Object Cloning for Ownership Systems

by

Paley Guangping Li

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy
in Computer Science.

Victoria University of Wellington
2015

Abstract

Modern object-oriented programming languages frequently need the ability to clone, duplicate, and copy objects. The usual approaches taken by languages are rudimentary, primarily because these approaches operate with little understanding of the object being cloned. Deep cloning naively copies every object that has a reachable reference path from the object being cloned, even if the objects being copied have no innate relationship with that object. For more sophisticated cloning operations, languages usually only provide the capacity for programmers to define their own cloning operations for specific objects, and with no help from the type system.

Sheep cloning is an automated operation that clones objects by leveraging information about those objects' structures, which the programmer imparts into their programs with ownership types. Ownership types are a language mechanism that defines an owner for every object in the program. Ownership types create a hierarchical structure for the heap.

In this thesis, we construct an extensible formal model for an object-oriented language with ownership types (`Core`), and use it to explore different formalisms of sheep cloning. We formalise three distinct operational semantics of sheep cloning, and for each approach we include proofs or proof outlines where appropriate, and provide a comparative analysis of each model's benefits. Our main contribution is the `descripSC` formal model of sheep cloning and its proof of type soundness.

The second contribution of this thesis is the formalism of Mojo-jojo, a multiple ownership system that includes existential quantification over types and context parameters, along with a constraint system for context

parameters. We prove type soundness for Mojo-jojo. Multiple ownership is a mechanism which allows objects to have more than one owner. Context parameters in Mojo-jojo can use binary operators such as: intersection, union, and disjointness.

Acknowledgments

First and foremost, I would like to thank my supervisors, James Noble and Nicholas Cameron. I know it has taken me exceptionally longer than usual to complete this thesis, and I am eternally grateful for all the support and encouragements they've given me through all my struggles. The amount of wisdom, knowledge, and passion they have shown for their work has allowed me to have more confidence in my own work. Over these last few years both James and Nick have played a much bigger role in my life than just being my supervisors, they have been my psychiatrists, my nutritionists, and my role models both in and out of the office.

I am grateful to Tobias Wrigstad and Sophia Drossopoulou for all the interesting and helpful discussions.

I would like to thank Rujuta Cameron for being so gracious with her home, and for all the no-nonsense advice during each stay, they have been greatly appreciated.

I am grateful for all the people I've shared an office with, and for all the late afternoon coffees with Timothy Jones and allowing me to rant about my proofs.

I would like to thank Stephen and Melanie Nelson for feeding me and giving me a place to relax during the hectic period of writing up.

Finally, I would like to thank my parents for all of their support during this entire process, and a special thank you to uncle Mike who has always put my needs above his own.

Contents

Chapter 1

Introduction

Objects are the essential building blocks for creating programs in object-oriented programming languages like Java and C++ [36, 81]. The relationships between the objects influence the structure of the program, but the interactions between objects can be difficult to understand, and can cause unexpected behaviors within programs [50]. *Aliasing* is when multiple references interact with the same object.

In Java, the expression `Object b = a`, shown in Fig. 1.1, assigns `b` to the object referenced by `a`, making `b` and `a` aliases of the same object. Any changes made via either `b` or `a` will be reflected via the other object.

```
Object a = new Object();
Object b = a;
```

Figure 1.1: Aliasing.

The ease of aliasing means it can occur in situations where it may be unwanted or inappropriate. Controlling abuses of aliasing, and the behaviors resulted from aliasing are difficult [43]. Java prevents fields being directly accessed if they are annotated with the `private` keyword, however, fields annotated `private` can still be exposed through methods that return the contents of those fields.

The need for greater aliasing control spurred the development of *ownership types* [70, 26]. Every object in an ownership type system must have an owner, which makes the runtime topology of those objects explicit. The owners are declared, during compile-time, in the types of those objects. This gives programmers more control over the topology of their programs' runtime object structures during compile-time, and can aid programmers in creating safer programs. The ability to reason about the heap in a more sophisticated manner is valuable in a number of domains, allowing ownership types to be incorporated into domains, such as: verification [25, 64], memory management [8, 55], concurrency [22, 24], and parallelism [6, 5, 7].

Most ownership systems model the heap as a tree, however, trees are often too restrictive to describe many real programs. Empirical studies of object-oriented programs [63, 62] have shown that their runtime object structures are more complex than trees. Objects may need to be shared between multiple threads or domains. *Multiple ownership* structures the heap as a directed acyclic graph (DAG) rather than the tree in single ownership systems. In a multiple ownership system, objects can be owned by more than one object, which accommodates for programs with a more complex heap structure, that are otherwise not possible in a single ownership system. MOJO [16] is the first system to support multiple ownership, and it featured multiple owners per object, owner-wildcards, an effect system, and a simple "owners as sets" model. MOJO, however, has several issues, such as: classes designed for multiple ownership cannot always have the same declarations as those designed for a single owner; types are more restrictive than necessary; the language does not support encapsulation policies by restricting inter-object references or modifications; and the formalization is untidy.

For the code example shown in Fig. 1.1, instead of sharing the same object as `a`, we might wish for `b` to have its own object, so `b` requires a copy of the object in `a`. Object copying is also known as *object cloning*. The two most common object cloning techniques are shallow cloning and deep

cloning. Shallow cloning copies the object and aliases the references held by the object. Deep cloning copies the object and the entire structure of the object, recursively copying all other objects referenced by the object. Shallow clones are often too shallow as they share their object structure with the object being cloned, whereas deep clones are often too deep as they copy every object reachable from the object being cloned regardless if that object is relevant to the original object.

To address these issues in object cloning and multiple ownership, this thesis makes the following contributions:

- Formalises sheep cloning with three different formal models of the operational semantics for sheep cloning.
- Proves type soundness for the sheep cloning formalism, `descripSC`.
- Formalises a new multiple ownership system, Mojo-jojo, a more expressive and disciplined evolution of MOJO.
- Proves type soundness for Mojo-jojo.

1.1 Sheep Cloning

Sheep cloning [68] is an automated object cloning operation that uses ownership types to create clones with the internal structure of the object being cloned. Ownership types explicitly state the runtime topologies of the objects in the program, and this allows sheep cloning to copy only the parts of the objects' runtime object topologies that are deemed necessary. Sheep cloning copies the object's internal representation and aliases the objects outside the representation that are referenced from within the representation. Sheep cloning provides a compromise between shallow and deep cloning. Sheep clones are neither too shallow nor too deep, containing just the right amount of an object's structure. Sheep cloning has not been fully formalised. In this thesis, we wish to bring clarity and validation to sheep

cloning with a formalism of sheep cloning and a proof of type soundness for that formalism.

1.2 Formalising Sheep Cloning

We create three formalisms for the dynamic semantics of sheep cloning. Each formalism has its own strengths and weaknesses, however, each improves on its predecessor.

We have constructed an austere formalism called `Core`, that is a class-based object-oriented programming language with deep ownership. The `Core` serves as the foundation for all three formalism of sheep cloning.

The first formalism of the sheep cloning semantics (`recurSC`) is inspired by the semantics of shallow and deep cloning. The `recurSC` formalism recursively copies every object in representation of the object being cloned. The `recurSC` formalism is satisfactory in its resemblance to an implementation of sheep cloning, however, the formalism is crude, bulky, and lacks cleanliness. The `recurSC` formalism does highlight the importance of the mappings between the objects being cloned and their clones, which became a critical element in our successor sheep cloning semantics.

The second formalism (`mapSC`) first creates a `map` that contains the mappings between the representation of the objects being cloned and the representation of the clone, then constructs the sheep clone directly from `map`. The `mapSC` formalism is easier to comprehend, but its proofs were complex and unwieldy.

The third formalism (`descripSC`) describes the sheep cloning semantics in three parts. The first part identifies the sub-heap that contains the representation of the object being cloned, the second part copies that sub-heap, and the final part renames every object in the new sub-heap to be the representation of the sheep clone. The `descripSC` formalism is much easier to read, understand and prove sound.

The `recurSC` formalism is published in the paper “Sheep Cloning

with Ownership Types" [52], while the `mapSC` and `descripSC` formalisms are published in the paper "Dynamic Semantics of Sheep Cloning: Proving Cloning" [53].

1.3 Mojo-jojo

The overarching design goal of Mojo-jojo is to unify as many features as possible while remaining a simple and general language with multiple ownership. Mojo-jojo's core calculus is constructed to be expressive and flexible, but is also simple and re-usable. Mojo-jojo contains a system of constraints to describe the heap topologies, based straightforwardly on the rules of set algebra. The constraint system is flexible enough to describe more sophisticated language features for Mojo-jojo in the future, such as owners-as-dominators encapsulation [26], owners-as-modifiers encapsulation [65], permissions systems [49], and effect systems [38, 25, 80]. The additions of generics and existential quantification in Mojo-jojo allow a single class to describe objects which may be singly or multiply owned. Mojo-jojo has a smaller and less complex formal foundation than MOJO, which makes for a smaller and tidier proof of type soundness. Mojo-jojo is published in the paper "Mojojojo - More Ownership for Multiple Owners" [51].

1.4 Structure of Thesis

The thesis is organised as follows:

- Chapter 2 presents the background to the thesis.
- Chapter 3 presents the `Core` formalism, which serves as the foundation for our sheep cloning formalism.

- Chapter 4 presents `recurSC`, a formalism of sheep cloning inspired by the semantics of shallow and deep cloning.
- Chapter 5 presents `mapSC`, a sheep cloning formalism constituted on creating the `map` and construct clones as defined by the `map`.
- Chapter 6 presents `descripSC`, a formalism of sheep cloning that focuses on describing the sheep clone itself, instead of describing the construction of the sheep clone.
- Chapter 7 presents Mojo-jojo, a multiple ownership system that allows existential owners and contains a constraint system. The descriptions in chapter 7 are derived mostly from the multiple ownership paper [51] by Nicolas Cameron, James Noble, and me, with substantial contributions by Nicolas Cameron in section 7.4.
- Chapter 8 concludes the thesis, and presents our related work and future work.

Chapter 2

Background

In this chapter, we present the background work for this thesis: aliasing, ownership types, multiple ownership, and object cloning, with each feature discussed in an individual section. The first section describes the formal definitions of programming languages, and how languages features are formalised. The second section describes aliasing, the necessity of aliasing in object-oriented programming languages, and problems associated with aliasing. The third section presents ownership types, the limitations of ownership types, and the different kinds of ownership type systems. The fourth section presents multiple ownership, and the advantages of multiple ownership systems over single ownership systems. The final section describes how object cloning is generally supported in object-oriented programming languages.

Objects in an object-oriented program are individually encapsulated components of the program. Every object has a set of variables that hold references to other objects, and a set of methods that make use of the variables. Encapsulation is a strength of object-oriented programming as the internal representation of an object is hidden from other objects. An object interacts with other objects by invoking methods of the object it wants to interact with. Otherwise, objects can be accessed directly through fields or indirectly through the variables of other objects. A sequence of vari-

ables can be evaluated down to a single object by recursively evaluating each variable in the sequence to provide a new context to evaluate the next variable, until the final variable is reached. An object can invoke its own methods by using the special variable "self" or "this". The interactions between objects provide the basis for any object-oriented program. In most object-oriented programming languages, like Java [36], method arguments and method results are passed "by-reference".

2.1 Formal Definitions of Programming Languages

There are several ways to design a programming language feature. One approach described by Pierce [73] explains how programming languages can be constructed from the simply typed lambda calculus. Pierce introduces operational semantics as the basis for programming languages and demonstrates the techniques required to prove type soundness for his language systems. Pierce demonstrates the process by first introducing the untyped lambda calculus, incrementally expanding it to become the simply typed lambda calculus, and then gradually introducing features such as subtyping, recursive types, universal types and existential types.

Another approach is to formulate programming language features over an existing formal type system. Featherweight Java (FJ) and Featherweight Generic Java (FGJ) are formalisms of a type system for an object-oriented programming language [45, 46] that are designed to highlight a specific language feature: generic types. FGJ was designed to omit as many features of Java as possible, without compromising the paradigms and principles of Java and its type system [45]. FGJ has top-level class definitions, object instantiation, field access, method invocation and type casts. There are six parts to the FGJ type system: syntax; auxiliary functions; subtyping; well formedness; typing rules; and reduction rules. The syntax contains types, class declarations, constructor declarations, method declarations and expressions. The auxiliary functions assist the reduction of the

expressions, type checking, and the declaration of methods, and classes. The subtyping rules state that subtyping is reflexive and transitive, and define the subtype relations between type variables and their bounds. Well-formedness rules judge when classes, methods, and types are well formed with respect to the current environment. The types of expressions, classes, and methods are defined in expression typing, class typing, and method typing respectively. The reduction rules indicate the process by which a program is able to be reduced to a value. FGJ is a subset of Java, therefore all legal programs in FGJ are legal programs in Java. FGJ provides the blueprint to incorporate and formalise features into a Java-like language, without the need to construct a formalism for the entire language from scratch.

2.2 Aliasing

An object is aliased if there exist multiple reference paths to that object. A strength of object-oriented programming is the ability to provide encapsulation over objects, whereby the representation of each object is retained within those objects; however, this is not always possible. Aliasing allows objects to change the state of an object without the knowledge of the other objects referring to that object.

Consider the example of a list object `list`, which has two variables `size` and `elements`. `size` describes the number of elements in `list`, and `elements` is an array containing the data of `list`, which makes `size` and `elements` part of the representation of `list`. If `size` is increased without the knowledge of `list` or `elements`, then `list` may become internally inconsistent. If there were another list, `list2`, with the variables `size2` and `elements2`, it would be possible for `elements` and `elements2` to be aliases of the same underlying array object, which means changes to `elements` would consequently change `elements2`. If we want to implement a function `addToList`, which takes an array and a list, that adds the array to

the `elements` of the list, then it would be possible for `addToList` to retain arguments passed into the function by reference, and therefore, an array passed into `addToList` can be retained by it. This means `addToList` can cause changes to `elements` without the knowledge of the list that contains `elements`. The purpose of this example is to show how aliasing removes the control that `list` has over its representation.

2.2.1 Islands, Balloons, and Eiffel

Hogg et al. [43] propose four treatments for aliasing: detection, advertisement, prevention, and control. Detection is the ability to detect any (potential) aliasing. Advertisement is the ability to annotate methods to indicate how they would treat aliases. Prevention is the ability to disallow occurrences of aliasing for a particular object. Control describes how methods can isolate and contain the effects of the aliases that occur in those methods. The implementation of any of these four treatments in practice is non-trivial.

Detection of aliasing at compile-time is NP-hard [50], resulting in an analysis that returns either *never*, *sometimes*, or *always* when asked if two variables alias the same object. If the result is *never* or *always*, then the computations that use these variables can be optimised; however, if *sometimes* is the result then further analysis will be required as *sometimes* refers to situations where variables are aliased during some invocations of the system but not during others.

Alias Advertisement requires the development of new annotations, and imposes overheads when applying these annotations. An annotation could be required when the arguments passed into a method will not be modified within the method, similarly an annotation could be required when the arguments passed into the method will not be aliased. The annotations can also be used to indicate when the variables returned from the method will not be modified outside of the method. These annotations can be used

to prevent some of the issues caused by aliasing.

Alias Prevention ensures that aliasing cannot occur in certain contexts within the system. One example of this approach to resolving the issues surrounding aliasing was presented in the paper "Islands: Aliasing Protection in Object-Oriented Languages" by Hogg [42]. Islands ensure that no references can exist between groups of closely related objects, where each group is declared to be an island. The isolation between islands is achieved by setting previous references to null when objects are passed in and out of their islands. Islands can be nested and are completely encapsulated, meaning that "Islands" should be scalable. Other prevention approaches remove the assignment operator from the language, replacing it with a swap operator that exchanges the bindings of the two sides of the operator. This is the case in "Copying and swapping: Influences on the design of reusable software components" by Harms and Weide [41], however, the cost in time and memory to achieve this paradigm can be extravagant.

Alias control permits aliases in some or all parts of a program but seeks to manage or mitigate their effects. "Balloon types: Controlling sharing of states in data types" by Almeida [3] presents one approach to alias control. Objects of balloon types cannot have any of their states referenced by external objects, i.e. objects outside their representation. Balloon types enforce stronger encapsulation than the types seen in object-oriented programming languages such as Java [36]. A key contribution of Balloon types is establishing the importance of a classification of data types in regard to an object's ability to share its state, however, the invariants of Balloon types are achieved with static analysis instead of being a feature of the programming language.

Expanded types are introduced in "Eiffel: The language" by Bertrand Meyer [60]. Variables of expanded type hold the object itself and not a reference to that object; parameter passing and assignment are achieved by copying the object, otherwise known as passing "by value". Expanded

types eliminate the possibility of aliasing, therefore resolving any problems associated with aliasing. Expanded types do not support subtype polymorphism, which is an essential part of the object-oriented paradigm. Furthermore, expanded types, unlike recursive types, cannot represent linked substructures, e.g. stacks or sets, as it is not possible for a variable to contain a value, but rather the value will need to be represented by a group of objects.

2.2.2 Flexible Alias Protection

Noble et al. [70] present "Flexible Alias Protection", a conceptual model that controls the degrees in which objects can be changed through aliasing. Objects are allowed to be aliased, and the undesired effects of aliasing are reduced. Noble et al. introduce the terminology of aggregate objects, aliasing shadow, and alias-protected containers. An aggregate object is an object with a set of known aliases, and these known aliases are required to be controlled. The aliasing shadow of an aggregate object is the internal representation of that aggregate object. Consider the `list` object example at the start of this section, the `list` object is the aggregate object and the `size` and `elements` objects are the aliasing shadow of `list`. An alias-protected container is a particular type of aggregate object, and is used to limit the amount of change aliases are allowed on the aliasing shadow. Alias-protected containers divide the aliasing shadow into two categories, the private representation and the public arguments. The private representation cannot be accessed outside the container, but within the container the private representation can be accessed freely, depended upon, duplicated, created anew, and change its state, so long as it is not exposed outside. The public arguments of the container can be accessed freely from anywhere within the system, and objects can be arguments to more than one container. A container can only depend upon its arguments as long as they are immutable; a container can never depend upon

an argument's mutable state. The implementation of the container needs to keep the two sets, representation and arguments, completely separated. Alias-protected containers can be generalised into three invariants, which must be true for a container at all times.

- `No Representation Exposure`: References to the container's mutable representation cannot exist outside of the container.
- `No Argument Dependence`: A container cannot depend upon its arguments' mutable state.
- `No Role Confusion`: A container cannot return an object in one role when it is passed into the container of another role.

Noble et al. also developed a set of aliasing modes, and techniques that check these modes. The checking techniques statically ensure the three invariants hold for every container in the system at all times. The checking technique verifies the aliasing properties of the object by propagating aliasing modes through the expressions and checking for consistency within the defining context. The aliasing modes decorate the type constructors in the language's type expressions. There are five aliasing modes: '`rep`', '`arg R`', '`free`', '`val`' and '`var R`'. The '`R`' in the '`arg`' and '`var`' mode is used to distinguish the different roles within the '`arg`' and '`var`' modes.

- `rep`: Is the mode for the representation object of a container; objects with the `rep` mode are part of another object's representation. Objects referred to by a `rep` expression can change within the container.
- `arg R`: Is the mode for the argument object of a container; objects with the `arg` expression refer to an argument of an aggregate object. Objects referred to by the `arg` expression are considered to be immutable through the `arg` expression. The `R` in the mode is used to differentiate the different roles of a similar mode.

- `free`: The `free` mode is used to handle object creation; the `free` expressions hold the only references to this object in the system.

- `val`: The `val` mode is used to handle value types; a `val` expression references to a variable of a value type.

- `var R`: The `var` mode refers to expressions that change freely and can be aliased; a `var` expression provides for a weaker aliasing guarantee. The `R` is used to differentiate the different roles of a similar mode.

Finally, the combinations of these modes form the three invariants over the container.

2.2.3 Summary

The encapsulation methods presented in Islands [42] and Balloons [3] are closely related, although the mechanism and details of these two systems may appear different. They both aim towards providing a complete encapsulation of aliasing, statically preventing external references into an object's internal representation. The encapsulation policy presented by Noble et al. [70], however, is different to both Islands and Balloons. An alias-protected container does not prevent every reference into an aggregate object's shadow, but rather the container is able to separate the representation and arguments, while also protect its representation.

Each of these systems has their own drawbacks. Some forbid the presence of some object-oriented programming idioms such as requiring every alias in the system to be fully encapsulated, or restricting aliasing modes on methods or variables, or high syntactic overhead. None of these systems are perfect, and the best defense against aliasing will always be careful programming and constant vigilance.

2.3 Ownership Types

In this section, we discuss ownership types, different ownership type systems, and features of ownership types. One of the motivations for ownership types are the issues associated with aliasing. "Flexible Alias Protection" [70] offered a novel approach to containing and managing aliasing, however it was only a conceptual model. "Ownership Types for Flexible Alias Protection" by Clarke et al. [26] develops this conceptual model into a formal type system, giving rise to the concept of *ownership types*.

In the previous section, we discussed how object references can freely access other objects in object-oriented programming languages like Java, which creates issues with aliasing as the representation of an object can be exposed without the knowledge of the aggregate object. The issues surrounding aliasing were a primary motivation for ownership types, but ownership types are capable of far more than just aliasing control. The applications for ownership types include memory management [26, 55], garbage collection [25], distributed system and parallel programming [7, 6], program transformation [65] and program verification [64].

2.3.1 Clarke's Ownership Types

Clarke et al. [26] introduce object contexts, whereby each object owns the context of its representation. This means every object is in a context and the owner of that context is the owner of the object. Ownership types annotate an object's type with context information. The context information restricts access to objects, allowing only objects within the same context to access the representation of that context. In the ownership system introduced by Clarke et al., the top-most context is the `root` context, denoted by the annotation *norep*. Any object owned by the `root` context is considered to be global and owned by the system. The `root` context allows free sharing of values, global variables, and the existence of objects that are not part of any other objects' representation.

Fig. 2.1 depicts an example of two objects in an ownership system. The outer-most box represents the program, and the `root` context owns everything in this box. Clarke et al. use the `rep` context to denote that an object is in the representation of another object. The `rep` context is object dependent; objects with different `rep`s will be in different contexts, which is enforced by having `rep` declared upon the creation of each new object. The `owner` context denotes the owner of the object that the context is declared in. Owning an object is different to having a reference to an object.

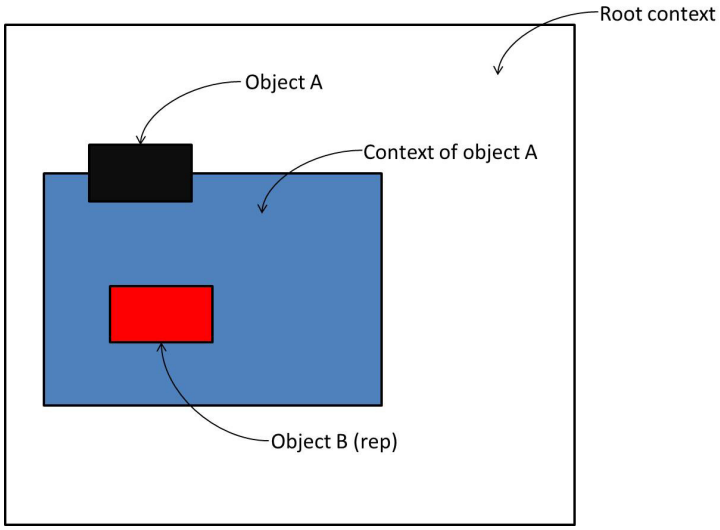


Figure 2.1: Ownership.

In Fig. 2.1, the `object B` is owned by `object A`, hence `object B` is in the context of `object A`. Ownership types consist of a class name, and the contexts represent the owner (`rep`, `norep` or `owner`) and the bindings for the context parameters. The owner of each object is declared as part of the type of that object. The owner is fixed for the entire existence of the object. No changes can be made to the owner of an object once the type of that object is declared, which allows ownership types to be statically enforced.

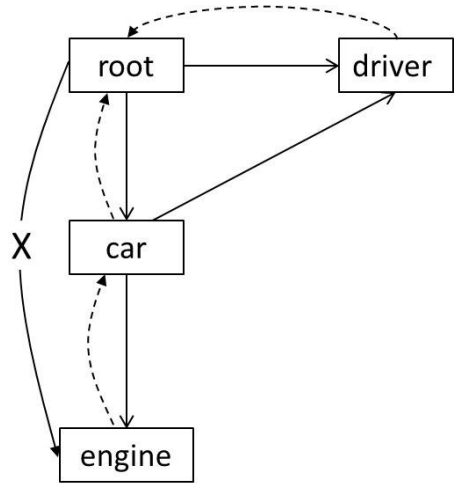


Figure 2.2: Car object graph from [70].

The diagram in Fig. 2.2 and the code in Fig. 2.3 present the example of a Car. In Fig. 2.2, solid arrows are references, dotted arrows are owner arrows, for objects to point to their owner, and solid arrows with a cross are illegal references. The `Engine` object is considered to be a part of the `Car` object, therefore, the `Engine` object is declared to have type `rep Engine` within class `Car`, thereby making it part of the `Car` object's context; this ensures that each `Car` object has it own `Engine` object. The `Driver` object is not part of `Car`, which means the `Driver` object has the context `norep`.

In Fig. 2.2, the reference arrow from the `Root` to the `Engine` is crossed out as this reference is illegal. The `Engine` is part of the representation of the `Car`, which means all references must go through the `Car` in order to reach the `Engine`. In Fig. 2.3, this is shown with the failed statement `car.engine.stop()`.

```
class Engine {
 void start() { ... }
 void stop() { ... }
}

class Driver { ... }

class Car {
 rep Engine engine; // representation
 norep Driver driver; // not representation

 Car() {
  //new engine is part of representation
  engine = new rep Engine(); //constructor
  driver = null;
 }

 rep Engine getEngine() { return engine; }
 void setEngine(rep Engine e) { engine = e; }

 void go(){
  if (driver !=  null) engine.start();
 }
}

class Main {
 void main() {
  norep Driver bob = new norep Driver();
  norep Car car = new norep car();

  car.driver = bob; //
  car.go();
  car.engine.stop(); // fails
  car.getEngine.stop(); // fails
  rep Engine e = new rep Engine();
  car.setEngine(e); // fails, different rep
 }
}
```

Figure 2.3: Car example from [70].

The statement `car.getEngine.stop()` fails for the reason that references cannot access the `Engine` directly, as they must go through `Car`. The statement `car.setEngine(e)` fails as the `rep` representing the object `e` in the class `Main` refers to a different context to the `rep` for the object `engine` in the `Car` class. The statement `car.go()` will succeed, even though the body of the `car.go()` statement equates to `car.engine.start()`.

Ownership type systems are well typed if three structural invariants are satisfied: role separation, restricted visibility, and representation containment [70]. Role separation states that different owners cannot appear in the same context. This is achieved by the construction of the type system, therefore, regardless of how the types are bound, the type system must ensure there is no coercion operating on the ownership types. Restricted visibility states that the objects in expressions (field assignments, arguments of method calls, field access) and objects returned from methods must be visible in the context of both the caller and callee. This means no dynamic aliasing of an object's representation is allowed; in other words, `rep` objects cannot be accessed outside their owning object. Finally, representation containment states that all paths from the root of the system to any object must pass through that object's owner. The representation containment invariant is proven inductively over the entire object graph of the program.

Prescriptive ownership [26] and *descriptive ownership* [65] are two ownership encapsulation policies that can be enforced in systems with ownership types. Descriptive ownership produces a purely topological ownership structure for the system, without enforcing any encapsulation properties over how objects may be accessed. Prescriptive ownership enforces encapsulation properties as well as ensures a topological ownership structure. The two most common prescriptive ownership policies are *owners-as-dominators* [26] and *owners-as-modifiers* [65]. Owners-as-dominators ensures that every reference to an object must go through its owner, while owners-as-modifiers ensures that mutations to an object can only occur

via the owner of that object.

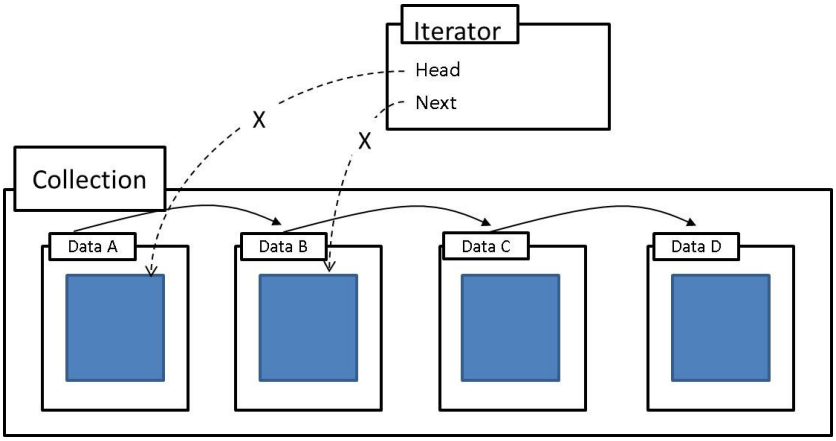


Figure 2.4: Iterator example.

Early versions of ownership types have several limitations [26]. One of these limitations is the inability to construct external iterators for the representation of substructures, e.g. stacks or sets. In Fig. 2.4, we present a collection of four elements and an iterator object. The iterator object needs access to the data (represented in blue) in each element, however, the iterator object does not own the element and therefore, cannot access the data. Another limitation of ownership types is that functions cannot operate over multiple representations as ownership types are not parametric polymorphic.

For example, consider a program that exchanges foreign currency. The program will require an addition function for adding money. Instead of making an addition function for each currency, you would like to have a generic function that operates over all currencies. A generic addition function that operates over all currencies is not possible in ownership types, as this would require the addition function to be the owner of every currency.

Along with "Ownership Types for Flexible Aliasing Protection" [26] there are several other papers by Clarke et al. on ownership types. "The ins and outs of objects" by Potter, Noble and Clarke [77], and "Object

Ownership for Dynamic Alias Protection" by Noble, Clarke, and Potter [68] are the two papers that are the most interesting to us. The former discusses the relationships between the topology of objects in the heap and the topology of objects in ownership types to identify and define object containment. The latter introduces an explicit, run time notion of object ownership to control aliases between objects in dynamically typed languages.

2.3.2 Ownership Domains

Aldrich et al. [2] introduce the concept of ownership domains, an extension of ownership types that allows objects to have multiple ownership domains. This allows a system where objects are divided into multiple subsystems, and where each subsystem can specify its own policy of interaction. The key benefit of multiple domains is the ability to have more flexible encapsulation by using link declarations between the domains, which allows for example: the implementation of iterators. A sequence abstract data type (ADT) can declare one domain for its internal representation and a separate domain for its iterator. The aliasing policy for the ADT can specify that the iterator's domain can access and write into the ADT, while prohibiting outside references from accessing the ADT's internal representation. Each object must specify a policy describing whether aliasing is permitted within its internal domain, and between its internal and external domains. Along with the user defined policies, there are two implicit policies that all systems follow:

- An object has permission to access other objects in the same domain; this allows internal representations to access external representations of the object.
- An object has permission to access objects in the domains that it declares; this allow internal representations to access other internal representations.

The paper goes on to formalize ownership domains as Featherweight Domain Java, an ownership domain language based on Featherweight Java, and prove the language is sound.

2.3.3 Generic Ownership

Generic types are a form of type polymorphism [18] which allows the use of type parameters. Generic types were introduced into Java by Bracha et al. [11] and then others [10]. "Generic Ownership for Generic Java" by Potanin, Noble, Clarke, and Biddle [75] introduces the concept of generic ownership types and its associated system, Ownership Generic Java (OGJ). OGJ incorporates generic ownership types into Java. Potanin et al. show how the ownership information of an object can be added into that object's generic types without any additional syntactic overhead or parameter space. The OGJ system treats ownership as a simple addition to generic types. The ownership parameter is declared at the end of a generic type's parameter list. Generic ownership allows the ownership parameters to be polymorphic, similarly, generic types allow types to be polymorphic. The beauty of OGJ is its simplicity in how it incorporates generic ownership types into the generic types in Java without compromising or altering generic types. OGJ uses a single parameter space for both the generic parameters of ownership and types. OGJ supports deep ownership (also known as owners-as-dominators) and provides encapsulation for transitively nested objects. OGJ achieve generic ownership types with three ownership parameters:

- `This`: Access to this object is restricted to the current instances of the class it is declared in. Similar to `rep` in [26].
- `World`: Access to this object is unrestricted. Again, this is similar to the `norep` in [26].
- `Package`: Access to this object is restricted to classes in the package.

2.3.4 Universe Types

During the development of ownership types, other approaches to resolve the issue of representation exposure were also being researched and developed. Universe Types introduced by Müller and Poetzsch-Heffter in "Universes: A type system for controlling representation exposure" [64], later re-formalised in "Universe Types for Topology and Encapsulation" by Cunningham, Dietl, Drossopoulou, Francalanza, Müller, and Summers [27]. Universe Types can be considered as a variety of ownership types, as Universe Types organise the object graph into an owners graph where each object is owned by at most one object. The key difference between Universe Types and the other ownership type systems at the time was that ownership types enforce the representation containment invariant, otherwise known as owners-as-dominators, while Universe Types enforce the owners-as-modifiers invariant, where all modifications of an object must be initiated by the object's owner. The paper "Universes: Lightweight Ownership for JML" by Dietl and Müller [32] was the first to use owner-as-modifier as well as the first to formalise the type system using viewpoint adaptation. Universe Types use the three modifiers (`rep`, `peer` and `any`) to denote the context of an object:

- `rep`: expresses the object is owned by the currently active object.
- `peer`: expresses the object has the same owner as the currently active object.
- `any`: expresses an object with a statically unknown owner; this is similar to wildcard types [17].

Generic types have been incorporated into Universe Types with "Generic Universe Types" by Dietl, Drossopoulou, and Müller [29] as well as in "Separating ownership topology and encapsulation with Generic Universe Types" with the same authors [30]. Dietl et al. describes the workings of

Universe Types with generics, and unlike OGJ where the ownership information and generic information are integrated together, Generic Universe Types separate the ownership information from the generic information without having separate parameters for ownership.

2.3.5 Effect Systems

The concept of an effect system is first introduced by Lucassen and Gifford in the paper ”Polymorphic effect systems” [57]. The effects for an object include the reading and writing of the object's mutable state. An effect system, like the one presented in ”An object oriented effect system“ by Greenhouse and Boyland [38], is an approach to infer the effects of the computation of a program, which allows changes to be made to a program if they do not change the behaviour of that program. Managing the order of computation, and understanding when two computations do not interfere with each other is crucial, especially when programmers wants to manipulate the order of execution for their programs. Ownership type systems have been developed that incorporate effect systems, e.g. ”Ownership, encapsulation and the disjointness of type and effect“ by Clarke and Drossopoulou [25], and ”Object invariants and effects“ by Potter and Lu [56].

The owner of an object in most ownership type systems cannot be transferred between objects, which can hinder features such as object copying and concurrent programming [26]. The papers ”External uniqueness is unique enough“ by Clarke and Wrigstad [23], ”Aliasing burying: Unique variables without destructive reads“ by Boyland [9], and ”Ownership transfer in universe types“ by Müller and Rudich [66], present different ways to transfer objects between owners. Wrigstad et al. explain that a reference to an object is externally unique if it is the only reference directed to the object that is outside the representation of the object. The external uniqueness system uses virtual owners as bounds on the outwards movements

of unique references. This means unique references can be changed inside the scope of the virtual owner. The virtual owners allow the transfer of the external unique references of an object, by ensuring the object being transferred has no references back to where it originated.

2.4 Multiple Ownership

Ever since the initial ownership systems [26] ownership types have continued incorporating many features such as generic types, effect systems, and ownership transfers. Previous limitations of ownership types, like the inability to implement iterators, have also been resolved. While each new feature increases the capabilities of ownership types, ownership type systems are still limited by allowing objects to only have a single owner. In this section, we will discuss an ownership type system that allows objects to have multiple owners.

Single ownership type systems produce tree structured heaps, however, this is not suitable for many common programs and design patterns. In "The runtime structure of object ownership" by Mitchell [62], Mitchell summarizes that the heap of commonly used programs has one of the four owner graph structures: halos, unique ownership, shared ownership and butterflies. Only shared ownership occurs regularly within dominator trees, which accounts for over 75 percent of the object reference graphs. A shared ownership structure is when an object is the root of a dominator forest but is not part of the representation of the forest, hence the responsibility for the forest is controlled by a single object. Potanin et al. [76] show further evidence that multiple ownership is essential. Potanin et al. demonstrate that object-oriented programs are scale-free and the size of these programs follow the power-law. As the size of these programs grow, the relationships between objects become more complex. A single ownership system is too restrictive to model these programs.

"Multiple Ownership" by Cameron, Drossopoulou, Noble, and Smith

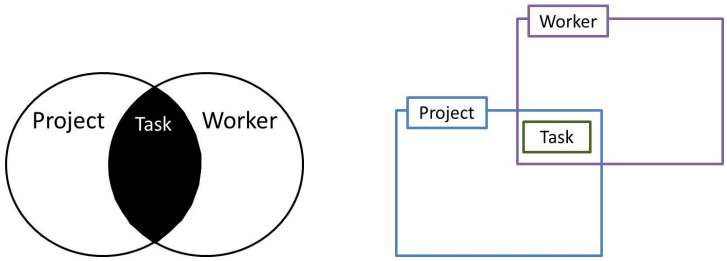


Figure 2.5: (Left) A Venn diagram of the set of Projects, Workers and Tasks. (Right) An objects-in-boxes model of the relationship in the Venn diagram.

[16] introduces multiple ownership and MOJO, a multiple ownership calculus. Cameron et al. model their ownership system using the objects-in-boxes model. The idea to use boxes to model objects is not new, and boxes can easily be replaced with more familiar terms like contexts, universes and domains.

In Fig. 2.5, the image on the left is a Venn diagram with a set of workers, a set of projects and the intersection between the two sets. The intersection is a set of tasks that is part of any project in the set of projects and worked on by any worker in the set of workers. The image on the right, in Fig. 2.5, is an objects-in-boxes model of the Venn diagram. The Project object and the Worker object co-own the task object. The task object is in the context of the intersection of the Project object and the Worker object. This is shown in Fig. 2.5 by the set of tasks being in the intersection of the set of projects and the set of workers. The similarities between set theory and the objects-in-boxes model is not a coincidence and is one of the main reasons multiple ownership is modeled with the objects-in-boxes model.

Fig. 2.6 presents an objects-in-boxes model of a single ownership structure. In this example there are three kinds of objects, the `Worker`, the `Project` and the `Task`. It is important to understand how the ownership relationships between these objects are developed. The objects $\texttt{Project}_1$ and $\texttt{Project}_2$ are inside the `Worker` object, or in this case the `Worker`

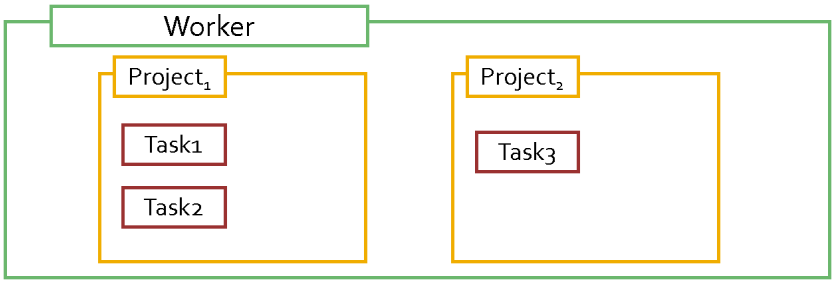


Figure 2.6: Single ownership example in the objects-in-boxes model.

box, symbolizes that the Worker owns both of these objects. Similarly, $\texttt{Project}_1$ owns the objects $\texttt{Task}_1$ and $\texttt{Task}_2$, while $\texttt{Project}_2$ owns $\texttt{Task}_3$. Conversely the owner of $\texttt{Task}_3$ is $\texttt{Project}_2$ and so on. A box is basically a set of objects: a box can hold other boxes and a box is the owner of all the objects it holds.

In Fig. 2.7, we present another example that builds on Fig. 2.6. The workers and projects are no longer in a hierarchical structure. Multiple workers are now able to collaborate on the same task, and multiple workers can also work on different projects at the same time.

The topology of Fig. 2.7 cannot be described by any ownership type systems that enforce a prescriptive ownership policy, such as owners-as-dominators or owners-as-modifiers. The system described in this section does not enforce any encapsulation upon the ownership, instead it is purely descriptive. Boxes as sets provide a simpler conceptual model of multiple ownership: similar to how elements can be in more than one set, objects can be in more than one box. If an object is in more than one box, then it has more than one owner, and the owners of that object are the boxes which contain that object. In Fig. 2.7, each `Task` object has two owners, a `Project` object and a `Worker` object. The $\texttt{Task}_1$ object is in the boxes of $\texttt{Project}_1$ and $\texttt{Worker}_1$, which means $\texttt{Task}_1$ is in the intersection of $\texttt{Project}_1$ and $\texttt{Worker}_1$, hence $\texttt{Task}_1 \in [\texttt{Project}_1] \cap [\texttt{Worker}_1]$.

MOJO (Multiple-Ownership-for-Java-like-Objects) is a multiple own-

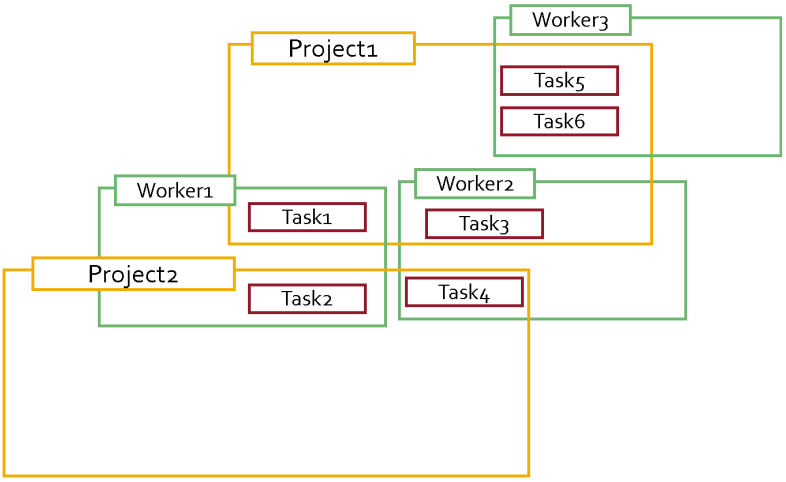


Figure 2.7: Multiple ownership example in the objects-in-boxes model.

ership system [16]. It is surprisingly similar to previously discussed single ownership systems. The ownership parameters in MOJO are declared in class instantiation, which makes MOJO type parametric with a single parameter space. Below we present the example of $\texttt{task}_1$ being owned by $\texttt{project}_1$ and $\texttt{worker}_1$, as shown in Fig. 2.7:

```
Project<this> project1 = new Project<this>();
Worker<this> worker1 = new Worker<this>();

Task<project1 & worker1> task1
              = new Task<project1 & worker1>();
```

Creating objects in MOJO follows the convention that the first parameter of the type declaration is the owner of the object. As well as introducing multiple ownership, MOJO also introduces wildcard owners, effect system, and basic constraints over contexts. The wildcard owners feature in MOJO is similar to the wildcard type feature in generic types of Java, with the difference that wildcard owners parameterise over contexts instead of types. In MOJO, `?` is a wildcard owner and it means an unknown owner,

which is similar to an unknown type in Java wildcards and `any` in Generic Universe Types. When a `?` is placed in the ownership parameter of an object, we can assume that the object has an owner, but we cannot determine who the owner is. The motivation for wildcard owners is to increase the expressiveness of MOJO by allowing types like:

```
Project<this> project1 = new Project<this>();
Worker<this> worker1 = new Worker<this>();

Task<project1 & ?> taskA;
Task<? & worker1> taskB;
```

The type of the object `taskA` is `Task<project1 & ?>`, indicating that the owners of the object `taskA` are the object `project1` and an unknown owner. Similarly the object `taskB` has the object `worker1` and an unknown object as its owners. The type `Task<project1 & worker1>` is a subtype of `Task<project1 & ?>` and `Task<? & worker1>`. The effects system of MOJO is far more conservative than single ownership systems, as for many objects there is simply not enough information from the multiple owners of the type to give an exact effect to the object or method.

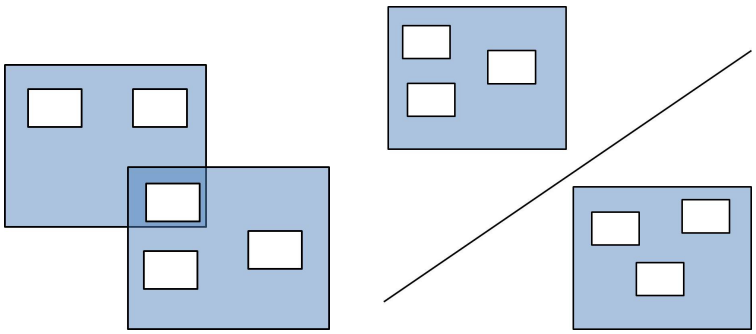


Figure 2.8: (Left) Joint boxes. (Right) Disjoint boxes

In MOJO's class declarations, the ownership contexts can be constrained in two ways. The `intersects` constraint guarantees that two boxes intersect, and the `disjoint` constraints guarantee that two boxes must be disjoint.

