all* $\mathcal{H}$, ι, ι', *and* ι''. **If** $\mathcal{H} \vdash \iota \preceq \iota'$ *and* $\mathcal{H} \vdash \iota \not\preceq \iota''$ **then** $\mathcal{H} \vdash \iota' \not\preceq \iota''$.

This lemma is proved by structural induction over the derivation of $\mathcal{H} \vdash \iota \preceq \iota'$, with a cases analysis on the last step.

4 Related work

In this section, we discuss other formal methods of object cloning.

Drossopoulou and Noble [7] 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 target object (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.

Jensen [11] 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 describes the semantics of sheep cloning in full: using sheep cloning, programmers never have to implement a clone method.

One of the first papers to identify the confusion between the semantics and the implementation of the copy function was Grogono and Chalin [8]. 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.

Finally, Grogono and Sakkinen [9] 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 of several programming languages.

5 Conclusion

We have presented three formalisations of sheep cloning. We compared the formalisms, and discussed how each subsequent formalism improves upon the previous formalism. An outline of the proof for each formalism provides a basis for the validity and soundness of these formalism. In hindsight, the declarative semantics may seem to be the "obvious" formalisation of sheep cloning. We could not, however, have come to such a conclusion without creating the preceding formalisms based on recursive traversal, and the `makeMap` and `makeHeap` functions.

In the future, we expect to present the complete formalism and soundness proofs. We are also especially keen to explore translating the properties described by the declarative formalism to the `makeMap` and `makeHeap` functions. If it is possible to show that the `map` produced by `makeMap` can correlate with the sub-heap that contains the sheep clone as described in the declarative formalism, then it should be possible to translate the lemmas describing the sub-heap to describe the `map`. This would allow us to prove soundness for the formalism that uses `makeMap` and `makeHeap`. We still strongly believe that the `map` is a very strong tool in formalising sheep cloning, and so we wish to prove soundness for the formalism based on `makeMap` and `makeHeap`, as well as the declarative formalism.

References

1. Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, 2004.
2. Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.
3. David Clarke. *Object Ownership and Containment.* PhD thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2001.
4. David Clarke and Tobias Wrigstad. External Uniqueness is Unique Enough. In *ECOOP*, 2003.
5. David G. Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *OOPSLA*, 2002.
6. David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, 1998.
7. Sophia Drossopoulou and James Noble. Trust the clones . In *Formal Verification of Object-Oriented Software (FoVEOOS)*, 2011.
8. 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.
9. Peter Grogono and Markku Sakkinen. Copying and comparing: Problems and solutions. In *ECOOP*. 2000.
10. Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *ECOOP*, 2010.

11. 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.
12. Paley Li, Nicholas Cameron, and James Noble. Sheep cloning with ownership types. In *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*, 2012.
13. Peter Müller and Arnd Poetzsch-Heffter. Universes: A Type System for Controlling Representation Exposure. In *Programming Languages and Fundamentals of Programming*, 1999.
14. Peter Müller and Arnd Poetzsch-Heffter. Universes: A Type System for Alias and Dependency Control. Technical Report 279, Fernuniversität Hagen, 2001.
15. 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.
16. Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic Ownership for Generic Java. In *OOPSLA*, 2006.