The type system of MOJO is constructed in a similar manner to the single ownership systems presented in section 2.3. MOJO has several limitations. MOJO does not enforce any ownership encapsulation policy, like owners-as-dominators or the owners-as-modifiers. MOJO only characterizes the topology of the heap, which means MOJO does not ensure the correctness of ownership in the traditional sense [26]. Below we present a definition for a list where the context of the list is not parametric, showing that MOJO requires two implementations of the list.

```
class Worker<o>{
 TaskList<this, this> taskList;
 VariantTaskList<this, this> varitaskList;
}

class TaskList<o, r0>{
 Task<r0> task;           //Single Owner
 TaskList<o, r0> next;
}

class VariantTaskList<o, r0>{
 Task<r0 & ?> task;       //Variant Owners
 VariantTaskList<o, r0> next;
}
```

2.5 Object Cloning

Cloning an object in many object-oriented languages either performs shallow cloning, where a single object is copied, or deep cloning, where that

object and every object it can (transitively) refer to are copied. In this section, we discuss previous work conducted on object cloning, different object cloning techniques, and the differences between cloning approaches guided by the object structure described by ownership type and cloning approaches guided by reference structures.

2.5.1 Copying and Cloning

One of the earliest papers to address issues surrounding cloning is "Copying, Sharing, and Aliasing" by Grogono and Chalin [39]. Grogono et al. explain how there is confusion within programming languages between the semantics and implementations of copying. The paper goes on to describe that it is more important to know if the object you are copying is immutable or mutable than it is if you are copying a value or a reference. The paper then goes on to talk about the importance of containing and sharing objects, which later became known as representation objects and argument objects. Finally the paper concludes with ways to implement these principles, suggesting copying should be guided more by effect systems.

Grogono and Sakkinen [40] 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.

2.5.2 Shallow Cloning

Shallow cloning is one of the most common form of object cloning found in object oriented programming languages. Shallow cloning an object would copy the object and alias the references of that object.

In Fig. 2.9, we present three diagrams to demonstrate the workings of

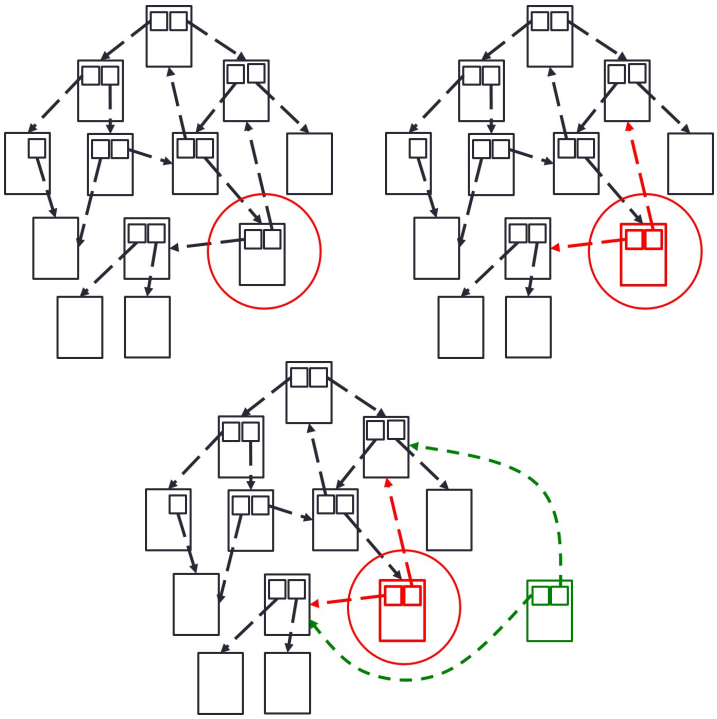


Figure 2.9: Shallow cloning.

shallow cloning. In each diagram the rectangles represents objects, the squares inside each rectangle are the fields of that object, and the arrow from the squares represents the reference of that field. Each diagram represents the heap of the program. The diagram on the top left is the initial state of the heap before shallow cloning. The object inside the red circle is the object to be shallow cloned. The diagram on the top right shows the objects and references that will be copied by highlighting them in red. In diagram on the bottom, the object structure highlighted in green is the shallow clone of the object inside the red circle.

Shallow clones are not perfect. There are situations when it is not ideal for an object and its clones to share parts of their object structure. The example in Fig. 2.10 demonstrates how the shallow clone of a `displayWindow` object with a reference to a `scrollBar` object can be an inadequate clone.

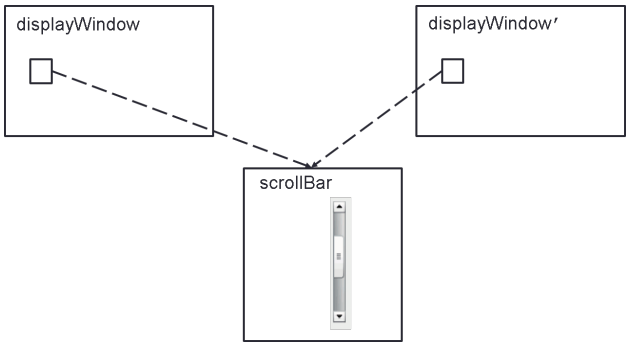


Figure 2.10: Shallow cloning displayWindow.

The `displayWindow'` object is the shallow clone of the `displayWindow` object, and it contains an aliases to the `scrollBar` object of `displayWindow`. Any mutation to the `scrollBar` object is observed by both `displayWindow` or `displayWindow'`, therefore the `scrollBar` of a particular display window could be unintentionally changed by another display window.

2.5.3 Deep Cloning

A common alternative to shallow cloning is deep cloning. Deep cloning an object would copy that object and every object transitively reachable from that object.

In Fig. 2.11, we demonstrate the deep cloning of the same object used to demonstrate shallow cloning in Fig. 2.9. The diagram on the top left shows the original heap, displayed in black, and the object to be deep cloned, inside the red circle. The object structure highlighted in red, in the diagram on the top right, is the object structure that will be copied during deep cloning. The two diagrams on the bottom are the heap after deep cloning. The object structure in red is the original heap, while the object structure in green is the deep clone created from deep cloning.

Deep cloning is also not without its flaws. Performing deep cloning can be very expensive in time and memory. Furthermore, there are cases

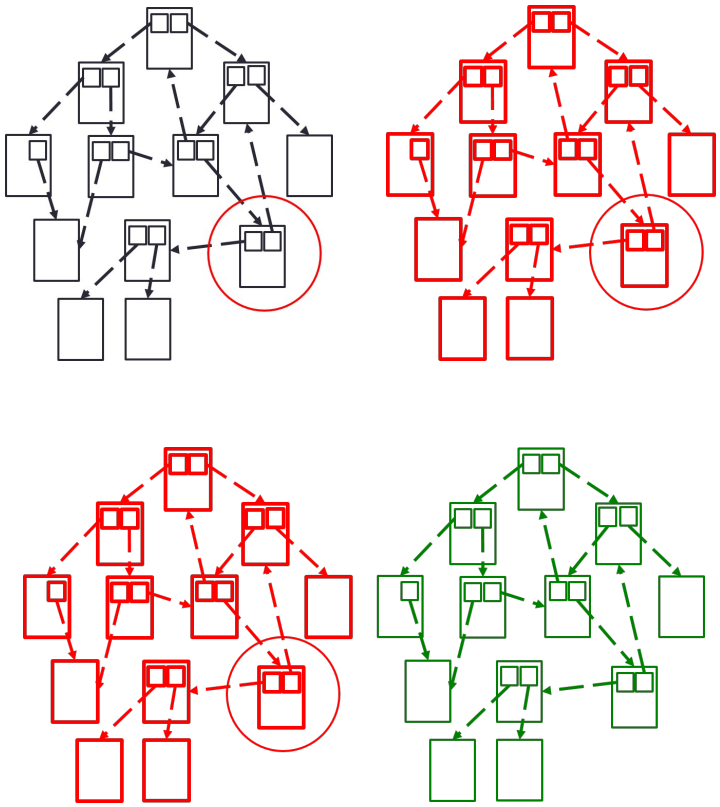


Figure 2.11: Deep cloning.

whereby having a replica of the cloned object's object structure can causes the deep clone to not satisfy its intended purpose or create unforeseen issues. In Fig. 2.12 we demonstrates the issue of deep cloning copying too much, or being too deep, with a simply example. The example presents the deep cloning of a `displayWindow` object that references an `imageDatabase` object containing the images displayed by the `displayWindow` object. The `displayWindow'` object is the deep clone of the `displayWindow` object. The `displayWindow'` object displays its own collections of images from a new image data (`imageDatabase'`) that is a replica of the `imageDatabase` object. Copying the image database is excessive. It would also give rise to situations where changes to an image in one of

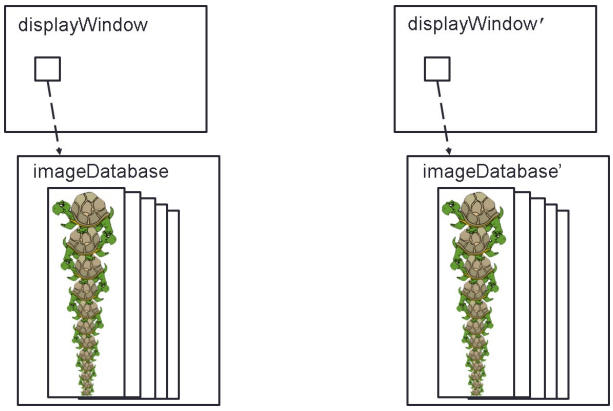


Figure 2.12: Deep cloning displayWindow.

the display window would not be reflected in the corresponding image shown in the other display window.

The deep cloning of objects that use the flyweight pattern is another example where deep cloning is too deep. The flyweight pattern aims to minimise memory usage by having similar objects sharing their data. For example, a `Character` object represents each individual character in a document. The `Character` object maintains the front, size, colour, and location of that character. Some of these attributes for many of the `Character` objects in a document are known, fixed and the same. The flyweight pattern is commonly used for the `Character` object to reference a shared object containing these attributes, instead of having each `Character` object containing their own. Deep cloning a `Character` object would create a copy of the attributes object, violating the principles of the flyweight pattern. Furthermore, if a collection of characters are individually deep cloned then each individual clone would have their own attributes object, undoing the benefits provided by the flyweight pattern.

2.5.4 Sheep Cloning

The inception for an ownership guided cloning was introduced in "Object Ownership for Dynamic Alias Protection" by Noble, Clarke and Potter [68]. Noble et al. coined the term "Sheep cloning" for an object cloning procedure utilising ownership types. Sheep cloning an object copies the objects owned (transitively) by the object being cloned, while all objects outside the representation of the object being cloned are aliased. When comparing sheep cloning against shallow and deep cloning, sheep cloning is far superior. Sheep cloning is more efficient in terms of both time and memory than deep cloning. The clones created from sheep cloning contains object structures that are richer than those in shallow clones. Noble et al. describe limitations of sheep cloning, such as the existence of a cycle within the object graph may hinder the feasibility of sheep clones. Noble et al. does not describe a formal model of sheep cloning, nor has sheep cloning gone through much development since its conception.

In Fig. 2.13, we present several diagrams to demonstrate the workings of sheep cloning. These diagrams are similar to those presented in Fig. 2.9 and Fig. 2.11 that demonstrate shallow and deep cloning respectively, with the addition of a dotted box that encloses a set of objects. The dotted box represents the ownership representation, also known as the context, of the object on its top side edge. The objects inside a dotted box are owned by the object on the top side edge of that dotted box. The diagram on the top left is the state of the heap before sheep cloning. The object inside the red circle is the object being sheep cloned. In the diagram on the top right, the objects and references highlighted in red will be copied during sheep cloning. The objects copied are also the objects inside the representation of the object being cloned (the object in the red circle). In the diagram on the bottom, the object structure highlighted in green is the sheep clone of the object in the red circle. The sheep clone has a copy of the representation of the object being cloned, and an alias for every reference going out of the representation of the object being cloned.

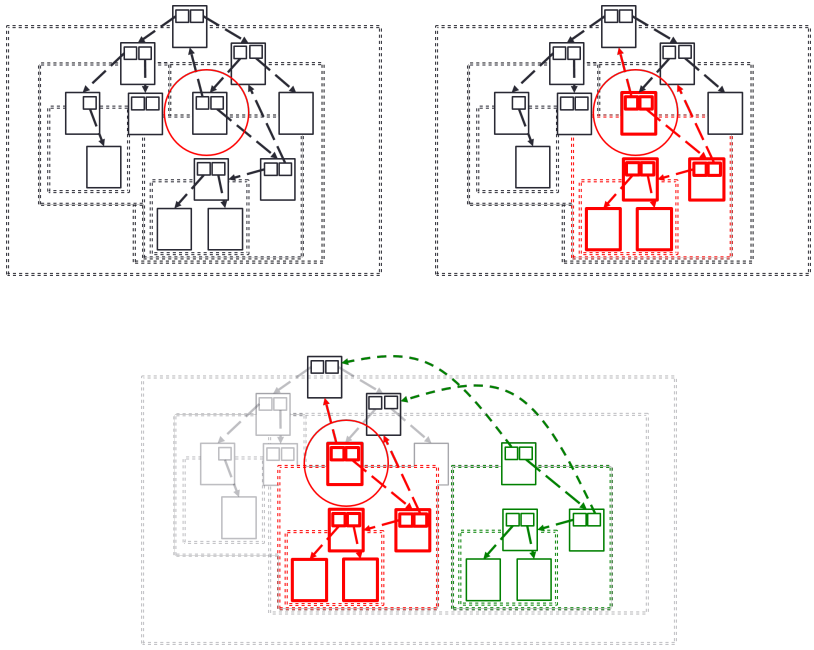


Figure 2.13: Sheep cloning.

The ability for sheep cloning to use ownership types to distinguish between copying and aliasing prevents sheep cloning from being too deep and being too shallow. Sheep cloning needs to be validated in order for it to be the default object cloning implementation of programming language. In this thesis, we have constructed several formal models of sheep cloning, as well as proven soundness for one of these models.

Rust

The `Rust` [37] programming language is designed to emphasise safety, in particular memory safety without the presence of garbage collection, and one of the ways `Rust` achieves this is through ownership types. `Rust` uses `box` to allocate data and represents the abstraction of memory within the heap.

Ownership types are enforced in Rust through external uniqueness [23]. There are two kinds of pointers in Rust: owned pointer and borrowed reference (declared with &). Pointer with the type Box<T> is an owned pointer pointing to a value (box) of type T allocated in the heap. The declared variable of the owned pointer takes ownership of the box. The ownership invariant of Rust ensures the uniqueness of the owned pointer, hence only one owned pointer can point to a box at any time.

```
let y : Box<i32> = Box::new(5i32);
```

Figure 2.14: Owned pointer.

The Rust code in Fig. 2.14 is an example of variable y with a box type of 32 bit integer (Box<i32>). The variable y is assigned five as a 32 bit integer.

Data sharing and pass-by-reference on functions can be achieved in Rust with borrow references. References with the borrowed pointer & operator can temporary reference a box, and are known as borrowed references. A borrowed reference cannot change the ownership of the box. There are several restrictions on the owner pointer that points to a box when there are borrowed pointers that are also pointing to the box. The main restriction on owned pointer is that it cannot deallocate the box or change the type of the box.

```
fn borrow(r : &i32) -> i32{

 *r;
}
```

Figure 2.15: Borrow reference function.

The Rust code in Fig. 2.15 is an example of a pass-by-reference function. The borrow function takes a borrowed reference to 32 bit integer and returns the owned pointer of the box. The operator dereferences a pointer. There is no restriction on the number of borrowed pointers pointing to a box, however, a box can only ever have a single owned pointer pointing to it. Rust uses move semantics to change the ownership of a box.

```
let y : Box<i32> = Box::new(5i32);
let mut x = y;
*x = 4i32;
println!("old value {}", y);
             //error: use of moved value: `y` [E0382]
```

Figure 2.16: Borrow reference function.

The Rust code in Fig. 2.16 is an example of the move semantics in Rust. The variable y is assigned a pointer to the value 5, in 32 bit integer, allocated on the heap. The assignment operator = moves the ownership of the box in y to the mutable variable x. The box can be changed because x is a mutable variable. When the variable y attempts to access the box an error will arise because the box has been moved to x.

The Clone trait in Rust is very similar to sheep cloning. When the clone() function is called on an object, its borrowed references are aliased while its owned pointers are copied.

The Rust code in Fig. 2.17 is an example of the Clone trait. The example is the cloning of a pair object. The pair object contains an owned pointer (first) and a borrowed reference (second), both with the type i32. The variable cloned_pair is a clone of the pair object. The second variable of the pair object and the clone_pair object is referencing the same box. While the first variable of the pair object and the clone_pair object is unique to them. It is important to note there is a Copy trait

```
#[derive(Clone)]
struct Pair<'a>(Box<i32>, &'a i32);

fn main() {

 let first = Box::new(1);
 let second = 2;
 let pair = Pair(first, &second);
 let cloned_pair = pair.clone();

}
```

Figure 2.17: Clone trait.

which is separate to the `Clone` trait. The `Copy` trait performs a bit-wise copy (i.e. memcpy) of the item being copied.

Chapter 3

The Foundation of Sheep Cloning

This chapter presents `Core`, a formalism of the type system for a class-based object-oriented programming language with deep ownership. `Core` is created in the style of Featherweight Java [46]. `Core` is the foundation for our formalisms of sheep cloning. The purpose of `Core` is to capture the common parts of the different semantic models, which avoids presenting repetitions of the same formalism in each in later chapters . Type soundness has not been shown for `Core`, however, the sheep cloning formalisms of the following chapters are extensions of `Core`, and we have either proved or outlined the proof of type soundness for each of the sheep cloning formalisms.

The design philosophy for `Core` is to be a simply object oriented language with ownership types that serves as the basis for a formalism featuring sheep cloning. `Core` consists of a static (compile-time) system and a dynamic (runtime) system. The static system of this formalism can be used by programmers to write programs. The dynamic system of this formalism cannot be used by programmers; it is used to model the execution of the system written by the static system. `Core` contains all the components required in a formalism of sheep cloning except the sheep cloning semantics, which is presented in the following chapters.

3.1 Syntax

Q	::=	`class C<`$\overline{\texttt{o}_l \preceq \texttt{x} \preceq \texttt{o}_u}$`>` $\{\overline{N\,\texttt{f;}}\ \overline{M}\}$	*class declaration*
M	::=	N `m(`N `x)` $\{$`return` e`;}`	*method declaration*
N	::=	`o:C<`$\overline{\texttt{o}}$`>`	*class type*
o	::=	γ | `world` | `owner`	*owners*
e	::=	`null` | γ | γ`.f` | γ`.f =` e | γ`.m(`e`)`	*expressions*
		| `new o:C<`$\overline{\texttt{o}}$`>` | v	
v	::=	`null` | ι	*values*
γ	::=	`x` | `this` | ι	*expression variables and addresses*
Γ	::=	$\overline{\gamma\texttt{:}N}, \overline{\texttt{o:}\top}$	*variable environments*
$\mathcal{E}$	::=	$\overline{\texttt{o} \preceq \texttt{o}}$	*context environments*
$\mathcal{H}$	::=	$\overline{\iota \rightarrow \{N,\ \overline{\texttt{f}\rightarrow v}\}}$	*heaps*
$\texttt{x} \preceq \texttt{o}$			*inside relation*
x			*variables*
ι			*object address*
`err`			*errors*
`null`			*null expression*
f			*field names*
m			*method names*
C			*class names*

Figure 3.1: Syntax.

Fig. 3.1 presents the syntax for `Core`, and it contains the syntax for the static and dynamic systems. The syntax in grey, apart from the `world`

parameter, is used exclusively in the dynamic system.

A class declaration in Core contains a class header and a class body. A class header consists of a sequence of owner parameters with lower (o_l) and upper (o_u) bounds for each parameter. These bounds are used to enforce owners-as-dominators and to prevent class declarations that cannot be instantiated. In the static system, the owner parameters are formal variables (x). In the dynamic system the variables are replaced by the actual objects (ι) they represent. The body of a class has a sequence of field names and a sequence of method declarations.

A method declaration contains a method header and a method body. A method header contains the return type of that method, the argument of the method, and the type of the argument, while the method body contains the return statement of the method.

The owner parameters for Core are: world, owner, this, and x. The world parameter is the unique owner at the top of the ownership hierarchy, and can be used in both the static and dynamic system. Objects with world as their owner are accessible by every object in the system. The owner parameter refers to the owner of the object that will be instantiated by the class where owner is used; therefore, objects owned by owner have the same owner as the object that instantiated them. The this parameter denotes the current object, and objects owned by this are owned by the object that instantiated them. In the dynamic system both the owner and this parameters are substituted by the actual objects they represent.

The syntax for types (N) consists of three components: the class name (C), the owner of the type (o), and a sequence of owner parameters ($\overline{o}$). The $\overline{o}$ are the owner parameters required for the header of class C.

The recursively defined expressions (e) of Core consist of: field look-up, field assignment, method invocation, and object creation. The field look-up expression (γ.f) is an expression variable (γ) with a field name (f). The field assignment expression (γ.f = e) assigns an expression (e) to a field look-up expression. The method invocation expression (γ.m(e))

has the expression variable (γ) invoking the method, the method name (m), and an expression (e), the argument for the method invocation. The object creation expression (new o:C<$\overline{o}$>) consists of the new keyword and the type of the object being created.

The remaining expressions are: values (v), expression variables (γ), and null. A value is either a null or an object address (ι). An expression variable is either the variable this, a general variable (x), or an object address (ι). An object address exists only in the dynamic system.

The inside relation ($x \preceq o$) is used to describe the ownership hierarchy of the program. The inside relation states whether an object is inside another object. There are separate inside relation judgments for the dynamic system and the static system.

Judgments in the static system are decided under the environments Γ and $\mathcal{E}$. The Γ environment contains a set of free variables and owner parameters mapped to their types. Owner parameters are initialised with the top type ($\top$). The $\mathcal{E}$ environment contains every inside relation in the static system, which are mostly the lower and upper bounds from the class declarations of the system.

Judgments in the dynamic system are decided under the heap ($\mathcal{H}$). The heap is a set of mapping of object addresses to a pair that represents the object. The pair contains the type of the object and a mapping from the fields of the object to their values.

Errors (err) in the system are used in the semantics when an expression is dereferencing a null. A program in Core is an expression and a set of class declarations. Execution of a program reduces the expression to a value.

3.2 Auxiliary functions

The auxiliary functions of Core are presented in Fig. 3.2. The $own_{\mathcal{H}}$ function returns the owner of an object address or a type. When given an object

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

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

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

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

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

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

Figure 3.2: Auxiliary functions.

address (ι), the $own_{\mathcal{H}}$ function uses the heap ($\mathcal{H}$) in its subscript to retrieve the owner of ι.

The *fields* function returns the fields of a class (C). The *fType* function takes a field (f_i) and the type (o:C<$\overline{o}$>) for the object that contains f_i, and returns the type of f_i.

The *mBody* function takes a method name (m) and the type (o:C<$\overline{o}$>) for the object that contains the method m, and returns the argument of the method and the expression in the body of the method. The method declaration of method m is retrieved with the class declaration on class C.

The *mType* function returns a mapping from a method's argument type to that method's return type. The *mType* function takes a method name (m) and the type (o:C<$\overline{o}$>) for the object that contains the method m. The argument type and return type of the method m are retrieved from the method header of method m, and the method declaration is retrieved from the class declaration of class C.

The type from the *fType* function, along with the pair from the *mBody* function, and the mapping from the *mType* function all require two possible substitutions. The first substitution is for the owner parameter owner in the type of the field or the return expression. The owner parameter is substituted for the owner of the object that contains the field or method. The second substitution is for the formal owner parameters (x) in type of the field or return expression of a method. The formal owner parameters are substituted for the actual owner parameters in the type of the object that contains that field or method.

3.3 Well-Formedness Judgments

Every judgment in the static system is judged under the environments Γ and $\mathcal{E}$. The class declaration judgment initialises both environments. The Γ environment is initialised with the formal owner parameters ($\overline{x}$) of the class with the top type ($\top$), and the owner parameter this with the type

`owner:C<`$\overline{\texttt{x}}$`>`. The $\mathcal{E}$ environment is initialised with the lower and upper bounds of the class, and the inside relation `owner` $\preceq$ `world`.

The judgments for well-formedness of class and method declarations are presented in Fig. 3.3.

Well-formed class: $\boxed{\vdash \texttt{class C<}\overline{o_l \preceq \texttt{x} \preceq o_u}\texttt{>} \ \{\overline{N\ \texttt{f;}}\ \ \overline{M}\}\ \text{OK}}$

$$\frac{\begin{array}{c}\Gamma = \texttt{this : owner : C<}\overline{\texttt{x}}\texttt{>},\ \overline{\texttt{x} : \top} \\ \mathcal{E} = \overline{o_l \preceq \texttt{x}},\ \overline{\texttt{x} \preceq o_u},\ \texttt{owner} \preceq \texttt{world} \qquad \mathcal{E};\Gamma \vdash \overline{N}, \overline{M}\ \text{OK} \\ \mathcal{E};\Gamma \vdash \overline{o_l}, \overline{o_u}\ \text{OK} \qquad \forall o_l \in \overline{o_l} : \mathcal{E};\Gamma \vdash \texttt{owner} \preceq o_l\end{array}}{\vdash \texttt{class C<}\overline{o_l \preceq \texttt{x} \preceq o_u}\texttt{>} \ \{\overline{N\ \texttt{f;}}\ \ \overline{M}\}\ \text{OK}}$$

(T-CLASS)

Well-formed method: $\boxed{\mathcal{E};\Gamma \vdash N\ \texttt{m(}N\ \texttt{x)}\ \ \{\texttt{return}\ \ e\texttt{;}\}\ \text{OK}}$

$$\frac{\Gamma' = \Gamma, \texttt{x} : N' \qquad \mathcal{E};\Gamma' \vdash N, N'\ \text{OK} \qquad \mathcal{E};\Gamma' \vdash e : N}{\mathcal{E};\Gamma \vdash N\ \texttt{m(}N'\ \texttt{x)}\ \ \{\texttt{return}\ \ e\texttt{;}\}\ \text{OK}}$$

(T-METHOD)

Figure 3.3: Well-formedness rules for classes and methods.

The T-CLASS judgment defines class declaration well-formedness. A class declaration is well formed if every method and the types of every field in the class are well-formed; each lower and upper bound of the class header is well-formed; and (in red) the owner parameter `owner` of this particular class is inside every lower bound of the class.

The T-METHOD judgment defines well-formedness for method declarations. A method declaration is well formed if the return type and the argument's type are well-formed; and the expression in the method body has the type of the method's return type. T-METHOD updates the Γ environment with the method's argument and its type.

The well-formedness judgments for owner parameters and types are presented in Fig. 3.4.

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

$$\frac{\gamma \in \mathtt{dom}(\Gamma)}{\mathcal{E};\Gamma \vdash \gamma\ \text{OK}}\ (\text{F-VAR}) \qquad \frac{}{\mathcal{E};\Gamma \vdash \mathtt{world}\ \text{OK}}\ (\text{F-WORLD})$$

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

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

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

Figure 3.4: Well-formed contexts and types.

The F-VAR judgment states that an expression variable is well-formed if it exists in the environment Γ. In the static system, the `world`, `owner`, and `this` owners are always well-formed.

The F-CLASS judgment states well-formedness for types ($\mathtt{o}:\mathtt{C{<}}\overline{\mathtt{o}}\mathtt{{>}}$). The premise of F-CLASS states that class `C` must be a valid class declaration; the owner ($\mathtt{o}$) and owner parameters ($\overline{\mathtt{o}}$) must be well-formed; the upper and

lower bounds of class C must be valid inside relations when the formal owner parameters ($\overline{\mathtt{x}}$) are substituted for the actual owner parameters ($\overline{\mathtt{o}}$).

Heap well-formedness is defined by the judgments F-HEAP and F-HEAPE in Fig. 3.5.

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

$$\frac{\mathcal{H} \vdash \mathcal{H}\ \text{OK}}{\vdash \mathcal{H}\ \text{OK}}$$

(F-HEAPE)

$$\frac{\begin{array}{cc}\mathcal{H}' \subseteq \mathcal{H} & \forall\, \iota \to \{N; \overline{\mathtt{f} \to v}\} \in \mathcal{H}' : (\mathcal{H} \vdash N\ \text{OK} \\ \overline{fType(\mathtt{f}, N) = N'} & \mathcal{H} \vdash \overline{v : [\iota/\mathtt{this}]N'} \qquad \mathcal{H} \nvdash own_{\mathcal{H}}(\iota) \preceq \iota \\ \forall\, v \in \overline{v} : v \neq \mathtt{null} \Rightarrow \{v \in dom(\mathcal{H}) \wedge \mathcal{H} \vdash \iota \preceq own_{\mathcal{H}}(v)\})\end{array}}{\mathcal{H} \vdash \mathcal{H}'\ \text{OK}}$$

(F-HEAP)

Figure 3.5: Well-formed heap.

The F-HEAPE judgment is syntactic sugar for judging heap well-formedness under itself. The F-HEAP judgment states a heap is well formed if every object in the heap has a well-formed type ($\mathcal{H} \vdash N$ OK); if the fields of those objects have the type stated by the *fType* function; (in red) if the owner of those objects cannot be shown to be inside them; and if the non-null fields of those objects are in the domain of the heap and (in red) those objects are inside the owner of their fields.

3.4 Static Inside Relation

The inside relation judgments for the static system are defined in Fig. 3.6. The inside relation defines the ownership hierarchy of the system, stating when an object (o) can be inside another object (o').

The inside relations: $\boxed{\mathcal{E}; \Gamma \vdash o \preceq o}$

$$\frac{}{\mathcal{E}; \Gamma \vdash o \preceq \texttt{world}} \quad \text{(IC-World)} \qquad \frac{}{\mathcal{E}; \Gamma \vdash o \preceq o} \quad \text{(IC-Refl)} \qquad \frac{\mathcal{E}; \Gamma \vdash o \preceq o'' \quad \mathcal{E}; \Gamma \vdash o'' \preceq o'}{\mathcal{E}; \Gamma \vdash o \preceq o'} \quad \text{(IC-Trans)}$$

$$\frac{o \preceq o' \in \mathcal{E}}{\mathcal{E}; \Gamma \vdash o \preceq o'} \quad \text{(IC-Env)} \qquad \frac{}{\mathcal{E}; \Gamma \vdash \texttt{this} \preceq \texttt{owner}} \quad \text{(IC-This)}$$

Figure 3.6: Inside Relation

The IC-Env judgment states that an inside relation is valid if it is in the environment $\mathcal{E}$. The IC-This judgment describes how every object is inside their owner, by stating that the `this` parameter is always inside the `owner` parameter.

Reflexivity of the inside relation is stated in IC-Refl, while transitivity of the inside relation is stated in IC-Trans. The IC-World judgment states that every well-formed owner parameter is inside `world`.

3.5 Sub-typing

The sub-typing judgments are presented in Fig. 3.7 and describe reflexivity (`SR-Refl`), transitivity (`SR-Trans`), and the top type (`SR-Top`) of sub-typing. Sub-typing is required by the expression typing judgments and in proving subject reduction.

Sub-type relations: $\boxed{\mathcal{E};\Gamma \vdash T <: T}$

$$\frac{}{\mathcal{E};\Gamma \vdash N <: N}$$

(SR-REFL)

$$\frac{\mathcal{E};\Gamma \vdash N <: N'' \quad \mathcal{E};\Gamma \vdash N'' <: N'}{\mathcal{E};\Gamma \vdash N <: N'}$$

(SR-TRANS)

$$\frac{}{\mathcal{E};\Gamma \vdash T <: \top}$$

(SR-TOP)

Figure 3.7: Sub-type Relations.

3.6 Expression Typing

The expression typing judgments are presented in Fig. 3.8. The T-NEW judgment is typing for the object creation expression (`new o :C<`$\overline{\texttt{o}}$`>`). An object creation expression has the type of the object being created.

The T-FIELD judgment is the typing rule for the field look-up expression (γ`.f`). The expression variable (γ) containing the field has to be well typed, and the field look-up expression must have the type obtained by the auxiliary function $fType$ on the field `f`.

The T-ASSIGN judgment is typing for field assignment (γ.f = e). The expression variable (γ) containing the field f and the expression (e) being assigned to the field must be well typed. The sub-typing judgment ensures covariance over the expression being assigned. The type of the field assignment expression must be the type of the expression (e) being assigned.

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

$$\frac{\mathcal{E};\Gamma \vdash \mathtt{o} : \mathtt{C}\langle\overline{\mathtt{o}}\rangle \text{ OK}}{\mathcal{E};\Gamma \vdash \mathtt{new}\,\mathtt{o} : \mathtt{C}\langle\overline{\mathtt{o}}\rangle : \mathtt{o} : \mathtt{C}\langle\overline{\mathtt{o}}\rangle}\ (\text{T-NEW}) \qquad \frac{}{\mathcal{E};\Gamma \vdash \gamma : \Gamma(\gamma)}\ (\text{T-VAR})$$

$$\frac{\mathcal{E};\Gamma \vdash \gamma : \mathtt{o} : \mathtt{C}\langle\overline{\mathtt{o}}\rangle \quad fType(\mathtt{f}, \mathtt{o} : \mathtt{C}\langle\overline{\mathtt{o}}\rangle) = N}{\mathcal{E};\Gamma \vdash \gamma\mathtt{.f} : [\gamma/\mathtt{this}]N}\ (\text{T-FIELD}) \qquad \frac{\mathcal{E};\Gamma \vdash N \text{ OK}}{\mathcal{E};\Gamma \vdash \mathtt{null} : N}\ (\text{T-NULL})$$

$$\frac{\mathcal{E};\Gamma \vdash e : N' \quad \mathcal{E};\Gamma \vdash N' <: N \qquad \mathcal{E};\Gamma \vdash N \text{ OK}}{\mathcal{E};\Gamma \vdash e : N}\ (\text{T-SUBS})$$

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

$$\frac{\mathcal{E};\Gamma \vdash \gamma : \mathtt{o} : \mathtt{C}\langle\overline{\mathtt{o}}\rangle \quad mType(\mathtt{m}, \mathtt{o} : \mathtt{C}\langle\overline{\mathtt{o}}\rangle) = N' \to N \quad \mathcal{E};\Gamma \vdash e : [\gamma/\mathtt{this}]N'}{\mathcal{E};\Gamma \vdash \gamma\mathtt{.m}(e) : [\gamma/\mathtt{this}]N}\ (\text{T-INVK})$$

Figure 3.8: Sheep expression typing rules.

The T-INVK judgment is the typing rule for method invocation (γ.m(e)). The expression variable (γ) with the method m and the argument (e) of the method must be well typed. The sub-typing judgment ensures covariance over the argument of the method. The method invocation expression must have the type obtained by the auxiliary function $mType$ on method m, which is the type of the expression returned in the body of method m.

In the judgments for the typing of the field look up expression, the field assignment expression, and the method invocation expression, any occurrence of the owner parameter this is substituted with the actual object (γ) containing those fields or methods.

The T-VAR judgment ensures every variable has the type as defined in the environment Γ. The T-SUBS judgment describes type subsumption for each expression in the formalism. Type subsumption allows expressions to have a super type of its current type, provided the super type is well formed. The T-NULL judgment states that a null expression must have well-formed types.

3.7 Reduction Rules

The operational semantics for expressions, excluding sheep cloning (sheep(e)), are presented in Fig. 3.9.

The R-FIELD judgment describes the reduction of the field look-up expression (ι.f$_i$). The field look-up expression is reduced to the value (v_i) corresponding to the field f$_i$ in the object representation ($\{N; \overline{\texttt{f}\rightarrow v}\}$) at the object address ι of the heap ($\mathcal{H}$). No changes are made to the heap.

The R-ASSIGN judgment reduces field assignment (ι.f$_i$ = v) to the value (v) being assigned. The resulting heap ($\mathcal{H}'$) is updated with the field f$_i$ now pointing to the value v.

The R-NEW judgment reduces object creation (new o:C<$\overline{\texttt{o}}$>) to the object address (ι) of the newly created object. The new object is a fresh object with its fields ($\overline{\texttt{f}}$) obtained using the $fields$ function on class C. The result-

Expression reduction: $\boxed{e; \mathcal{H} \rightsquigarrow e'; \mathcal{H}'}$

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

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

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

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

Figure 3.9: Reduction rules.

ing heap ($\mathcal{H}'$) is updated with a mapping from ι to an object representation with the type $\mathtt{o}:\mathtt{C}\texttt{<}\overline{\mathtt{o}}\texttt{>}$ and its fields $\overline{\mathtt{f}}$ initialised to `null`.

The R-INVK judgment reduces method invocation ($\iota\texttt{.m(}v\texttt{)}$) to the expression ($e$) returned by the method `m`. The type of ι is obtained from the heap ($\mathcal{H}$), and the $mBody$ function retrieves the expression e from the method `m` with the type of ι. There are two substitutions for e: occurrences of the `this` parameter are substituted by ι, and occurrences of the argument (`x`) are substituted by the actual argument (v). No updates are made to the heap.

3.8 Propagation Rules

In Fig. 3.10, we present the expression propagation and error propagation reductions for `Core`.

$$\frac{}{\texttt{sheep(null)};\mathcal{H} \rightsquigarrow \texttt{null};\mathcal{H}}\ (\textsc{R-Sheep-Null}) \qquad \frac{}{\texttt{null.f};\mathcal{H} \rightsquigarrow \texttt{err};\mathcal{H}}\ (\textsc{R-Field-Null})$$

$$\frac{}{\texttt{null.f = } e;\mathcal{H} \rightsquigarrow \texttt{err};\mathcal{H}}\ (\textsc{R-Assign-Null}) \qquad \frac{}{\texttt{null.m(}e\texttt{)};\mathcal{H} \rightsquigarrow \texttt{err};\mathcal{H}}\ (\textsc{R-Invk-Null})$$

$$\frac{e;\mathcal{H} \rightsquigarrow e';\mathcal{H}' \qquad e' \neq \texttt{err}}{\iota\texttt{.m(}e\texttt{)};\mathcal{H} \rightsquigarrow \iota\texttt{.m(}e'\texttt{)};\mathcal{H}'}\ (\textsc{RC-Invk}) \qquad \frac{e;\mathcal{H} \rightsquigarrow e';\mathcal{H}' \qquad e' \neq \texttt{err}}{\iota\texttt{.f = } e;\mathcal{H} \rightsquigarrow \iota\texttt{.f = } e';\mathcal{H}'}\ (\textsc{RC-Assign})$$

$$\frac{e;\mathcal{H} \rightsquigarrow e';\mathcal{H}' \qquad e' \neq \texttt{err}}{\texttt{sheep(}e\texttt{)};\mathcal{H} \rightsquigarrow \texttt{sheep(}e'\texttt{)};\mathcal{H}'}\ (\textsc{RC-Sheep}) \qquad \frac{e;\mathcal{H} \rightsquigarrow \texttt{err};\mathcal{H}'}{\iota\texttt{.m(}e\texttt{)};\mathcal{H} \rightsquigarrow \texttt{err};\mathcal{H}'}\ (\textsc{RC-Invk-Err})$$

$$\frac{e;\mathcal{H} \rightsquigarrow \texttt{err};\mathcal{H}'}{\iota\texttt{.f = } e;\mathcal{H} \rightsquigarrow \texttt{err};\mathcal{H}'}\ (\textsc{RC-Assign-Err}) \qquad \frac{e;\mathcal{H} \rightsquigarrow \texttt{err};\mathcal{H}'}{\texttt{sheep(}e\texttt{)};\mathcal{H} \rightsquigarrow \texttt{err};\mathcal{H}'}\ (\textsc{RC-Sheep-Err})$$

Figure 3.10: Propagation rules.

The judgments R-SHEEP-NULL, R-FIELD-NULL, R-ASSIGN-NULL, and R-INVK-NULL describe when the field look up expression, the field assignment expression, the method invocation expression, and the sheep cloning expressions are dereferencing a `null` respectively. The judgments of R-FIELD-NULL, R-ASSIGN-NULL, and R-INVK-NULL all reduce to the `err` expression, however the R-SHEEP-NULL judgment reduces to a `null` because the sheep clone of `null` is `null`. The judgments RC-INVK, RC-ASSIGN, and RC-SHEEP describe the reduction for expression propagation of the method invocation expression, the field assignment expression, and the sheep cloning expression respectively. The judgments RC-INVK-ERR, RC-ASSIGN-ERR, and RC-SHEEP-ERR describe the reduction for error propagation of the method invocation expression, the field assignment expression, and sheep cloning respectively. Each reduction reduces to an `err` when their inner expression reduces to an `err`.

3.9 Dynamic Inside Relation

The judgments for the dynamic inside relation are presented in Fig. 3.11. The judgments in the dynamic system are judged under the heap ($\mathcal{H}$). The ownership hierarchy of the heap is determined by the dynamic inside relation. The I-REF judgment states that dynamic inside relation is reflexive. The I-WORLD judgment states that every object address is inside the `world` parameter. The I-TRANS judgment states that dynamic inside relation is transitive. Finally, the I-REC judgment states that all objects are inside their owner.

Dynamic Inside Relation $\boxed{\mathcal{H} \vdash \iota \preceq \iota'}$

$$\frac{}{\mathcal{H} \vdash \iota \preceq \iota} \quad \text{(I-REF)} \qquad \frac{}{\mathcal{H} \vdash \iota \preceq \texttt{world}} \quad \text{(I-WORLD)}$$

$$\frac{\mathcal{H} \vdash \iota \preceq \iota'' \quad \mathcal{H} \vdash \iota'' \preceq \iota'}{\mathcal{H} \vdash \iota \preceq \iota'} \quad \text{(I-TRANS)} \qquad \frac{\mathcal{H}(\iota) = \{\texttt{o:C<}\overline{\texttt{o}}\texttt{>; } \overline{\texttt{f}\rightarrow v}\}}{\mathcal{H} \vdash \iota \preceq \texttt{o}} \quad \text{(I-REC)}$$

Figure 3.11: Dynamic Inside Relation

The judgment for well-formedness of dynamic objects is presented in Fig. 3.12. An object address (ι) is well-formed if ι exists in the domain of the heap ($\mathcal{H}$).

Dynamic Object Well-formedness $\boxed{\mathcal{H} \vdash \iota \text{ OK}}$

$$\frac{\iota \in \texttt{dom}(\mathcal{H})}{\mathcal{H} \vdash \iota \text{ OK}} \quad \text{(F-ADDR)}$$

Figure 3.12: Dynamic Object Well-formedness

3.10 Sheep Cloning Syntax and Expression

In Fig. 3.13, we present the syntax and expression typing for sheep cloning.

Sheep cloning syntax:

`sheep(`e`)`	*Sheep cloning expression*

Sheep cloning expression typing:

$$\frac{\mathcal{E}; \Gamma \vdash e : N}{\mathcal{E}; \Gamma \vdash \texttt{sheep}\,(e) : N}$$

(T-SHEEP)

Figure 3.13: Syntax and typing for sheep cloning.

The sheep cloning expression consists of the `sheep` keyword and the expression of the object being sheep cloned. The T-SHEEP judgment states the typing rule for the sheep cloning expression. The sheep cloning expression has the type of the expression being sheep cloned.

The syntax and typing rules for the sheep cloning expression only differs from the shallow and deep cloning expressions in the keywords that denotes the name of the cloning procedure. The most interesting aspect of formalising sheep cloning is the reduction rule, which states the semantics for sheep cloning. In the next few chapters, we will discuss in detail formalisms that describes the semantics for sheep cloning. These formalisms are extensions of the `Core` formalism, they only describe the sheep cloning semantics and judgments required by the sheep cloning semantics, all other judgments remain as how they are stated in the `Core` formalism.

3.11 Properties of `Core`

3.11.1 Ownership Property of `Core`

The well-formedness judgment for class declaration (T-CLASS), presented in Fig. 3.3, and the heap (F-HEAP), presented in Fig. 3.5, describes properties that are essential for ownership types, owners-as-dominators, and sheep cloning.

The possible owner for an object when it is created in a class is either: `this`, `world`, `owner`, or a formal owner parameter ($\overline{\mathtt{x}}$). The T-CLASS judgment ensures every parameter in $\overline{\mathtt{x}}$ follows the naming convention of ownership types, whereby $\overline{\mathtt{x}}$ cannot contain any parameter that could create an object which would cause a cyclic ownership structure or break any prescriptive ownership policy the system follows, such as owners-as-dominators.

The formal representation of the naming convention of ownership type is the red premise $\mathcal{E}; \Gamma \vdash \overline{\mathtt{owner} \preceq \mathtt{o}_l}$ in the T-CLASS judgment. This red premise ensures the owner of the objects of this class is always inside the lower bounds of the formal owner parameters ($\overline{\mathtt{x}}$). Therefore, the formal owner parameters are guaranteed to be substituted with objects that are outside the owner of this particular object, and the fields of a class cannot be owned by objects that do not yet exist in the system or have no inside relation with the owner of the objects instantiated with that class. Most ownership systems [26, 23] use this naming convention to prevent circularity in their ownership hierarchy.

The two red premises in the F-HEAP judgment require that the heap preserves owners-as-dominators and has an acyclic ownership structure. The red premise $\mathcal{H} \nvdash own_{\mathcal{H}}(\iota) \preceq \iota$ states that for all objects in the heap it is not possible to show that any of those objects are outside their owner, and this property ensures the topology of the heap's ownership structure is acyclic. The acyclicity of the ownership structure for the heap needs to be explicitly stated as sheep cloning is a dynamic operation and the heap is

required to maintain an acyclic ownership structure after any occurrence of sheep cloning.

The red premise ($\mathcal{H} \vdash \iota \preceq own_{\mathcal{H}}(v)$) states that every object in the heap is inside the owner of each its fields. This property maintains owners-as-dominators over the heap by preventing any object in the heap from referencing objects in another object's representation.

3.11.2 Reflexivity of Inside Relation

The inside relation is critical for sheep cloning and deep ownership. The inside relation determines which objects are copied during sheep cloning, and an owners-as-dominators system requires every object to be inside the owners of the objects it is referencing.

We encountered an interesting conundrum while proving our sheep cloning formalism, should the inside relation be reflexive? It is standard for the inside relation to be reflexive, however, when considering the inside relation, as shown in Fig. 3.11, in terms of owners owning objects, reflexivity over the inside relation does not always seem as intuitive .

I-WORLD states that all objects are inside `world`, as every object is owned, directly or transitively, by `world`. I-REC states that all objects are inside their owner for every ownership system. I-TRANS states that the inside relation is transitive for the structural property of the inside relation.

There is an important distinction between the inside relation between owners, and access between objects. The inside relation between owners is transitive, the owner of an object also owns the objects owned by that object, i.e. three levels deep in the ownership hierarchy. In an owners-as-dominators system, access to an object's representation is not transitive. The owners-as-dominators prescriptive policy ensures objects can only access the objects they own, therefore, an owner cannot access the representation of objects owned by the objects it owns, i.e. objects can only access the objects one level below themselves in the ownership hierarchy.

An intuitive visualisation for ownership types is to view objects as boxes [21]. The act of owning an object is represented by placing the box that represents that object inside the box of the object that owns it. In this visualisation, the inside relation is represented by the nesting of boxes inside one and another, much like a Matryoshka doll. When considering the reflexivity of the inside relation in this model, how would an object that is inside itself be represented? A box can never be inside nor outside of itself, therefore, for a single box should it have an inside relation?

The reflexivity of the inside relation also has interesting implications over ownership structures being cyclic. Ownership structures cannot be cyclic and should never be allowed. Consider an example where object `a` is inside object `b`, then the owner of `a` should be inside `b`, unless `a` is `b`. This example can be formalised into a lemma:

Lemma 3.1.2a
if :
$\mathcal{H} \vdash \texttt{a} \preceq \texttt{b}$
$\texttt{a} \neq \texttt{b}$
then : $\mathcal{H} \vdash \texttt{own}_{\mathcal{H}}\texttt{(a)} \preceq \texttt{b}$.

Lemma 3.1.2a is proved by structural induction over the derivation of $\mathcal{H} \vdash \texttt{a} \preceq \texttt{b}$, as shown in Fig. 3.11, with cases analysis on the last step.

The reflexive (I-REF), world (I-WORLD), and owner (I-REC) case are trivial as $\texttt{a} \neq \texttt{b}$, $\mathcal{H} \vdash \texttt{a} \preceq \texttt{world}$, and $\texttt{b} = \texttt{own}_{\mathcal{H}}\texttt{(a)}$ by the premise of their respective case.

The transitive (I-TRANS) case states $\mathcal{H} \vdash \texttt{a} \preceq \texttt{c}$ and $\mathcal{H} \vdash \texttt{c} \preceq \texttt{b}$, for some `c`. The inductive hypothesis on $\mathcal{H} \vdash \texttt{a} \preceq \texttt{c}$ gives $\mathcal{H} \vdash \texttt{own}_{\mathcal{H}}\texttt{(a)} \preceq \texttt{c}$, but requires $\texttt{a} \neq \texttt{c}$. A case analysis on `c`, for when $\texttt{c} \neq \texttt{a}$ gives $\mathcal{H} \vdash \texttt{own}_{\mathcal{H}}\texttt{(a)} \preceq \texttt{b}$ with transitivity over $\mathcal{H} \vdash \texttt{c} \preceq \texttt{b}$, and when $\texttt{c} = \texttt{a}$ we get $\mathcal{H} \vdash \texttt{a} \preceq \texttt{b}$, which causes the proof to be cyclic, and therefore, we do not always have a valid argument for the transitive case of **lemma 3.1.2a**.

If the inside relation is never reflexive, i.e. if $\mathcal{H} \vdash \texttt{a} \preceq \texttt{b}$ then $\texttt{a} \neq \texttt{b}$, then a slight adjustment to the lemmas above would produce the lemma:

Lemma 3.1.2b
if :
$\mathcal{H} \vdash \texttt{a} \preceq \texttt{b}$
$\texttt{a} \neq \texttt{b}$
then : $\mathcal{H} \vdash \texttt{own}_{\mathcal{H}}(\texttt{a}) \preceq \texttt{b}$, *or* $\texttt{b} = \texttt{own}_{\mathcal{H}}(\texttt{a})$.

Lemma 3.1.2b is also proved by structural induction over $\mathcal{H} \vdash \texttt{a} \preceq \texttt{b}$. The world (I-WORLD), and owner (I-REC) cases are still trivial. For the transitive (I-TRANS) case, we get $\mathcal{H} \vdash \texttt{a} \preceq \texttt{c}$ and $\mathcal{H} \vdash \texttt{c} \preceq \texttt{b}$, for some $\texttt{c}$. The induction hypothesis without reflexivity over the inside relation gives either $\mathcal{H} \vdash \texttt{own}_{\mathcal{H}}(\texttt{a}) \preceq \texttt{c}$ or $\texttt{c} = \texttt{own}_{\mathcal{H}}(\texttt{a})$. For $\mathcal{H} \vdash \texttt{own}_{\mathcal{H}}(\texttt{a}) \preceq \texttt{c}$, we get $\mathcal{H} \vdash \texttt{own}_{\mathcal{H}}(\texttt{a}) \preceq \texttt{b}$ by transitivity on $\mathcal{H} \vdash \texttt{c} \preceq \texttt{b}$, otherwise $\mathcal{H} \vdash \texttt{own}_{\mathcal{H}}(\texttt{a}) \preceq \texttt{b}$ is given by $\texttt{c} = \texttt{own}_{\mathcal{H}}(\texttt{a})$ and $\mathcal{H} \vdash \texttt{c} \preceq \texttt{b}$.

Resolving **lemma 3.1.2a** and **lemma 3.1.2b** by making the inside relation not reflexive is a sign of poor design, as other aspects of the formalism would be greatly hindered if the inside relation were not reflexive. For example, when an object references its owner, the heap, as deemed by owners-as-dominators, requires the owner of that object to be inside that object's owner, ie. $\mathcal{H} \vdash \texttt{own}_{\mathcal{H}}(\iota) \preceq \texttt{own}_{\mathcal{H}}(\iota)$, and this judgment is extremely difficult to show if the inside relation is not reflexive by default.

3.11.3 Garbage and Reachability in Deep Ownership

The owners-as-dominators prescriptive policy and the presence of garbage collection ensures every object inside ι^* is transitively reachable from ι^*. Reachability is a property of deep ownership and garbage collection, as any dead object inside ι^* would be garbage collected. Deep ownership states that all reference paths to an object must go through that object's owner, which ensures every object inside an object is also reachable from that object.

3.12 Formalism of Object Cloning

The inspiration for formalising and understanding sheep cloning stemmed from the frustration caused by the naivety of existing object cloning idioms: shallow cloning and deep cloning, however, before formalising our semantics for object cloning, we must understand these existing cloning idioms in order to overcome their weaknesses.

Shallow cloning an object performs a field by field copy, without copying any referenced objects,where the object in that memory location is copied. Deep cloning an object performs a deep copy over the entire structure of that object. There exists a yin and yang quality between shallow cloning and deep cloning: it is common for shallow cloning to copy not enough of the object's structure, creating clones that are too shallow, while deep cloning copies too much of the object's structure, creating clones that are too deep. Sheep cloning copies the object's representation and aliases the objects that are outside the representation but referenced from within the representation.

It is possible for an object's sheep clone to be the same as that object's shallow clone or deep clone. The sheep clone of an object that does not own any other object is identical to the shallow clone of this object. Similarly, the sheep clone of an object that does not reference (directly or indirectly) any object outside its representation is identical to the deep clone of this object.

3.12.1 Shallow Cloning

In Fig. 3.14, we present the syntax, expression typing, and the reduction rule for shallow cloning.

The shallow cloning expression `shallow(e)` represents the shalow clone of the object denoted by `e`. The type of the shallow cloning expression has the type of the expression being cloned. The reduction rule for shallow cloning states that the shallow cloning expression is reduced to a

Shallow cloning syntax:

`shallow(`e`)`	*Shallow cloning expression*

Shallow cloning expression typing:

$$\frac{\mathcal{E};\Gamma \vdash e : N}{\mathcal{E};\Gamma \vdash \texttt{shallow(}e\texttt{)} : N}$$

(T-SHALLOW)

Shallow cloning reduction:

$$\frac{\begin{array}{c}\mathcal{H}(\iota) = \{N,\ \overline{\texttt{f}\rightarrow \iota}\} \\ \iota' \notin \texttt{dom}(\mathcal{H}) \\ \mathcal{H}' = \mathcal{H},\ \iota' \rightarrow \{N,\ \overline{\texttt{f}\rightarrow \iota}\}\end{array}}{\texttt{shallow(}\iota\texttt{);}\ \mathcal{H} \rightsquigarrow \iota';\ \mathcal{H}'}$$

Figure 3.14: Syntax, typing, and reduction rule for shallow cloning.

fresh object ι' that represents the shallow clone. The type and fields of ι' are a duplicate of the type and fields of ι. The original heap ($\mathcal{H}$) is updated with ι' to create the resulting heap ($\mathcal{H}'$) of the reduction.

3.12.2 Deep Cloning

In Fig. 3.15, we present the syntax, expression typing, and the reduction rule for deep cloning.

The deep cloning expression consists of the keyword `deep` and the object being deep cloned (ι). The typing of the deep cloning expression has the type of the expression being cloned. The reduction of the `deep(`ι`)`

expression uses the `deepAux` function to create the deep clones.

The `deepAux` function recursively traverses through the heap to every object reachable from ι while simultaneously copy each object. Objects are created on the way back up the traversal, and not when the traversal initially reaches an object. The `deepAux` function takes three arguments: the current object (ι_0) being traversed; the heap ($\mathcal{H}_1$) at the current stage of the traversal; a list of mappings between objects (`visited`), where the object traversed is mapped to their deep clone. The `deepAux` function returns the deep clone (ι') of ι_0, and a new heap ($\mathcal{H}'$) containing $\mathcal{H}_1$ and the deep clone of ι_0.

The `deepAux` function has three cases in total. One general case and two special cases. The general case for the `deepAux` function has seven premises. The first premise obtains the type and fields of ι_0. The second and third premise respectively ensure the deep clone (ι') of ι_0 is a fresh object and has not been cloned. The fourth premise determines the number of fields in ι_0, which is also the size of the recursive traversal required to copy ι_0. To prevent objects from being copied multiple times, looping in the traversal is prevented by maintaining `visited`. The fifth premise updates `visited` with ι_0 mapping to its deep clone ι'. The sixth premise recursively calls the `deepAux` function over every field of ι'. The final premise creates an updated heap ($\mathcal{H}'$) containing ι' and the heap ($\mathcal{H}_{n+1}$) with the deep clones for the fields of ι'.

The first special case describes when the traversal has reached an object that it has already traversed. In this case, the deep clone of the already traversed object can be retrieved from `visited`. The `deepAux` function returns the deep clone of the traversed object along with the original heap. The second special case for the `deepAux` function states that the deep clone of `null` is `null`.

Deep cloning syntax:

$\texttt{deep}(e)$ *Deep cloning expression*

Deep cloning expression typing:

$$\frac{\mathcal{E};\Gamma \vdash e : N}{\mathcal{E};\Gamma \vdash \texttt{deep}(e) : N}$$

(T-DEEP)

Deep cloning reduction:

$$\frac{}{\texttt{deep}(\iota);\ \mathcal{H} \rightsquigarrow \texttt{deepAux}(\iota,\ \mathcal{H},\ \{\})}$$

$$\frac{\begin{array}{c}\mathcal{H}_1(\iota_0) = \{N,\ \overline{\texttt{f} \rightarrow \iota}\} \\ \iota' \notin \texttt{dom}(\mathcal{H}_1) \\ \iota_0 \notin \texttt{dom}(\texttt{visited}) \\ \texttt{n} = |\overline{\texttt{f} \rightarrow \iota}| \\ \texttt{visited}' = \texttt{visited},\ \iota_0 \rightarrow \iota' \\ \forall_j : 1 \leq \texttt{j} \leq \texttt{n}: \quad \texttt{deepAux}(\iota_j,\ \mathcal{H}_j,\ \texttt{visited}') = \iota'_j;\ \mathcal{H}_{j+1} \\ \mathcal{H}' = \mathcal{H}_{n+1},\ \iota' \rightarrow \{N,\ \overline{\texttt{f} \rightarrow \iota'}\}\end{array}}{\texttt{deepAux}(\iota_0,\ \mathcal{H}_1,\ \texttt{visited}) = \iota';\ \mathcal{H}'}$$

$$\frac{\iota \in \texttt{visited}}{\texttt{deepAux}(\iota,\ \mathcal{H},\ \texttt{visited}) = \iota;\ \mathcal{H}}$$

$$\frac{}{\texttt{deepAux}(\texttt{null},\ \mathcal{H},\ \texttt{visited}) = \texttt{null};\ \mathcal{H}}$$

Figure 3.15: Syntax, typing, and reduction rule for deep cloning.

3.13 Chapter Summary

In this chapter, we present `Core`, the formalism that serves as the foundation for our sheep cloning formalisms. `Core` is formalised in the style of Featherweight Java. We discuss our observations regarding reflexivity and the inside relation. We outline a proof for type soundness of `Core`. We present the semantics for shallow cloning and deep cloning, and explain reachability for an deep ownership system.

Chapter 4

`recurSC` Formalism

In this section, we present `recurSC`, our first formalism for the semantics of sheep cloning. We have not proven type soundness for `recurSC`, but a discussion for the reasons behind its omission can be found in section 4.1. Much of the formalism described by `recurSC` is identical to `Core` formalism in chapter 3. This includes the syntax, the auxiliary functions, the judgments for expression typing, well formedness of types, owner parameters, class declarations, method declarations, and heap; sub-typing; dynamic and static inside relations; and the reduction of every expression except for the sheep cloning expression.

The sheep cloning semantics in `recurSC` is heavily influenced by the deep cloning semantics in section 3.12.2. The fundamental principle behind `recurSC` consists of two parts. The first part is to deep clone the representation of the object being cloned. The second part is to alias the objects outside the representation of the object being cloned that are directly reachable from objects inside that representation. The reduction of the sheep cloning expression for `recurSC` is presented in Fig. 4.3.

4.0.1 Sheep Cloning in recurSC

The principle idea behind the sheep cloning semantics of `recurSC` is to perform a depth-first traversal over the part of the heap that contains the ownership representation of the object being cloned. Every object reached by the traversal of the representation is copied and every object outside the representation that is reachable directly by objects inside the representation is aliased. The traversal starts at the object being cloned and backtrack once it reaches: `null`, an object outside the representation, or an object that has already been traversed.

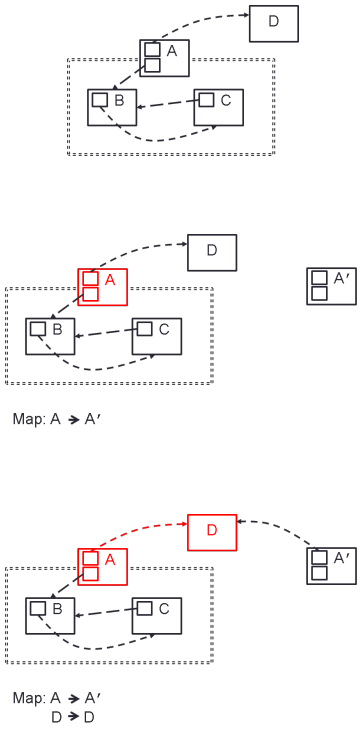


Figure 4.1: Sheep cloning in `recurSC` (part 1).

In Fig. 4.1 and Fig. 4.2, we present an example through a series of images to illustrates how the `recurSC` formalism creates sheep clones. The object being sheep cloned in the example is object `A`. Each individual image corresponds to a single step in the process of sheep cloning object `A`. The style of this example is the same as the examples presented in Fig. 2.9, Fig. 2.11, and Fig. 2.13. The rectangles represent objects, the square in a rectangle represent the field of that object, the arrow coming out of a square represent the reference of that field, and the dotted line represents the ownership representation of the object on its top edge. The example consists of four objects: `A`, `B`, `C`, and `D`. The top most image in Fig. 4.1 is show the heap before sheep cloning object `A`. Object `A` references object `D` and object `B`, as well as owning object `B` and object `C`. Object `B` references object `C`. Object `C` references object `B`.

The middle image in Fig. 4.1 shows when the depth-first traversal has reached object `A`. The objects in red indicate they have been traversed. Object `A` is copied because the reflexivity of the inside relation states that an object is always inside itself. Object `A′` is the copy of object `A`. A map is used to prevent looping in the traversal, whereby every object traversed is recorded. The map contains a set of mapping from the objects traversed to their copy otherwise to the object itself.

The bottom image in Fig. 4.1 shows when the depth-first traversal has reached object `D`. Object `D` is not copied because it is outside the representation of object `A`. An alias to object `D` is created for object `A′`, the sheep clone of object `A`. The map is updated with a mapping of object `D` to object `D`. The traversal now backtracks back to object `A` as it has reached an object that is outside the representation.

The top most image in Fig. 4.2 shows when the depth-first traversal has reached object `B`. Object `B` is copied as it is inside the representation of object `A`. The object `B′` is copy of object `B`. The map is updated with a mapping of object `B` to object `B′`. The middle image in Fig. 4.2 shows when the traversal has reached object `C`. Object `C` is copied as it is inside

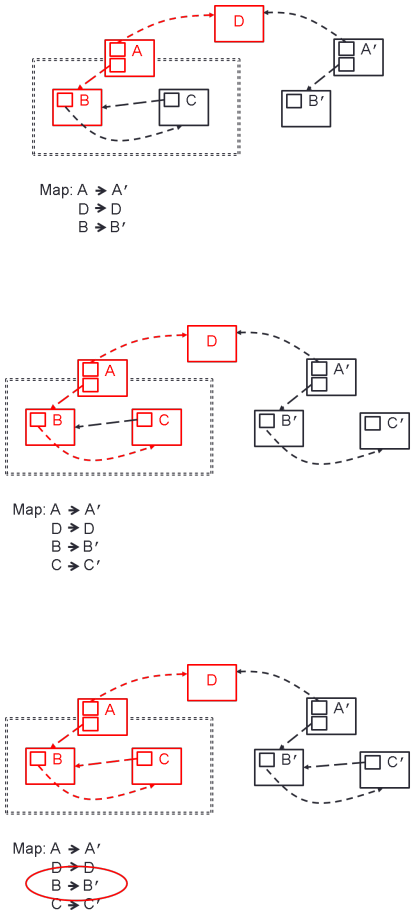


Figure 4.2: Sheep cloning in `recurSC` (part 2).

the representation of object `A`. The object `C′` is copy of object `C`. The map is updated with a mapping of object `C` to object `C′`. The bottom image in Fig. 4.2 shows when the traversal has reached object `B` again, but this time from object `C`. The sheep clone of object `B` is retrieved from the map, and an alias is created from the sheep clone of object `C` (object `C′`) to the sheep clone of object `B` (object `B′`). The traversal will finally backtrack back up to object `A` and terminate.

4.0.2 The Formalism

Sheep cloning reduction:

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

(R-SHEEP)

Figure 4.3: Sheep cloning reduction for `recurSC`.

The reduction rule in Fig. 4.3 shows how the sheep cloning expression is reduced to the sheep clone using the `SheepAux` function. The `SheepAux` function is influenced directly by the `deepAux` function in Fig. 3.15. The `SheepAux` function has the three cases of the `deepAux` function with an addition case for when the traversal reached an object outside the representation of the object being cloned. The four cases of the `SheepAux` function are: traversal case, visited case, outside case, and `null` case. The visited case is to prevent any looping in the traversal of the heap. The outside case is for when the traversal has reached an object outside the representation of the object being cloned.

The `SheepAux` function, as presented in Fig. 4.4, takes four arguments and returns a 3-tuple. The first argument is the object being cloned (ι). The second argument is the object being traversed (ι'): this object determines which case of the `SheepAux` function will be applied by its relationship with ι. The third argument is the heap ($\mathcal{H}$) at the current state of sheep cloning. The fourth argument is the `map`. The domain of the `map` contains the representation of the object being cloned as well as every object directly reachable from the representation. The range of the `map` contains the representation of that object's sheep clone, and the objects directly reachable from the sheep clone's representation.

The `SheepAux` function returns a 3-tuple: the sheep clone of ι'; an up-

dated heap; and an updated map. The heap and map are updated with the newly created sheep clone (ι'').

Fig. 4.4 presents the four cases of the SheepAux function. The R-SHEEPNULL case describes when SheepAux traverses a null, where a null is returned with no updates to the heap or the map.

The R-SHEEPREF case occurs when SheepAux traverses an object that exists in the domain of the map. This indicates the object has been traversed, and the sheep clone of this object exists in the map. The SheepAux function returns the object in the mapping of this object and no updates are made to the heap or map.

The R-SHEEPOUTSIDE case describes when SheepAux traverses an object outside the representation of the object being cloned (ι). The reference to this object will need to be copied. The SheepAux function returns this object, no update is made to the heap as no sheep clone is created. The map is updated with a mapping of this object to itself, which ensures this object is aliased as well as preventing further traversals of this object.

The R-SHEEPINSIDE case describes when SheepAux traverses an object inside ι. 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 then called recursively on every field of the traversed object, and the sheep clones of those fields are assigned to their corresponding fields in the sheep clone. The map is updated with this new object, and the sheep clones of the traversed objects' fields. Similarly, the heap is updated with this new object, and the sheep clones of the traversed objects' fields.

In Fig. 4.5, we present the mapT function that takes a map and a type (N) to create a mapped type of that type. A mapped type has its owner and owner parameters substituted for the corresponding object in the range of the map. The mapped type of an object is the type of that object's sheep clone.

Sheep Auxiliary function:

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

(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)

Figure 4.4: recurSC sheep cloning semantics

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

Figure 4.5: mapT function.

In the R-SHEEPINSIDE case of the SheepAux function, the mapT function is used to create the type of the sheep clone, and it is interesting to note that the mapT function takes map, instead of $\texttt{map}_1$, when creating the type for the sheep clone. The mapT function could take $\texttt{map}_1$ instead of map, and the same type would have been created. In an owners-as-dominators system, it is impossible for the type of an object to contain the object itself, either as the owner or an owner parameter.

4.1 Reflecting on recurSC

We found the recurSC formalism to be an inadequate formalism of sheep cloning, primarily due to the crude nature of the SheepAux function that constructs the sheep clones. The inductive case of SheepAux, R-SHEEPINside, is monolithic and complicated, exhibiting the classic software engineering mistake of a function that is doing too much.

To prove soundness for the sheep cloning semantics of the SheepAux function, the heap created by the SheepAux function is required to be shown that it is well formed. This can be shown by structural induction over the derivation of the SheepAux function, with case analysis over the last step. The difficulty with this proof is the large amount of interleaving inductive cases required for the recursive case, R-SHEEPINSIDE. An induction is required for every field of every object copied.

The map is used in R-SHEEPINSIDE to prevent looping in the traversal of the representation of the object being cloned, as well as in the mapT function to create the type of the sheep clone. An abstract representation of

the sheep clone can be created when the `map` is applied over the structure of the object being cloned.

The key problem for `recurSC` is how the `SheepAux` function constructs the representation of the sheep clone in concert with the traversal of the representation of the object being cloned, leading to a formalism that is unrefined and a proof that is tedious. To prove type soundness for `recurSC` we are eventually required to show the heap returned by the `SheepAux` function is well formed for the object that is being sheep cloned. Proving type soundness is difficult and time consuming. The `SheepAux` function recursively call itself over every field of the object passed in, and this requires a proof that consists of a numerical induction for each field in that object. The numerical induction is necessary as the ordering on the traversal of the fields must be preserved because each iterative call of the `SheepAux` function requires the heap produced by the `SheepAux` function on the previous field. The heap is only updated with the sheep clone once the `SheepAux` function has been called on every field of the object being sheep cloned.

This inspires our second attempt, where sheep clones are created using the `map` as a blueprint. Separating the traversal of the representation of the object being cloned from the construction of the sheep clone allows for a simpler formalism and an easier proof. We will discuss this new formalism in the next chapter.

4.2 Correctness Properties

In this section, we present seven correctness properties for the sheep cloning semantics of the `recurSC` formalism, and outline proofs of each property.

The **first correctness property** states a sheep clone is a fresh object, and cannot be the object it was cloned from.

Correctness property 1: Sheep cloning creates new objects.

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 for the reduction of the expression $\texttt{sheep}(\iota)$, which happens to be the function $\texttt{SheepAux}(\iota, \iota, \mathcal{H}, \emptyset)$. The three base cases R-SHEEPNULL, R-SHEepRef, and R-SHEEPOUTSIDE are all not applicable. For the recursive case R-SHEEPINSIDE, the premise states that $\mathcal{H}(\iota')$ `undefined`, therefore, $\iota' \notin \texttt{dom}(\mathcal{H})$ and from this we can deduce that $\iota \neq \iota'$ as $\iota' \notin \texttt{dom}(\mathcal{H})$ and the fact that `SheepAux` states $\iota \in \texttt{dom}(\mathcal{H})$.

The **second correctness property** states sheep clones preserve the owners-as-dominators policy:

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

For all $\mathcal{H}$, $\mathcal{H}'$, ι, *and* ι', **if** $\vdash \mathcal{H}$ OK *and* $\texttt{sheep}(\iota); \mathcal{H} \rightsquigarrow \iota'; \mathcal{H}'$ **then** $\forall \iota \mapsto \{N; \overline{\texttt{f} \mapsto v}\} \in \mathcal{H}', \forall \iota' \in \overline{v} : \mathcal{H}' \vdash \iota \preceq own_{\mathcal{H}}(\iota')$.

Proof outline: This property is proved by proving the **Theorem:** *expression reduction preserves heap well-formedness*. The **Theorem** shows the heap preserves owners-as-dominators for every expression reduction. For the sheep cloning expression we need to show each cases of the `SheepAux` function preserves owners-as-dominators over the returned heap. The R-SHEEPNULL case, the R-SHEEPREF case, and the R-SHEEPOUTSIDE case are trivial to prove. The proof for owners-as-dominators on the heap ($\mathcal{H}'$) produced by the recursive R-SHEEPINSIDE case is achieved in two parts. The first part requires the newly created clone (ι') to preserve owners-as-dominators, by showing the owner of ι' is inside the owner of the original

object, i.e., the owner of the object that was passed in the sheep cloning expression. The second part requires the fields of ι' to satisfy the owners-as-dominators property, and is shown by the transitivity of two inside relations. The first inside relation describes that the owner of the values is outside the owner of the fields they are assigned to. The second relation states that the owner of the field of ι' is outside of ι'. The transitivity of these two inside relations ensures owners-as-dominators for ι'.

The **third correctness property** states how sheep cloning creates a new sub-heap that contains the sheep clone:

> **Correctness property 3:** Sheep cloning creates a new sub-heap.
>
> *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 performing a case analysis over the `SheepAux` function, looking at the heap created by the `SheepAux` function. The three base cases of R-SHEEPNULL, R-SHEEPREF, and R-SHEEPOUTSIDE are trivial to prove as the heap remains the same after the reduction. For the recursive case R-SHEEPINSIDE, we require the **Theorem:** *SheepAux creates the same or larger heaps* that states **if** $\texttt{SheepAux}(\iota, \iota', \mathcal{H}, \texttt{map}) = \iota'', \mathcal{H}', \texttt{map}'$ **then** $\mathcal{H} \subseteq \mathcal{H}'$, where we get $\mathcal{H}_1 = \mathcal{H}, \iota' \mapsto \{...\}$, $\mathcal{H} \subseteq \mathcal{H}_1$ and $\iota' \in \texttt{dom}(\mathcal{H}_1 \setminus \mathcal{H})$. The premise of this property shows that $\mathcal{H}_1 \subseteq \mathcal{H}'$ and $\mathcal{H}_1 \setminus \mathcal{H} \subseteq \mathcal{H}''$, which means $\iota' \in \texttt{dom}(\mathcal{H}'')$.

The **fourth correctness property** states that the clone cannot reference objects in the representation of the object being cloned.

> **Correctness property 4:** Sheep cloning cannot create new references into the representation of 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* $\forall\ \texttt{f} \mapsto \iota'' \in \texttt{range}{\downarrow_2}\ (\mathcal{H}'')$ *where* $\iota'' \in \texttt{dom}(\mathcal{H})$ **then** $\mathcal{H}' \vdash \iota \preceq \iota''$.

Proof outline: This property is proved by a case analysis on how ι'' is added into $\mathcal{H}'$. The case analysis is over the construction of the sheep clone by the `SheepAux` function. The R-SHEEPNULL case and the R-SHEEPINSIDE case 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 gives $\mathcal{H}' \vdash \iota \preceq \iota''$, as $\mathcal{H} \subseteq \mathcal{H}'$. The function $\texttt{range}{\downarrow_2}\ (\mathcal{H}'')$ retrieves the second element in the range of $\mathcal{H}''$.

The **fifth correctness property** states that the clone can reference objects in the newly created heap ($\mathcal{H}''$), if and only if those objects are inside the representation of the clone.

> **Correctness property 5:** All new objects created by sheep cloning are in the representation of the clone, and all objects in the representation of the clone 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 shown in two parts, with a case analysis on the `SheepAux` function for each part. The first part shows that

$\mathcal{H}' \vdash \iota'' \preceq \iota'$ when $\iota'' \in \texttt{dom}(\mathcal{H}'')$. This part is possible in the recursive case of SheepAux, R-SHEEPINSIDE. By the premises of R-SHEEPINSIDE, we know that $\mathcal{H} \vdash \iota^* \preceq \iota$ when $\texttt{map}(\iota^*) = \iota''$. By **lemma:** *address mapping preserves inside relation*, we get $\mathcal{H} \vdash \texttt{map}(\iota^*) \preceq \texttt{map}(\iota)$, which gives $\mathcal{H} \vdash \iota'' \preceq \iota'$, and therefore $\mathcal{H}' \vdash \iota'' \preceq \iota'$, as $\mathcal{H} \subseteq \mathcal{H}'$. The second part requires us to show $\iota'' \in \texttt{dom}(\mathcal{H}'')$ when $\mathcal{H}' \vdash \iota'' \preceq \iota'$. This is also only possible for the R-SHEEPINSIDE case. Using a similar 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 $\mathcal{H}_1 \subseteq \mathcal{H}'$, and therefore, $\iota' \mapsto \{N, \overline{\texttt{f} \mapsto v}\} \in \texttt{dom}(\mathcal{H}' \setminus \mathcal{H})$, which means $\iota' \mapsto \{N, \overline{\texttt{f} \mapsto v}\} \in \texttt{dom}(\mathcal{H}'')$.

The **sixth correctness property** ensures every object outside the cloned object is also outside the cloned object's clone.

> **Correctness property 6:** Objects outside the cloned object remain 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 with a case analysis. Assume that $\mathcal{H}' \nvdash \iota' \preceq \iota^*$ for some ι^*, where $\iota^* \in \texttt{dom}(\mathcal{H})$, $\iota^* \neq \iota$, and $\mathcal{H}' \vdash \iota \preceq \iota^*$. 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 is no ownership relation between ι' and ι^*. 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 correctness property** ensures the original object and its clones have corresponding references to objects that are outside their respective representation.

> **Correctness property 7:** For all references from the representation of the clone to objects outside that representation, there must exist references to those same objects from the representation of 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 `SheepAux` function. The premise of the property seven states $\mathcal{H}' \vdash \iota' \preceq \iota''$, and from this we deduce $\iota'' \mapsto \iota'' \in \texttt{map}$ and $\mathcal{H}' \vdash \texttt{map}(\iota) \preceq \texttt{map}(\iota'')$, as $\texttt{map}(\iota') = \iota$. By **lemma:** *address mapping preserves inside relation*, we can state that $\mathcal{H}' \vdash \iota \preceq \iota''$. Next we consider how the fields (`f`) are constructed, which is only possible by R-SHEEPINSIDE, the recursive case of the `SheepAux` function. R-SHEEPINSIDE only transverses objects of the original heap, therefore ι'' must be a value of a field of an object in the original heap.

4.3 Chapter Summary

In this chapter, we present `recurSC`, a formalism of the sheep cloning semantics inspired by the semantics of shallow cloning and deep cloning. We describe the difficulties in proving type soundness for `recurSC`, as well as seven correctness properties for sheep cloning, and an outline for the proof of each correctness property with regards to `recurSC`.

Chapter 5

mapSC Formalism

In this chapter, we present the mapSC formalism, our second attempt at formalising sheep cloning. We encountered increasingly more difficult problems during the construction of our proof for the subject reduction of mapSC. Details of the difficulties in our proofs are presented in Section 5.1. mapSC is a formalism of sheep cloning based on the map that was part of the recurSC formalism of chapter 4.1. The mapSC formalism is based on the observation that the mapping of the type and fields of an object gives the type and fields of that object's sheep clone. The sheep cloning semantics of mapSC creates a map for the object being cloned and then the sheep clone is constructed from that map.

The map is a set of mappings ($\overline{\iota \mapsto \iota'}$) from the objects ($\bar{\iota}$) in the representation of the object being cloned to the objects ($\overline{\iota'}$) in the representation of that object's sheep clone.

Fig. 5.1 presents an illustration of a map. The left oval is the domain of the map and the representation of the object being cloned. The right oval is the range of that map as well as the representation of the sheep clone. The arrows between the ovals are the mappings from the cloned objects to their sheep clones.

We employed the map in the SheepAux function of the recurSC formalism to prevent looping when the SheepAux function traverses the

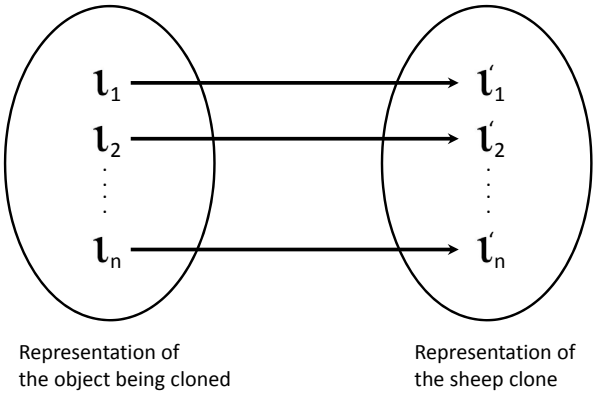


Figure 5.1: A map.

heap. Nearing the completion of the `recurSC` formalism, we discovered we could use the `map` to transform an object's type to the type of that object's sheep clone (the `mapT` function, Fig. 4.5). When the `map` is applied to the type of an object in the representation of the object being cloned, the owner and owner parameters of that type are substituted for the owner and owner parameters of its sheep clone, creating the type of that object's sheep clone. The type of every object in the representation of the sheep clone is created by applying the `map` to the type of every object in the representation of the object being cloned. Furthermore, applying the `map` to the fields of the object being cloned gives the fields of the sheep clone, and similarly, applying the `map` to the fields of every object in the representation of the original gives the fields of every object in the representation of the clone.

In this chapter, we again only present the judgments that are different to those of the `core` formalism of chapter 3. The syntax, well-formedness, expression typing, class and method declarations, static and dynamic inside relation, sub-typing, and non-sheep cloning reduction are the same for both formalisms; please refer back to the `core` formalism for the details of these judgments.

5.0.1 Sheep Cloning in `mapSC`

The principle idea of the sheep cloning semantics in `mapSC` is to expend the role played by the map of the `recurSC` formalism in creating sheep clones. The goal is to saturate the map with the representation of the object being cloned through a depth-first traversal of the heap that contains the representation. Once the map is complete, the sheep clone is constructed directly from the information gathered in the map, rather than creating the sheep clone during the traversal of the heap as shown in `recurSC`.

We will first present the formalism semantics for sheep cloning, then an example to illustrate how sheep clones are created in `mapSC`. In Fig. 6.6, we present the reduction for the sheep cloning expression (`R-Sheep`) of `mapSC`.

Sheep cloning reduction:

$$\frac{\texttt{makeMap}(\iota,\ \emptyset)_{\mathcal{H},\iota} = \texttt{map} \qquad \texttt{makeHeap}(\texttt{map})_{\mathcal{H},\texttt{map}} = \mathcal{H}' \qquad \texttt{map}(\iota) = \iota'}{\texttt{sheep}(\iota);\ \mathcal{H} \rightsquigarrow \iota';\ \mathcal{H}'}\ (\text{R-SHEEP})$$

Figure 5.2: Sheep cloning reduction for `mapSC`.

The premises of R-SHEEP relies on two functions, `makeMap` and `makeHeap`. The `makeMap` function, shown in Fig. 5.4, creates the `map`, and the `makeHeap` function takes the `map` and constructs the sub-heap that contains the sheep clone as described by the `map`.

The `makeMap` function creates the `map` by recursively traversing through the object graph of the object being cloned. For each object inside the object being cloned given by the inside relation in Fig. 3.11, it adds a mapping between the traversed object and a fresh object. The `makeMap` func-

tion takes two arguments: an object address (ι) and a `map`, and returns that `map` updated with the sheep clone of ι. The `makeMap` function also takes two subscripts that remain constant through the construction of the `map`. The first subscript is the original heap before the reduction of the sheep cloning expression. The second subscript is the object address in the sheep cloning expression, which is the object being sheep cloned. The `map` for ι is created by invoking the `makeMap` function with ι and $\emptyset$ (an empty `map`).

The `makeHeap` function, shown in Fig. 5.7, takes the `map` created by the `makeMap` function on ι, and returns a heap ($\mathcal{H}'$) comprised of the original heap ($\mathcal{H}$) and a new sub-heap containing the representation of the sheep clone. The `makeHeap` function also takes two subscripts: the original heap ($\mathcal{H}$) and the `map` of ι produced by `makeMap`. The third premise of R-SHEEP states that the sheep clone of ι is `map(`ι`)`.

5.0.2 Map Well-Formedness

Dynamic Map Well-formedness:

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

Figure 5.3: Dynamic Map Well-formedness:

Fig. 5.3 presents the judgments for `map` well formedness. The `map` is only created during runtime, which means it is judged under the heap. The `map` is well-formed if it is empty (F-EMPTY) or a bijective function (F-MAP).

The map is considered well-formed when it is a bijective function. The map well-formedness judgment could be a stronger property, however, we did not want map well-formedness to be defined by the map's construction or its use. Map well-formedness should not contain aspects of ownership types nor information regarding the object which it is created from. Sheep clones should only be created from well-formed maps, however, the judgment for map well-formedness should not be the same as the judgment for well-formed sheep clones, otherwise we will encounter the same problem as the recurSC formalism, where the construction and the definition of the sheep clones are interleaved.

5.0.3 The makeMap Function

In Fig. 5.4 we present the definition for the makeMap function. The makeMap function traverses the entire representation of the object being cloned, creating a mapping for each object in the representation.

The makeMap function has three base cases: SC-MAPNULL, SC-MAPEXIST, and SC-MAPOUTSIDE, and a recursive case: SC-MAPINSIDE.

For the base cases, the map is unchanged, the same map that was passed into makeMap is returned.

The SC-MAPNULL case occurs when the makeMap function traverses a null. The map does not create a mapping for null.

The SC-MAPEXIST case occurs when the makeMap function traverses an object that is already in the domain of the map. For the object to already be in the map, the object must have already been traversed. To prevent looping in the traversal, objects that have already been traversed are ignored.

The SC-MAPOUTSIDE case occurs when the makeMap function traverses an object that is not inside the representation of the object being sheep cloned. Sheep cloning aliases objects that are not inside the representation being cloned but are reachable directly from inside the represen-

MakeMap:

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

(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)

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

(SC-MAPINSIDE)

Figure 5.4: MakeMap.

tation. It is possible to add a mapping from the object to itself into the `map` for this case, however, there is no need for this substitution of an object for itself. It is important when proving the `makeMap` function that the `map` only contains the objects inside the object being cloned.

The SC-MAPINSIDE case describes when the `makeMap` function reaches

an object (ι') that is inside the representation being cloned ($\mathcal{H} \vdash \iota' \preceq \iota$) and not already in the domain of the `map` ($\iota' \notin$ `dom(map)`). The `map` is updated ($\texttt{map}_1$) with a mapping from ι' to a fresh object (ι''). Finally, the `makeMap` function is recursively called on each of the fields of ι' with the `map` returned from the `makeMap` function on the previous field. The `makeMap` function on the first field is invoked with $\texttt{map}_1$.

To better understanding how the `mapSC` formalism creates sheep clones, we present an example illustrating how the `makeMap` function construct the map used by the `makeHeap` function to create sheep clones. The example is the same as the example in Fig. 4.1, whereby the aim of the example is to sheep clone object `A`.

In Fig. 5.5, we present the first three steps taken by the `makeMap` function in creating the map for object `A`. The top image shows the initial call of the `makeMap` function. The traversal reached object `A` and the map is initially empty ($\emptyset$). A mapping of object `A` to a fresh variable ($\texttt{A}'$) to is be created as the reflexivity of the inside relation states that object `A` is inside itself. The middle image shows when the `makeMap` function has traversed object `D` from object `A`. No mapping for object `D` is added into the map as object `D` is outside the representation of object `A`. The traversal backtracks to object `A` as it has reached an object outside the representation of the object being cloned. The bottom image shows the `makeMap` function has traversed object `B` from object `A`. A mapping of object `B` to a fresh variable ($\texttt{B}'$) will be added to the map as object `B` is inside the representation of object `A`.

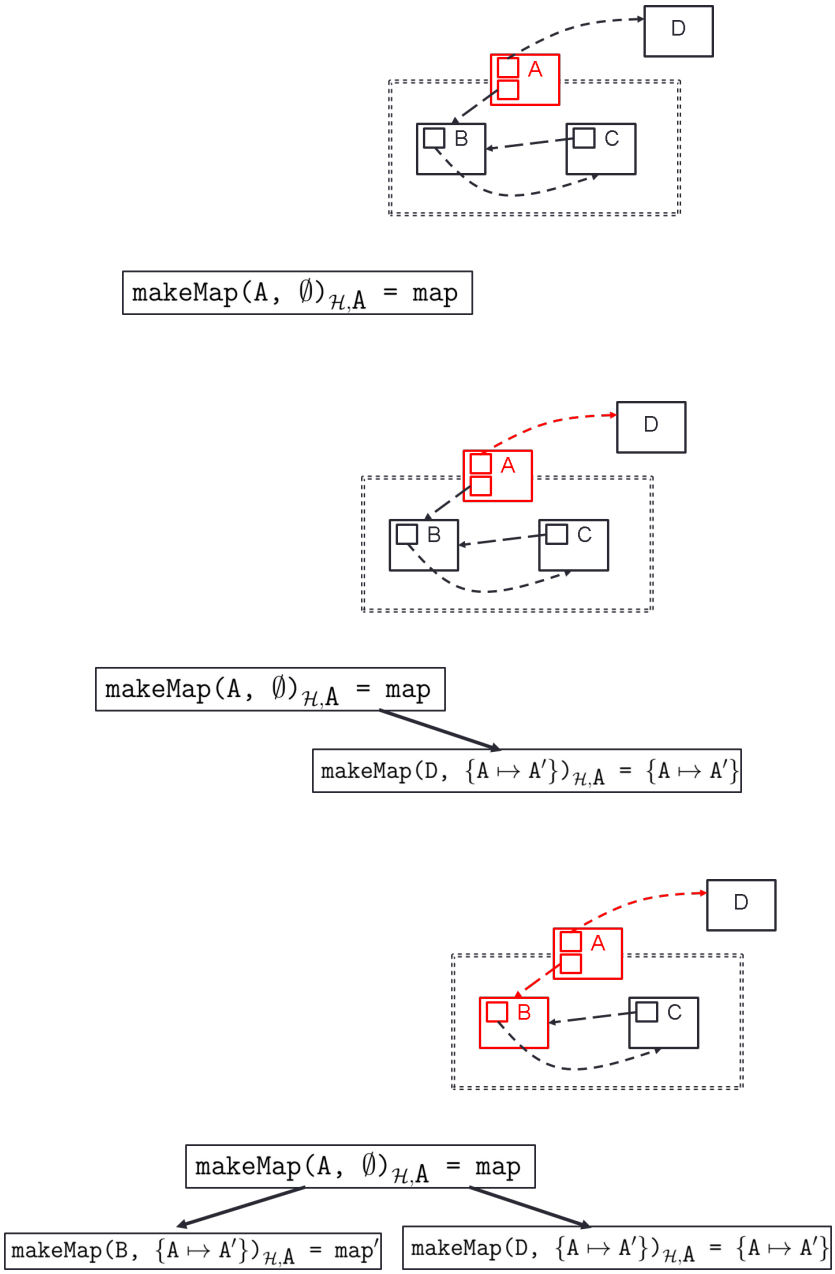


Figure 5.5: Sheep cloning in `mapSC` (part 1).

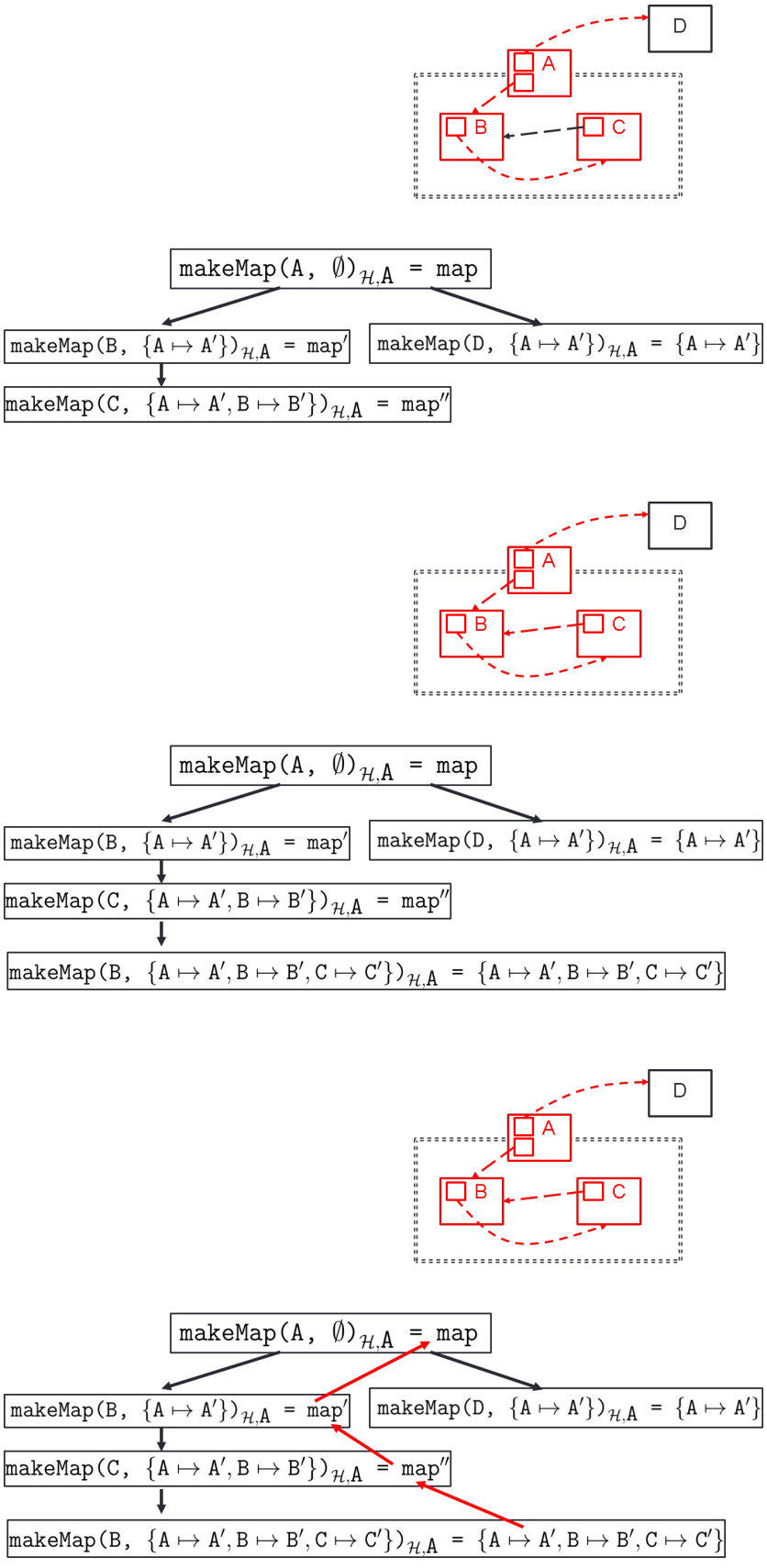


Figure 5.6: Sheep cloning in `mapSC` (part 2).

In Fig. 5.6, we present the final three steps taken by the makeMap function in creating the map for object A. The top image shows when the makeMap function has traversed object C from object B. The map contains two mapping, A to A′ and B to B′. A mapping for object C to a fresh variable (C′) is added into the map as object C is inside the representation of object A. The middle image shows when the makeMap function has traversed back to object B from object C. No new mapping is created for object B since object B is already in the domain of the map. The traversal backtrack to object A as it has reached an object that is already in the map. The final image at the bottom shows when the makeMap function has backtracked the traversal up to object A. The traversal stops as it has already traversed object A and that there are no more objects from object A that have not been traversed. The map created by the mapMake function on object A contains the mapping of A to A′, B to B′, and C to C′.

5.0.4 The makeHeap Function

The definition for the makeHeap function in Fig. 5.7 consists of two cases: SC-MAKEHEAPE and SC-MAKEHEAP. The makeHeap function recursively creates a new object for each mapping in the map. The new object is the object denoted in the range of each mapping, and the object in the range of the map is the sheep clone of the object that maps to it. Creating a new object for every mapping in map creates the representation of the sheep clone.

The base case SC-MAKEHEAPE occurs when the map is empty and the original heap described in the subscript of the makeHeap function is returned. The recursive case SC-MAKEHEAP creates a new object (ι') for each mapping ($\iota \mapsto \iota'$) in the map. The type and fields for ι' are the mapping of the type and fields for ι respectively. The map function is the same as the mapT function in Fig. 4.5 of the recurSC formalism in chapter 4.1.

MakeHeap:

$$\frac{\mathcal{H}(\iota) = \{N;\ \overline{\mathtt{f}\rightarrow v}\} \qquad \mathcal{H}' = \mathtt{makeHeap(map')}_{\mathcal{H},\mathtt{map}},\ \iota' \rightarrow \{\mathtt{map}(N);\ \overline{\mathtt{f}\rightarrow\mathtt{map}(v)}\}}{\mathtt{makeHeap}(\iota \mapsto \iota',\ \mathtt{map'})_{\mathcal{H},\mathtt{map}} = \mathcal{H}'}$$

(SC-MAKEHEAP)

$$\frac{}{\mathtt{makeHeap}(\emptyset)_{\mathcal{H},\mathtt{map}} = \mathcal{H}}$$

(SC-MAKEHEAPE)

Figure 5.7: MakeHeap.

5.1 Proof Outline for Type Soundness of mapSC

The mapSC formalism creates an object's sheep clone from the map of that object. The direct relationship between the representation of the sheep clone and the map allows properties describing the sheep clone to depend upon the makeMap function that creates the map, and the makeHeap function that creates the sheep clone.

In Section 5, we stated that the correctness, or well-formedness, of the map should be solely on the merit of being a map. Well-formedness of the map should not be overcomplicated with features, such as ownership types.

In this Section, we attempt to bridge the difference between the map's ability to describe the structural properties of the sheep clone and the algorithmic process in constructing the sheep clones from the map. We also describe some of the reasons for why we have failed to prove type soundness for mapSC.

To prove type soundness for mapSC, we are required to establish progress

and type preservation properties. Proving type preservation consists of two parts, the first part shows the resulting expression of each reduction rule preserves the type of the expression being reduced, and the second part requires the resulting heap of each reduction rule to be well-formed.

The proof for type preservation of mapSC requires the makeHeap function, in Fig. 5.7, to produce well-formed heaps.

Lemma 3.1: makeHeap creates well-formed heaps.

For all $\mathcal{H}$, $\mathcal{H}'$, map, *and* map$'$.
If :
$\texttt{makeHeap}\,(\texttt{map})_{\mathcal{H},\texttt{map}} = \mathcal{H}''$
$\texttt{makeHeap}\,(\texttt{map}')_{\mathcal{H},\texttt{map}} = \mathcal{H}'$,
$\vdash \mathcal{H}$ OK
$\vdash \texttt{map}'$ OK
then : $\mathcal{H}'' \vdash \mathcal{H}'$ OK.

Lemma 3.1 is proved by structural induction over the derivation of $\texttt{makeHeap}\,(\texttt{map}')_{\mathcal{H},\texttt{map}} = \mathcal{H}'$, with a cases analysis on the last step. The proof for the base case, SC-MAKEHEAPE, is trivial as the premise of **lemma 3.1** states that $\vdash \mathcal{H}$ OK, and the resulting heap ($\mathcal{H}'$) is the same as the original heap ($\mathcal{H}$), which is passed to the makeHeap function as a subscript.

For the recursive case, SC-MAKEHEAP, we need to show the object $(\iota' \rightarrow \{\texttt{map}(N);\ \overline{\texttt{f}\rightarrow\texttt{map}\,(v)}\})$ created by that iteration of the makeHeap function is well-formed according to the definition of heap well-formedness. Invoking the makeHeap function on the remaining elements of the map can be assumed to be well-formed by the induction hypothesis.

The mapSC formalism has the same definition for heap well-formedness as the core formalism in chapter 3. Fig. 5.8 repeats the judgment for heap well-formedness.

The F-HEAP judgment states that the object ι' in $\mathcal{H}'$ is well-formed if:

1. ι' has a well-formed type (N).

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

$$\frac{\begin{array}{cc}\mathcal{H}' \subseteq \mathcal{H} & \forall\, \iota \to \{N; \overline{\mathtt{f}\to v}\} \in \mathcal{H}' : \{\ \mathcal{H} \vdash N\ \text{OK} \\ \overline{fType(\mathtt{f}, N) = N'} & \mathcal{H} \vdash \overline{v : [\iota/\mathtt{this}]N'\iota} \qquad \mathcal{H} \nvdash own_{\mathcal{H}}(\iota) \preceq \iota \\ \multicolumn{2}{c}{\forall\, v \in \overline{v} : v \neq \mathtt{null} \Rightarrow \{v \in dom(\mathcal{H}) \wedge \mathcal{H} \vdash \iota \preceq own_{\mathcal{H}}(v)\}\}}\end{array}}{\mathcal{H} \vdash \mathcal{H}'\ \text{OK}}$$

(F-HEAP)

Figure 5.8: Well-formed heap.

2. The fields of ι' have the type (N') stated by the *fType* function.
3. The owner of ι' can not be shown to be inside ι', $\mathcal{H} \nvdash own_{\mathcal{H}}(\iota) \preceq \iota$.
4. For each non-`null` field of ι', the field must be in the domain of $\mathcal{H}'$ and ι' is inside the owner of that field.

In order to show the heaps created by the `makeHeap` function are well-formed, the heap well-formedness judgment must allow heaps to be judged under a larger heap. The heaps created by the `makeHeap` function can only show well-formedness when it is judged under the heap created from the initial `map` in the subscript of `makeHeap`. This is because the `makeHeap` function creates objects from the `map` in a non-deterministic order, and therefore, it is possible for an iteration of `makeHeap` to create an object that contains fields that may not yet have been created. The `makeHeap` function does ensure that every object in the `map` will eventually be created.

To show every $\iota' \to \{\mathtt{map}(N); \overline{\mathtt{f}\to\mathtt{map}\,(v)}\}$ in $\mathcal{H}'$ is well-formed under $\mathcal{H}''$, as stated in **lemma 3.1**, then we need:

1. To show `map`(N) is a well-formed type, we require a lemma that shows type well-formedness is preserved by mapping or address substitution.

2. To show $\texttt{map}(v)$ has the type stated by the *fType* function on $\texttt{map}(N)$, we need a lemma that shows the type given by *fType* is preserved by address substitution.

3. To show that it is not possible for the owner of ι' to be inside ι', we need a lemma that states the ownership structure of the heap created by `makeHeap` is acyclic.

4. Finally, to show ι' is inside the owner of the non-`null` $\texttt{map}(v)$, we need a lemma that show the inside relation is preserved by mapping or address substitution.

It is important to distinguish between an object (ι) that is outside of another object (ι'), $\mathcal{H} \vdash \iota' \preceq \iota$, against not being able to show ι' that is inside of ι, $\mathcal{H} \nvdash \iota' \preceq \iota$. For example, consider two objects with the same owner, in ownership terminology these two objects are considered to be peers or siblings. It is not possible to show an inside relation between these two objects, as they are neither inside nor outside of each other.

Lemma 3.2: Non-inside relation is preserved by mapping.

If :
$\mathcal{H} \nvdash \iota \preceq \iota'$
$\vdash \texttt{map}$ OK
then : $\mathcal{H} \nvdash \texttt{map}(\iota) \preceq \texttt{map}(\iota')$.

Any properties for the non-inside relation that we are unable to show rely upon the same property, however, on the inside relation, therefore, to prove **lemma 3.2** we must first prove how the inside relation is preserved by mapping.

Lemma 3.3: Inside relation is preserved by mapping.

If :
$\mathcal{H} \vdash \iota \preceq \iota'$

$\vdash$ map OK
then : $\mathcal{H} \vdash \texttt{map}(\iota) \preceq \texttt{map}(\iota')$.

Lemma 3.3 is proved by structural induction over the definition of $\mathcal{H} \vdash \iota \preceq \iota'$, with a case analysis on the last step. The dynamic inside relation judgments are presented in Fig. 3.11 of chapter 3. The reflexive judgment (I-REF) and the world judgment (I-WORLD) are both trivial.

The owner judgment (I-REC) and the transitive judgment (I-TRANS) require a case analysis for when $\iota \in \texttt{dom(map)}$, $\iota \notin \texttt{dom(map)}$, $\iota' \in \texttt{dom(map)}$, and $\iota' \notin \texttt{dom(map)}$. It is possible to reduce the number of case analysis required by **lemma 3.3** if $\iota \in \texttt{dom(map)}$ is a premise.

To show $\mathcal{H} \vdash \texttt{map}(\iota) \preceq \texttt{map}(\iota')$ when $\iota \notin \texttt{dom(map)}$ requires showing that $\iota' = \texttt{map}(\iota')$. Since $\iota \notin \texttt{dom(map)}$ we can deduce from the construction of the map that $\mathcal{H} \nvdash \iota \preceq \iota^*$, and since $\mathcal{H} \vdash \iota \preceq \iota'$, then $\mathcal{H} \nvdash \iota' \preceq \iota^*$, and finally, $\iota' \notin \texttt{dom(map)}$. The key in that sequence is establishing the connection between objects in the domain of the map and the objects inside ι^*. Conceptually these two sets should be identical by the construction of the map. The makeMap function constructs the map by creating only mappings of the objects that are inside ι^*. The SC-MAPINSIDE case of makeMap prevents objects not inside ι^* from being in the domain of the map. The owners-as-dominators prescriptive policy and the assumption that the heap does not contain garbage ensures every object inside ι^* is transitively reachable from ι^*, and therefore, traversed by the makeMap function.

Lemma 3.4 is a stronger well-formedness property on the map, which express the connection between the representation of ι^* (i.e. all the objects inside ι^*) and the map created by the makeMap function on ι^*. This property states that an object is inside the domain of a map if and only if that object is inside the object which the map is created for:

Lemma 3.4: Correctness property for the map.

$\mathcal{H} \vdash \iota \preceq \iota^*$ **iff** $\iota \in \texttt{dom(map)}$
where : $\texttt{makeMap}(\iota^*, \emptyset)_{\mathcal{H},\iota^*} = \texttt{map}$.

To prove **lemma 3.4**, we are required to show the lemma holds for both directions. Consider this lemma from left to right: if $\mathcal{H} \vdash \iota \preceq \iota^*$ then $\iota \in \texttt{dom(map)}$, where $\texttt{makeMap}(\iota^*, \emptyset)_{\mathcal{H},\iota^*} = \texttt{map}$. This property can be shown by structural induction over the derivation of the makeMap function, with a case analysis on ι and the last step. The intuition behind the proof of this property is to identify when ι is added into the map by the makeMap function. By $\mathcal{H} \vdash \iota \preceq \iota^*$, ι will eventually be traversed by the $\texttt{makeMap}(\iota^*, \emptyset)_{\mathcal{H},\iota^*}$ function. The three base cases of the makeMap function are either not applicable for this property or they are trivial to prove.

To show **lemma 3.4** for the recursive case SC-MAPINSIDE, a case analysis over ι is required. The case $\iota = \iota^*$ requires showing $\iota^* \in \texttt{dom(map)}$, which is trivial by I-Ref and the premise of SC-MAPINSIDE. Unfortunately, the case $\iota \neq \iota^*$ exposes **lemma 3.4** as being too strong. A weaker version of **lemma 3.4** is required in order to invoke the inductive hypothesis on makeMap.

Lemma 3.4.1: Correctness property for the map, version two.

if :
$\mathcal{H} \vdash \iota \preceq \iota^*$
$\texttt{makeMap}(\iota'', \texttt{map}')_{\mathcal{H},\iota^*} = \texttt{map}$
$\mathcal{H} \vdash \iota \texttt{ R } \iota''$
then : $\iota \in \texttt{dom(map)}$.

Lemma 3.4.1 states that $\iota \in \texttt{dom(map)}$, if ι is inside ι^* ($\mathcal{H} \vdash \iota \preceq \iota^*$) and the object ($\iota''$) currently being traversed by the makeMap function is reachable by ι ($\mathcal{H} \vdash \iota \texttt{ R } \iota''$). **Lemma 3.4.1** is also proved by structural induction over makeMap, with a case analysis on ι'' and the last step. The base case SC-MAPNULL remains trivial, as it is not applicable because ι'' can not be a null.

The two remaining base cases, however, are a lot more problematic. The SC-MAPOUTSIDE case states ι'' is not inside ι^* and that $\texttt{map}' = \texttt{map}$, however, there is not enough information to determine if $\iota \in \texttt{dom(map)}$ or $\iota \in \texttt{dom(map}')$. Similarly, the SC-MAPEXIST case states that $\iota'' \in \texttt{dom(map)}$ and $\texttt{map}' = \texttt{map}$, and once again it is possible to show either $\iota \in \texttt{dom(map)}$ or $\iota \in \texttt{dom(map}')$.

An illustration of the issues with the SC-MAPEXIST and SC-MAPOUTSIDE cases is presented in Fig. 5.9. The diagram on the left is an example of the problem with SC-MAPEXIST, while the diagram on the right is an example of the problem with SC-MAPOUTSIDE. For both of these diagrams, ι is inside ι^* and ι is reachable from ι'', however, the SC-MAPEXIST case would occur when the `makeMap` function is invoked on ι'' for the diagram on the left, and similar the SC-MAPOUTSIDE case would occur for the diagram on the right.

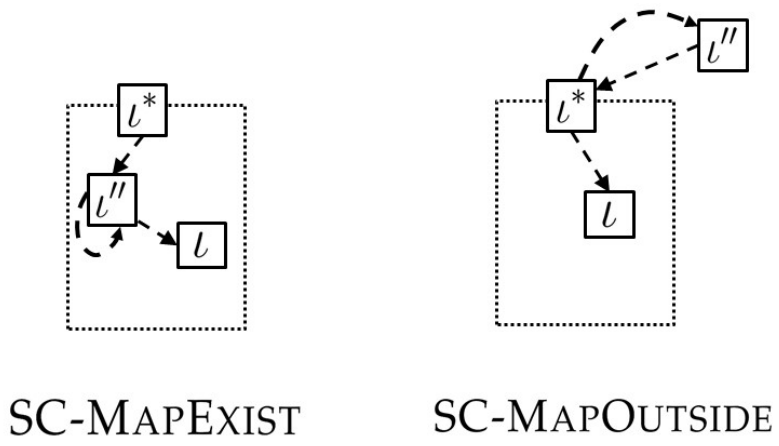


Figure 5.9: Examples of the base cases for the `makeMap` function.

These two base cases, SC-MAPEXIST and SC-MAPOUTSIDE, can be excluded from **lemma 3.4.1** by strengthening its premise. The proof for the

resulting **lemma 3.4.2** is to identify the reference path from ι'' to ι, since ι is reachable from ι''. The need to strengthen the premise of a lemma is a common indication that the lemma cannot be proven, and at this point in the proof for type preservation, we started to have some serious doubts regarding the use of the `makeMap` function to formalise sheep cloning.

Lemma 3.4.2: Correctness property for the `map`, version three.

if :
$\mathcal{H} \vdash \iota \preceq \iota^*$
$\texttt{makeMap}(\iota'', \texttt{map}')_{\mathcal{H},\iota^*} = \texttt{map}$
$\mathcal{H} \vdash \iota \texttt{ R } \iota''$
$\mathcal{H} \vdash \iota \preceq \iota^*$
$\iota \notin \texttt{dom}(\texttt{map}')$
then : $\iota \in \texttt{dom}(\texttt{map})$.

We attempted to prove **lemma 3.4.2** by structural induction over the derivation of the `makeMap` function, with a case analysis on ι'' and the last step. None of the three base cases of the `makeMap` function are applicable. For the recursive case SC-MAPINSIDE, if $\iota = \iota''$ by the case analysis over ι'', then $\iota \in \texttt{dom}(\texttt{map})$ is trivial. If $\iota \neq \iota''$, then $\iota \in \texttt{dom}(\texttt{map})$ can be shown by the inductive hypothesis over the particular field of ι'' that is: able to reach ι, not be in the domain of `map`, and not be outside of ι^*. The `makeMap` function traverses over an object's representation non-deterministically, and therefore, it is not possible for the `makeMap` function to select the field that satisfies those three requirements. The `mapSC` formalism is flawed, as its sheep cloning semantics are so algorithmic that their proof depends upon the method in which the heap is traversed.

The biggest motivation for our `descripSC` formalism came when we had to prove our `mapSC` formalism. When proving the `mapSC` formalism, we came to the realisation that we should not focus on the construction of the sheep clone. We are not interested in "how" sheep clones are created

or the details of the implementation of a sheep cloning algorithm, instead, our focus should have been on describing and formulating what a sheep clone is.

5.2 Chapter Summary

In this chapter, we present `mapSC`, a formalism of the sheep cloning semantics that creates sheep clones from the object structure described in the `map`. The `map` describes the representation of the sheep clone by recording the representation of the object that is being cloned. We encountered problems when proving `mapSC`, as we were required to describe how the traversal used in creating the `map` would affect the structure of the sheep clone. The semantics for sheep cloning should not be dictated by how the representation is being traversed, i.e. depth-first or breadths-first, but instead the semantics should be describing properties for the structure of that representation.

Chapter 6

descripSC Formalism

In this chapter, we present descripSC, our third and final semantics for sheep cloning. The descripSC formalism presents the semantics of sheep cloning in a descriptive style. We have proved type soundness for the descripSC formalism, and the complete proof is in Appendix A.

The sheep cloning semantics of descripSC consists of three steps: identify the objects required to be copied; copy those objects; rename them to create the sheep clone. Only after creating the mapSC formalism and the difficulties we encountered when proving the mapSC formalism did we realise the possibility of creating sheep clones with this procedure.

The descripSC formalism is designed to present a proof that is as succinct and concise as possible. This differs from the aim of the previous formalisms, recurSC and mapSC, where the aim was to formalise sheep cloning from existing cloning idioms with a hypothetical implementation. The sheep cloning semantics of mapSC was an improvement over the semantics in recurSC, however the focus of mapSC was still directed towards how sheep clones are constructed in practice.

The mapSC formalism uses the map as an abstraction for the representation of sheep clone. The map contains the objects that are required to be copied, as well as all the typing and field information of the sheep clone's representation, and therefore, once the map is created, creating the rep-

resentation of the sheep clone becomes trivial. The complications in the proof of soundness for the mapSC formalism forced us to realise that the sheep cloning semantics of mapSC needed to be abstracted further. Most of the complications arise from requiring too much attention to be paid to the construction of the map. The map may contain a blueprint for creating sheep clones, but the details in the construction of the map create an unnecessary burden for the sheep cloning semantics.

The descripSC formalism is the result from a shift in the paradigm of how we viewed sheep cloning. The algorithmic details of creating sheep clones are no longer considered vital for sheep cloning. The formalism for the sheep cloning semantics should not take into consideration whether a depth-first traversal or breath-first traversal is used. Copying the representation of the object being cloned is not important, however, the method that identifying whether an object is in the representation of the object being cloned remains very important. The ownership structure is responsible for identifying whether an object is in a particular object's representation, so any deep ownership system can incorporate sheep cloning as the ownership structure displayed by the inside relation is viable in every deep ownership system. The dynamic ownership inside relation, in Fig. 3.11, identifies the objects of an object's representation. Every deep ownership system enforces a tree-like structure over their heap, which means every deep ownership system, whether implicitly or explicitly, defines an inside relation for its objects.

6.0.1 Sheep Cloning Semantics for descripSC

The principle idea before the sheep cloning semantics of descripSC is to identify the sub-heap that contains every object in the representation of the object being cloned. Copy the sub heap and rename the objects in it with fresh names.

In Fig. 6.1, we present R-SHEEP, the reduction of the sheep cloning

expression for `descripSC`. R-SHEEP can be described in three steps: the first step identifies the objects required to be copied, the second step copies these objects, and the third step renames the new objects to be distinct from the objects they were copied from.

$$\frac{\begin{array}{c}\overline{\iota' \rightarrow \{N;\ \overline{\mathtt{f} \rightarrow v}\}} = \{\iota' \rightarrow \{N;\ \overline{\mathtt{f} \rightarrow v}\} \in \mathcal{H} \mid \mathcal{H} \vdash \iota' \preceq \iota\} \\ \mathcal{H}(\overline{\iota''})\ \mathtt{undefined} \\ \mathcal{H}' = \overline{\iota'' \rightarrow [\overline{\iota''/\iota'}]\{N;\ \overline{\mathtt{f} \rightarrow v}\}}\end{array}}{\mathtt{sheep}(\iota);\ \mathcal{H} \rightsquigarrow [\overline{\iota''/\iota'}]\iota;\ \mathcal{H},\ \mathcal{H}'}\ \text{(R-SHEEP)}$$

Figure 6.1: Sheep Cloning Reduction.

The first premise of `R-Sheep` identifies the objects in the representation of the object being cloned (ι). The sub-heap described by $\overline{\iota' \rightarrow \{N;\ \overline{\mathtt{f} \rightarrow v}\}}$ contains every object in the representation of ι using the set notion over the whole heap ($\mathcal{H}$) to obtain every object inside ι, ie. $\iota' \preceq \iota$. The second premise ensures the sheep clone and its representation are fresh objects, and the third premise copies the representation of ι and renames every object in the new representation to be the sheep clone. The sub-heap $\overline{\iota' \rightarrow \{N;\ \overline{\mathtt{f} \rightarrow v}\}}$ contains the representation of ι, and is copied to create the new sub-heap $\mathcal{H}'$. The objects in $\mathcal{H}'$ are then renamed to be the representation of the sheep clone of ι. The resulting heap from the reduction of the sheep cloning expression (ι) is the concatenation of the original heap ($\mathcal{H}$) and the new sub-heap ($\mathcal{H}'$) containing the representation of the sheep clone.

To better understand how the `descripSC` formalism create sheep clones,

we will create the sheep clone of our running example using the descipSC formalism. presented in Fig. 4.1 of section 4.0.1 and Fig. 5.5 of section 5.0.3. Just like in the examples in the previous sections, rectangles are objects, arrows are references, and the dotted line represents the ownership representation of the object on its top edge. The example consists of four objects: A, B, C, and D. The aim of the example is to creates the sheep clone of object A.

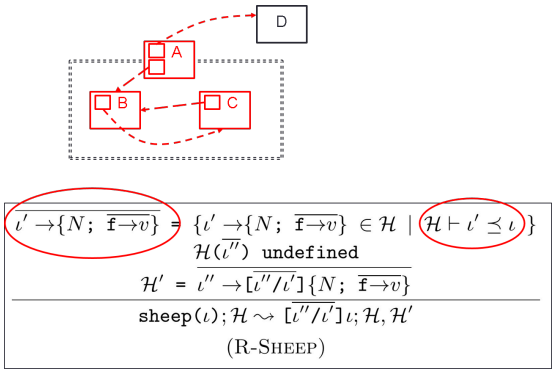
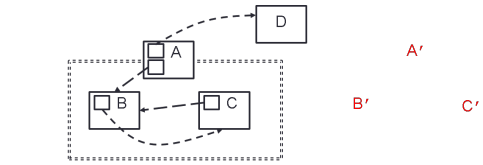


Figure 6.2: Sheep cloning in descripSC (part 1).

In Fig. 6.2, we illustrate how the first line of the sheep cloning semantics (R-Sheep) is identifying every object inside the representation of the object cloned (object A). The objects in red represent the objects inside the representation of object A, and these objects are organised into the subheap $\overline{\iota' \rightarrow\{N;\ \overline{\mathtt{f}\rightarrow v}\}}$.

In Fig. 6.3, we illustrate how the second line of R-Sheep is establishing a fresh set of object names ($\overline{\iota''}$). The fresh object names are the three variables in red: A′, B′, and C′. These three names will later become the name of the objects making up the representation of the sheep clone.

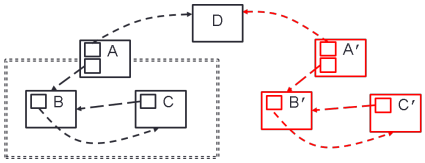
In Fig. 6.4, we illustrate how the final line of R-Sheep is creating the representation of the sheep clone of object A. The representation is created by duplicating of the representation of object A that was established in the first line of R-Sheep, and then renaming ($[\overline{\iota''/\iota'}]$) every object in the



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

Figure 6.3: Sheep cloning in `descripSC` (part 2).

new representation with the fresh names ($\overline{\iota''}$) created in the second line of `R-Sheep`. The object `A′` is the sheep clone of object `A`.



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

Figure 6.4: Sheep cloning in `descripSC` (part 3).

6.1 Proof of Type Soundness for descripSC

This section presents the proof of type soundness for the descripSC formalism. A type system is sound if it has the properties of progress and type preservation. Type preservation, also known as subject reduction, requires the resulting heap of every reduction rule to be well-formed, and for the resulting expression to have the same type as the expression being reduced.

Theorem: Subject Reduction.

If:
(a) $\mathcal{H} \vdash \mathtt{e} : N$
(b) $\vdash \mathcal{H}$ OK
(c) $\mathtt{e}\,;\ \mathcal{H} \rightsquigarrow \mathtt{e}'\,;\ \mathcal{H}'$

then:
(d) $\vdash \mathcal{H}'$ OK
(e) $\mathcal{H}' \vdash \mathtt{e}' : N$
or
(f) $\mathtt{e}' = \mathtt{err}$

We will only discuss the proofs for subject reduction of the sheep cloning reduction. The proofs for the reductions of the other expressions are standard for a type system in the style of Featherweight Java, and are presented in Appendix A.

Subject reduction for the sheep cloning expression (R-SHEEP) shown in Fig. 6.1, requires the resulting heap $\mathcal{H}$, $\mathcal{H}'$ to be well-formed and the resulting sheep clone $[\overline{\iota''/\iota'}]\,\iota$ to have the same type as the sheep cloning expression $\mathtt{sheep}(\iota)$.

To show $[\overline{\iota''/\iota'}]\,\iota$ has the type N, where $\mathcal{H} \vdash \mathtt{sheep}(\iota) : N$, we must first determine the type of $[\overline{\iota''/\iota'}]\,\iota$, then show that type is the same as N. From the premises of R-SHEEP, we can observe $[\overline{\iota''/\iota'}]\,N$ being the type

for $[\overline{\iota''/\iota'}]\iota$, and therefore, we must show $[\overline{\iota''/\iota'}]N = N$. We address this as follows:

1. Use the *Inversion* lemma of the sheep cloning expression to obtain the premises for $\mathcal{H} \vdash \texttt{sheep(}\iota\texttt{)} : N$, and the type of ι.

2. Use the lemma *Expression Typing has Well-formed Types* to show the type of ι is well-formed.

3. The lemma *Owner Parameters are Outside the Owner* states that for every well-formed type ($\texttt{o:C<}\overline{\texttt{o}}\texttt{>}$) in `descripSC` the owner of the type must be inside the type parameters of the type.

   Lemma: Owner Parameters are Outside the Owner.
   If:
   (a) $\mathcal{H} \vdash \texttt{o:C<}\overline{\texttt{o}}\texttt{>}$ OK
   then:
   (b) $\forall\ \texttt{o}_i \in \overline{\texttt{o}} : \mathcal{H} \vdash \texttt{o} \preceq \texttt{o}_i$

   The *Owner Parameters are Outside the Owner* lemma explicitly prevents types from having illegal type parameters.

4. Using the property that the owner of an object can not be inside that object, the lemma *Owner Parameters are Outside the Owner* on ι, and the lemma *Structural Consistency of the Inside Relation*, we can show the type parameters of the type for ι are also not inside ι.

   Lemma: Structural Consistency of the Inside Relation.
   If:
   (a) $\mathcal{H} \vdash \iota \preceq \iota'$
   (b) $\mathcal{H} \vdash \iota \not\preceq \iota''$
   then:
   (c) $\mathcal{H} \vdash \iota' \not\preceq \iota''$

Structural Consistency of the Inside Relation states that if object ι is inside object ι', and object ι is not inside object ι'', then object ι' cannot be inside object ι''.

5. Finally, we can state that $[\overline{\iota''/\iota'}]\,N = N$ because the owner of ι and the type parameters for the type of ι are all outside of ι, and from R-SHEEP, we know that $\overline{\iota'}$ are inside ι, as $\forall\, \iota' \in \overline{\iota'} : \mathcal{H} \vdash \iota' \preceq \iota$.

6.1.1 Sheep Cloning Preserves Heap Well-Formedness

Due to the size and complexity of the proof that shows the sheep cloning reduction preserves heap well-formedness, this property requires a separate lemma.

Lemma: Sheep cloning reduction preserves heap well-formedness.

If:
(a) $\texttt{sheep}(\iota)\texttt{;}\ \mathcal{H} \rightsquigarrow [\overline{\iota''/\iota'}]\iota\texttt{;}\ \mathcal{H}, \mathcal{H}'$
(b) $\vdash \mathcal{H}$ OK

then:
(c) $\vdash \mathcal{H}, \mathcal{H}'$ OK

$\mathcal{H}$ and $\mathcal{H}'$ are well formed heaps if every object inside each sub-heap is well-formed. It is trivial to show well-formedness for $\mathcal{H}$. From the premise of R-SHEEP, we can show that $\mathcal{H}'$ consists of $\overline{\iota'' \rightarrow [\overline{\iota''/\iota'}]\{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\}}$, and that $\overline{\iota' \rightarrow \{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\}}$ is a well-formed sub-heap of $\mathcal{H}$. To show every ι'' in $\overline{\iota''}$ is well-formed, F-HEAP states that the type of each ι'' must be well-formed, the fields of each ι'' must have the typing defined by the $fType$ function on that ι'', and that the owner of each ι'' must be inside the owner of every non-null field of that ι''.

The lemma *Address Substitution preserves Type Well-Formedness* is used to show the type of ι'' is well-formed.

Lemma: Address Substitution preserves Type Well-Formedness.

If:
(a) $\mathcal{H} \vdash N$ OK
(b) $\vdash \mathcal{H}$ OK
(c) $\mathcal{H}\,(\overline{\iota})$ `undefined`
(d) $\mathcal{H}' = \overline{\iota \rightarrow [\overline{\iota/\iota'}]\,\{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\}}$

then:
(e) $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]N$ OK

$\mathcal{H} \vdash N$ OK is obtained from the premise of F-HEAP on $\mathcal{H}$ with ι', and from the premise of R-SHEEP, we obtain $\mathcal{H}(\overline{\iota})$ `undefined` and $\mathcal{H}' = \overline{\iota \rightarrow [\overline{\iota/\iota'}]\,\{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\}}$.

The lemmas *Address Substitution preserves* $fType$ and *Address Substitution preserves Value Typing* are used to show the fields of ι'' have the typing defined by $fType$ on ι'' and $[\overline{\iota/\iota'}]N$. From the premises of F-HEAP on $\mathcal{H}$, we can obtain the $fType$ function for every field of $N$and the typing for the fields of ι'.

Lemma: Address Substitution preserves $fType$.

If:
(a) $fType(\texttt{f}_i, N) = N'$
(b) $\mathcal{H}(\iota) = \{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\}$
(c) $\vdash \mathcal{H}$ OK
(d) $\mathcal{H} \vdash N$ OK
(e) $\mathcal{H}\,(\overline{\iota})$ `undefined`
(f) $\overline{\iota' \rightarrow\{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\}} = \{\iota' \rightarrow\{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\} \in \mathcal{H} \mid \mathcal{H} \vdash \iota' \preceq \iota^* \}$

then:
(g) $fType(\texttt{f}_i, [\overline{\iota/\iota'}]N) = [\overline{\iota/\iota'}]\,N'$

Lemma: Address Substitution preserves Value Typing.

If:
(a) $\mathcal{H} \vdash v : N$
(b) $\vdash \mathcal{H}$ OK
(c) $\overline{\iota' \rightarrow \{N\texttt{;}\ \ \overline{\texttt{f}\rightarrow v}\}} = \{\iota' \rightarrow \{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\} \in \mathcal{H} \mid \mathcal{H} \vdash \iota' \preceq \iota^* \}$
(e) $\mathcal{H}\,(\bar{\iota})\ \texttt{undefined}$
(f) $\mathcal{H}' = \overline{\iota \rightarrow [\overline{\iota/\iota'}]\{N\texttt{;}\ \ \overline{\texttt{f}\rightarrow v}\}}$

then:
(g) $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]v : [\overline{\iota/\iota'}]N$

Finally, the lemma *Address Substitution preserves Inside Relation* is used to show that every object in $\mathcal{H}'$ preserves the deep ownership invariant where the owner of ι' must be inside the owner of the fields of ι'.

Lemma: Address Substitution preserves Inside Relation.

If:
(a) $\mathcal{H} \vdash \iota \preceq \iota''$
(b) $\vdash \mathcal{H}$ OK
(c) $\overline{\iota' \rightarrow \{N\texttt{;}\ \ \overline{\texttt{f}\rightarrow v}\}} = \{\iota' \rightarrow \{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\} \in \mathcal{H} \mid \mathcal{H} \vdash \iota' \preceq \iota^* \}$
(e) $\mathcal{H}\,(\bar{\iota})\ \texttt{undefined}$
(f) $\mathcal{H}' = \overline{\iota \rightarrow [\overline{\iota/\iota'}]\{N\texttt{;}\ \ \overline{\texttt{f}\rightarrow v}\}}$

then:
(g) $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}](\iota \preceq \iota'')$

The lemma *Owner Parameters are Outside the Owner*, the lemma *Structural Consistency of the Inside Relation*, and the conjecture *Preservation of Ownership Acyclicity for Sheep Cloning* are the most interesting properties in our proof of subject reduction for `descripSC`. Ownership types are commonly considered to be very constricting for the structure of the objects in the system, however, we found the strictness of ownership types allows us to reason about manipulations to ownership structures in a very straightforward manner.

Lemma: Owner Parameters are Outside the Owner.

If:
(a) $\mathcal{H} \vdash \mathtt{o} : \mathtt{C}\langle\overline{\mathtt{o}}\rangle$ OK

then:
(b) $\forall\, \mathtt{o}_i \in \overline{\mathtt{o}} : \mathcal{H} \vdash \mathtt{o} \preceq \mathtt{o}_i$

The lemma *Owner Parameters are Outside the Owner* is proved by natural deduction over the definition for the class declaration and well-formed types of descripSC. The property that ownership structure is structurally consistent requires two lemmas: the first lemma states that the substitution of the owner is the same as the owner of the substitution; the second lemma states that an object's structure is persistent when it cannot be shown to be inside another object's structure.

Lemma: Address Substitution preserves Owner Function.

If:
(a) $\iota \in \mathtt{dom}(\mathcal{H})$
(b) $\vdash \mathcal{H}$ OK
(c) $\overline{\iota' \rightarrow \{N;\ \overline{\mathtt{f} \rightarrow v}\}} = \{\iota' \rightarrow \{N;\ \overline{\mathtt{f} \rightarrow v}\} \in \mathcal{H} \mid \mathcal{H} \vdash \iota' \preceq \iota^* \}$
(d) $\mathcal{H}(\overline{\iota})$ undefined
(e) $\mathcal{H}' = \overline{\iota \rightarrow [\overline{\iota/\iota'}]\{N;\ \overline{\mathtt{f} \rightarrow v}\}}$

then:
(f) $\mathtt{own}_{\mathcal{H},\mathcal{H}'}([\overline{\iota/\iota'}]\iota) = [\overline{\iota/\iota'}](\mathtt{own}_{\mathcal{H},\mathcal{H}'}(\iota))$

The lemma *Address Substitution preserves Owner Function* is proved by natural deduction with a case analysis for $\iota \in \overline{\iota'}$ and $\iota \notin \overline{\iota'}$. When $\iota \notin \overline{\iota'}$, we require the lemma *Structural Consistency of the Inside Relation* to show that no substitution would occur for the owner.

Lemma: Structural Consistency of the Inside Relation.

If:
(a) $\mathcal{H} \nvdash \iota \preceq \iota'$
(b) $\mathcal{H} \vdash \iota \preceq \circ$

then:
(c) $\mathcal{H} \nvdash \circ \preceq \iota'$

The lemma *Structural Consistency of the Inside Relation* can be shown by contradiction. Let's assume $\mathcal{H} \vdash \circ \preceq \iota'$, then by the transitive property of the inside relation on $\mathcal{H} \vdash \circ \preceq \iota'$ and **(b)**, we can show $\mathcal{H} \vdash \iota \preceq \iota'$, which contradicts **(a)**.

Finally, sheep cloning must preserve acyclicity of the heap's ownership structure. This property is described with a conjecture, we have no formal proof for this property.

Conjecture: Preservation of Ownership Acyclicity for Sheep Cloning.

If:
(a) $\mathcal{H} \nvdash own_{\mathcal{H}}(\iota) \preceq \iota$
(b) $\vdash \mathcal{H}$ OK
(c) $\mathcal{H} \vdash \iota \preceq \iota^*$
(d) $\iota^* \in \mathtt{dom}(\mathcal{H})$
(e) $\overline{\iota' \rightarrow \{N\,;\ \ \overline{\mathtt{f} \rightarrow v}\}} = \{\iota' \rightarrow \{N\,;\ \overline{\mathtt{f} \rightarrow v}\} \in \mathcal{H} \mid \mathcal{H} \vdash \iota' \preceq \iota^*\}$
(f) $\mathcal{H}(\bar{\iota})$ `undefined`
(g) $\mathcal{H}' = \iota \rightarrow [\overline{\iota/\iota'}]\{N\,;\ \ \overline{\mathtt{f} \rightarrow v}\}$

then:
(h) $\mathcal{H}, \mathcal{H}' \nvdash own_{\mathcal{H},\mathcal{H}'}([\overline{\iota/\iota'}]\iota) \preceq [\overline{\iota/\iota'}]\iota$

The ownership structure of a heap is acyclic if no object in the heap has their owner inside themselves (as defined by the dynamic inside relation). Acyclicity of the ownership structure is formally described in Fig. 3.5

of the F-HEAP judgment by the premise: $\mathcal{H} \nvdash own_{\mathcal{H}}(\iota) \preceq \iota$. The function $own_{\mathcal{H}}(\iota)$ retrieves the owner of ι. The sheep cloning semantics of the descripSC formalism consist of three steps. The first step establishes the sub-heap ($\mathcal{H}_1$) containing the representation of the object being cloned. The second step creates an exact duplicate ($\mathcal{H}_2$) of $\mathcal{H}_1$. The third step renames every object in $\mathcal{H}_2$ with a fresh name. The ownership structure of $\mathcal{H}_1$ is acyclic as the heap must always be well-formed prior to sheep cloning. The F-HEAP judgment ensures a well-formed heap has an acyclic ownership structure. Intuitively, the ownership structure of $\mathcal{H}_2$ should also be acyclic. The ownership structure of $\mathcal{H}_2$ is an exact duplicate of the ownership structure in $\mathcal{H}_1$. Furthermore, the renamed $\mathcal{H}_2$ should preserve the structural properties of $\mathcal{H}_1$ as the names used in the renaming are all fresh and unique names with respect to every other object in the heap.

Ownership types as described in the descripSC formalism will always create a heap where its ownership structure is a tree. Ownership types ensure every object has an owner, but only one owner, and the owner of an object cannot change over the lifetime of that object. A tree, by definition, is a connected graph free of cycles [19]. In graph theory, two graphs, G and H with graph vertices V_n, are isomorphic [84], if there is a permutation (bijection function) p of V_n (p: $V(\mathtt{G}) \rightarrow V(\mathtt{H})$) such that any two vertices u and v of G are adjacent in G if and only if $p(u)$ and $p(v)$ are adjacent in H. In accordance with the general principle of isomorphism, the bijection function is also known as an "edge-preserving bijection" [85], whereby the function is a structure-preserving bijection. This means any structural properties, such as acyclicity of edges, of a graph will be preserved in its isomorphism.

Sheep cloning preserves acyclicity of ownership structure, if the ownership structure for the representation of the object being cloned is isomorphic to the ownership structure for the representation of the object's sheep clone. The representations of the sheep clones created in the mapSC formalism are isomorphic to the representations of the objects being cloned.

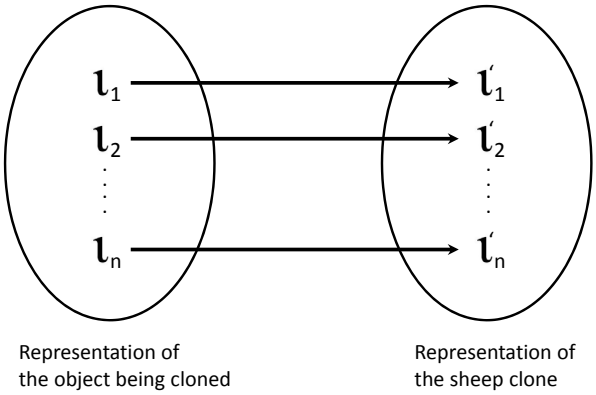


Figure 6.5: Map from the `makeMap` function.

The bijection function p required for isomorphism is described in the `mapSC` formalism by the `map` created from the `makeMap` function. The `map`, as illustrated in Fig. 6.5, is a bijection function that maps the representation of the object being cloned to the representation of that object's sheep clone. In the `descipSC` formalism, the bijection function p is described implicitly by the substitution function ($[\overline{\iota/\iota'}]$) that renames the duplicate representation.

The problem of determining if two graphs are isomorphic is thought to be neither an NP-complete problem nor a P-problem [82, 35], but a complexity class of its own called the graph isomorphism complete (GI-complete). The graph isomorphism (GI) complexity class is define as the set of problems that reduces to the graph isomorphism problem in polynomial time. A problem is considered GI-hard if that problem can be reduced in polynomial time to every problem in GI. Finally, a problem is a GI-complete problem if it is GI-hard and the polynomial time solution for the GI problem is also a polynomial time solution for that problem. An effective technique to determine graph isomorphism between two graphs is to use canonical labeling [74]. There exists several implementations of canonical labeling for graph isomorphism, such as the nauty program and

the Traces program [59]. An interesting future work would be to model the sheep cloning semantics described in the `mapSC` formalism and the `descripSC` formalism in either nauty or Traces.

6.2 Semantics as Specification

The similarities between the proof of type soundness for `descripSC` and our attempt at proving type soundness for `mapSC` leads us to believe that it is possible to use the sheep cloning semantics of `descripSC` as a specification for the more algorithmic sheep cloning semantics in `mapSC`. In this section we explore the possibility of using the sheep cloning semantics of `descipSC` as invariants for the sheep cloning semantics of `mapSC`. In Fig. 6.1, the sheep cloning reduction shows that the `sheep(`ι`)` expression and a heap ($\mathcal{H}$) are reduced to the sheep clone (`[`$\overline{\iota''/\iota'}$`]`$\iota$) of ι and the resulting heap ($\mathcal{H}$, $\mathcal{H}'$) that contains $\mathcal{H}$ along with the new sub-heap that contains the representation of the sheep clone.

The R-SHEEP of `descripSC` contains three premises:

- The first premise uses the inside relation to identify every object inside ι, forming a sub-heap that is the representation of ι.
- The second premise identifies a fresh object for each object in the sub-heap identified in the first premise.
- The third premise describes a new sub-heap that is the sub-heap of the first premise with every object substituted by a new object as described in the second premise.

In Fig. 6.6, we present the sheep cloning reduction of the `mapSC` formalism. The `sheep(`ι`)` expression and a heap ($\mathcal{H}$) are reduced to the sheep clone (ι') of ι and the resulting heap ($\mathcal{H}'$). The representation of the sheep clone is described by the `map` that is created by the `makeMap` function,

and the sub-heap that contains the representation of the sheep clone are constructed by the makeHeap function with the map.

Sheep cloning reduction:

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

(R-SHEEP)

Figure 6.6: Sheep cloning reduction for mapSC.

The R-SHEEP of mapSC also contains three premises:

- The first premise creates the map with the makeMap function on ι.
- The second premise creates the new heap by using the map.
- The third premise retrieves the sheep clones (ι') of ι from within the map.

We believe the descriptive nature of the sheep cloning semantics in descripSC would allow it to be used as a specification for the sheep cloning semantics for mapSC. The first premise of descripSC's R-SHEEP would specify for the domain of the map created by the makeMap function in the first premise of mapSC's R-SHEEP. The domain of the map should be all the objects inside $\bar{\iota}$. The second and third premise descripSC's R-SHEEP would specify for the heap produced by the makeHeap function in the second premise of mapSC's R-SHEEP. Finally, the sheep clone described in descripSC's R-SHEEP would specify the third premise of mapSC's R-SHEEP. Specifying the sheep cloning semantics of the mapSC formalism using the proven sound sheep cloning semantics would lend

a certain validity to the `mapSC` formalism, and in turn any actual implementation (which we would expect to be close to the semantics described in `mapSC`).

6.2.1 Reasoning about Sheep Cloning

Local reasoning [71] is the research field for proving the correctness of computer programs whereby the specifications and proofs of a property mention only portions of memory used by that property, and not the entire global state of the system. Local reasoning can be used to reason about sheep clones as the `descripSC` formalism has shown that sheep cloning is only interested in the portion of the heap that contains the representation of the object being cloned. In this subsection, we explore how *separation logic* [78], a form of local reasoning, can be used to reason about properties of sheep cloning.

Separation logic is an extension of *Hoare logic* [44, 86] with the two new logical connectives: *separating conjunction* and *separating implication*. Separating conjunction has the form $P * Q$ and states that the propositions P and Q hold for disjoint portions of the heap. Separating implication has the form $P \mathrel{-\!\!*} Q$ and states that if the heap is extended with a disjoint portion which holds for the proposition P, then the proposition Q holds for the new larger heap.

$$\frac{\{\texttt{P}\}\mathbf{c}\{\texttt{Q}\}}{\{\texttt{P * R}\}\mathbf{c}\{\texttt{Q * R}\}}$$

Figure 6.7: Frame rule.

In Fig. 6.7, we present the *frame rule*, an inference rule in the form of Hoare triples. The premise of the frame rule states that if `P` is true and the program **c** runs, then `Q` is true in the final state provided that **c** halts. The frame rule allow unaffected regions of the heap (`R`) to remain valid when

proving portions of the heap that are effected by the program. A frame rule is only valid if no free variable in R is modified by the program **c**.

$$\frac{\{\iota \to \{...\}\}\ \texttt{sheep}(\iota)\ \{\iota \to \{...\},\ \iota' \to \{...\}\}}{\{\iota \to \{...\}\ *\ \mathcal{H}\}\ \texttt{sheep}(\iota)\ \{\iota \to \{...\},\ \iota' \to \{...\}\ *\ \mathcal{H}\}}$$

Figure 6.8: Frame rule example with sheep cloning.

In Fig. 6.8, we present an example illustrating how the frame rule could be used to prove properties about sheep cloning. The premise of the frame rule in Fig. 6.8 describes a scenario where sheep cloning is performed on the only object (ι) in the heap. Sheep cloning ι creates a heap containing two objects, ι and its sheep clone (ι'). The conclusion of the frame rule shows that any property of a heap ($\mathcal{H}$) that is disjoint from ι would be preserved after sheep cloning ι. More importantly the frame rule shows that if it is possible to clearly identify the portion of the heap that sheep cloning affects, then it is possible to state that the unaffected portions of the heap remains valid after sheep cloning. If we abstract ι to be an ownership representation instead of a single object, then we can prove properties about sheep cloning using separation logic without any regard for the objects that are not inside the representation of the object being cloned and the representation of the sheep clone. It is important to recognise that separation logic cannot be used to create sheep clones, separation logic is merely a method in which sheep cloning can be reasoned.

6.3 Implementing Sheep Cloning

Sheep cloning is a convenient and natural way to clone objects in systems with ownership types. The descripSC formalism shows how sheep clones can be easily created once the ownership representation of the object being cloned is determined, whereby that representation is duplicated

and renamed. The method in which the objects in the ownership representation is duplicated is an implementation detail and not of great interest to our formalism, however, the method in which the ownership representation is represented and subsequently identified is important to our formalism. Sheep cloning is not restricted to systems with the owners-as-dominators perspective ownership policy, and in section 8.2.1 we discuss in detail sheep cloning in systems with weaker prescriptive ownership policies. It is also possible to perform sheep cloning in systems with ownership types that enforces no prescriptive ownership policies, whereby ownership types are purely descriptive and does not restrict aliasing, accessing, or mutation on the objects in the system. A system with descriptive ownership only ensures every object in the system has an owner. The heap of a system with descriptive ownership would still have the same tree-like structure as those systems with ownership types that enforces prescriptive policies. The primary requirement for a system to perform sheep cloning is that the ownership representation of the object being cloned must be able to be identified. The `descipSC` formalism uses the inside relation to identify the ownership representation of its objects, and the inside relation can be defined in systems with descriptive ownership.

Sheep cloning as described in the `descipSC` formalism will always create sheep clones that preserves the prescriptive ownership policy of the system. The reason for this is very simple, the representation of the sheep clone is a duplication of a representation that already enforces the prescriptive policy of the system. So long as the ownership representation of the object being cloned enforces the perspective policy, then any exact duplication of that representation will enforce the same perspective policy. The sheep cloning semantics in the `descipSC` formalism does not introduce any

We implemented sheep cloning in `Hopper` [48], an interpreter written in `Javascript` [1] for the programming language `Grace` [4]. `Grace` can form simply descriptive ownership structures over objects using the

`owned` annotation.

The `owned` annotation can only be denoted on fields in an object's constructor assigned with the object creation expression. The `owned` annotation is similar to the ownership context `this`, objects with the `owned` annotation are owned by the object created from that object constructor. The ownership structure created with the `owned` annotation is free from cycles as only new objects can be owned.

```
def compositeFigure = object {
 def imageFigure is owned = object {
  def image = object { ... }
   ...
  }
  def textAreaFigure is owned = object { ... }
 ...
}

def cloneComposite = compositeFigure.sheepClone

print(compositeFigure == cloneComposite)        // false
print(compositeFigure.imageFigure ==
       cloneComposite.imageFigure)              // false
print(compositeFigure.textAreaFigure ==
          cloneComposite.textAreaFigure)        // false
print(compositeFigure.imageFigure.image ==
             cloneComposite.imageFigure.image)  // true
```

Figure 6.9: owned annotation.

The `Grace` code in Fig. 6.9 is an example of performing sheep cloning in `Hopper`. The example consists of a composite figure object (`composit-`

eFigure) with an image object (imageFigure) and a text figure object (textAreaFigure). Both imageFigure and textAreaFigure are annotated with owned in compositeFigure. The imageFigure object contains an image object that is not owned by imageFigure, but simply displays the image referenced by image. There are four identity tests at the bottom of the example. The first test returns false as the compositeFigure object is compared against its sheep clone (cloneComposite). The second test returns false as the imageFigure of compositeFigure is compared against the imageFigure of cloneComposite. The third test returns false as the textAreaFigure of compositeFigure is compared against the textAreaFigure of cloneComposite. The fourth test returns true when comparing the image object in the textAreaFigure of compositeFigure against the image object in the textAreaFigure of cloneComposite. The image object of compositeFigure and cloneComposite are both referencing the same image.

```
addMethod(owned, "annotateDef()", function (def) {
  if(!(def.mode.value instanceof ast.ObjectConstructor)){
   throw new Error("Owned annotation can only
                              be placed on new object.");
  }
   ...
 }
);
```

Figure 6.10: owned annotation.

The Javascript code in Fig. 6.10 is the implementation of the owned annotation in Hopper. The addMethod method takes three arguments: the name of the annotation (owned), the type of the annotation, and a function to describe the functionality of the annotation. The owned anno-

tation is purely descriptive and only needs to ensure it is denoted over a newly created object. This is described by the `if` statement which ensures the `value` with the annotation is an `ObjectConstructor` ast node, else an error is thrown.

The `Javascript` code in Fig. 6.11 implements sheep cloning in `Hopper` through the `sheepClone` function. The `sheepClone` function is a default method on `Grace` objects. The `delegate` function returns a fresh copy of the object passed into the function. The object calling the `sheepClone` function is copied with the statement `delegate(this)`. If the object is marked with the `owned` annotation then the `if` statement over `field.isOwned` ensures the `sheepClone` function is recursively called on every field of that object.

```
function delegate(object) {
 function Clone() {}
 Clone.prototype = object;
 return new Clone();
}

addMethod(GraceObject, "sheepClone", 0, function () {
 var clone, cloneField, field,
  ...
  clone = delegate(this);

  if(field.isOwned){
   cloneField = rt.method(field.identifier, 0, function () {
    return cloneField.value;
   });
 ...

   return field.value.sheepClone().then(function (value) {
    cloneField.value = value;
   });
 ...
 }

 return clone;
});
```

Figure 6.11: owned annotation.

6.4 Critique of descripSC

In hindsight, the declarative semantics described in the descripSC formalism may appear to be the "obvious" formalisation for the semantics of sheep cloning. We would not have reached this conclusion, however, without first creating the two previous formalisms. We believe that it is possible to prove type soundness for recurSC and mapSC, however, their proofs would not be nearly as tidy or as concise as that of descripSC.

The lack of a proof for the conjecture that states ownership acyclicity for sheep clones is disappointing. The conjecture does a serviceable job of explaining on a higher level why sheep clones' ownership structures cannot be cyclic. We have attempted to prove this property as a lemma several different ways. The main difficultly is in showing the object addresses used to rename the sub-heap of the sheep clone are not inside the object being cloned, under both the original heap and the new sub-heap.

The descipSC formalism can creates sheep clones that contain dead objects, if the representation of the object being cloned contains dead objects. The representation of the sheep clone is created by copying and renaming the objects inside the representation of object being cloned regardless if any of those objects are dead or alive.

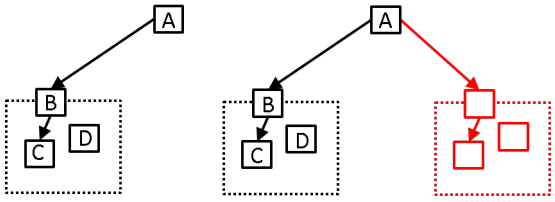


Figure 6.12: Sheep clones with dead objects.

In Fig. 6.12, we present an example of an object whose sheep clone contains a dead object. The square boxes are objects, and this example consists of four objects: A, B, C, and D. The dotted line represents the ownership representation of the object on the top edge, and the arrows are the

references. The objects in red are the sheep clone of object B created by object A. Object D is a dead object and it is also inside the ownership representation of object B. When object B is being sheep cloned, object D will be copied into the representation of the sheep clone.

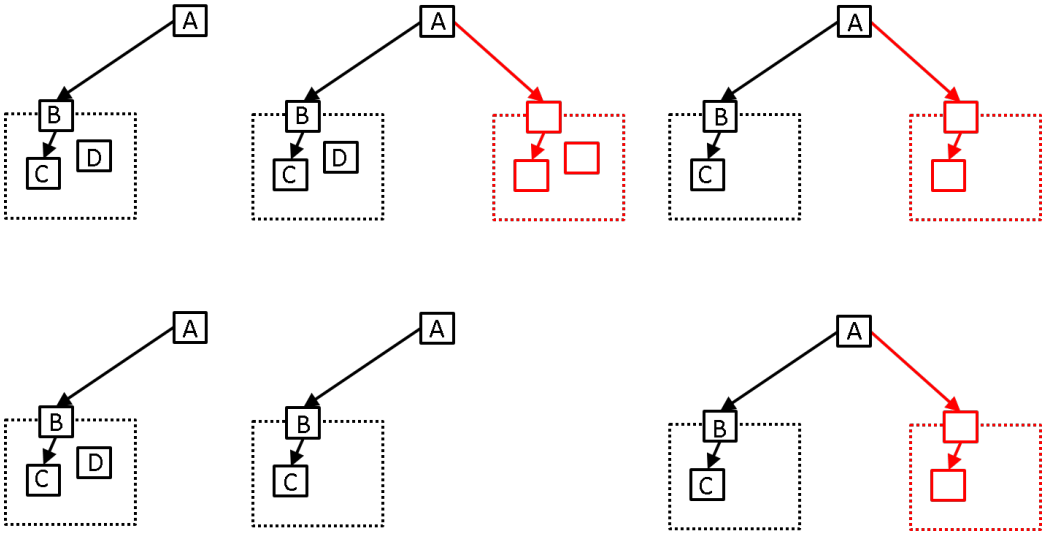


Figure 6.13: Eliminating dead objects in sheep clones.

The dead objects in the representation of the sheep clone can be eliminated without altering the sheep cloning semantics using garbage collection. In Fig. 6.13, we show how garbage collection can eliminate the dead object (object D) in the sheep clone presented in Fig. 6.12. The top images of Fig. 6.13 show when garbage collection occurs after the sheep cloning object B. Sheep cloning object B will copy object D into the sheep clone, and then the garbage collection will remove both object D and its copy in the sheep clone. The bottom images show when garbage collection occurs before the sheep cloning. In this scenario object D will not be copied into the sheep clone as it does not exist when sheep cloning on object B is called.

Alternatives to garbage collection is to to introduce concepts such as reachability into the sheep cloning semantics. The semantics would ensure that the objects being copied must be (transitively) reachable from

the object being cloned as well as being inside the representation of the object being cloned. The issue of dead objects in sheep clones is further discussed in section 8.2.1, where we introduce sheep cloning to systems with weaker prescriptive ownership policies.

6.5 Chapter Summary

In this chapter, we present descripSC, a formalism of the sheep cloning semantics in a descriptive style. The sheep cloning semantics in descripSC describes the sheep clone by using properties that are described over the entire heap. These properties will identify the sub-heap which contains the object structure that needs to be copied. The objects that make up the sheep clone are obtained by renaming the copy of the previously identified object structure. We present our proof of type soundness, along with a conjecture that states sheep cloning is acyclic. The details for the proof of type soundness for descripSC are presented in appendix A.

Chapter 7

Multiple Ownership

Multiple ownership lifts the restriction of hierarchies by allowing objects to have multiple owners and consequently allowing the heap to be described by a directed acyclic graph (DAG). MOJO [16] was the first multiple ownership system to have been created and it is not without its flaws. The variant types in MOJO is very restrictive, whereby classes cannot be agnostic as to whether they support single or multiple ownership. There is also no support for ownership-based encapsulation in MOJO, and the complexity of its formalism makes proving MOJO very difficult.

In this chapter, we present Mojo-jojo, a multiple ownership formalism that succeeds MOJO. Mojo-jojo constrains owner parameters similar to MOJO, however, the constraint system of Mojo-jojo is based on set algebra and therefore is more expressive. Mojo-jojo features generics and existential types, which does not exists in MOJO. The goal of Mojo-jojo is to be a lightweight and concise formalism of a multiple ownership system. Features that are unnecessary to multiple ownership are mostly excluded from Mojo-jojo. For example, Mojo-jojo does not have an effect system, but if an effect system is required of Mojo-jojo then an effect system can easily be incorporated into Mojo-jojo through a permissions system. Similarly, Mojo-jojo supports only descriptive ownership, a permissions system could also be used to describe prescriptive ownership policies such as

owners-as-dominators.

If an object is owned by multiple objects then we imagine that it is in the *intersection* (as in standard set theory) of those objects' contexts. We write `a∩b:Record` for the type of records owned by `a` *and* `b`. In MOJO, any number of contexts (including zero) may be intersected together to make a single context.

The code example below uses MOJO's less expressive wildcard notation, rather than the explicit quantification we present in Mojo-jojo.

```
class Doctor {
    this:List<this∩?:Record> recordList;
}

class Record {
    ?:Patient p;
    this:Object data;
}
```

A Doctor shares ownership over the records of the patients he/she treats with other unknown objects, possibly other doctors who treat the same patient. In the `Record` class the owner of patient `p` is unknown and is represented with a wildcard (quantified context). Since a patient may be treated by one or many doctors and the other owners are unknown, a patient is owned by '`this`' intersected with another wildcard. Given the `Doctor` and `Record` classes, we present an example where a Doctor object named doc is owned by a hospital box, and we attempt to a Record into the collection of doc's Records. The first two additions are legal, the second two illegal as neither include `doc` in the owner.

```
    hospital:Doctor doc = new hospital:Doctor();
    doc.recordList.add(new doc:Record());
    doc.recordList.add(
        new (doc∩otherDoc∩anotherDoc):Record());
```

```
//doc.recordList.add(new ∅:Record());
//doc.recordList.add(
//  new (otherDoc∩anotherDoc):Record());
```

In order to support multiple ownership, the type system needs a little more information from the programmer about contexts. The programmer must specify which combinations of contexts may intersect, which are known to be disjoint, and which are neither. If two contexts are declared to intersect, then their intersection can be used as a context parameter; if two contexts are not known to intersect, then their intersection cannot be used as a context parameter. Disjointness information is used to calculate which effects are disjoint.

Although MOJO is a useful improvement over standard, single-owner ownership types it has several flaws: its support for variant types is restrictive and means that classes can not be agnostic as to whether they support single or multiple ownership, there is no support for ownership-based encapsulation, and the formalisation is complex.

7.1 Mojo-jojo

In this section, we discuss two key features of Mojo-jojo, and present the formalism of Mojo-jojo. The two features are algebraic constraints for the topology of Mojo-jojo's ownership structure, and existential quantification of the contexts and types.

7.1.1 Constraint System

Constraint systems for multiple ownership were first introduced by MOJO. The constraint system in MOJO only supports constraints over contexts, specifying how contexts intersect or are disjoint. If two contexts are declared to intersect, then the intersection of these two contexts can be used

as a context parameter itself. The converse is also true: so if two contexts are not known to intersect then their intersection cannot be used as a context parameter. The disjointness constraints allow the effect system to identify when contexts have no effect on each other.

The constraint system in MOJO provides the basis for the constraint system in Mojo-jojo. The constraint system in Mojo-jojo allows constraints to express relationships between boxes, and therefore, the constraints are over the topology of the boxes, not objects.

A box is an abstraction for a group of objects. The heap in Mojo-jojo is composed of objects in boxes. A box may be associated with an object (the owner) or a composition of boxes. If `a` and `b` are variables in our program, we use `a` and `b` to denote the boxes they own. In Mojo-jojo, $\texttt{a} \cap \texttt{b}$ denotes the intersection of the two boxes. Objects in the box $\texttt{a} \cap \texttt{b}$ are owned by both `a` and `b`. We also allow the union of two boxes $\texttt{a} \cup \texttt{b}$, objects in the box $\texttt{a} \cup \texttt{b}$ are owned by either `a` or `b`.

The syntax $\texttt{a} \cap \texttt{b} \neq \emptyset$ and $\texttt{a} \cap \texttt{b} = \emptyset$ describes the constraints on the intersection between boxes in Mojo-jojo. Our constraints also extend to unions between boxes and sub-boxing.

The sub-boxing constraints in Mojo-jojo describe whether a box is inside or outside another box, as in other ownership systems [20, 25]. Our constraint system follows the rules of set algebra: the constraints on the boxes follow the same laws as subsets and set equalities, including the commutative laws, the associative laws, and the distributive laws.

The close relationship between standard set theory and our constraint system is one of the important improvements of Mojo-jojo over MOJO.

The relationship between ownership and set algebra based constraints is not always intuitive. The constraints do not guarantee an object is present in any particular box, only that an object is allowed to be in that box. For example, in most ownership systems when `a` is 'inside' `b` then the objects in `a` are not considered to be in `b` [20], which means the intersection of `a` and `b` would be empty. However, our constraints describe the ownership

topology, and not the object topology of the heap. The constraints follow set theory, and therefore, the intersection of `a` and `b` is non-empty when `a` is inside `b`.

7.1.2 Generics and Existential Owners

In multiple ownership, variant owners are used to refer to an object owned by both known and unknown owners [14]. In MOJO, these unknown owners are denoted with a wildcard: `?`. Unfortunately, this syntax has its drawbacks. Classes cannot be polymorphic in the variance of their owners. For example, a list of objects with known owners must have a separate class definition from a list of objects with partially known owners [16].

In Jo$\exists$ [14], existential quantification of owners is used to represent unknown and variant contexts. We adapt this quantification to multiple owners by using existentially quantified owners to hide a combination of contexts. In Mojo-jojo the inside and outside bounds on the traditional single ownership hierarchy are not flexible enough for the directed acyclic graph (DAG) of multiple ownership, therefore we use constraints as bounds for existential types. The constraints describe where a quantified context appears in the DAG.

A benefit of existential quantification is a cleaner formalisation. The well-known concepts [73] of packing and unpacking are used implicitly to implement the restricted access to quantified variables. Implicit packing and unpacking follows the formalisations of Java wildcards [15], which is a contrast to the explicit packing and unpacking in traditional existential types formalisations [61]. In MOJO, the same restrictions required a complex system of strict lookups and ad hoc type transformations.

In our `Doctor` and `Record` example, we could add the following constraints to the existential type of the record to ensure that only objects which are inside the current object's owner may share in owning each record, and that these other objects must be allowed to overlap with the

current object:

```
this:List<∃o;(o⊆owner,o∩this≠∅).(this∩o):Record>
 recordList;
```

We add generics (parametric types) to Mojo-jojo, not for the usual benefits of more precise and flexible types (although these are welcome), but because, by combining generics with existential quantification, our classes are variance-polymorphic. The result of this is that class declarations for single and multiple ownership systems are identical and therefore classes written for single ownership can be reused in Mojo-jojo.

Given a simple `List` declaration in Mojo-jojo, we can write the following types (in each case, the list itself is owned by `a`):

`a:List<this:Record>`	list of records owned by `this`
`∃o.a:List<o:Record>`	list of records owned by a single unknown owner
`a:List<∃o.o:Record>`	list of records owned by several different unknown owners

In MOJO, we would need two list classes, one which handled the first case and one which handled the third; the second case could not be encoded at all.

7.2 Formalism for Mojo-jojo

In this section, we present the formalism of Mojo-jojo, a calculus in the tradition of Featherweight Java [46].

Q	::=	`class C<`$\overline{\texttt{X}}$ $\overline{\mathcal{C}}$`>` `{`$\overline{T\,\texttt{f;}}$ $\overline{M}$`}`	*class declarations*
M	::=	T `m(`T `x)` $\overline{\mathcal{C}}$ `{return` e`;}`	*method declarations*
a	::=	γ | `world` | $\emptyset$ | T`.owner`	*contexts*
b	::=	a | $b \cap b$ | $b \cup b$	*boxes*
r	::=	ι | $r \cap r$ | $r \cup r$	*runtime boxes*
$\mathcal{C}$	::=	$b \subseteq b$ | $b = \emptyset$ | $b \neq \emptyset$	*constraints*
N	::=	b`:C<`$\overline{T}$`>`	*class types*
T	::=	$\exists\emptyset\texttt{;}\emptyset\texttt{.X}$ | $\exists\overline{o}\texttt{;}\overline{\mathcal{C}}.N$ | $\top$	*types*
R	::=	$\exists\emptyset\texttt{;}\emptyset\texttt{.}r\texttt{:C<}\overline{T}\texttt{>}$	*runtime types*
e	::=	`null` | γ | γ`.f` | γ`.f =` e | γ`.m(`e`)` | `new` b`:C<`$\overline{T}$`>` | `err`	*expressions*
v	::=	`null` | ι	*values*
γ	::=	`x` | ι	*expression variables and addresses*
Γ	::=	$\overline{\gamma\texttt{:}T}$	*environments*
Δ	::=	$\overline{\mathcal{C}}$	*constraint environments*
$\mathcal{H}$	::=	$\overline{\iota \rightarrow \{R,\ \overline{\texttt{f}\rightarrow v}\}}$	*heaps*
`x,o,owner,this`			*variables*
`X`			*type variables*
`f`			*field names*
`m`			*method names*
`C`			*class names*

Figure 7.1: Syntax of Mojo-jojo.

7.2.1 Syntax of Mojo-jojo

Fig. 7.1 presents the syntax for Mojo-jojo. Constraints ($\mathcal{C}$) in class declarations and existential types are associated with formal context variables ($\mathtt{o}$). All class types are generic and existential. Non-existential types can be simulated by quantifying with the empty set of variables and constraints. Type variables (X) are always quantified by empty lists of variables and constraints; they are quantified for consistency with class types. We use a top type ($\top$) for variables that are only used as owners and are therefore not associated with a type. Syntax in grey cannot be written by the programmer and is used to represent running programs.

Context parameters (a) denote the owners in Mojo-jojo, they include: the variables ($\mathtt{x}$), `this`, and `owner`; the default owners `world`, T`.owner`, and $\emptyset$. T`.owner` denotes the owner of the type T. The boxes (b) in Mojo-jojo are: context parameters; intersections of boxes; or unions of boxes. Constraints ($\mathcal{C}$) in Mojo-jojo are described over boxes, and they include: sub-boxing; empty boxes, and non-empty boxes. The expressions (e) in Mojo-jojo are: `null`; variables; field look-up; field assignment; method invocation; and object creation.

Mojo-jojo judgments are decided under three environments: Γ maps variables to their types, Δ stores the current constraints on contexts, and $\overline{\mathtt{X}}$ is a list of type variables currently in scope (we do not support bounds on type variables). Variables in Γ may be either expression variables ($\mathtt{x}$) or context variables ($\mathtt{o}$, which always have type $\top$); the latter only appear as a result of unpacking existential types and are not used in expressions. There is, however, no syntactic distinction between $\mathtt{x}$ and $\mathtt{o}$. At runtime, Γ maps addresses (ι) to their types.

Mojo-jojo is not Turing complete, for example, Mojo-jojo cannot encode conditionals as it does not support inheritance or first class functions. Mojo-jojo is proposed to be a formalisation of the interesting aspects of a multiple ownership language, while keeping the formalism as small as possible. We do not consider Turing completeness to be crucial for Mojo-

jojo, as type safety is orthogonal to Turing completeness and we do not intend programs to be written with Mojo-jojo.

7.2.2 Auxiliary Functions of Mojo-jojo

Fig. 7.2 presents the auxiliary functions used in this formalism. The *fields* function returns the fields of class `C`. The *fType* function gives the type for the field $\texttt{f}_i$, in the class `C`.

$$\frac{\texttt{class C<}\overline{\texttt{X}}\ \overline{\mathcal{C}}\texttt{>}\ \{\overline{U\ \texttt{f;}}\ \overline{M}\}}{\mathit{fields}(\texttt{C}) = \overline{\texttt{f}}}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}}\ \overline{\mathcal{C}}\texttt{>}\ \{\overline{U\ \texttt{f;}}\ \overline{M}\}}{\mathit{fType}(\texttt{f}_i, b\texttt{:C<}\overline{T}\texttt{>}) = \texttt{[}b\texttt{/owner,}\overline{T\texttt{/X}}\texttt{]}U_i}$$

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}}\ \overline{\mathcal{C}}\texttt{>}\ \{\overline{U\ \texttt{f;}}\ \overline{M}\} \\ T\ \texttt{m(}T'\ \texttt{x)}\ \ \overline{\mathcal{C}''}\ \ \{\texttt{return}\ \ e\texttt{;}\} \in \overline{M}\end{array}}{\mathit{mBody}(\texttt{m}, b\texttt{:C<}\overline{T}\texttt{>}) = (\texttt{x; [}b\texttt{/owner,}\overline{T\texttt{/X}}\texttt{]}e)}$$

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}}\ \overline{\mathcal{C}}\texttt{>}\ \{\overline{U'\ \texttt{f;}}\ \overline{M}\} \\ T\ \texttt{m(}T'\ \texttt{x)}\ \ \overline{\mathcal{C}''}\ \ \{\texttt{return}\ \ e\texttt{;}\} \in \overline{M}\end{array}}{\mathit{mType}(\texttt{m}, b\texttt{:C<}\overline{U}\texttt{>}) = \texttt{[}b\texttt{/owner,}\overline{U\texttt{/X}}\texttt{]}(\overline{\mathcal{C}''}.T' \to T)}$$

$$\frac{}{\Downarrow_{\overline{\texttt{o}};\overline{\mathcal{C}}}\ \exists\emptyset;\emptyset\texttt{.X} = \exists\emptyset;\emptyset\texttt{.X}}$$

$$\frac{}{\Downarrow_{\overline{\texttt{o}};\overline{\mathcal{C}}}\ \exists\overline{\texttt{o}'};\overline{\mathcal{C}'}.N = \exists\overline{\texttt{o}}, \overline{\texttt{o}'};\overline{\mathcal{C}}, \overline{\mathcal{C}'}.N}$$

$$\frac{}{\Downarrow_{\overline{\texttt{o}};\overline{\mathcal{C}}}\ \top = \top}$$

Figure 7.2: Auxiliary functions for Mojo-jojo.

The *mBody* function returns the argument and return expression of method `m` in class `C`. The *mType* function returns the argument type and return type of method `m` in class `C`. The $\Downarrow$ (close) operation denotes the packing operation where variables and constraints that are required to be packed ($\overline{o}$ and $\overline{\mathcal{C}}$ in $\Downarrow_{\overline{o};\overline{\mathcal{C}}}$) are added to those quantifying a class type or ignored in the case of type variables, as type variables have no free expression variables.

7.2.3 Sub-boxes of Mojo-jojo

Fig. 7.3 presents the sub-boxing relationship for boxes. These rules follow directly from set theory as sub-boxing corresponds to the subset relation. We added B-ENV and B-OWNER to take into account relations declared by the programmer, and that object's contexts are within their owner's context.

Fig. 7.4 presents the equality relation axioms over boxes. The equality relation axioms form a bounded distributive lattice. The algebraic structure of a set is a lattice if the two binary operations $\cap$ and $\cup$ over the set follows the axiomatic identities of the commutative laws, the associative laws, and the absorption laws. The lattice is a distributive if it follows for the distributive laws, and furthermore the lattice is bounded if it also follows the identity laws. For the equality relation axioms, the $\emptyset$ is the lattice's bottom and the `world` context is the lattice's top. A lattice is considered complete if every subset of the partially ordered set (i.e. boxes) has both a greatest lower bound (the infimum) and a least upper bound (the supremum). The equality relation axioms is a complete lattice, for any power set of a given set of boxes the supremum is given by the union and the infimum by the intersection of subsets.

A key concept in Mojo-jojo is the constraint ($\mathcal{C}$). Constraints may be assumed within the body of the class, method, or unpacked scope, and must be satisfied when the class is instantiated, method invoked, or exis-

tential type packed. Constraints are established to be valid using the rules in Fig. 7.5. These rules describe how topological constraints can be valid under the sub-box and equality rules, declarations, and some intuitive notions about the constraints. Note that for a box to be judged equal or not equal to the empty set ($\Delta; \Gamma; \overline{\mathtt{X}} \models \mathtt{a} = \emptyset$) is different from box-equality with the empty set ($\mathtt{a} = \emptyset$). For $\Delta; \Gamma; \overline{\mathtt{X}} \models \mathtt{a} = \emptyset$, the C-SB-E judgment states that the constraint $\mathtt{a} = \emptyset$ can be proved under the environments Δ, Γ, and $\overline{\mathtt{X}}$, while $\mathtt{a} = \emptyset$ states that $\mathtt{a}$ equals $\emptyset$ using the set equality rules in Fig. 7.4.

It is important to recognise that a well-formed constraint does not mean the constraint is necessarily valid. Well-formedness is a purely syntactic judgment, which checks mainly that component variables are in scope. Validity is a semantic judgment which checks that a well-formed constraint can be proved from the relevant environments.

Sub-boxes: $\boxed{\Delta; \Gamma; \overline{\mathrm{X}} \vdash b \subseteq b}$

$$\frac{}{\Delta; \Gamma; \overline{\mathrm{X}} \vdash b \subseteq b}$$
(B-REFLEX)

$$\frac{}{\Delta; \Gamma; \overline{\mathrm{X}} \vdash b \subseteq \texttt{world}}$$
(B-TOP)

$$\frac{\Delta; \Gamma; \overline{\mathrm{X}} \vdash b_1 \subseteq b_2 \quad \Delta; \Gamma; \overline{\mathrm{X}} \vdash b_2 \subseteq b_3}{\Delta; \Gamma; \overline{\mathrm{X}} \vdash b_1 \subseteq b_3}$$
(B-TRANS)

$$\frac{b_1 \subseteq b_2 \in \Delta}{\Delta; \Gamma; \overline{\mathrm{X}} \vdash b_1 \subseteq b_2}$$
(B-ENV)

$$\frac{}{\Delta; \Gamma; \overline{\mathrm{X}} \vdash b_1 \subseteq b_1 \cup b_2}$$
(B-JOIN-1)

$$\frac{\Delta; \Gamma; \overline{\mathrm{X}} \vdash b_1 \subseteq b_3 \quad \Delta; \Gamma; \overline{\mathrm{X}} \vdash b_2 \subseteq b_3}{\Delta; \Gamma; \overline{\mathrm{X}} \vdash b_1 \cup b_2 \subseteq b_3}$$
(B-JOIN-2)

$$\frac{}{\Delta; \Gamma; \overline{\mathrm{X}} \vdash b_1 \cap b_2 \subseteq b_1}$$
(B-MEET-1)

$$\frac{\Delta; \Gamma; \overline{\mathrm{X}} \vdash b_1 \subseteq b_2 \quad \Delta; \Gamma; \overline{\mathrm{X}} \vdash b_1 \subseteq b_3}{\Delta; \Gamma; \overline{\mathrm{X}} \vdash b_1 \subseteq b_2 \cap b_3}$$
(B-MEET-2)

$$\frac{\Gamma(\gamma) = b\texttt{:C<}\overline{\mathrm{T}}\texttt{>}}{\Delta; \Gamma; \overline{\mathrm{X}} \vdash \gamma \subseteq b}$$
(B-OWNER)

$$\frac{}{\Delta; \Gamma; \overline{\mathrm{X}} \vdash \emptyset \subseteq b}$$
(B-EMPTY)

Figure 7.3: Mojo-jojo sub-boxing between boxes.

Equalities: $\boxed{b = b}$

$b_1 \cap b_2 = b_2 \cap b_1$	EQ-COMM-I
$b_1 \cup b_2 = b_2 \cup b_1$	EQ-COMM-U
$b_1 \cap (b_2 \cap b_3) = (b_1 \cap b_2) \cap b_3$	EQ-ASSOC-I
$b_1 \cup (b_2 \cup b_3) = (b_1 \cup b_2) \cup b_3$	EQ-ASSOC-U
$b_1 \cap (b_2 \cup b_3) = (b_1 \cap b_2) \cup (b_1 \cap b_3)$	EQ-DISTRIB-I
$b_1 \cup (b_2 \cap b_3) = (b_1 \cup b_2) \cap (b_1 \cup b_3)$	EQ-DISTRIB-U
$b \cap \texttt{world} = b$	EQ-ID-I
$b \cup \emptyset = b$	EQ-ID-U
$b \cap b = b$	EQ-IDEM-I
$b \cup b = b$	EQ-IDEM-U
$b \cap \emptyset = \emptyset$	EQ-DOM-I
$b \cup \texttt{world} = \texttt{world}$	EQ-DOM-U
$b_1 \cap (b_1 \cup b_2) = b_1$	EQ-ABS-I
$b_1 \cup (b_1 \cap b_2) = b_1$	EQ-ABS-U
$b\texttt{:C<}\overline{T}\texttt{>.owner} = b$	EQ-OWNER

Figure 7.4: Mojo-jojo equalities between boxes.

Valid constraints: $\boxed{\Gamma \models \mathcal{C}}$

$$\frac{\mathcal{C} \in \Delta}{\Gamma \models \mathcal{C}} \text{(C-ENV)} \qquad \frac{\Gamma \models \mathcal{C}' \qquad \mathcal{C}' = \mathcal{C}}{\Gamma \models \mathcal{C}} \text{(C-EQ)}$$

$$\frac{\Gamma \models b' = \emptyset \qquad \Delta; \Gamma; \overline{X} \vdash b \subseteq b'}{\Gamma \models b = \emptyset} \text{(C-SB-E)} \qquad \frac{\Gamma \models b' \neq \emptyset \qquad \Delta; \Gamma; \overline{X} \vdash b' \subseteq b}{\Gamma \models b \neq \emptyset} \text{(C-SB-NE)}$$

$$\frac{\Delta; \Gamma; \overline{X} \vdash b_1 \subseteq b_2}{\Gamma \models b_1 \subseteq b_2} \text{(C-SB)}$$

Figure 7.5: Mojo-jojo valid constraints.

7.2.4 Expression Typing of Mojo-jojo

Expression typing is defined in Fig. 7.6. In T-FIELD, T-ASSIGN, and T-INVK the type of the receiver is unpacked before performing a field or method lookup. The resulting type may contain free variables, and if this type forms part of the conclusion, then it must be packed to prevent free variable escape. In the premises of the type rules (intuitively, between unpacking and repacking) we must be careful to use correct, enlarged environments which include the unpacked context variables. Ownership types are dependent types and `this` can be used in types, and constructing runtime types involves substituting the receiver for `this` in types, thus we restrict receivers to be variables. This is not a problem in practice because methods can be used like a let expression to assign any expression to a variable, which can then be used as receiver. Apart from the packing and unpacking operations, the expressions rules for Mojo-jojo are identical to those in the `Core` formalism defined in Chapter 3.

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

$$\frac{\mathcal{E};\Gamma \vdash N \text{ OK}}{\mathcal{E};\Gamma \vdash \texttt{new}\ \ N : N}\ (\text{T-NEW}) \qquad \frac{}{\mathcal{E};\Gamma \vdash \gamma : \Gamma(\gamma)}\ (\text{T-VAR}) \qquad \frac{\mathcal{E};\Gamma \vdash T \text{ OK}}{\mathcal{E};\Gamma \vdash \texttt{null} : T}\ (\text{T-NULL})$$

$$\frac{\begin{array}{c}\mathcal{E};\Gamma \vdash \gamma : \exists\overline{o};\overline{\mathcal{C}}\,.\,b\texttt{:C<}\overline{\texttt{T}}\texttt{>} \\ fType(\texttt{f}, b\texttt{:C<}\overline{\texttt{T}}\texttt{>}) = T\end{array}}{\mathcal{E};\Gamma \vdash \gamma\texttt{.f} :\Downarrow_{\overline{o};\overline{\mathcal{C}},\gamma\subseteq b}\ \texttt{[}\gamma\texttt{/this]}T}\ (\text{T-FIELD})$$

$$\frac{\begin{array}{c}\mathcal{E};\Gamma \vdash \gamma : \exists\overline{o};\overline{\mathcal{C}}.b\texttt{:C<}\overline{\texttt{T}}\texttt{>} \qquad fType(\texttt{f}, b\texttt{:C<}\overline{\texttt{T}}\texttt{>}) = T' \\ \mathcal{E};\Gamma \vdash e : T \qquad \Delta, \overline{\mathcal{C}}, \gamma \subseteq b; \Gamma, \overline{\texttt{o}\texttt{:}\top}; \overline{\texttt{X}} \vdash T <: \texttt{[}\gamma\texttt{/this]}T'\end{array}}{\mathcal{E};\Gamma \vdash \gamma\texttt{.f = } e : T}\ (\text{T-ASSIGN})$$

$$\frac{\begin{array}{c}\mathcal{E};\Gamma \vdash \gamma : \exists\overline{o};\overline{\mathcal{C}}.b\texttt{:C<}\overline{\texttt{T}}\texttt{>} \qquad \mathcal{E};\Gamma \vdash e : T'' \\ mType(\texttt{m}, b\texttt{:C<}\overline{\texttt{T}}\texttt{>}) = \overline{\mathcal{C}'}.T' \to\ T \\ \Delta, \overline{\mathcal{C}}, \gamma \subseteq b; \Gamma, \overline{\texttt{o}\texttt{:}\top}; \overline{\texttt{X}} \vdash T'' <: \texttt{[}\gamma\texttt{/this]}T' \\ \Delta, \overline{\mathcal{C}}, \gamma \subseteq b; \Gamma, \overline{\texttt{o}\texttt{:}\top}; \overline{\texttt{X}} \models \texttt{[}\gamma\texttt{/this]}\overline{\mathcal{C}'}\end{array}}{\mathcal{E};\Gamma \vdash \gamma\texttt{.m(}e\texttt{)} :\Downarrow_{\overline{o};\overline{\mathcal{C}},\gamma\subseteq b}\ \texttt{[}\gamma\texttt{/this]}T}\ (\text{T-INVK})$$

$$\frac{\begin{array}{c}\mathcal{E};\Gamma \vdash e : T' \\ \mathcal{E};\Gamma \vdash T' <: T \qquad \mathcal{E};\Gamma \vdash T \text{ OK}\end{array}}{\mathcal{E};\Gamma \vdash e : T}\ (\text{T-SUBS})$$

Figure 7.6: Mojo-jojo expression typing rules.

7.2.5 Subtyping of Mojo-jojo

Subtyping: $\boxed{\mathcal{E}; \Gamma \vdash T <: T}$

$$\frac{}{\mathcal{E}; \Gamma \vdash T <: T} \text{(S-REFLEX)} \qquad \frac{}{\mathcal{E}; \Gamma \vdash T <: \top} \text{(S-TOP)}$$

$$\frac{\overline{o} \cap fv(\exists \overline{o'}; \overline{C'}.N) = \emptyset \quad \Delta, \overline{C}; \Gamma, \overline{o : \top}; \overline{X} \models [\overline{b/o'}]\overline{C'}}{\mathcal{E}; \Gamma \vdash \exists \overline{o}; \overline{C}.[\overline{b/o'}]N <: \exists \overline{o'}; \overline{C'}.N} \text{(S-ENV)}$$

Figure 7.7: Mojo-jojo subtyping.

Subtyping is defined in Fig. 7.7, S-REFLEX describes reflexivity of subtyping and S-TOP describes the top type over subtyping. Mojo-jojo does not support sub-classing, which makes its handling of existential types the most interesting aspect of subtyping. We use S-ENV, a rule taken from the formalisation of Java wildcards [58], to introduce existential types (packing) and partial abstraction of existential types (corresponding to covariant and contravariant changes to bounds in traditional existential type systems). In Mojo-jojo, strictness of bounds is indicated by the set of constraints in the subtype being stronger than (that is, can prove valid) the set of constraints in the supertype. The *fv* function returns the free variables of the type passed in.

The subtyping checks for the nominal typing aspect of Mojo-jojo (S-REFLEX and S-TOP) are trivial as Mojo-jojo does not supports inheritance or subclassing. The primary purpose of the subtyping rules in Mojo-jojo is to ensure valid constraints in the expression typing rules for field assignment (T-ASSIGN), method invocation (T-INVK), and subsumption (T-

SUB). The S-ENV rule gives a subtyping relationship between two existentially quantified class types, where the constraints parameters of the subtype are required to be more precise than those in the supertype. The notion of being more precise is expressed using the substitution $[\overline{b/\circ'}]$, where $\overline{\circ'}$ are the contexts parameters of the supertype and $\overline{b}$ are the corresponding boxes in the subtype. The S-ENV rule uses the substitution to ensure $\overline{b}$ is more precise than $\overline{\circ'}$ inside the constraints $\overline{\mathcal{C}'}$.

Incorporate subclassing into Mojo-jojo is non-trivial. Additional lemmas are required to link subtyping with subclassing as most of the typing rules and lemmas are expressed in terms of subtyping, while the features of the language, such as field and method lookup, are defined with subclassing. One possibility is to have a lemma that shows when two types are subtypes then the upper bounds of the types are subclasses of each other. The difficulty is in locating the upper bounds of the types, whereby an auxiliary function is required to recursively traverse through the upper bounds of the type until a non-variable type is reached.

7.2.6 Well-formedness of Mojo-jojo

Well-formed constraints are presented in Fig. 7.8.

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

$$\frac{\mathcal{E};\Gamma \vdash b\ \text{OK}}{\mathcal{E};\Gamma \vdash b = \emptyset\ \text{OK}}\ (\text{F-EQ}) \qquad \frac{\mathcal{E};\Gamma \vdash b\ \text{OK}}{\mathcal{E};\Gamma \vdash b \neq \emptyset\ \text{OK}}\ (\text{F-NEQ})$$

$$\frac{\mathcal{E};\Gamma \vdash b_1\ \text{OK} \qquad \mathcal{E};\Gamma \vdash b_2\ \text{OK}}{\mathcal{E};\Gamma \vdash b_1 \subseteq b_2\ \text{OK}}\ (\text{F-SUB})$$

Figure 7.8: Mojo-jojo well-formed constraints.

F-EQ and F-NEQ respectively state that the constraints for boxes being empty or non-empty are well-formed, if those boxes are well-formed. Similarly, F-SUB states that the sub-boxing constraint is well-formed, if both boxes described in the constraint are well-formed.

Well-formed heaps are presented in Fig. 7.9.

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

$$\frac{\mathcal{H} \vdash N\ \text{OK} \qquad \begin{array}{c}\forall \iota \to \{N; \overline{\mathtt{f} \to v}\} \in \mathcal{H} : \\ \overline{fType(\mathtt{f}, N) = T'} \qquad \mathcal{H} \vdash \overline{v : [\iota/\mathtt{this}]T'\iota} \\ \forall v \in \overline{v} : v \neq \mathtt{null} \Rightarrow v \in dom(\mathcal{H})\end{array}}{\vdash \mathcal{H}\ \text{OK}}\ (\text{F-HEAP})$$

Figure 7.9: Mojo-jojo well-formed heaps.

F-HEAP states that a heap ($\mathcal{H}$) is well-formed if every object in $\mathcal{H}$ has a well-formed type, and its fields are in $\mathcal{H}$ with types determined by the *fType* function.

Fig. 7.10 presents well-formed boxes and types for Mojo-jojo. Each judgment checks that variables are correctly bound, constraints are satisfied where appropriate (the equivalent of bounds checking), and named classes are declared in the program.

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

$$\frac{\Gamma(\gamma) \neq \mathtt{X}}{\mathcal{E};\Gamma \vdash \gamma \text{ OK}} \text{ (F-VAR)} \qquad \frac{}{\mathcal{E};\Gamma \vdash \mathtt{world} \text{ OK}} \text{ (F-WORLD)}$$

$$\frac{}{\mathcal{E};\Gamma \vdash \emptyset \text{ OK}} \text{ (F-EMPTY)} \qquad \frac{\mathcal{E};\Gamma \vdash T \text{ OK}}{\mathcal{E};\Gamma \vdash T\mathtt{.owner} \text{ OK}} \text{ (F-TOWNER)}$$

$$\frac{\mathcal{E};\Gamma \vdash b \text{ OK} \quad \mathcal{E};\Gamma \vdash b' \text{ OK} \quad \Delta;\Gamma;\overline{\mathtt{X}} \models b \cap b' \neq \emptyset}{\mathcal{E};\Gamma \vdash b \cap b' \text{ OK}} \text{ (F-INTERSECT)}$$

$$\frac{\mathcal{E};\Gamma \vdash b \text{ OK} \quad \mathcal{E};\Gamma \vdash b' \text{ OK}}{\mathcal{E};\Gamma \vdash b \cup b' \text{ OK}} \text{ (F-UNION)}$$

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

$$\frac{\mathtt{X} \in \overline{\mathtt{X}}}{\mathcal{E};\Gamma \vdash \mathtt{X} \text{ OK}} \text{ (F-TYPE-VAR)} \qquad \frac{\Gamma' = \Gamma, \overline{\mathtt{o}:\top} \quad \Delta,\overline{\mathcal{C}};\Gamma';\overline{\mathtt{X}} \vdash N \text{ OK} \quad \Delta,\overline{\mathcal{C}};\Gamma';\overline{\mathtt{X}} \vdash \overline{\mathcal{C}} \text{ OK}}{\mathcal{E};\Gamma \vdash \exists\overline{\mathtt{o}};\overline{\mathcal{C}}.N \text{ OK}} \text{ (F-EXISTS)}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{Y}}\ \overline{\mathcal{C}}\mathtt{>\ ...} \quad \mathcal{E};\Gamma \vdash \overline{T} \text{ OK} \quad \mathcal{E};\Gamma \vdash b \text{ OK} \quad \Delta;\Gamma,\mathtt{this}:b:\mathtt{C<}\overline{T}\mathtt{>};\overline{\mathtt{X}} \models [b/\mathtt{owner},\overline{\mathtt{T}/\mathtt{Y}}]\overline{\mathcal{C}}}{\mathcal{E};\Gamma \vdash b:\mathtt{C<}\overline{T}\mathtt{>} \text{ OK}} \text{ (F-CLASS)}$$

Figure 7.10: Mojo-jojo well-formed boxes and types.

7.2.7 Class and Method Well-formedness of Mojo-jojo

Type rules for methods and classes are given in Fig. 7.11.

$$\frac{\begin{array}{c}\Gamma = \texttt{owner:}\top, \texttt{this} : \texttt{owner:C<}\overline{\texttt{X}}\texttt{>} \\ \texttt{this} \neq \emptyset, \overline{\mathcal{C}}; \Gamma; \overline{\texttt{X}} \vdash \overline{T}, \overline{M}\ \text{OK} \qquad \texttt{this} \neq \emptyset; \Gamma; \overline{\texttt{X}} \vdash \overline{\mathcal{C}}\ \text{OK} \\ \texttt{this} \neq \emptyset, \overline{\mathcal{C}}; \Gamma; \overline{\texttt{X}} \not\models \emptyset \neq \emptyset \end{array}}{\vdash \texttt{class C<}\overline{\texttt{X}}\ \overline{\mathcal{C}}\texttt{>}\ \{\overline{T\,\texttt{f;}}\ \ \overline{M}\}\ \text{OK}}$$

(T-CLASS)

$$\frac{\begin{array}{cc}\Gamma' = \Gamma, \texttt{x}:T' & \Delta' = \Delta, \overline{\mathcal{C}} \\ \Delta'; \Gamma; \overline{\texttt{X}} \vdash T, T'\ \text{OK} & \Delta'; \Gamma'; \overline{\texttt{X}} \vdash e : T \\ \Delta'\Gamma; \overline{\texttt{X}} \not\models \emptyset \neq \emptyset & \mathcal{E}; \Gamma \vdash \overline{\mathcal{C}}\ \text{OK} \end{array}}{\Delta; \Gamma; \overline{\texttt{X}} \vdash T\texttt{m(}T'\,\texttt{x)}\ \overline{\mathcal{C}}\,\{\texttt{return}\ e\texttt{;}\}\ \text{OK}}$$

(T-METHOD)

Figure 7.11: Mojo-jojo typing rules for classes and methods.

Declared constraints must be consistent as well as being well-formed. Consistency is checked by requiring that the empty set cannot be proved not equal to itself (in both T-CLASS and T-METHOD); essentially meaning that 'false' cannot be shown.

7.2.8 Semantics of Mojo-jojo

The semantics for Mojo-jojo are presented in Fig. 7.12, and Fig. 7.13. We use a large step semantics because we need to know which object is the current '`this`' object at each stage of execution to prove properties of the permissions system.

$$\frac{\mathcal{H}(\iota) = \{\mathtt{R}; \overline{\mathtt{f}{\rightarrow}\mathtt{v}}\}}{\iota\,.\,\mathtt{f}_i; \mathcal{H} \rightsquigarrow \mathtt{v}_i; \mathcal{H}} \quad \text{(R-FIELD)}$$

$$\frac{\mathcal{H}(\iota) = \{\mathtt{R}; \overline{\mathtt{f}{\rightarrow}\mathtt{v}}\} \qquad \mathcal{H}' = \mathcal{H}[\iota \mapsto \{\mathtt{R}; \overline{\mathtt{f}{\rightarrow}\mathtt{v}}[\mathtt{f}_i \mapsto \mathtt{v}]\}]}{\iota\,.\,\mathtt{f}_i \;=\; \mathtt{v}; \mathcal{H} \rightsquigarrow \mathtt{v}; \mathcal{H}'} \quad \text{(R-ASSIGN)}$$

$$\frac{\mathcal{H}(\iota)\ undefined \qquad fields(\mathtt{C}) = \overline{\mathtt{f}} \qquad \mathcal{H}' = \mathcal{H}, \iota \rightarrow \{\mathtt{P:C<}\overline{\mathtt{T}}\mathtt{>}; \overline{\mathtt{f}{\rightarrow}\mathtt{null}}\}}{\mathtt{new\ P:C<}\overline{\mathtt{T}}\mathtt{>}; \mathcal{H} \rightsquigarrow \iota; \mathcal{H}'} \quad \text{(R-NEW)}$$

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

Figure 7.12: Mojo-jojo reduction rules.

$$\frac{}{\texttt{null.f};\mathcal{H} \rightsquigarrow \texttt{err};\mathcal{H}}\ (\text{R-FIELD-NULL}) \qquad \frac{}{\texttt{null.f = e};\mathcal{H} \rightsquigarrow \texttt{err};\mathcal{H}}\ (\text{R-ASSIGN-NULL})$$

$$\frac{}{\texttt{null.<}\overline{\texttt{T}}\texttt{>m(}\overline{\texttt{e}}\texttt{)};\mathcal{H} \rightsquigarrow \texttt{err};\mathcal{H}}\ (\text{R-INVK-NULL})$$

$$\frac{\texttt{e}_i;\mathcal{H} \rightsquigarrow \texttt{e}_i';\mathcal{H}' \qquad \texttt{e}_i' \neq \texttt{err}}{\iota\texttt{.<}\overline{\texttt{T}}\texttt{>m(}\overline{\texttt{v}}\texttt{, e}_i\texttt{, }\overline{\texttt{e}}\texttt{)};\mathcal{H} \rightsquigarrow \iota\texttt{.<}\overline{\texttt{T}}\texttt{>m(}\overline{\texttt{v}}\texttt{, e}_i'\texttt{, }\overline{\texttt{e}}\texttt{)};\mathcal{H}'}\ (\text{RC-INVK})$$

$$\frac{\texttt{e};\mathcal{H} \rightsquigarrow \texttt{e}';\mathcal{H}' \qquad \texttt{e}' \neq \texttt{err}}{\iota\texttt{.f = e};\mathcal{H} \rightsquigarrow \iota\texttt{.f = e}';\mathcal{H}'}\ (\text{RC-ASSIGN})$$

Figure 7.13: Mojo-jojo reduction rules (cont.).

7.3 Proof Properties of Mojo-jojo

In this section, we discuss our proof of the subject reduction theorem for Mojo-jojo:

> **Theorem 2.3.1:** subject reduction
>
> *For all* $\mathcal{H}$, $\mathcal{H}'$, e, v, *and* T, *if* $\mathcal{H} \vdash e : T$ *and* $e; \mathcal{H} \rightsquigarrow v; \mathcal{H}'$ *and* $\vdash \mathcal{H}$ OK *then* $\mathcal{H}' \vdash v : T$ *and* $\vdash \mathcal{H}'$ OK.

The proof proceeds in a standard manner by structural induction over the derivation of reductions. We have a large number of lemmas, most of them standard for such systems (see for example, [16, 13, 14]). We state and discuss some of the interesting lemmas below. Subject reduction shows that reduction preserves types, that is an expression cannot change type as it executes (up to subtyping). In ownership terms, since we have proved preservation, we can be sure that the owners described by the static types in our language reflect the actual heap at runtime. This is a necessary condition for supporting an effect system or any prescriptive property.

> **Lemma 2.3.2:** strengthening (type checking) *For all* Δ, Γ, $\overline{\mathrm{X}}$, e, T, *and* $\overline{\mathcal{C}}$, *if* $\Delta, \overline{\mathcal{C}}; \Gamma; \overline{\mathrm{X}} \vdash e : T$ *and* $\Delta; \Gamma; \overline{\mathrm{X}} \models \overline{\mathcal{C}}$ *then* $\Delta; \Gamma; \overline{\mathrm{X}} \vdash e : T$

We use a system of constraints rather than bounds on variables, this causes several differences in the statement of lemmas (see below) and the overall shape of our proofs. One of the nice properties of our system is strengthening: if a constraint can be proven by an environment, then it can be removed from that environment when proving judgments. We give the strengthening lemma for the type checking judgment above, lemmas for the other judgments are similar.

> **Lemma 2.3.3:** box substitution preserves type checking *For all* Δ, Γ, $\overline{\mathrm{X}}$, e, T, $\overline{b}$, $\overline{\mathrm{o}}$, *and* $\overline{\mathcal{C}}$, *if* $\Delta, \overline{\mathcal{C}}; \Gamma, \overline{\mathrm{o} : \top}; \overline{\mathrm{X}} \vdash e : T$ *and* $\mathcal{E}; \Gamma \vdash \overline{b}$ OK *and* $\Delta; \Gamma; \overline{\mathrm{X}} \models [\overline{b/\mathrm{o}}]\overline{\mathcal{C}}$ *then* $[\overline{b/\mathrm{o}}]\Delta; [\overline{b/\mathrm{o}}]\Gamma; \overline{\mathrm{X}} \vdash [\overline{b/\mathrm{o}}]e : [\overline{b/\mathrm{o}}]T$

We also prove lemmas for the substitution of types for type variables, values for expressions variables, and boxes for expression variables; we prove a preservation lemma for each judgment for each kind of substitution. The substitution lemmas for types and values are standard. The lemmas for boxes are more interesting because the variables being replaced do not have bounds but are subject to constraints. Our lemma therefore requires that the constraints associated with $\overline{o}$ can be satisfied, rather than the bounds of $\overline{o}$.

Lemma 2.3.4: closing preserves well-formedness *For all* Δ, Γ, $\overline{X}$, T, $\overline{o}$, *and* $\overline{\mathcal{C}}$, *if* $\Delta, \overline{\mathcal{C}}; \Gamma, \overline{o : \top}; \overline{X} \vdash T$ OK *and* $\Delta, \overline{\mathcal{C}}; \Gamma, \overline{o : \top}; \overline{X} \vdash \overline{\mathcal{C}}$ OK *then* $\mathcal{E}; \Gamma \vdash \Downarrow_{\overline{o};\overline{\mathcal{C}}} T$ OK

Lemma 2.3.5: closing gives subtypes *For all* Δ, Γ, $\overline{X}$, T, $\overline{b}$, $\overline{o}$, *and* $\overline{\mathcal{C}}$, *if* $\Delta; \Gamma; \overline{X} \models [\overline{b/o}]\overline{\mathcal{C}}$ *then* $\mathcal{E}; \Gamma \vdash [\overline{b/o}]T <: \Downarrow_{\overline{o};\overline{\mathcal{C}}} T$

In Mojo-jojo, we introduce existential quantification using the $\Downarrow$ operator. We must account for closing in our proofs and this is done (in part) by the above two lemmas. **Lemma 2.3.4** shows that closing preserves the well-formedness of types it operates on. **Lemma 2.3.5** shows how closing fits with subtyping: closing and the S-ENV subtyping rule are complements, they introduce existential types in the same way, this lemma formalises that connection.

7.4 Effects, permissions, and Prescriptive Policies for Mojo-jojo

This subsection explores the possibility of adding effects, permissions, and prescriptive ownership policies to Mojo-jojo. The main benefit of ownership types comes from their ability to restrict access to objects by restricting which references can access those objects. Descriptive ownership type systems can not enforce such restrictions, and this greatly diminishes their

usefulness in common applications of ownership types. The ownership types described in Mojo-jojo are purely descriptive, and to compensate for this weakness, Mojo-jojo needs to support an effect system, or prescriptive ownership policy such as owners-as-dominators or owners-as-modifiers. An effect system describes how regions of the heap are affected with read, write, and reference operators. Prescriptive ownership policies restrict the objects that can be referenced or modified, according to their position in the ownership hierarchy. We have not proven type soundness for any of the extensions discussed in this section.

Effects

The syntax for effects are defined as:

$$M \quad ::= \quad T\,\texttt{m(}\overline{T\,\texttt{x}}\texttt{)}\ \ \overline{\mathcal{C}}\ \ \overline{\mathcal{E}}\ \ \texttt{\{return } e\texttt{;\}} \qquad \textit{method declarations}$$

$$p \quad ::= \quad \texttt{rd} \mid \texttt{wr} \mid \texttt{rf} \qquad \textit{effects kinds}$$

$$\mathcal{E} \quad ::= \quad p\texttt{:}\ b \qquad \textit{effects}$$

There are three kinds of effect: read (`rd`), write (`wr`), or reference (`rf`). The read and write effects respectively describe the objects in a method that are read and written. The reference effect describes which objects are referenced during execution.

We call our multiple ownership system with an effect system Mojo-jojo-$\mathcal{P}$. The effect system for Mojo-jojo-$\mathcal{P}$ is similar to the effect system for multiple ownership that MOJO supports. Boxes (b) denote the regions the effects system tracks. Methods are required to be annotated with effects, which are inferred on the expressions in those methods.

The most interesting aspect of the effect system is in the T-INVK rule where we must account for unpacked variables in the effect annotations.

Unpacked variables are treated in the same way as in types, by using the close operation ($\Downarrow$). We use the following rule to close boxes (and thus effects), the conclusion should be read as using $\Delta; \Gamma; \overline{\mathrm{X}}$ to judge the closing operation to $\mathtt{b}$; the result of the closing is $\mathtt{b}'$:

$$\frac{\Delta; \Gamma; \overline{\mathrm{X}} \vdash b \subseteq b' \qquad \mathit{fv}(b') \cap \overline{\mathrm{o}} = \emptyset}{\Delta; \Gamma; \overline{\mathrm{X}} \vdash \Downarrow_{\overline{\mathrm{o}};\overline{\mathcal{C}}} b = b'}$$

The close operator finds a super-box of b that contains no free $\overline{\mathrm{o}}$. If b describes an effect, then b' describes a larger effect which does not include unpacked variables. Our effect system is thus safe and conservative in the presence of existential quantification of ownership variables. This rule works in the same way as finding a closed super-type of a type variable in systems with existential types [13].

MOJO's effect system uses wildcards over owner parameters to describes its effects. The equivalent in Mojo-jojo-$\mathcal{P}$ would be to allow existential quantification for effects. We found this approach not only fails to improve the expressiveness of the effect system, but also increased its complexity.

As in MOJO, the most common use of the effects system in Mojo-jojo-$\mathcal{P}$ is to predict the disjointness of expressions if their effects do not overlap.

Policies

The most common encapsulation policies (owners-as-dominators and owners-as-modifiers) do not scale straightforwardly to multiple ownership.

These policies ensure that access to (permission to reference or write) an object is restricted to objects which (transitively) own or are owned by that object. The key question in extending these policies to multiple ownership is how shared ownership is handled. We must decide what makes

an 'owner' in terms of 'owners-as' policies. There are two choices: either an object must have exclusive ownership over an object (by being the only direct owner, or transitively owning all owners) or shared ownership is good enough. A case can be made for each side: is the policy strict enough to provide relabel guarantees and is it permissive enough to be safe? Requiring exclusive ownership means that an object cannot access objects it partially owns, but since a common use for multiple ownership is for an object to have partial ownership of its fields, this may well be too restrictive. Allowing shared ownership, however, means the type system cannot easily reason about access because there may be other (unknown) objects with access to the shared objects. In short, there is no easy answer: we need to experiment with systems that requires different levels of exclusivity of ownership.

Permissions

Permissions can be thought of as the dual of effects, where the effect describes what has been done, and a permission describes what may be done. A feature of Mojo-jojo-$\mathcal{P}$ is that it describes both permissions and effects. However, permissions and effects are not a perfect fit: For permissions, we care about the object that is doing the action (e.g., a may be allowed to modify b, but c may not be) whereas the subject is irrelevant for effects. In terms of checking, there is no real overlap between effect and permission information.

It would be safe to use a method's effect to conservatively estimate if the method can be called by a given receiver and parameters. This estimate, however, is too conservative to be useful — it is possible that a method is safe to call even if it has an effect that the receiver or parameters do not have permission for. Instead, we use method-level constraints to check whether it is safe to call a method, and it is always safe if we can satisfy its constraints at its call site. This check allows permissions to be safer and more precise than using effects, however, methods are required

to be annotated with both effects and permissions.

Permissions are added as just another kind of constraint. Constraints in Mojo-jojo-$\mathcal{P}$ are therefore not only topological, but also describe other kinds of constraints on a program. By encapsulating permissions within constraints, different kinds of permissions can be included without changing the majority of the type system. Permissions on a class, similar to effects on methods, allow boxes of objects to read, write, or reference instantiations of that class. Permissions may also be given per-method, which have to be satisfied at the call site. Permission constraints may also be present in existential types.

We define the syntax of permissions as:

p	::=	`rd` | `wr` | `rf`	*permission kinds*
vb	::=	$b \mid b^{\triangledown} \mid b^{\triangle}$	*variant boxes*
$\mathcal{P}$	::=	$vb\ p\ b$	*permissions*
$\mathcal{C}$	::=	... | $\mathcal{P}$	*constraints*

For example, `a rd b` means that all objects in box `a` may read object `b`. The use of variant boxes in permissions allows the programmer to specify that not only a specific object, but any object in a sub- or super-box (using $b^{\triangledown}$or $b^{\triangle}$, respectively) has a permission; for example, $\texttt{a}^{\triangledown}$ includes `a`, $\texttt{a} \cap \texttt{b}$, and $\texttt{a} \cap \texttt{c}$, and $\texttt{a}^{\triangle}$ includes $\texttt{a} \cup \texttt{b}$ and the owner of `a`.

As opposed to other common encapsulation policies, permissions do not propagate up or down the ownership graph. For example, just because object `a` has permission to access object `b`, objects owned by `a` do not automatically have permission to access `b`. The programmer can always add more permissions, or a pre-processing step could enforce a policy by adding propagated permissions.

Checking of permissions in the type rules is done implicitly by checking that class and method level constraints are satisfied in T-NEW (via F-CLASS) and T-INVK, respectively. Our constraint satisfaction rules must expand to accommodate checking of permissions, including accounting for variant boxes, and rules to allow an object complete access to itself and its owner. At runtime, our semantics must keep track of the current object (`this`) in order to type check expressions at intermediate reduction steps; this motivates our use of large-step semantics.

The most interest parts of the formalism are the expressions typing rules for field lookup, method invocation, and field assignment presented in Fig. 7.14, as well as the well-formedness for method and class declarations presented in Fig. 7.15. The field lookup and method invocation expressions require the close operators, while field assignment requires an interesting set of permissions. The interesting part for class declaration is the initialisation of the permissions in the constraints of class.

7.5 Chapter Summary

In this chapter, we present Mojo-jojo, our formalism of a multiple ownership system. Mojo-jojo is the first multiple ownership system that incorporates a constraints system and contains existential quantifications over the types and contexts. The constraints system describes constraints over the context parameters of a class declaration, and is similar to the constraints described in set algebra. We also discuss the proofs for type soundness of Mojo-jojo, and present the key lemmas for those proofs. The details for the proof of subject reduction for Mojo-jojo are presented in appendix B.

Expression typing: $\boxed{\Delta; \Gamma; \overline{\mathtt{X}} \vdash e : T\, \mathcal{E}}$

$$\frac{\begin{array}{c}\Delta; \Gamma; \overline{\mathtt{X}} \vdash \gamma_0 : \exists \overline{\mathtt{o}}; \overline{\mathcal{C}}\,.\, N\, \overline{\mathcal{E}} \\ \mathtt{fType(f,}\ N\mathtt{)} = T \\ \Delta, \overline{\mathcal{C}}; \Gamma, \overline{\mathtt{o}:\top}; \overline{\mathtt{X}} \vdash own(\Gamma(\gamma))\ \mathtt{rd}\ own(N)\end{array}}{\Delta; \Gamma; \overline{\mathtt{X}} \vdash \gamma_0\mathtt{.f} :\Downarrow_{\overline{\mathtt{o}}}\ [\gamma_0/\gamma]T\, !\, \overline{\mathcal{E}}, \mathtt{rd}:\gamma_0}$$

(T-FIELD)

$$\frac{\begin{array}{c}\Delta; \Gamma; \overline{\mathtt{X}} \vdash \gamma_0 : \exists \overline{\mathtt{o}}; \overline{\mathcal{C}}.N\, \overline{\mathcal{E}}_1 \\ fType(\mathtt{f}, N) = T' \\ \Delta; \Gamma; \overline{\mathtt{X}} \vdash e : T\, \overline{\mathcal{E}}_2 \\ \Delta, \overline{\mathcal{C}}; \Gamma, \overline{\mathtt{o}:\top}; \overline{\mathtt{X}} \vdash T <: [\gamma_0/\gamma]T' \\ \Delta, \overline{\mathcal{C}}; \Gamma, \overline{\mathtt{o}:\top}; \overline{\mathtt{X}} \vdash own(\Gamma(\gamma))\ \mathtt{wr}\ own(N) \\ \Delta, \overline{\mathcal{C}}; \Gamma, \overline{\mathtt{o}:\top}; \overline{\mathtt{X}} \vdash own(N)\ \mathtt{rf}\ own([\gamma_0/\gamma]T)\end{array}}{\Delta; \Gamma; \overline{\mathtt{X}} \vdash \gamma_0\mathtt{.f} = e : T\, \overline{\mathcal{E}}_1, \overline{\mathcal{E}}_2, \mathtt{wr}:\gamma_0}$$

(T-ASSIGN)

$$\frac{\begin{array}{c}\Delta; \Gamma; \overline{\mathtt{X}} \vdash \gamma_0 : \exists \overline{\mathtt{o}}; \overline{\mathcal{C}}.N\, \overline{\mathcal{E}} \\ \Delta; \Gamma; \overline{\mathtt{X}} \vdash \overline{e : T\ !\ \overline{\mathcal{E}}} \\ mType(\mathtt{m}, N) = \overline{\mathcal{C}}.\overline{T'}\ \rightarrow\ T\, !\, \overline{\mathcal{E}}_m \\ \Delta, \overline{\mathcal{C}}; \Gamma, \overline{\mathtt{o}:\top}; \overline{\mathtt{X}} \vdash \overline{T <: [\gamma_0/\gamma]T'} \\ \Delta, \overline{\mathcal{C}}; \Gamma, \overline{\mathtt{o}:\top}; \overline{\mathtt{X}} \models [\gamma_0/\gamma]\overline{\mathcal{C}} \\ \Delta, \overline{\mathcal{C}}; \Gamma, \overline{\mathtt{o}:\top}; \overline{\mathtt{X}} \models\Downarrow_{\overline{\mathtt{o}}}\ [\gamma_0/\gamma]\, \overline{\mathcal{E}}_m = \mathcal{E}'\end{array}}{\Delta; \Gamma; \overline{\mathtt{X}} \vdash \gamma_0\mathtt{.m(}\overline{e}\mathtt{)} :\Downarrow_{\overline{\mathtt{o}}}\ [\gamma_0/\gamma]T\, \overline{\mathcal{E}}, \overline{\overline{\mathcal{E}}}, \mathcal{E}'}$$

(T-INVK)

Figure 7.14: Expression typing of Mojo-jojo-$\mathcal{P}$.

$$\frac{\begin{array}{c}\Gamma = \texttt{this}{\rightarrow}\texttt{owner:C<}\overline{\texttt{X}}\texttt{>} \\ \overline{\mathcal{C}}, \overline{\mathcal{C}'}; \Gamma; \overline{\texttt{X}} \vdash \overline{T}, \overline{M}\ \text{OK} \qquad \overline{\mathcal{C}}, \overline{\mathcal{C}'}; \Gamma; \overline{\texttt{X}} \not\vdash \emptyset \neq \emptyset \qquad \emptyset; \Gamma; \overline{\texttt{X}} \vdash \overline{\mathcal{C}}, \overline{\mathcal{C}'}\ \text{OK} \\ \Delta = \texttt{this} \subseteq \emptyset, \quad \texttt{world} \subseteq \texttt{this}, \quad \texttt{world}^{\triangledown}\texttt{rd this}, \\ \texttt{world}^{\triangledown}\texttt{wr this}, \quad \texttt{world}^{\triangledown}\texttt{rf this} \\ \Delta; \Gamma; \emptyset \models \overline{\mathcal{C}'} \\ \forall b \in \overline{\mathcal{E}}.\, \overline{\mathcal{C}}, \overline{\mathcal{C}'}; \Gamma; \overline{\texttt{X}} \vdash b\ \text{OK}\end{array}}{\vdash \texttt{class C<}\overline{\texttt{X}}\ \overline{\mathcal{C}}\texttt{>}\ \overline{\mathcal{C}'}\ \{\overline{T\,\texttt{f;}}\ \ \overline{M}\}\ \text{OK}}$$

(T-CLASS)

$$\frac{\begin{array}{c}\Gamma' = \Gamma, \overline{\texttt{x}{\rightarrow}T} \qquad \Delta' = \Delta, \overline{\mathcal{C}} \\ \Delta'; \Gamma'; \overline{\texttt{X}} \vdash T, \overline{T}, \overline{\mathcal{E}}\ \text{OK} \qquad \Delta'; \Gamma'; \overline{\texttt{X}} \vdash e : T\texttt{this}\overline{\mathcal{E}} \\ \Delta'\Gamma; \overline{\texttt{X}} \not\vdash \emptyset \neq \emptyset \qquad \mathcal{E}; \Gamma \vdash \overline{\mathcal{C}}\ \text{OK}\end{array}}{\Delta; \Gamma; \overline{\texttt{X}} \vdash T\texttt{m}\,(\overline{T\,\texttt{x}})\ \overline{\mathcal{C}}\,\overline{\mathcal{E}}\,\{\texttt{return}\ e\texttt{;}\}\ \text{OK}}$$

(T-METHOD)

Figure 7.15: Class and method typing of Mojo-jojo-$\mathcal{P}$.

Chapter 8

Conclusion

In this thesis we have presented two type sound language formalisms: `descripSC` and Mojo-jojo. `descripSC` is a class-based object-oriented programming language with the ability to perform sheep cloning. Mojo-jojo is a class-based object-oriented programming language with multiple ownership, existential quantification of owners with type parametricity, and an algebraic constraint system over the multiple ownership topology of the system.

Sheep cloning uses the owner topology provided ownership types to identify and copy an object's internal structure. Sheep clones are structurally closer to the original object than shallow clones, and sheep clones are more appealing than deep clones in terms of practical applications. We have constructed three formalism of sheep cloning: `recurSC`, `mapSC`, and `descripSC`. For each formalism, we discussed its motivating factors, its flaws, and how it subsequently influences its successors. The `descripSC` formalism is the only sheep cloning formalism shown to be type sound. We presented a proof outline for the `mapSC` and `recurSC` formalisms, and argued why their proofs were so difficult.

Mojo-jojo contains standard type theoretic tools (existential quantification, generics) and an elegant system of constraints, based directly on set algebra, to model complex owner topologies. We presented a discussion

of its proof. We sketched how effects and permissions can be added into Mojo-jojo. Permissions can be incorporated into the existing constraints system, which would extends it from a description of the runtime topology of the system to describing the program's general runtime behaviour.

For this final chapter, we discuss research related to sheep cloning, and present directions for future work in sheep cloning and multiple ownership.

8.1 Related Work

In this section, we discuss the areas where sheep cloning is applicable, and how our sheep cloning formalism compares against other object cloning formalism. The related work for Mojo-jojo has been discussed with the introduction of MOJO.

8.1.1 Possible Applications for Sheep Cloning

In "Exceptions in ownership types systems" [31] Dietl and Müller outline several possible solutions to exception handling for Universe Types. One of these solutions is to clone the exception object when it appears, then propagate the clone through the stack to the exception handler. They explain how supporting exceptions with cloning would require no changes to their ownership system, however, they chose not to handle exceptions with cloning, citing the need for every object in the system to be `cloneable` as being too costly, especially if an exception is propagated multiple times before it is caught.

In "Minimal Ownership for Active Objects" [24], Clarke et al. developed 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 et al. propose using unique references and immutable objects, and cloning the active object only when necessary. They discuss three cases where they would clone active objects using a "minimal clone operation". The minimal clone operation determines whether an object's fields are cloned or aliased based on their ownership annotation. Their minimal clone operation is very similar to sheep cloning, and Clarke et al. even mention how sheep cloning can be used in its place.

In his PhD. thesis [67], Nienaltowski 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.

8.1.2 Comparison with Formalisms of Cloning

A more recent study detailing procedures for object cloning was by Jensen et al. [47], where they 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.

In Fig. 8.1, we present a linked list example described in Jensen's system. The `List` class contains a `Node` that is the head of the linked list. The type parameter `V` denotes the type of the values in the linked list. The `Node` class contains two fields, the field `next` is the next node of the linked list and the field `value` with type `V` is the value contained in this particular node of the list. Cloning a `List` object should copy every node while aliasing the values of each node.

```
class List<V>{
 Node<V> head;
}

class Node<V>{
 Node<V> next;
 V value;
}
```

Figure 8.1: Linked list in Jensen's system.

In Fig. 8.2, we present the copy policy for the `List` and `Node` classes, as well as the cloning method a `List` object and a `Node` object. `DL` is the copy policy of the `List` class. `DL` states the field `head` is to be deep copied. `deepList()` is the cloning method for a `List` object, and it follows the copy policy set by `DL`. The head node is deep copied by the `deepNode()` method, which deep copies `Node` objects.

`DN` is the copy policy for the `Node` class, and it states that the field `next` is deep copied and the field `value` is to be aliased. The cloning method `deepNode()` would deep copy the `next` object and alias the `value` ob-

ject.

```
DL: {@Deep Node head;};
@Copy(DL) List<V> deepList(){
 return new List<V> (head == null ? null :
                                  head.deepNode());
}

DN: {@Deep Node next; @Alias V value;};
@Copy(DN) Node<V> deepNode(){
 return new Node<V> (value == null ? null :
                                          value;
                     next == null ? null :
                                  next.deepNode());
}
```

Figure 8.2: Copy policy and cloning method for the `List` and `Node` class.

The copy policy and cloning method defined by Jensen differs from sheep cloning as our formalism describes the semantics of sheep cloning in full: using sheep cloning, programmers would never have to implement a clone method. In Fig. 8.3, we show an example of the `List` class with ownership types. The `List` class is parameterised with an owner parameter `c` which denotes the owner of the value in the linked list. The field `head` is a `Node` object, and it takes two owner parameters, `this` and `c`. The `this` parameter in the type of `head` denotes that the `head` object is owned by this instantiation of the `List` object. The `Node` class is parameterised with two owner parameters, `c1` and `c2`, where `c1` is the owner of the nodes of the linked list and `c2` is the owner of the value in the linked list. A value in the linked list has the type `V` and the owner `c2`.

For comparison in Fig. 8.4, we sheep clone a `List` object in a system described in our formalism. We create an instantiation of the `List` class

```
class List<c>{
 Node<this, c> head;
}

class Node<c1, c2>{
 Node<c1, c2> next;
 V<c2> value;
}
```

Figure 8.3: Linked list in a system with ownership types.

(`list`) in the Class `Foo`. The `Foo` class uses the `sheep` expression on the `list` object to creates a sheep clone. The sheep clone is constructed completely by the system with no input from the user, or any additional methods and annotations.

```
Class Foo{
 List<this> list = new List<this>();
 ....
 sheep(list);
}
```

Figure 8.4: Sheep cloning the `list` object.

Drossopoulou and Noble [33] also propose a static object cloning implementation. They introduce the concept of cloning domain, and the method in which objects are cloned by cloning their domain. Just as ownership types enforce a topological structure upon the heap, the cloning domain provides an hierarchical structure for the objects in the program. This is achieved by placing cloning annotations on every field of every class, and using these cloning annotations to create the cloning paths for

each field of the class. 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 the `Boolean` arguments set by the originator.

The parametric clone method is of the form `clone(Boolean` s_1`, ..., Boolean` s_n`, Map m)`. The variables s_1, ..., 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 (s_i). In contrast, the expression for sheep cloning is `sheep(`ι`)`, where ι is the object to be cloned.

The linked list example as described in Drossopoulou and Noble's system is identically to the link list example described in Fig. 8.3 for our sheep cloning formalism, as both systems are based in ownership types. The `List` class contains the head of the list. The `Node` class contains the next node (`next`) of the linked list and the value (`value`) of the current node. Cloning the list should copy every node in the list, however, the value of each node are aliased, and not copied, by the node of the new list.

In Fig. 8.5 shows Drossopoulou and Noble's the interface to the cloning method of the `Node` object. This differs from Java's default implementation of object cloning, as the default cloning method creates only shallow clones.

In Fig. 8.6, Drossopoulou and Noble presents the actual cloning method that will be generated for the `Node` objects. The actual cloning methods will not be visible outside of the cloning library that is performing the object cloning. The cloning method for an object requires a boolean argument for each field that exists in the class. For the `Node` class the variable `s1` denotes how the `next` object is copied, while the variable `s2` is for the

```
Node clone( ){
 this.clone(false, false, new IdentityHashMap())
}
```

Figure 8.5: Interface clone method for `Node`.

`value` object. The variable `m` is used to prevent looping during the cloning process.

```
Node clone(Boolean s1, Boolean s2, Map m){
 Object n = m.get(this);

 if ( n != null) then {
   return (Node)n;
 } else {
   Node clone = new Node();
   m.put(this,clone);
   clone.next = s1 ? this.next.clone(s1,s2,m) :
                                          this.next;
   clone.value = s2 ? this.value.clone(s2,m) :
                                         this.value;
   return clone;
 }
}
```

Figure 8.6: Actual clone method for `Node`.

Similar to the copy policy described in Jensen's system, the cloning domains and parametric clone method defined by Drossopoulou and Noble can have excessive syntactical overhead when compared against sheep

cloning. Simply having ownership types is enough to obtain the full benefits provided by sheep cloning. The single expression, `sheep(list)`, shown in Fig. 8.4 is all that's required to perform sheep cloning.

8.1.3 Cyclic Structures

To address the issue of cycles within object graphs, consider "A type system for Reachability and Acyclicity" by Yi Lu and John Potter [54]. Lu et al. present a type system, called the Acyclic Region Type System (ARTS), that allows programmers to restrict cycles within the heap, through the use of constraints on the reachability of references and pointers. The paper uses region based types to achieve acyclic reachability for regions. Regions are disjoint sets of objects, with the properties that regions are acyclically reachable, i.e. regions cannot form cycles, while reference cycles are allowed within each region. A novel contribution of this paper is that whenever a new region is created the existing regions will perform a refinement of the acyclic reachability ordering to incorporate the new region. That is because the ordering of the definition of each region ensures the regions are acyclic, similar to the way an inheritance relationship is acyclic. The rest of the paper presents the ARTS formal system and its proof, followed by an interesting comparison of this system and ownership domains. Finally the paper explains that the occurrence of a cycle in the run time object graph implies there must be a cycle in the type dependency graph, so if there is no cycle in the type dependency graph then there can be none in the run time object graph. If there are type level cycles, however, the type system is powerless to prevent cyclic references, even if they are undesirable.

8.2 Future Work

In this section, we discuss several ideas associated with sheep cloning that we would have liked to explore further. We discuss approaches to show correctness for the `mapSC` formalism, ways to implement sheep cloning in practice, and discuss sheep cloning in systems with weaker prescriptive ownership constraints, or even systems without ownership types.

8.2.1 Sheep Cloning without Ownership Types

Sheep cloning is a convenient, organic and natural method to cloning objects in a system with owners-as-dominators. Sheep cloning benefits greatly from owners-as-dominators enforcing a tree-like structure on the heap by restricting incoming references from outside an object's representation, however, we believe it is possible to create sheep clones without owners-as-dominators. Many languages consider owners-as-dominators too restrictive on the structure of the heap, giving rise to weaker prescriptive policies like owners-as-modifiers [20, 32], owners-as-accessors [69], and owners-as-ombudsmen [72].

In an owners-as-dominators system, sheep cloning copies the representation of the object being cloned, and aliases the reachable objects that are outside that representation. For systems without owners-as-dominators identifying which objects to alias and which objects to copy for a particular object would be one of the most difficult aspect of sheep cloning.

In an owners-as-modifiers system there exists scenarios where not every object in an object's representation can be reachable from the object that they represent, and in those scenarios the entire heap would need to be traversed to identify the representation of that object. There are also issues with copying an object's representation in owners-as-modifiers systems once that object's representation has been identified. Copying the entire representation of an object in an owners-as-modifiers system might create objects with no incoming references. Objects without any incoming

references are unreachable and commonly garbage collected, therefore, the representations of the sheep clones in an owner-as-modifiers system are not always equivalent to the representations they are cloned from.

From the perspective of the object requesting the sheep clone, ie. the caller of the sheep cloning function, the representation of the sheep clone should not contain objects that are not part of the object being cloned. If the caller knows more about the representation of the object being cloned than that object, the caller can pass that additional information to the sheep clone once it is created. It would be unreasonable, however, for the caller to expect the object being cloned to know of objects that it can not reach. The objects inside the representation of a sheep clone observe the prescriptive policy that the sheep clone was created in, however, a newly created sheep clone would not contain any incoming references to its representation regardless of the prescriptive policy of the system that created the sheep clone.

In Fig. 8.7, we present an example to illustrate the similarities between the sheep clones of an owners-as-dominators system and that of an owners-as-modifiers system. The solid boxes are objects and the letters inside those boxes are the name of those objects, the dotted box represents the representation of the object that is on the edge of the dotted box, and the directed arrows are directed references between objects. In this example, object `A` is sheep cloning object `B`, and the objects in red represent the sheep clone of object `B`.

The top three diagrams show the sheep cloning of object `B` in an owners-as-modifiers system. The left-most diagram presents the representation of object `B`, where object `D` is referenced by object `A` and object `C` is referenced by an unknown objects outside the representation of object `B`. The middle diagram presents the sheep clone of object `B` before garbage collection, where it contains an unreferenced object (copy of object `D`). The left-most diagram shows the proper sheep clone of object `B`.

The bottom three diagrams show the sheep cloning of object `B` in an

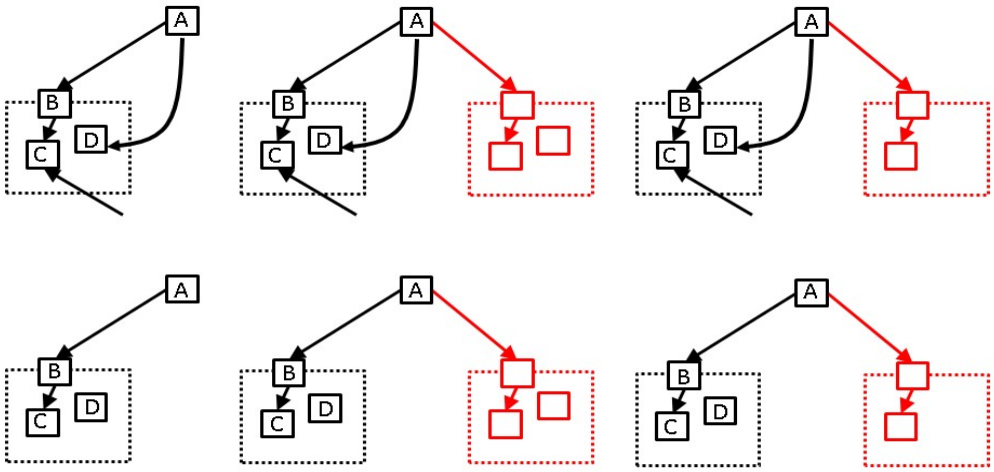


Figure 8.7: Sheep cloning with contrasting prescriptive ownership policies (part 1).

owners-as-dominators system. For this example to satisfy owners-as-dominators, all incoming references to the representation of object `B` that does not bypass object `B` are discarded. The semantics used to sheep clone object `B` is from the `descripSC` formalism. The left-most diagram shows the representation of object `B` where object `C` and `D` are not referenced by any object outside the representation of object `B`. The `descripSC` formalism copies every object inside the representation of object `B` to create the sheep clone shown in the middle diagram. Similar to the sheep clone in the owners-as-modifiers system, the copy of object `D` in this sheep clone is also garbage collected. The sheep clone in the right-most diagram is the proper sheep clone of object `B`. Object `D` could have been garbage collected before the representation is copied, which would have given the sheep clone in the right-most diagram. It is interesting to note that the sheep cloning semantics of the `mapSC` formalism would not have copied object `D`, creating

the sheep clone in the rightmost diagram. Objects that cannot be reached by object `B` would not have been added into the `map`.

The copy of object `D` in the sheep clone should be oblivious to most of the objects requesting the sheep clone. Object `A` knows the existence of object `D` and it may even know the sheep clone of object `B` contains a copy of object `D`, however, even object `A` can not access that copy of object `D` in the sheep clone.

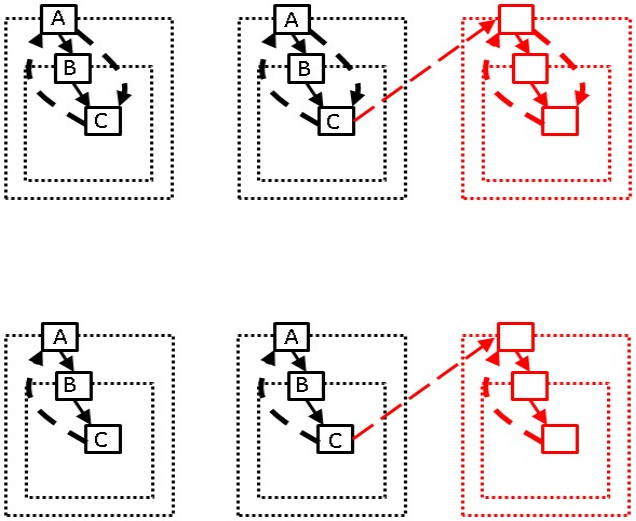


Figure 8.8: Sheep cloning with contrasting prescriptive ownership policies (part 2).

The example in Fig. 8.7 shows the same sheep clone for essentially the same object (with some minor adjustments) in two different prescriptive ownership system. In Fig. 8.8, we present an example that contrasts the difference between the sheep clone created in an owners-as-modifiers system to that created in an owners-as-dominators system. The sheep clone created in the owners-as-modifiers system would display owners-as-modifiers properties. The solid boxes, arrows, and dotted boxes in

Fig. 8.8 represent objects, references, and the representation of objects respectively. The top two diagrams represent sheep cloning in an owners-as-modifiers system, while the bottom two diagrams represent sheep cloning in an owners-as-dominators system. In this example, the caller of the sheep cloning function (object `C`) comes from inside the representation of the object being cloned (object `A`). The sheep clone for both prescriptive ownership system is a duplicate of their respective object `A`. The example in Fig. 8.8 illustrates how the representation of the sheep clone follows the prescriptive policy of the system.

Objects inside a representation that are unreachable from the object that representation is representing, should not be considered, at least when sheep cloning, to be part of that representation. Occurrence of this scenarios should be considered a flaw in the design, or possibly a misuse of ownership types. The semantics for sheep cloning in an owner-as-modifiers system should not copy an object only to be immediately garbage collected. Sheep cloning in an owners-as-modifiers system should instead only copy the objects in the representation that are (transitively) reachable from the object being cloned.

8.2.2 Serialisation

Serialisation is the process of converting data and data structures to data streams [79]. Serialisation is commonly used in object-oriented languages to perform object cloning where the object to be cloned is serialised, copied and then deserialised to create the clone [12].

An object's structure is serialized based on the selected rule set for the type of that object [28]. A rule set can be user definable and provided by the author of the class within a class definition, or by an external file called a surrogate. Creating sheep clones with serialisation in a system that has ownership types would in practice be similar to how we created sheep clones in our `mapSC` formalism. In the `mapSC` formalism, a map is created

for the object being cloned, the map contains the object structure of the object being cloned and the object structure of that object's sheep clone. The map can be viewed as the data streams for object being cloned and the clone. The co-domain of the map is the data stream of the object being cloned, and the range of the map is the data stream of that object's clone. Creating the map is the serialisation and deserialisation process in object cloning.

Sheep cloning with serialisation becomes more difficult when there are no ownership types. The obvious approach would be to serialise with a surrogate that contains ownership information of every object in the system. Serialisation by itself is not enough to create sheep clones. The additional information required of the object being cloned must come from some where else, but once that is established then serialisation is a sustitute for object cloning.

8.2.3 Prototypes

In a prototype-based programming language, such as JavaScript [1] and Self [83], objects are created by cloning prototypes, whereby eliminating the need for classes. A prototype can either be an existing object or a blank object. The blank object is commonly the root object of the system, such as the *Object* prototype in JavaScript. Objects created from *Object* contain a set of default slots (methods and properties).

Every object in a prototype-based language contains a clone or a copy method describing how that object is cloned, and is called when creating a new object of this prototype. For Self, the default `copy` message is a shallow copy, and the primitive `_Clone` method is also a shallow clone. For a more expressive `copy` message, each object is responsible for implementing it themself. In JavaScript, the most common way to create objects is to use the `new` keyword along with the object constructor of the desired prototype. Alternatively, objects can be created with the statement

`Object.create`, which explicitly creates an instance for the prototype stated in `Object`.

Sheep cloning is an automated object cloning procedure that creates clones with functionalities as well as the structure that closely resemble the desired purpose of the cloned object. Sheep cloning should be considered the default cloning operation for object creation in a prototype-based language. For an object to have a meaningful prototype, the programmer is required to define the cloning operation for that object. Tailoring a cloning implementation could possibly be strenuous for the programmer.

For a prototype-based language to perform sheep cloning it would require ownership. The additional restriction on the structure of the heap from ownership can be considered as an additional burden for the language. We believe, however, there are some inherent connections between object creation through prototyping and certain benefits provided by ownership. Object creation from existing objects requires the programmer to have an in-depth knowledge of every prototype in the system. Eliminating classes in prototype-based languages also eliminates some information regarding object structures that would otherwise be provided by the class hierarchies of the system. The tree-like structure of the heap provided by ownership allows for easier reasoning of the objects in the system.

The Prototype pattern is a design pattern [34] used to create new objects of similar nature by copying an existing prototype. The prototype of a Prototype pattern is a base class that is commonly implemented as an abstract class with a virtual clone method. Each concrete prototype definition would extend the prototype and implement their clone method as require for that instantiation of the prototype. A new object is created when the prototype's clone method is called.

One of the most difficult aspects of implementing the Prototype pattern is implementing the clone operation correctly for each concrete prototype. The standard approach is to use the cloning operation supported by the language used to implement the pattern. In most cases the default cloning

operation would be shallow cloning. We believe sheep cloning would be a more suitable cloning operation for the Prototype pattern. Sheep cloning would decide for each concrete prototype the objects that are necessary to be shared as well as those that are required to be copied.

Appendix A

Proofs for `descripSC`

In this appendix, we present the proof for type soundness of the `descripSC` formalism. The proofs for the type preservation theorem are presented in full detail, however, the lesser lemmas have not been presented. The complete proof of this system only exists in the form of hand-written hard copy.

Type Preservation

Subject Reduction

Theorem 1 : Subject Reduction

If:

a. $\mathcal{H} \vdash \mathtt{e} : N$

b. $\vdash \mathcal{H}$ OK

c. $\mathtt{e}\mathtt{;}\ \mathcal{H} \rightsquigarrow \mathtt{e}'\mathtt{;}\ \mathcal{H}'$

then:

$\vdash \mathcal{H}'$ OK

$\mathcal{H}' \vdash \mathtt{e}' : N$

or:

$\mathtt{e}' = \mathtt{err}$

Proof by induction on the derivation of **c**, with a case analysis on the last step:

Case: R-Field

1.	$\mathtt{e} = \iota.\mathtt{f}$	} by the definition of R-FIELD on **c**
2.	$\mathtt{e}' = v_i$	
3.	$\mathcal{H}' = \mathcal{H}$	
4.	$\mathcal{H}(\iota) = \{N'; \overline{\mathtt{f} \to v}\}$	by the premise of R-FIELD on **c**, **1 - 3**
5.	$\mathcal{H} \vdash \iota.\mathtt{f} : N$	by **a**, **1**
6.	$\mathcal{H} \vdash [\gamma/\mathtt{this}]N'' <: N$	} by LEMMA 10 on **5**
7.	$\mathcal{H} \vdash \iota : \mathtt{o}:\mathtt{C}<\overline{\mathtt{o}}>$	
8.	$fType(\mathtt{f}_i, \mathtt{o}:\mathtt{C}<\overline{\mathtt{o}}>) = N''$	
9.	$\forall \iota' \to \{N^3; \overline{\mathtt{f}' \to v'}\} \in \mathcal{H}$:	} by the premise of F-HEAP on **b**
10.	$\mathcal{H} \vdash N^3$ OK	
11.	$\overline{fType(\mathtt{f}', N^3) = N^4}$	
12.	$\mathcal{H} \vdash \overline{v' : [\iota'/\mathtt{this}]N^4}$	
13.	**let** $\iota' \to \{N^3; \overline{\mathtt{f}' \to v'}\} = \iota \to \{N'; \overline{\mathtt{f} \to v}\}$ by **9**, **4**	
14.	$\iota' = \iota$	} by **16**
15.	$N^3 = N'$	
16.	$\overline{\mathtt{f}} = \overline{\mathtt{f}'}$	
17.	$\overline{v} = \overline{v'}$	
18.	$\mathcal{H} \vdash N' <: \mathtt{o}:\mathtt{C}<\overline{\mathtt{o}}>$	by LEMMA 17 on **7**, **4**, **b**
19.	$fType(\mathtt{f}_i, N') = N^4$	by **11**, **17**, **20**
20.	$N^4 = N''$	by LEMMA 19 on **22**, **18**, **8**
21.	$\mathcal{H} \vdash \mathtt{v}_i : [\iota'/\mathtt{this}]N^4$	by **12**, **21**
22.	$\mathcal{H} \vdash \mathtt{v}_i : [\iota'/\mathtt{this}]N''$	by **25**, **23**
23.	$\mathcal{H} \vdash N$ OK	by LEMMA 24 on **a**, **b**
24.	$\mathcal{H} \vdash v_i : N$	by T-SUBS on **26**, **27**, **6**

25. $\mathcal{H}' \vdash v_i : N$ by LEMMA 25 on **a**, **b**, **c**, **1**, **2**, **3**

26. $\mathcal{H}' \vdash \mathtt{e}' : N$ by **30**, **2**

27. $\vdash \mathcal{H}'$ OK by **b**, **3**

28. done by **32**, **31**

Case: R-Assign

1. $\mathtt{e} = (\iota\,.\,\mathtt{f}_i = v)$
2. $\mathtt{e}' = v$
} by the definition of R-FIELD on **c**

3. $\mathcal{H}(\iota) = \{N'; \overline{\mathtt{f} \to v}\}$
4. $\mathcal{H}' = \mathcal{H}[\iota \mapsto \{N'; \overline{\mathtt{f} \to v}[\mathtt{f}_i \mapsto v]\}]$
} by the premise of R-ASSIGN on **c** with **1**

5. $\mathcal{H} \vdash \iota\,.\,\mathtt{f}_i = v : N$ by **c**, **1**

6. $\mathcal{H} \vdash \iota : \mathtt{o}\mathtt{:}\mathtt{C}\mathtt{<}\overline{\mathtt{o}}\mathtt{>}$
7. $\mathcal{H} \vdash v : N''$
8. $fType(\mathtt{f}_i, \mathtt{o}\mathtt{:}\mathtt{C}\mathtt{<}\overline{\mathtt{o}}\mathtt{>}) = N^3$
9. $\mathcal{H} \vdash N'' <: \mathtt{[}\iota\mathtt{/this]}\,N^3$
10. $\mathcal{H} \vdash N'' <: N$
} by LEMMA 11 on **5**, **b**

11. $\mathcal{H} \vdash N$ OK by LEMMA 24 on **a**, **b**

12. $\mathcal{H} \vdash v : N$ by T-SUBS on **7**, **10**, **11**

13. $\mathcal{H}' \vdash v : N$ by LEMMA 25 on **a**, **b**, **c**, **1**, **2**, **4**

14. $\mathcal{H}' \vdash \mathtt{e}' : N$ by **13**, **2**

15. $\forall \iota' \to \{N^4; \overline{\mathtt{f}' \to v'}\} \in \mathcal{H}$**:**
16. $\mathcal{H} \vdash N^4$ OK
17. $\overline{fType\,(\mathtt{f}',\ N^4)\ =\ N^5}$
18. $\mathcal{H} \vdash \overline{v' : \mathtt{[}\iota'\mathtt{/this]}\,N^5}$
19. $\forall\, v' \in \overline{v'}$**:**
20. $v' \neq \mathtt{null} \Rightarrow v' \in \mathtt{dom}(\mathcal{H})$
21. $\mathcal{H} \vdash \iota' \preceq own_{\mathcal{H}}(v')$
} by the premise of F-HEAP on **b**

22. **let** $\iota' \to \{N^4; \overline{\mathtt{f}' \to v'}\} = \iota \to \{N'; \overline{\mathtt{f} \to v}\}$ by **12**, **3**

23. $\iota' = \iota$ } by **19**
24. $N^4 = N'$ } by **19**
25. $\overline{\mathtt{f}} = \overline{\mathtt{f}'}$ } by **19**
26. $\overline{v} = \overline{v'}$ } by **19**
27. $\mathcal{H} \vdash N'$ OK by **13**, **22**
28. $\mathcal{H}' \vdash N'$ OK by LEMMA 26 on **34**, **b**, **c**, **1**, **2**, **4**
29. $\mathcal{H} \vdash N' <: \mathtt{o}\mathtt{:}\mathtt{C}\mathtt{<}\overline{\mathtt{o}}\mathtt{>}$ by LEMMA 17 on **6**, **3**, **b**
30. $fType(\mathtt{f}_i, N') = N^5$ by **14**, **22**, **23**
31. $N^5 = N^3$ by LEMMA 19 on **25**, **26**, **8**
32. $fType(\mathtt{f}_i, N') = N^3$ by **26**, **27**
33. $\mathcal{H} \vdash v_i' : [\iota'/\mathtt{this}]N^5$ by **15**
34. $\mathcal{H} \vdash v_i : [\iota/\mathtt{this}]N^3$ by **28**, **24**, **23**, **27**
35. $\mathcal{H} \vdash [\iota/\mathtt{this}]N^3$ OK by LEMMA 24 on **29**, **b**
36. $\mathcal{H} \vdash v : [\iota/\mathtt{this}]N^3$ by T-SUBS on **7**, **8**, **30**
37. $\mathcal{H}' \vdash v : [\iota/\mathtt{this}]N^3$ by LEMMA 25 on **32**, **b**, **c**, **1**, **2**, **4**

Case analysis on $v = \mathtt{null}$
xxxviii. $\vdash \mathcal{H}'$ OK by **4**, **b**, **35**, **32**, **33**, `case`
xxxix. done by **14**, **xxxviii**

Case analysis on $v \neq \mathtt{null}$
xl. $v \in \mathtt{dom}(\mathcal{H}')$ by the premise of T-VAR on **13**, `case`
xli. $\mathcal{H} \vdash \iota : N'$ by T-SUBS on **6**, **25**, **34**
xlii. $\mathcal{H}' \vdash \iota : N'$ by LEMMA 25 on **xli**, **b**, **c**, **1**, **2**, **4**
xliii. $\mathcal{H}' \vdash \iota \preceq own_{\mathcal{H}'}(v)$ by LEMMA 28 on **xlii**, **32**, **33**
xliv. $\vdash \mathcal{H}'$ OK by **4**, **b**, **35**, **32**, **33**, `case`, **xl**, **xliii**

xlv. done — by **14**, **xliv**

Case: R-New

1. $\texttt{e} = \texttt{new o:C<}\overline{\texttt{o}}\texttt{>}$
2. $\texttt{e}' = \iota$
} by the definition of R-NEW on **c**

3. $\mathcal{H}(\iota)\ undefined$
4. $fields(\texttt{C}) = \overline{\texttt{f}}$
5. $\mathcal{H}' = \mathcal{H}, \iota \rightarrow \{\texttt{o:C<}\overline{\texttt{o}}\texttt{>; }\overline{\texttt{f}\rightarrow\texttt{null}}\}$
} by the premise of R-NEW on **c** with **1**

6. $\mathcal{H} \vdash \texttt{new o:C<}\overline{\texttt{o}}\texttt{>} : N$ — by **a**, **1**

7. $\mathcal{H} \vdash \texttt{o:C<}\overline{\texttt{o}}\texttt{>}$ OK
8. $\mathcal{H} \vdash \texttt{o:C<}\overline{\texttt{o}}\texttt{>} <: N$
} by LEMMA 12 on **5**, **b**

9. $\mathcal{H}'(\iota) = \{\texttt{o:C<}\overline{\texttt{o}}\texttt{>; }\overline{\texttt{f}\rightarrow\texttt{null}}\}$ — by **4**

10. $\mathcal{H}' \vdash \iota : \texttt{o:C<}\overline{\texttt{o}}\texttt{>}$ — by T-VAR on **8**

11. $\mathcal{H} \vdash N$ OK — by LEMMA 24 on **a**, **b**

12. $\mathcal{H}' \vdash N$ OK — by LEMMA 26 on **10**, **b**, **c**, **1**, **2**, **4**

13. $\mathcal{H}' \vdash \texttt{o:C<}\overline{\texttt{o}}\texttt{>} <: N$ — by LEMMA 27 on **7**, **b**, **c**, **1**, **2**, **4**

14. $\mathcal{H}' \vdash \iota : N$ — by T-SUBS on **9**, **13**, **12**

15. $\texttt{class C<...> \{}\overline{N'\ \texttt{f}}\texttt{; ... \}}$ — by **4**, $\texttt{for some }\overline{N'}$

16. $\overline{fType\texttt{(f, o:C<}\overline{\texttt{o}}\texttt{>)}} = \overline{N'}$ — by **15**

17. $\mathcal{H} \vdash \overline{N'}$ OK — by LEMMA 20 on **b**, **6**, **16**

18. $\mathcal{H}' \vdash \overline{N'}$ OK — by LEMMA 26 on **17**, **b**, **c**, **1**, **2**, **4**

19. $\mathcal{H}' \vdash \overline{\texttt{[}\iota\texttt{/this]}N'}$ OK — by LEMMA 29 on **14**, **18**

20. $\mathcal{H}' \vdash \overline{\texttt{null : [}\iota\texttt{/this]}N'}$ — by T-NULL on **19**

21. $\mathcal{H}' \vdash \texttt{o:C<}\overline{\texttt{o}}\texttt{>}$ OK — by LEMMA 26 on **20**, **b**, **c**, **1**, **2**, **4**

22. $\vdash \mathcal{H}'$ OK — by F-HEAP on **8**, **21**, **16**, **20**, with **4** and **b**

23. $\mathcal{H}' \vdash \texttt{e}' : N$ — by **14**, **2**

24. done — by **22**, **23**

Case: R-Invk

1. $\mathtt{e} = \iota.\mathtt{m}(v)$ } by the definition of R-INVK on **c**

2. $\mathtt{e}' = [v/\mathtt{x},\ \iota/\mathtt{this}]\mathtt{e}''$ } by the definition of R-INVK on **c**

3. $\mathcal{H}' = \mathcal{H}$ } by the definition of R-INVK on **c**

4. $\mathcal{H}(\iota) = \{\mathtt{o}:\mathtt{C}\langle\overline{\mathtt{o}}\rangle;\ ...\}$ } by the premise of R-INVK on **c**, **1**, **2**

5. $mBody(\mathtt{m}, \mathtt{o}:\mathtt{C}\langle\overline{\mathtt{o}}\rangle) = (\mathtt{x};\ \mathtt{e}'')$ } by the premise of R-INVK on **c**, **1**, **2**

6. $\mathcal{H} \vdash \iota.\mathtt{m}(v) : N$ by **a**, **1**

7. $\mathcal{H} \vdash \iota : \mathtt{o}':\mathtt{C}'\langle\overline{\mathtt{o}'}\rangle$ } by LEMMA 13 on **7**, **b**

8. $\mathcal{H} \vdash v : N''$ } by LEMMA 13 on **7**, **b**

9. $mType(\mathtt{m}, \mathtt{o}':\mathtt{C}'\langle\overline{\mathtt{o}'}\rangle) = N' \to N^3$ } by LEMMA 13 on **7**, **b**

10. $\mathcal{H} \vdash N'' <: [\iota/\mathtt{this}]N'$ } by LEMMA 13 on **7**, **b**

11. $\mathcal{H} \vdash [\iota/\mathtt{this}]N^3 <: N$ } by LEMMA 13 on **7**, **b**

12. $\mathcal{H} \vdash \mathtt{o}:\mathtt{C}\langle\overline{\mathtt{o}}\rangle <: \mathtt{o}':\mathtt{C}'\langle\overline{\mathtt{o}'}\rangle$ by LEMMA 17 on **8**, **5**, **b**

13. $mType(\mathtt{m}, \mathtt{o}:\mathtt{C}\langle\overline{\mathtt{o}}\rangle) = N' \to N^3$ by LEMMA 21 on **13**, **10**

14. $\mathcal{H} \vdash \mathtt{o}:\mathtt{C}\langle\overline{\mathtt{o}}\rangle$ OK by the premise of F-HEAP on **b**, **5**

15. $\mathcal{H}, \mathtt{this}:\mathtt{o}:\mathtt{C}\langle\overline{\mathtt{o}}\rangle, \mathtt{x}:N' \vdash \mathtt{e}'' : N^3$ by LEMMA 23 on **14**, **6**, **16**

16. $\mathcal{H} \vdash \iota : \mathtt{o}:\mathtt{C}\langle\overline{\mathtt{o}}\rangle$ by T-SUBS on **8**, **13**, **16**

17. $\mathcal{H}, \mathtt{x}:[\iota/\mathtt{this}]N' \vdash [\iota/\mathtt{this}]\mathtt{e}'' : [\iota/\mathtt{this}]N^3$ by LEMMA 30 on **18**, **17**

18. $\mathcal{H}, \mathtt{this}:\mathtt{o}:\mathtt{C}\langle\overline{\mathtt{o}}\rangle \vdash N'$ OK } by LEMMA 22 on **16**, **14**

19. $\mathcal{H}, \mathtt{this}:\mathtt{o}:\mathtt{C}\langle\overline{\mathtt{o}}\rangle \vdash N^3$ OK } by LEMMA 22 on **16**, **14**

20. $\mathcal{H} \vdash [\iota/\mathtt{this}]N'$ OK by LEMMA 29 on **18**, **20**

21. $\mathcal{H} \vdash [\iota/\mathtt{this}]N^3$ OK by LEMMA 29 on **18**, **19**

22. $\mathcal{H} \vdash v : [\iota/\mathtt{this}]N'$ by T-SUBS on **9**, **11**, **20**

23. $\mathcal{H} \vdash [v/\mathtt{x}]([\iota/\mathtt{this}]\mathtt{e}'') : [v/x]([\iota/\mathtt{this}]N^3)$ by LEMMA 31 on **22**, **19**

24. $fv([\iota/\mathtt{this}]N') = \emptyset$ by LEMMA 18 on **20**

25. $\mathcal{H} \vdash [v/\mathtt{x}]([\iota/\mathtt{this}]\mathtt{e}'') : [\iota/\mathtt{this}]N^3$ by **23**, **24**

26. $\mathcal{H} \vdash [v/\mathtt{x},\ \iota/\mathtt{this}]\mathtt{e}'' : [\iota/\mathtt{this}]N^3$ by substitution convention on **25**

27. $\mathcal{H} \vdash N$ OK by LEMMA 24 on **a**, **b**

28.	$\mathcal{H} \vdash \texttt{[}v\texttt{/x, }\iota\texttt{/this]e}'' : N$	by T-SUBS on **26**, **12**, **27**
29.	$\mathcal{H}' \vdash \texttt{[}v\texttt{/x, }\iota\texttt{/this]e}'' : N$	by **28**, **3**
30.	$\mathcal{H}' \vdash \texttt{e}' : N$	by **29**, **2**
31.	$\vdash \mathcal{H}'$ OK	by **b**, **3**
32.	done	by **30**, **31**

Case: R-Sheep

1. $\mathtt{e} = \mathtt{sheep}(\iota)$
2. $\mathtt{e}' = [\overline{\iota''/\iota'}]\,\iota$
3. $\mathcal{H}' = \mathcal{H}, \mathcal{H}''$
} by the definition of R-SHEEP on **c**

4. $\overline{\iota' \rightarrow \{N';\ \overline{\mathtt{f} \rightarrow v}\}} = \{\iota' \rightarrow \{N';\ \overline{\mathtt{f} \rightarrow v}\} \in \mathcal{H} \mid \mathcal{H} \vdash \iota' \preceq \iota\}$ by the premise of R-SHEEP on **c**, **1**, **2**, **3**

5. $\mathcal{H}(\overline{\iota''})$ undefined
6. $\mathcal{H}'' = \overline{\iota'' \rightarrow [\overline{\iota''/\iota'}]\{N';\ \overline{\mathtt{f} \rightarrow v}\}}$
} by the premise of R-SHEEP on **c**, **1**, **2**, **3**

7. $\mathcal{H} \vdash \mathtt{sheep}(\iota) : N$ by **a**, **1**

8. $\mathcal{H} \vdash \iota : N''$
9. $\mathcal{H} \vdash N'' <: N$
} by LEMMA 14 on **7**, **b**

10. $\mathcal{H} \vdash N$ OK by LEMMA 24 on **a**, **b**

11. $\mathcal{H} \vdash \iota : N$ by T-SUBS on **8**, **9**, **10**

12. $\mathcal{H} \vdash \iota \preceq \iota$ by I-REFL on ι

13. $\iota \in \overline{\iota'}$ by **4**, **12**

14. $\mathcal{H}(\iota) = \{N;\ \overline{\mathtt{f} \rightarrow v}\}$ by **11**, **4**

15. **let** $[\overline{\iota''/\iota'}]\iota = \iota''$

16. $\mathcal{H}''(\iota'') = [\overline{\iota''/\iota'}]\{N;\ \overline{\mathtt{f} \rightarrow v}\}$ by **6**, **14**, **15**

17. $\mathcal{H}'' \vdash \iota'' : [\overline{\iota''/\iota'}]N$ by T-VAR on **16**

18. **let** $N' = \mathtt{o} : \mathtt{C}\langle\overline{\mathtt{o}}\rangle$

19. $\mathcal{H} \vdash \iota : \mathtt{o} : \mathtt{C}\langle\overline{\mathtt{o}}\rangle$ by **11**, **18**

20. $\forall\, \mathtt{o}_i \in \overline{\mathtt{o}} : \mathcal{H} \vdash \mathtt{o} \preceq \mathtt{o}_i$ by LEMMA 9 on **10**, **18**

21. $\mathcal{H} \nvdash own_{\mathcal{H}}(\iota) \preceq \iota$ by the premise of F-HEAP on **b**

22. $own_{\mathcal{H}}(\iota) = \mathtt{o}$ by **19**, **18**

23. $\mathcal{H} \nvdash \mathtt{o} \preceq \iota$ by **21**, **22**

24. $\forall\, \mathtt{o}_i \in \overline{\mathtt{o}} : \mathcal{H} \nvdash \mathtt{o}_i \preceq \iota$ by LEMMA 7 on **23**, **20**

25. $\forall \iota' \in \overline{\iota'} : \iota' \notin \{\overline{\mathtt{o}}, \mathtt{o}\}$ by **23**, **24**, **4**

26. $[\overline{\iota''/\iota'}](\mathtt{o} : \mathtt{C}\langle\overline{\mathtt{o}}\rangle) = \mathtt{o} : \mathtt{C}\langle\overline{\mathtt{o}}\rangle$ by **25**

27. $\mathcal{H}'' \vdash \iota'' : [\overline{\iota''/\iota'}](\mathtt{o} : \mathtt{C}\langle\overline{\mathtt{o}}\rangle)$ by **17**, **18**

28. $\mathcal{H}'' \vdash \iota'' : \mathtt{o} : \mathtt{C}\langle\overline{\mathtt{o}}\rangle$ by **27**, **26**

29.	$\mathcal{H}, \mathcal{H}'' \vdash \iota'' : \mathtt{o}\,\mathtt{:}\,\mathtt{C}\!<\!\overline{\mathtt{o}}\!>$	by LEMMA 36 on **28**, **5**, **6**
30.	$\mathcal{H}, \mathcal{H}'' \vdash [\overline{\iota''/\iota'}]\iota : N$	by **29**, **15**, **18**
31.	$\mathcal{H}' \vdash [\overline{\iota''/\iota'}]\iota : N$	by **30**, **3**
32.	$\vdash \mathcal{H}, \mathcal{H}''$ OK	by LEMMA 1 on **c**, **1**, **2**, **3**, **b**
33.	$\vdash \mathcal{H}'$ OK	by **32**, **3**
34.	done	by **33**, **31**

Case: R-Field-Null/R-Assign-Null/R-Invk-Null

Trivial as `e` = `err` by definition of these cases.

Case: RC-Invk/RC-Assign/RC-Sheep

Trivial by inductive hypothesis over the premise of these cases.

Case: RC-Invk-Err/RC-Assign-Err/RC-Sheep-Err

Trivial as `e` = `err` by definition of these cases.

Sheep Reduction Heap Well-formedness

Lemma 1 : Sheep cloning reduction preserves heap well-formedness

If:

a. $\texttt{sheep}(\iota)\texttt{;}\ \mathcal{H} \rightsquigarrow [\overline{\iota''/\iota'}]\iota\texttt{;}\ \mathcal{H}, \mathcal{H}'$

b. $\vdash \mathcal{H}$ OK

then:

$\vdash \mathcal{H}, \mathcal{H}'$ OK

Proof by induction on the derivation of **a**, with a case analysis on the last step:

1. $\overline{\iota' \rightarrow \{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\}} = \{\iota' \rightarrow \{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\} \in \mathcal{H} \mid \mathcal{H} \vdash \iota' \preceq \iota^*\}$

2. $\mathcal{H}(\overline{\iota''})\ \texttt{undefined}$

3. $\mathcal{H}' = \overline{\iota'' \rightarrow [\overline{\iota''/\iota'}]\{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\}}$

} by the premises of R-SHEEP on **a**

4. **let** $\mathcal{H}'' = \overline{\iota' \rightarrow \{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\}}$

5. $\mathcal{H}'' \subseteq \mathcal{H}$ by **4, 1**

6. $\mathcal{H} \vdash \mathcal{H}''$ OK by the definition of F-HEAP on **b** with **5**

7. $\mathcal{H}, \mathcal{H}' \vdash \mathcal{H}''$ OK by WEAKENING LEMMA 34 on **6, 3, 2**

8. $\texttt{dom}(\mathcal{H}) \cap \texttt{dom}(\mathcal{H}') = \emptyset$ by **2, 3**

9. $\mathcal{H} \vdash \mathcal{H}$ OK by the premise of F-HEAPE on **b**

10. $\mathcal{H}, \mathcal{H}' \vdash \mathcal{H}$ OK by WEAKENING LEMMA 34 on **9, 8**

11. $\forall \iota' \to \{N\texttt{;}\ \overline{\texttt{f}\to v}\} \in \mathcal{H}''$**:**
12. $\mathcal{H} \vdash N$ OK
13. $\overline{fType(\texttt{f},\ N)\ =\ N'}$
14. $\mathcal{H} \vdash \overline{v\ \texttt{:}\ \texttt{[}\iota'\texttt{/this]}N'}$
15. $\forall v \in \overline{v}\texttt{:}\ v \neq \texttt{null} \Rightarrow \{v \in \texttt{dom}(\mathcal{H}) \wedge \mathcal{H} \vdash \iota' \preceq own_{\mathcal{H}}(v)\ \}$
} by the premise of F-HEAP on **6**

16. $\mathcal{H}' \vdash \overline{\iota''}$ OK — by F-ADDRESS on **3**

17. $\mathcal{H}, \mathcal{H}' \vdash \overline{\iota''}$ OK — by WEAKENING LEMMA 35 on **16, b, 2, 3**

18. $\overline{\iota''} = \texttt{dom}(\mathcal{H}')$ — by **3**

19. $\overline{\iota'} \in \texttt{dom}(\mathcal{H})$ — by **4, 5**

20. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota''/\iota'}]N$ OK — by LEMMA 2 on **12, b, 1, 2, 3**

21. $\overline{fType(\texttt{f},\ \texttt{[}\overline{\iota''/\iota'}\texttt{]}N)\ =\ \texttt{[}\overline{\iota''/\iota'}\texttt{]}N'}$ — by LEMMA 6 on **13, b, 1, 2, 3**

22. $\mathcal{H}, \mathcal{H}' \vdash \overline{\texttt{[}\overline{\iota''/\iota'}\texttt{]}v\ \texttt{:}\ \texttt{[}\overline{\iota''/\iota'}\texttt{]}\ (\texttt{[}\iota'\texttt{/this]}N')}$ by LEMMA 3 on **14, b, 1, 2, 3**

23. $\forall v \in \overline{\texttt{[}\overline{\iota''/\iota'}\texttt{]}v}\texttt{:}\ v \neq \texttt{null} \Rightarrow \{\ v \in \texttt{dom}(\mathcal{H}, \mathcal{H}') \wedge \mathcal{H}, \mathcal{H}' \vdash [\overline{\iota''/\iota'}](\iota' \preceq own_{\mathcal{H},\mathcal{H}'}(v))\}$ by **15, 3, 22** and LEMMA 4 on **15, b, 2, 3**

24. **let** $\overline{v'} = \overline{\texttt{[}\overline{\iota''/\iota'}\texttt{]}v}$

25. **let** $N'' = \texttt{[}\overline{\iota''/\iota'}\texttt{]}N$

26. **let** $\overline{N^3} = \overline{\texttt{[}\overline{\iota''/\iota'}\texttt{]}N'}$

27. $\mathcal{H}' = \overline{\iota'' \to\{N''\texttt{;}\ \ \overline{\texttt{f}\to v'}\}}$ — by **3, 24, 25**

28. $\forall \iota'' \to \{N''\texttt{;}\ \overline{\texttt{f}\to v'}\} \in \mathcal{H}'\texttt{:}$ — by **27**

29. $\mathcal{H}, \mathcal{H}' \vdash N''$ OK — by **20, 25, 28**

30. $\overline{fType(\texttt{f},\ N'')\ =\ N^3}$ — by **21, 25, 26, 28**

31. $\mathcal{H}, \mathcal{H}' \vdash \overline{v'\ \texttt{:}\ \texttt{[}\iota''\texttt{/this]}N^3}$ — by **22, 24, 26, 28**

32. $\forall v' \in \overline{v'}\texttt{:}\ v' \neq \texttt{null} \Rightarrow \{v' \in \texttt{dom}(\mathcal{H}, \mathcal{H}') \wedge \mathcal{H}, \mathcal{H}' \vdash \iota'' \preceq own_{\mathcal{H},\mathcal{H}'}(v')\}$ by **23, 24, 28**

33. $\mathcal{H}, \mathcal{H}' \vdash \mathcal{H}'$ OK — by F-HEAP on **27 - 32**

34. $\mathcal{H}, \mathcal{H}' \vdash \mathcal{H}, \mathcal{H}'$ OK — by F-HEAP on **33, 10, 8**

35. done by **34**

Lemma 2 : Address Substitution preserves Type Well-Formedness

If:

a. $\mathcal{H} \vdash N$ OK

b. $\vdash \mathcal{H}$ OK

c. $\mathcal{H}\,(\overline{\iota})$ `undefined`

d. $\mathcal{H}' = \overline{\iota \rightarrow \texttt{[}\overline{\iota/\iota'}\texttt{]}\{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\}}$

then:

$\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]N$ OK

Proof by natural deduction.

1. **let** $N = \circ$: C<$\overline{\circ}$>

2. `class` C<$\overline{\circ_l \preceq \texttt{x} \preceq \circ_u}$> $\{\overline{N\ \texttt{f;}}\ \ \overline{M}\}$

3. $\mathcal{H} \vdash \circ$ OK

4. $\mathcal{H} \vdash \overline{\circ}$ OK

5. $\mathcal{H} \vdash \overline{[\circ/\texttt{x}]}(\overline{\circ_l \preceq \texttt{x}})$

6. $\mathcal{H} \vdash \overline{[\circ/\texttt{x}]}(\overline{\texttt{x} \preceq \circ_u})$

} by the premise of F-CLASS on **a** and **1**

7. $\forall\, \circ_i \in \overline{\circ} : \mathcal{H} \vdash \circ \preceq \circ_i$ by

8. $\mathcal{H} \vdash \overline{\overline{[\circ/\texttt{x}]}\circ_l \preceq \overline{[\circ/\texttt{x}]}\texttt{x}}$ by substitution convention on **5**

9. $\mathcal{H} \vdash \overline{\overline{[\circ/\texttt{x}]}\texttt{x} \preceq \overline{[\circ/\texttt{x}]}\circ_u}$ by substitution convention on **6**

10. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]\circ$ OK by LEMMA 5 on **3**, **b**, **c**, **d**

11. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]\overline{\circ}$ OK by LEMMA 5 on **4**, **b**, **c**, **d**

12. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}](\overline{\overline{[\circ/\texttt{x}]}\circ_l \preceq \overline{[\circ/\texttt{x}]}\texttt{x}})$ by LEMMA 4 on **8**, **b**, **c**, **d**

13. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}](\overline{[\circ/\texttt{x}]}(\overline{\circ_l \preceq \texttt{x}}))$ by substitution convention on **12**

14. $\mathcal{H}, \mathcal{H}' \vdash \overline{[\texttt{[}\overline{\iota/\iota'}\texttt{]}\circ/\texttt{x}]}(\overline{[\overline{\iota/\iota'}]\circ_l \preceq \texttt{x}})$ by substitution convention on **13** and $\overline{\iota'} \cap \overline{\texttt{x}} = \emptyset$

15. $\mathcal{H}, \mathcal{H}' \vdash \overline{[\overline{\iota/\iota'}]\circ/\mathtt{x}}](\overline{\circ_l \preceq \mathtt{x}})$ by substitution convention on **14** and $\overline{\iota'} \cap \overline{\circ_l} = \emptyset$

16. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}](\overline{[\circ/\mathtt{x}]\mathtt{x} \preceq [\circ/\mathtt{x}]\circ_u})$ by LEMMA 4 on **9**, **b**, **c**, **d**

17. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}](\overline{[\circ/\mathtt{x}](\mathtt{x} \preceq \circ_u)})$ by substitution convention on **16**

18. $\mathcal{H}, \mathcal{H}' \vdash \overline{[\overline{[\iota/\iota']}\circ/\mathtt{x}]}(\overline{\mathtt{x} \preceq [\iota/\iota']\circ_u})$ by substitution convention on **17** and $\overline{\iota'} \cap \overline{\mathtt{x}} = \emptyset$

19. $\mathcal{H}, \mathcal{H}' \vdash \overline{[\overline{[\iota/\iota']}\circ/\mathtt{x}]}(\overline{\mathtt{x} \preceq \circ_u})$ by substitution convention on **18** and $\overline{\iota'} \cap \overline{\circ_u} = \emptyset$

20. $\forall\, \circ_i \in \overline{\circ} : \mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}](\circ \preceq \circ_i)$ by LEMMA 4 on **7**, **b**, **c**, **d**

21. $\forall\, \circ_i \in \overline{\circ} : \mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]\circ \preceq [\overline{\iota/\iota'}]\circ_i$ by substitution convention on **20**

22. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]\circ : \mathtt{C}\overline{<[\overline{\iota/\iota'}]\circ>}$ OK by F-CLASS on **2**, **10**, **11**, **15**, **19**, **21**

23. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}](\circ : \mathtt{C}\overline{<\circ>})$ OK by substitution convention on **22**

24. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]N$ OK by **23**, **1**

25. done by **24**

Lemma 3 : Address Substitution preserves Value Typing

If:

- **a**. $\mathcal{H} \vdash v : N$
- **b**. $\vdash \mathcal{H}$ OK
- **c**. $\overline{\iota' \rightarrow \{N;\ \overline{\mathtt{f} \rightarrow v}\}} = \{\iota' \rightarrow \{N;\ \overline{\mathtt{f} \rightarrow v}\} \in \mathcal{H} \mid \mathcal{H} \vdash \iota' \preceq \iota^*\ \}$
- **d**. $\mathcal{H}\ (\overline{\iota})\ \ \mathtt{undefined}$
- **e**. $\mathcal{H}' = \overline{\iota \rightarrow [\overline{\iota/\iota'}]\{N;\ \overline{\mathtt{f} \rightarrow v}\}}$

then:

$\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]v : [\overline{\iota/\iota'}]N$

Proof by induction on the derivation of **a**, with a case analysis on the last step:

Case: D-Null

1. $v = \texttt{null}$ by the definition of D-NULL on **a**

2. $\mathcal{H} \vdash N$ OK by the premise of D-NULL on **a**

3. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]N$ OK by LEMMA 2 on **2**, **b**, **c**, **d**, **e**

4. $\texttt{null} \notin \overline{\iota'}$ by the definition of SYNTAX on

5. $[\overline{\iota/\iota'}]\texttt{null} = \texttt{null}$ by **4**

6. $\mathcal{H}, \mathcal{H}' \vdash \texttt{null} : [\overline{\iota/\iota'}]N$ by D-NULL on **3**

7. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]v : [\overline{\iota/\iota'}]N$ by **6**, **5**, **1**

8. done by **7**

Case: D-Addr

1. $v = \iota$ by the definition of D-ADDR on **a**

2. $\mathcal{H}(\iota) = \{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\}$ by the premise of D-ADDR on **a**

3. $[\overline{\iota/\iota'}]\iota \in \texttt{dom}(\mathcal{H}, \mathcal{H}')$ by **2**, **e**

Case analysis on ι

Case: $\iota \in \overline{\iota'}$

i. **let** $\iota = \iota'_n$ by `case`

ii. $N = N_n$ by **i**, **c**

iii. $[\overline{\iota/\iota'}]\iota'_n = \iota_n$ by **i**

iv. $\mathcal{H}' \vdash \iota_n : [\overline{\iota/\iota'}]N_n$ by T-VAR on **e**

v. $\mathcal{H}, \mathcal{H}' \vdash \iota_n : [\overline{\iota/\iota'}]N_n$ by LEMMA 37 on **iv**, **e**, **d**

vi. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]v : [\overline{\iota/\iota'}]N_n$ by **v**, **iii**, **i**, **1**

vii. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]v : [\overline{\iota/\iota'}]N$ by **vi**, **ii**

viii.	done	by **vii**

Case: $\iota \notin \overline{\iota'}$

i.	$[\overline{\iota/\iota'}]\iota = \iota$	by `case`
ii.	$\mathcal{H} \not\vdash \iota \preceq \iota^*$	by `case`, **c**, **2**
iii.	$\mathcal{H} \vdash \iota \preceq \mathtt{own}_{\mathcal{H}}(\iota)$	by I-OWNER on ι
iv.	$\mathcal{H} \not\vdash \mathtt{own}_{\mathcal{H}}(\iota) \preceq \iota^*$	by LEMMA 7 on **ii**, **iii**
v.	$\circ\mathtt{own}_{\mathcal{H}}(\iota) \notin \overline{\iota'}$	by **iv**, **c**
vi.	**let** $N = \circ : \mathtt{C}\langle\overline{\circ}\rangle$	
vii.	$\circ\mathtt{own}_{\mathcal{H}}(\iota) = \circ$	by definition of $\mathtt{own}_{\mathcal{H}}(\iota)$ with **2** and **vi**
viii.	$\circ \notin \overline{\iota'}$	by **v** and **vii**
ix.	$\mathcal{H} \vdash N$ OK	by the premise of F-HEAP on **b**, **2**
x.	$\mathcal{H} \vdash \circ : \mathtt{C}\langle\overline{\circ}\rangle$ OK	by **ix**, **vi**
xi.	$\forall\, \circ_i \in \overline{\circ} : \mathcal{H} \vdash \circ \preceq \circ_i$	by LEMMA 9 on **ix**, **vi**
xii.	$\forall\, \circ_i \in \overline{\circ} : \mathcal{H} \vdash \circ\mathtt{own}_{\mathcal{H}}(\iota) \preceq \circ_i$	by **xi**, **vi**
xiii.	$\forall\, \circ_i \in \overline{\circ} : \mathcal{H} \not\vdash \circ_i \preceq \iota^*$	by LEMMA 7 on **iv**, **xii**
xiv.	$\overline{\circ} \notin \overline{\iota'}$	by **xiii**, **c**
xv.	$[\overline{\iota/\iota'}]N = N$	by **viii**, **xiv**, **vi**
xvi.	$\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]v : [\overline{\iota/\iota'}]N$	by **a**, **i**, **xv**
xvii.	done	by **xvi**

Lemma 4 : Address Substitution preserves Inside Relation

If:

a.	$\mathcal{H} \vdash \iota \preceq \iota''$
b.	$\vdash \mathcal{H}$ OK
c.	$\overline{\iota' \rightarrow \{N\,;\ \overline{\mathtt{f} \rightarrow v}\}} = \{\iota' \rightarrow \{N\,;\ \overline{\mathtt{f} \rightarrow v}\} \in \mathcal{H} \mid \mathcal{H} \vdash \iota' \preceq \iota^*\}$
d.	$\mathcal{H}\,(\overline{\iota})$ `undefined`
e.	$\mathcal{H}' = \overline{\iota \rightarrow \{N\,;\ \overline{\mathtt{f} \rightarrow v}\}}$

then:

$\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}](\iota \preceq \iota'')$

Proof by induction on the derivation of **a**, with a case analysis on the last step:

Case: I-Ref

1.	$\iota = \iota'$	by the definition of I-REFL on **a**
2.	$\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]\iota \preceq [\overline{\iota/\iota'}]\iota$	by I-REFL on $[\overline{\iota/\iota'}]\iota$
3.	$\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}](\iota \preceq \iota)$	by substitution convention on **2**
4.	$\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}](\iota \preceq \iota')$	by **3, 1**
5.	done	by **4**

Case: I-Trans

1.	$\mathcal{H} \vdash \iota \preceq \iota''$	} by the premise of I-TRANS on **a**
2.	$\mathcal{H} \vdash \iota'' \preceq \iota'$	
3.	$\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}](\iota \preceq \iota'')$	by the inductive hyp on **1, b - e**
4.	$\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}](\iota'' \preceq \iota')$	by the inductive hyp on **2, b - e**
5.	$\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]\iota \preceq [\overline{\iota/\iota'}]\iota''$	by substitution convention on **3**
6.	$\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]\iota'' \preceq [\overline{\iota/\iota'}]\iota'$	by substitution convention on **4**
7.	$\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]\iota \preceq [\overline{\iota/\iota'}]\iota'$	by I-TRANS on **5, 6**
8.	$\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}](\iota \preceq \iota')$	by substitution convention on **7**
9.	done	by **8**

Case: I-Rec

1.	$\iota' = \mathtt{own}_{\mathcal{H}}(\iota)$	by the definition of I-REC

on **a**

2. **let** $[\overline{\iota/\iota'}]\iota = \iota''$

3. $\iota \in \mathtt{dom}(\mathcal{H})$ by **1**

4. $\iota'' \in \mathtt{dom}(\mathcal{H}, \mathcal{H}')$ by **3, 2, d, e**

5. $\mathcal{H}, \mathcal{H}' \vdash \iota'' \preceq \mathtt{own}_{\mathcal{H},\mathcal{H}'}(\iota'')$ by I-REC on ι''

6. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]\iota \preceq \mathtt{own}_{\mathcal{H},\mathcal{H}'}([\overline{\iota/\iota'}]\iota)$ by **5, 2**

7. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]\iota \preceq [\overline{\iota/\iota'}](\mathtt{own}_{\mathcal{H},\mathcal{H}'}(\iota))$ by LEMMA 8 on **6, 3, c, d, e**

8. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}](\iota \preceq \mathtt{own}_{\mathcal{H},\mathcal{H}'}(\iota))$ by substitution convention on **7**

9. $\mathtt{own}_{\mathcal{H},\mathcal{H}'}(\iota) = \mathtt{own}_{\mathcal{H}}(\iota)$ by **4, d, e**

10. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}](\iota \preceq \iota')$ by **8, 9, 1**

11. done by **10**

Case: I-World

1. $\iota' = \mathtt{world}$ by the definition of I-WORLD on **a**

2. $[\overline{\iota/\iota'}]\ \mathtt{world} = \mathtt{world}$ by syntax of the system

3. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]\iota \preceq \mathtt{world}$ by the definition of I-WORLD on $[\overline{\iota/\iota'}]\iota$

4. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]\iota \preceq [\overline{\iota/\iota'}]\iota'$ by **3, 2, 1**

5. $\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}](\iota \preceq \iota')$ by substitution convention on **4**

6. done by **5**

Lemma 5 : Address Substitution preserves Context Well-Formedness

If:

a. $\mathcal{H} \vdash \circ$ OK

b. $\vdash \mathcal{H}$ OK

c. $\overline{\iota' \rightarrow \{N;\ \overline{\mathtt{f} \rightarrow v}\}} = \{\iota' \rightarrow \{N;\ \overline{\mathtt{f} \rightarrow v}\} \in \mathcal{H} \mid \mathcal{H} \vdash \iota' \preceq \iota^* \}$

d. $\mathcal{H}\,(\overline{\iota})\ \mathtt{undefined}$

e. $\mathcal{H}' = \overline{\iota \rightarrow [\overline{\iota/\iota'}]\{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\}}$

then:

$\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]\texttt{o}$ OK

Proof by natural deduction.

1.	$\texttt{o} \in \texttt{dom}(\mathcal{H})$	by the premise of F-ADDR on **a**
2.	$\mathcal{H}, \mathcal{H}' \vdash \texttt{o}$ OK	by WEAKENING LEMMA 33 on **a**, **e**, **d**
3.	$\mathcal{H}, \mathcal{H}' \vdash [\overline{\iota/\iota'}]\texttt{o}$ OK	by F-ADDR on **2**, **e**
4.	done	by **3**

Lemma 6 : Address Substitution preserves $fType$

If:

a. $fType(\texttt{f}_i, N) = N'$

b. $\mathcal{H}(\iota) = \{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\}$

c. $\vdash \mathcal{H}$ OK

d. $\mathcal{H} \vdash N$ OK

e. $\mathcal{H}\,(\overline{\iota})$ `undefined`

f. $\overline{\iota' \rightarrow\{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\}} = \{\iota' \rightarrow\{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\} \in \mathcal{H} \mid \mathcal{H} \vdash \iota' \preceq \iota^* \}$

then:

$fType(\texttt{f}_i, [\overline{\iota/\iota'}]N) = [\overline{\iota/\iota'}]\,N'$

Proof by natural deduction.

1.	**let** $N = \texttt{o}:\texttt{C}\texttt{<}\overline{\texttt{o}}\texttt{>}$	
2.	$\texttt{class C}\texttt{<}\overline{\texttt{o}_l \preceq \texttt{x} \preceq \texttt{o}_u}\texttt{>}\ \{\overline{N\ \texttt{f;}}\ \overline{M}\}$	} by the definition of $fType$ on **a**, **1**
3.	$N' = [\texttt{o}/\texttt{owner}, \overline{\texttt{o}/\texttt{x}}]N_i$	
4.	$[\overline{\iota/\iota'}]N = [\overline{\iota/\iota'}]\texttt{o}:\texttt{C}\texttt{<}\overline{[\overline{\iota/\iota'}]\,\texttt{o}}\texttt{>}$	by substitution convention of $[\overline{\iota/\iota'}]$ on **1**
5.	**let** $fType(\texttt{f}_i, [\overline{\iota/\iota'}]N) = N''$	

6. $N'' = [([\overline{\iota/\iota'}]\circ)/\texttt{owner}, \overline{\texttt{(}[\iota/\iota']\circ\texttt{)/x}}]N_i$ by the definition of $fType$ on **4**, **5**, **2**

7. $\texttt{owner}, \overline{\texttt{x}} \notin \overline{\iota'}$ by syntax convention on $\texttt{owner}, \overline{\texttt{x}}$

8. $[\overline{\iota/\iota'}]N' = [\overline{\iota/\iota'}]([\circ/\texttt{owner}, \overline{\circ/\texttt{x}}]N_i)$ by substitution convention of $[\overline{\iota/\iota'}]$ on **3**

9. $[\overline{\iota/\iota'}]N' = [\overline{\iota/\iota'}, ([\overline{\iota/\iota'}]\circ)/\texttt{owner}, \overline{\texttt{(}[\iota/\iota']\circ\texttt{)/x}}]N_i$ by substitution convention on **8** and **7**

10. **let** $N_i = \circ_i$ **:** $\texttt{C<}\overline{\circ_i}\texttt{>}$

11. $\circ_i \in \{\overline{\texttt{x}}, \texttt{this}, \texttt{owner}, \texttt{world}\}$

12. $\overline{\circ_i} \subseteq \{\overline{\texttt{x}}, \texttt{this}, \texttt{owner}, \texttt{world}\}$

} by the premise of T-CLASS on **2**, **10**

13. $\circ_i \notin \overline{\iota'}$ by syntax convention on **11**

14. $\forall\, \circ' \in \overline{\circ_i} : \circ' \notin \overline{\iota'}$ by syntax convention on **12**

15. $[\overline{\iota/\iota'}]\circ_i = \circ_i$ by **13**

16. $[\overline{\iota/\iota'}]\overline{\circ_i} = \overline{\circ_i}$ by **14**

17. $[\overline{\iota/\iota'}]N_i = N_i$ by **15**, **16**, **10**

18. $[\overline{\iota/\iota'}]N' = [([\overline{\iota/\iota'}]\circ)/\texttt{owner}, \overline{\texttt{(}[\iota/\iota']\circ\texttt{)/x}}]N_i$ by **9**, **17**

19. $fType(\texttt{f}_i, [\overline{\iota/\iota'}]N) = \texttt{[}\overline{\iota/\iota'}\texttt{]}\, N'$ by **5**, **6**, **18**

20. done by **19**

Lemma 7 : Inside Relation Structural Property One

If:

a. $\mathcal{H} \nvdash \iota \preceq \iota'$

b. $\mathcal{H} \vdash \iota \preceq \circ$

then:

$\mathcal{H} \nvdash \circ \preceq \iota'$

Proof by contradiction.

1. Assume that: $\mathcal{H} \vdash \circ \preceq \iota'$ by the contradiction assumption on $\mathcal{H} \nvdash \circ \preceq \iota'$

2. $\mathcal{H} \vdash \iota \preceq \iota'$ by I-TRANS on **b 1**

3. **false** by contradicting statements **2** and **a**

4. $\mathcal{H} \nvdash \circ \preceq \iota'$ by **1, 3**

5. done by **4**

Lemma 8 : Owner substitution equals substitution of owners

If:

a. $\iota \in \mathtt{dom}(\mathcal{H})$

b. $\vdash \mathcal{H}$ OK

c. $\overline{\iota' \rightarrow \{N\,;\ \overline{\mathtt{f} \rightarrow v}\}} = \{\iota' \rightarrow \{N\,;\ \overline{\mathtt{f} \rightarrow v}\} \in \mathcal{H} \mid \mathcal{H} \vdash \iota' \preceq \iota^* \}$

d. $\mathcal{H}(\bar{\iota})$ `undefined`

e. $\mathcal{H}' = \overline{\iota \rightarrow [\overline{\iota/\iota'}]\{N\,;\ \overline{\mathtt{f} \rightarrow v}\}}$

then:

$\mathtt{own}_{\mathcal{H},\mathcal{H}'}([\overline{\iota/\iota'}]\iota) = [\overline{\iota/\iota'}](\mathtt{own}_{\mathcal{H},\mathcal{H}'}(\iota))$

Proof by natural deduction with a case analysis on ι.

1. **let** $\mathcal{H}(\iota) = \{N\,;\ \overline{\mathtt{f} \rightarrow v}\}$ by **a**

2. **let** $N = \circ : \mathtt{C}\langle\overline{\circ}\rangle$

3. $\mathtt{own}_{\mathcal{H}}(\iota) = \circ$ by the definition of $own_{\mathcal{H}}$ on **1, 2**

4. $\mathtt{own}_{\mathcal{H},\mathcal{H}'}(\iota) = \circ$ by **3, e, d**

Case analysis on ι

Case: $\iota \in \overline{\iota'}$

i. $\mathcal{H}, \mathcal{H}'([\overline{\iota/\iota'}]\iota) = [\overline{\iota/\iota'}]\{N\,;\ \overline{\mathtt{f} \rightarrow v}\}$ by `case`, **1, e**

ii. $\mathtt{own}_{\mathcal{H},\mathcal{H}'}([\overline{\iota/\iota'}]\iota) = [\overline{\iota/\iota'}]\circ$ by the definition of $own_{\mathcal{H}}$ on **i, 2**

iii.	$\texttt{own}_{\mathcal{H},\mathcal{H}'}([\overline{\iota/\iota'}]\iota) = [\overline{\iota/\iota'}](\texttt{own}_{\mathcal{H},\mathcal{H}'}(\iota))$	by **ii, 4**
iv.	done	by **iii**

Case: $\iota \notin \overline{\iota'}$

i.	$[\overline{\iota/\iota'}]\iota = \iota$	by `case`
ii.	$\texttt{own}_{\mathcal{H},\mathcal{H}'}([\overline{\iota/\iota'}]\iota) = \texttt{o}$	by **i, 4**
iii.	$\mathcal{H} \nvdash \iota \preceq \iota^*$	by `case`, **c**
iv.	$\mathcal{H} \vdash \iota \preceq \texttt{o}$	by I-REC on ι with **1, 2**
v.	$\mathcal{H} \nvdash \texttt{o} \preceq \iota^*$	by LEMMA 7 on **iii, iv**
vi.	$\texttt{o} \notin \overline{\iota'}$	by **v, d**
vii.	$[\overline{\iota/\iota'}]\texttt{o} = \texttt{o}$	by **vi**
viii.	$[\overline{\iota/\iota'}](\texttt{own}_{\mathcal{H},\mathcal{H}'}(\iota)) = \texttt{o}$	by **vii, 4**
ix.	$\texttt{own}_{\mathcal{H},\mathcal{H}'}([\overline{\iota/\iota'}]\iota) = [\overline{\iota/\iota'}](\texttt{own}_{\mathcal{H},\mathcal{H}'}(\iota))$	by **ii, viii**
x.	done	by **ix**

Lemma 9 : Owner Parameters are Outside the Owner

If:

a.	$\mathcal{H} \vdash \texttt{o}\,\texttt{:C<}\overline{\texttt{o}}\texttt{>}$ OK
b.	$\vdash \mathcal{H}$ OK

then:

$\forall\, \texttt{o}_i \in \overline{\texttt{o}} : \mathcal{H} \vdash \texttt{o} \preceq \texttt{o}_i$

Proof by natural deduction.

1.	$\texttt{class C<}\overline{\texttt{o}_l \preceq \texttt{x} \preceq \texttt{o}_u}\texttt{>}\ \{\overline{N\ \texttt{f;}}\ \ \overline{M}\}$	by the premise of F-CLASS on **a**
2.	$\mathcal{H} \vdash \overline{[\texttt{o}/\texttt{x}]\texttt{o}_l \preceq \texttt{o}}$	by the premise of F-CLASS on **a**
3.	$\mathcal{H} \vdash \overline{\texttt{o}}, \texttt{o}$ OK	by the premise of F-CLASS on **a**
4.	$\forall\, \texttt{o}_l \in \overline{\texttt{o}_l} : \mathcal{H} \vdash \texttt{owner} \preceq \texttt{o}_l$	by the premise of T-CLASS on **1**
5.	$\mathcal{H} \vdash \texttt{[o/owner]}(\overline{[\texttt{o}/\texttt{x}]\texttt{o}_l \preceq \texttt{o}})$	by LEMMA 32 on **3,** `owner`
6.	$\texttt{owner} \notin \overline{\texttt{o}}$	by syntax convention on `owner` and $\overline{\texttt{o}}$

7. $\mathcal{H} \vdash \overline{[\texttt{[o/owner]o/x}, \texttt{o/owner}]\texttt{o}_l \preceq \texttt{o}}$ by **5, 6**

8. $\forall\, \overline{[\texttt{o/x}]\texttt{o}_l} \in \overline{\texttt{[o/x]o}_l} : \mathcal{H} \vdash [\overline{\texttt{o/x}}](\texttt{owner} \preceq \texttt{o}_l)$
by LEMMA 32 on **3**, $\overline{\texttt{x}}$

9. $\texttt{owner} \notin \overline{\texttt{x}}$ by syntax convention on owner and $\overline{\texttt{x}}$

10. $\forall\, \overline{[\texttt{o/x}]\texttt{o}_l} \in \overline{\texttt{[o/x]o}_l} : \mathcal{H} \vdash \texttt{owner} \preceq [\overline{\texttt{o/x}}]\texttt{o}_l$ by **8, 9**

11. $\forall\, [\texttt{o/owner}](\overline{[\texttt{o/x}]\texttt{o}_l}) \in \overline{\texttt{[o/owner]([o/x]o}_l\texttt{)}} : \mathcal{H} \vdash [\texttt{o/owner}](\texttt{owner} \preceq \overline{[\texttt{o/x}]\texttt{o}_l})$
by LEMMA 32 on **3**, owner

12. $\forall\, \overline{[\texttt{[o/owner]o/x}, \texttt{o/owner}]\texttt{o}_l} \in \overline{[\texttt{[o/owner]o/x}, \texttt{o/owner}]\texttt{o}_l} : \mathcal{H} \vdash \texttt{o} \preceq \overline{[\texttt{[o/owner}}$
by substitution convention on **11**

13. $\forall\, \texttt{o}_i \in \overline{\texttt{o}} : \mathcal{H} \vdash \texttt{o} \preceq \texttt{o}_i$ by IR-TRANS on **12, 7**

14. done by **13**

Conjecture 9 : Preservation of Ownership Acyclicity for Sheep Cloning

If:

a. $\mathcal{H} \nvdash own_{\mathcal{H}}(\iota) \preceq \iota$

b. $\vdash \mathcal{H}$ OK

c. $\mathcal{H} \vdash \iota \preceq \iota^*$

d. $\iota^* \in \mathrm{dom}(\mathcal{H})$

e. $\overline{\iota' \rightarrow \{N\texttt{;}\ \ \overline{\texttt{f}\rightarrow v}\}} = \{\iota' \rightarrow \{N\texttt{;}\ \overline{\texttt{f}\rightarrow v}\} \in \mathcal{H} \mid \mathcal{H} \vdash \iota' \preceq \iota^*\}$

f. $\mathcal{H}(\overline{\iota})\ \ \texttt{undefined}$

g. $\mathcal{H}' = \overline{\iota \rightarrow [\overline{\iota/\iota'}]\{N\texttt{;}\ \ \overline{\texttt{f}\rightarrow v}\}}$

then:

$\mathcal{H}, \mathcal{H}' \nvdash own_{\mathcal{H},\mathcal{H}'}([\overline{\iota/\iota'}]\iota) \preceq [\overline{\iota/\iota'}]\iota$

Inversion Lemmas

Lemma 10 : Inversion on the Field Lookup Expression

If:

a. $\mathcal{E};\Gamma \vdash \gamma\texttt{.f} : N$

then:

$\mathcal{E};\Gamma \vdash \gamma : \texttt{o:C<}\overline{\texttt{o}}\texttt{>}$

$fType(\texttt{f, o:C<}\overline{\texttt{o}}\texttt{>}) = N'$

$\mathcal{E};\Gamma \vdash \texttt{[}\gamma\texttt{/this]}N' <: N$

Proof by induction on the derivation of **a**, with a case analysis on the last step:

Case: T-Field

1.	$N = \texttt{[}\gamma\texttt{/this]}N'$	by the definition of T-FIELD on **a**
2.	$\mathcal{E};\Gamma \vdash \gamma : \texttt{o:C<}\overline{\texttt{o}}\texttt{>}$	by the premise of T-FIELD on **a**, **1**
3.	$fType(\texttt{f, o:C<}\overline{\texttt{o}}\texttt{>}) = N'$	
4.	$\mathcal{E};\Gamma \vdash \texttt{[}\gamma\texttt{/this]}N' <: \texttt{[}\gamma\texttt{/this]}N'$	by SR-REFL on $\texttt{[}\gamma\texttt{/this]}N'$
5.	$\mathcal{E};\Gamma \vdash \texttt{[}\gamma\texttt{/this]}N' <: N$	by **4**, **1**
6.	done	by **5**, **3**, **2**

Case: T-Subs

1.	$\mathcal{E};\Gamma \vdash \gamma\texttt{.f} : N''$	by the premise of T-SUBS on **a**
2.	$\mathcal{E};\Gamma \vdash N'' <: N$	
3.	$\mathcal{E};\Gamma \vdash N$ OK	
4.	$\mathcal{E};\Gamma \vdash \gamma : \texttt{o:C<}\overline{\texttt{o}}\texttt{>}$	by the inductive hyp on **1**
5.	$fType(\texttt{f, o:C<}\overline{\texttt{o}}\texttt{>}) = N'$	
6.	$\mathcal{E};\Gamma \vdash \texttt{[}\gamma\texttt{/this]}N' <: N''$	
7.	$\mathcal{E};\Gamma \vdash \texttt{[}\gamma\texttt{/this]}N' <: N$	by SR-TRANS on **2, 6**
8.	done	by **4**, **5**, **7**

Lemma 11 : Inversion on the Field Assignment Expression

If:

a. $\mathcal{E};\Gamma \vdash \gamma\texttt{.f = e} : N$

then:

$\mathcal{E};\Gamma \vdash \gamma : \texttt{o:C<}\overline{\texttt{o}}\texttt{>}$

$\mathcal{E};\Gamma \vdash \texttt{e} : N''$

$fType(\texttt{f, o:C<}\overline{\texttt{o}}\texttt{>}) = N'$

$\mathcal{E};\Gamma \vdash N'' <: \texttt{[}\gamma\texttt{/this]}N'$

$\mathcal{E};\Gamma \vdash N'' <: N$

Proof by induction on the derivation of **a**, with a case analysis on the last step:

Case: T-Assign

1.	$\mathcal{E};\Gamma \vdash \gamma : \texttt{o:C<}\overline{\texttt{o}}\texttt{>}$	} by the premise of T-ASSIGN on **a**
2.	$fType(\texttt{f, o:C<}\overline{\texttt{o}}\texttt{>}) = N'$	
3.	$\mathcal{E};\Gamma \vdash \texttt{e} : N$	
4.	$\mathcal{E};\Gamma \vdash N <: \texttt{[}\gamma\texttt{/this]}N'$	
5.	**let** $N = N''$	
6.	$\mathcal{E};\Gamma \vdash N <: N$	by SR-REFL on N
7.	$\mathcal{E};\Gamma \vdash N'' <: N$	by **6**, **5**
8.	$\mathcal{E};\Gamma \vdash \texttt{e} : N''$	by **3**, **5**
9.	$\mathcal{E};\Gamma \vdash N'' <: \texttt{[}\gamma\texttt{/this]}N'$	by **4**, **5**
10.	done	by **1**, **8**, **2**, **9**, **7**

Case: T-Subs

1.	$\mathcal{E};\Gamma \vdash \gamma\texttt{.f = e} : N^*$	} by the premise of T-SUBS on **a**
2.	$\mathcal{E};\Gamma \vdash N^* <: N$	
3.	$\mathcal{E};\Gamma \vdash N$ OK	

4.	$\mathcal{E};\Gamma \vdash \gamma : \mathtt{o:C<}\overline{\mathtt{o}}\mathtt{>}$	} by the inductive hyp on **1**
5.	$\mathcal{E};\Gamma \vdash \mathtt{e} : N''$	
6.	$fType(\mathtt{f, o:C<}\overline{\mathtt{o}}\mathtt{>}) = N'$	
7.	$\mathcal{E};\Gamma \vdash N'' <: \mathtt{[}\gamma\mathtt{/this]}N'$	
8.	$\mathcal{E};\Gamma \vdash N'' <: N^*$	
9.	$\mathcal{E};\Gamma \vdash N'' <: N$	by SR-TRANS on **8**, **2**
10.	done	by **4 - 7**, **9**

Lemma 12 : Inversion on the Object Creation Expression

If:

a. $\mathcal{E};\Gamma \vdash \mathtt{new\ o:C<}\overline{\mathtt{o}}\mathtt{>} : N$

then:

$\mathcal{E};\Gamma \vdash \mathtt{o:C<}\overline{\mathtt{o}}\mathtt{>}$ OK

$\mathcal{E};\Gamma \vdash \mathtt{o:C<}\overline{\mathtt{o}}\mathtt{>} <: N$

Proof by induction on the derivation of **a**, with a case analysis on the last step:

Case: T-New

1.	$N = \mathtt{o:C<}\overline{\mathtt{o}}\mathtt{>}$	by the definition of T-NEW on **a**
2.	$\mathcal{E};\Gamma \vdash \mathtt{o:C<}\overline{\mathtt{o}}\mathtt{>}$ OK	by the premise of T-NEW on **a**, **1**
3.	$\mathcal{E};\Gamma \vdash \mathtt{o:C<}\overline{\mathtt{o}}\mathtt{>} <: \mathtt{o:C<}\overline{\mathtt{o}}\mathtt{>}$	by SR-REFL on $\mathtt{o:C<}\overline{\mathtt{o}}\mathtt{>}$
4.	$\mathcal{E};\Gamma \vdash \mathtt{o:C<}\overline{\mathtt{o}}\mathtt{>} <: N$	by **3**, **1**
5.	done	by **2**, **4**

Case: T-Subs

1.	$\mathcal{E};\Gamma \vdash \mathtt{new\ o:C<}\overline{\mathtt{o}}\mathtt{>} : N''$	} by the premise of T-SUBS on **a**
2.	$\mathcal{E};\Gamma \vdash N'' <: N$	
3.	$\mathcal{E};\Gamma \vdash N$ OK	

4.	$\mathcal{E};\Gamma \vdash \texttt{o:C<}\overline{\texttt{o}}\texttt{>}$ OK	} by the inductive hyp on **1**
5.	$\mathcal{E};\Gamma \vdash \texttt{o:C<}\overline{\texttt{o}}\texttt{>} <: N''$	} by the inductive hyp on **1**
6.	$\mathcal{E};\Gamma \vdash \texttt{o:C<}\overline{\texttt{o}}\texttt{>} <: N$	by SR-TRANS on **2, 5**
7.	done	by **4, 6**

Lemma 13 : Inversion on the Method Invocation Expression

If:

a. $\mathcal{E};\Gamma \vdash \gamma\texttt{.m(e)} : N$

then:

$\mathcal{E};\Gamma \vdash \gamma : \texttt{o:C<}\overline{\texttt{o}}\texttt{>}$

$\mathcal{E};\Gamma \vdash \texttt{e} : N''$

$mType(\texttt{m, o:C<}\overline{\texttt{o}}\texttt{>}) = N' \rightarrow\ N^*$

$\mathcal{E};\Gamma \vdash N'' <: \texttt{[}\gamma\texttt{/this]}N'$

$\mathcal{E};\Gamma \vdash \texttt{[}\gamma\texttt{/this]}N^* <: N$

Proof by induction on the derivation of **a**, with a case analysis on the last step:

Case: T-Invk

1.	$N = \texttt{[}\gamma\texttt{/this]}N^*$	by the definition of T-INVK on **a**
2.	$\mathcal{E};\Gamma \vdash \gamma : \texttt{o:C<}\overline{\texttt{o}}\texttt{>}$	} by the premise of T-INVK on **a, 1**
3.	$\mathcal{E};\Gamma \vdash \texttt{e} : N''$	} by the premise of T-INVK on **a, 1**
4.	$mType(\texttt{m, o:C<}\overline{\texttt{o}}\texttt{>}) = N' \rightarrow\ N^*$	} by the premise of T-INVK on **a, 1**
5.	$\mathcal{E};\Gamma \vdash N'' <: \texttt{[}\gamma\texttt{/this]}N'$	} by the premise of T-INVK on **a, 1**
6.	$\mathcal{E};\Gamma \vdash \texttt{[}\gamma\texttt{/this]}N^* <: \texttt{[}\gamma\texttt{/this]}N^*$	by SR-REFL on $\texttt{[}\gamma\texttt{/this]}N^*$
7.	$\mathcal{E};\Gamma \vdash \texttt{[}\gamma\texttt{/this]}N^* <: N$	by **6, 1**
8.	done	by **7, 2 - 5**

Case: T-Subs

1.	$\mathcal{E};\Gamma \vdash \gamma\texttt{.m(e)} : N^{**}$	} by the premise of T-SUBS on **a**
2.	$\mathcal{E};\Gamma \vdash N^{**} <: N$	} by the premise of T-SUBS on **a**
3.	$\mathcal{E};\Gamma \vdash N$ OK	} by the premise of T-SUBS on **a**

4.	$\mathcal{E};\Gamma \vdash \gamma : \texttt{o:C<}\overline{\texttt{o}}\texttt{>}$	} by the inductive hyp on **1**
5.	$\mathcal{E};\Gamma \vdash \texttt{e} : N''$	
6.	$mType(\texttt{m, o:C<}\overline{\texttt{o}}\texttt{>}) = N' \rightarrow N^*$	
7.	$\mathcal{E};\Gamma \vdash N'' <: \texttt{[}\gamma\texttt{/this]}N'$	
8.	$\mathcal{E};\Gamma \vdash \texttt{[}\gamma\texttt{/this]}N^* <: N^{**}$	
9.	$\mathcal{E};\Gamma \vdash \texttt{[}\gamma\texttt{/this]}N^* <: N$	by SR-TRANS on **8**, **2**
10.	done	by **9**, **4 - 7**

Lemma 14 : Inversion on the Sheep Cloning Expression

If:

a. $\mathcal{E};\Gamma \vdash \texttt{sheep(e)} : N$

then:

$\mathcal{E};\Gamma \vdash \texttt{e} : N'$

$\mathcal{E};\Gamma \vdash N' <: N$

Proof by induction on the derivation of **a**, with a case analysis on the last step:

Case: T-Sheep

1.	$\mathcal{E};\Gamma \vdash \texttt{e} : N$	by the premise of T-SHEEP on **a**
2.	**let** $N = N'$	
3.	$\mathcal{E};\Gamma \vdash N <: N$	by SR-REFL on N
4.	$\mathcal{E};\Gamma \vdash N' <: N$	by **3**, **2**
5.	$\mathcal{E};\Gamma \vdash \texttt{e} : N'$	by **1**, **2**
6.	done	by **4**, **5**

Case: T-Subs

1.	$\mathcal{E};\Gamma \vdash \texttt{sheep(e)} : N''$	} by the premise of T-SUBS on **a**
2.	$\mathcal{E};\Gamma \vdash N'' <: N$	
3.	$\mathcal{E};\Gamma \vdash N$ OK	

4.	$\mathcal{E}; \Gamma \vdash \mathtt{e} : N'$	} by the inductive hyp on **1**
5.	$\mathcal{E}; \Gamma \vdash N' <: N''$	
6.	$\mathcal{E}; \Gamma \vdash N' <: N$	by SR-TRANS on **5, 2**
7.	done	by **4, 6**

Lemma 15 : Inversion on Variable Expression

If:

a. $\mathcal{E}; \Gamma \vdash \gamma : N$

then:

$\mathcal{E}; \Gamma \vdash \Gamma(\gamma) <: N$

Proof by induction on the derivation of **a**, with a case analysis on the last step:

Case: T-Var

1.	$N = \Gamma(\gamma)$	by the definition of T-VAR on **a**
2.	$\mathcal{E}; \Gamma \vdash \Gamma(\gamma) <: \Gamma(\gamma)$	by SR-REFL on $\Gamma(\gamma)$
3.	$\mathcal{E}; \Gamma \vdash \Gamma(\gamma) <: N$	by **2, 1**
4.	done	by **3**

Case: T-Subs

1.	$\mathcal{E}; \Gamma \vdash \gamma : N'$	} by the premise of T-SUBS on **a**
2.	$\mathcal{E}; \Gamma \vdash N' <: N$	
3.	$\mathcal{E}; \Gamma \vdash N$ OK	
4.	$\mathcal{E}; \Gamma \vdash \Gamma(\gamma) <: N'$	by the inductive hyp on **1**
5.	$\mathcal{E}; \Gamma \vdash \Gamma(\gamma) <: N$	by SR-TRANS on **4, 2**
6.	done	by **5**

Lemma 16 : Inversion on Null Expression

If:

a. $\mathcal{E}; \Gamma \vdash \texttt{null} : N$

then:

$\mathcal{E}; \Gamma \vdash N' \text{ OK}$

$\mathcal{E}; \Gamma \vdash N' <: N$

Proof by induction on the derivation of **a**, with a case analysis on the last step:

Case: T-Null

1.	$\mathcal{E}; \Gamma \vdash N \text{ OK}$	by the premise of T-NULL on **a**
2.	$\mathcal{E}; \Gamma \vdash N <: N$	by SR-REFL on N
3.	**let** $N' = N$	
4.	$\mathcal{E}; \Gamma \vdash N' \text{ OK}$	by **1**, **3**
5.	$\mathcal{E}; \Gamma \vdash N' <: N$	by **2**, **3**
6.	done	by **4**, **5**

Case: T-Subs

1.	$\mathcal{E}; \Gamma \vdash \texttt{null} : N''$	} by the premise of T-SUBS on **a**
2.	$\mathcal{E}; \Gamma \vdash N'' <: N$	
3.	$\mathcal{E}; \Gamma \vdash N \text{ OK}$	
4.	$\mathcal{E}; \Gamma \vdash N' \text{ OK}$	} by the inductive hyp on **1**
5.	$\mathcal{E}; \Gamma \vdash N' <: N''$	
6.	$\mathcal{E}; \Gamma \vdash N' <: N$	by SR-TRANS on **5**, **2**
7.	done	by **4**, **6**

Progress

Theorem 2 : Progress

If:

a. $\mathcal{H} \vdash \mathtt{e} : N$

b. $\vdash \mathcal{H}$ OK

then:

$\mathtt{e}\,;\ \mathcal{H} \rightsquigarrow \mathtt{e}'\,;\ \mathcal{H}'$

or:

$\exists\, v$: $\mathtt{e} = v$

or:

$\mathtt{e}\,;\ \mathcal{H} \rightsquigarrow \mathtt{err}\,;\ \mathcal{H}$

Proof by induction on the derivation of **a**, with a case analysis on the last step:

Case: T-Var

1.	$\mathtt{e} = \gamma$	} by the definition of T-VAR on **a**
2.	$\mathcal{H}(\gamma) = \{N\,;\ \ldots\}$	
3.	$\gamma \neq \mathtt{this}$	} by syntax convention on **1**, **a**, **b**
4.	$\gamma \neq \mathtt{x}$	

Case analysis on $\gamma = \mathtt{null}$

v.	$\mathtt{e} = \mathtt{null}$	by `case`, **1**
vi.	$\mathtt{e} = v$	by **v**, and `syntax`
vii.	done	by **vi**

Case analysis on $\gamma = \iota$

viii.	$\mathtt{e} = \iota$	by **1**, `case`
ix.	$\mathtt{e} = v$	by **viii**, and `syntax`
x.	done	by **ix**

Case: T-Null

1. $\texttt{e} = \texttt{null}$ by the definition of T-NULL

on **a**

2. $\texttt{null} = v$ by `syntax`

3. $\texttt{e} = v$ by **2**, **1**

4. done by **3**

Case: T-New

1. $\texttt{e} = \texttt{new o:C<}\overline{\texttt{o}}\texttt{>}$ } by the definition of T-NEW on **a**

2. $N = \texttt{o:C<}\overline{\texttt{o}}\texttt{>}$ }

3. $\mathcal{H} \vdash \texttt{o:C<}\overline{\texttt{o}}\texttt{>}$ OK by the premise of T-NEW

on **1**

4. $\texttt{class C<}\overline{\texttt{o}_l \preceq \texttt{x} \preceq \texttt{o}_u}\texttt{>}\ \{\overline{N\ \texttt{f;}}\ \ \overline{M}\}$ by the premise of F-CLASS

on **3**

5. $fields(\texttt{C}) = \overline{\texttt{f}}$ by $fields$ function on `class C`

6. **let** $\mathcal{H}(\iota)\ undefined$

7. **let** $\mathcal{H}' = \mathcal{H}, \iota \rightarrow \{\texttt{o:C<}\overline{\texttt{o}}\texttt{>;}\ \overline{\texttt{f}\rightarrow\texttt{null}}\}$

8. $\texttt{new o:C<}\overline{\texttt{o}}\texttt{>;}\ \mathcal{H} \leadsto \iota;\ \mathcal{H}'$ by R-NEW on **5**, **6**, and **7**

9. **let** $\texttt{e}' = \iota$

10. $\texttt{e};\ \mathcal{H} \leadsto \texttt{e}';\ \mathcal{H}'$ by **8**, **9**, **1**

11. done by **10**

Case: T-Field

1. $\texttt{e} = \gamma\texttt{.f}$ } by the definition of T-FIELD on **a**

2. $N = \texttt{[}\gamma\texttt{/this]}N'$ }

3. $\mathcal{H} \vdash \gamma : N''$ } by the premise of T-FIELD on **1**

4. $fType(\texttt{f}, N'') = N'$ }

5. $\mathcal{H}(\gamma) = \{N''; \ \ldots\}$ by the definition of T-VAR

on **3**

6. $\gamma \neq \texttt{this}$ } by syntax convention on **5**, **a**, **b**

7. $\gamma \neq \texttt{x}$ }

Case analysis on $\gamma = \texttt{null}$

viii. $\texttt{e} = \texttt{null.f}$ by `case`, **1**

ix. $\texttt{null.f};\mathcal{H} \leadsto \texttt{err};\ \mathcal{H}$ by R-FIELD-NULL on **viii**

x. $\texttt{e};\ \mathcal{H} \leadsto \texttt{err};\ \mathcal{H}$ by **ix**, **viii**

xi. done by **x**

Case analysis on $\gamma = \iota$

xii. $\mathcal{H}(\iota) = \{N''; \overline{\mathtt{f} \to v}\}$ by **5**, case, and for some $\overline{v}$ and $\overline{\mathtt{f}}$, where $\mathtt{f} \in \overline{\mathtt{f}}$

xiii. $\iota\,.\,\mathtt{f}; \mathcal{H} \rightsquigarrow v; \mathcal{H}$ by R-FIELD on **xii**

xiv. **let** $\mathcal{H}' = \mathcal{H}$

xv. **let** $\mathtt{e}' = v$ **where** $\mathtt{f} \to v$ as shown in **xii**

xvi. $\mathtt{e}; \mathcal{H} \rightsquigarrow \mathtt{e}'; \mathcal{H}'$ by **xiii**, case, **1**, **xiv**, **xv**

xvii. done by **xvi**

Case: T-Subs

1. $\mathcal{H} \vdash \mathtt{e} : N'$ by the premise of T-SUB

on **a**

2. $\mathtt{e}; \mathcal{H} \rightsquigarrow \mathtt{e}'; \mathcal{H}'$

3. $\mathtt{e} = v$ } by the inductive hyp on **1**, **b**

4. $\mathtt{e}; \mathcal{H} \rightsquigarrow \mathtt{err}; \mathcal{H}$

5. done by **2**, **3**, **4**

Case: T-Assign

1. $\mathtt{e} = (\gamma\,.\,\mathtt{f} = \mathtt{e}'')$ by the definition of T-ASSIGN

on **a**

2. $\mathcal{H} \vdash \gamma : N'$

3. $\mathcal{H} \vdash \mathtt{e}'' : N$

4. $fType(\mathtt{f},\ N') = N''$ } by the premise of T-ASSIGN on **1**

5. $\mathcal{H} \vdash N <: [\gamma/\mathtt{this}]N''$

6. $\mathcal{H}(\gamma) = \{N';\ \overline{\mathtt{f} \to v}\}$ by the definition of T-VAR

on **2**

7. $\gamma \neq \mathtt{this}$

8. $\gamma \neq \mathtt{x}$ } by syntax convention on **6**, **a**, **b**

9. $\gamma = \iota$

Case analysis on $\gamma = \mathtt{null}$

x. $\mathtt{e} = (\mathtt{null.f} = \mathtt{e}'')$ by case, **1**

xi. $\mathtt{null.f = e}''; \mathcal{H} \rightsquigarrow \mathtt{err}; \mathcal{H}$ by R-ASSIGN-NULL on **x**

xii. $\mathtt{e}; \mathcal{H} \rightsquigarrow \mathtt{err}; \mathcal{H}$ by **xi**, **x**

xiii. done — by NF5

Case analysis on $\gamma = \iota$

xiv. $\mathtt{e}''; \mathcal{H} \rightsquigarrow \mathtt{e}'''; \mathcal{H}''$
xv. $\mathtt{e}'' = v'$
xvi. $\mathtt{e}''; \mathcal{H} \rightsquigarrow \mathtt{err}; \mathcal{H}$
} by the inductive hyp on **3**, **b**

Case analysis on $\mathtt{e}''; \mathcal{H} \rightsquigarrow \mathtt{e}'''; \mathcal{H}''$

xvii. $\mathtt{e}''' \neq \mathtt{err}$ — by `case`, **xvi**

xviii. $\iota.\mathtt{f} = \mathtt{e}''; \mathcal{H} \rightsquigarrow \iota.\mathtt{f} = \mathtt{e}'''; \mathcal{H}''$ — by RC-ASSIGN on `case`, **xvii**, **1**, $\mathtt{case}:\gamma = \iota$

xix. **let** $\mathtt{e}' = (\iota.\mathtt{f} = \mathtt{e}''')$

xx. **let** $\mathcal{H}' = \mathcal{H}''$

xxi. $\mathtt{e}; \mathcal{H} \rightsquigarrow \mathtt{e}'; \mathcal{H}'$ — by **xviii**, $\mathtt{case}:\gamma = \iota$, **1**, **xix**, **xx**

xxii. done — by **xxi**

Case analysis on $\mathtt{e}'' = v'$

xxiii. $\mathtt{e} = (\gamma.\mathtt{f} = v')$ — by `case`, **1**

xxiv. $\mathcal{H} \vdash v' : N$ — by `case`, **3**

xxv. **let** $\mathcal{H}' = \mathcal{H}[\iota \mapsto \{N; \overline{\mathtt{f} \to v}[\mathtt{f} \mapsto v']\}]$ — by **xxiv**, **4**, **5**, **6**, $\mathtt{case}:\gamma = \iota$

xxvi. $\iota.\mathtt{f} = v'; \mathcal{H} \rightsquigarrow v'; \mathcal{H}'$ — by R-ASSIGN on **xxv**, **6**, $\mathtt{case}:\gamma = \iota$

xxvii. **let** $\mathtt{e}' = v'$

xxviii. $\mathtt{e}; \mathcal{H} \rightsquigarrow \mathtt{e}'; \mathcal{H}'$ — by **xxvi**, **xxvii**, `case`, **1**, $\mathtt{case}:\gamma = \iota$

xxix. done — by **xxviii**

Case analysis on $\mathtt{e}''; \mathcal{H} \rightsquigarrow \mathtt{err}; \mathcal{H}$

xxx. $\iota.\mathtt{f} = \mathtt{e}''; \mathcal{H} \rightsquigarrow \mathtt{err}; \mathcal{H}$ — by RC-ASSIGN-ERR on `case`, **1**, $\mathtt{case}:\gamma = \iota$

xxxi. $\mathtt{e}; \mathcal{H} \rightsquigarrow \mathtt{err}; \mathcal{H}$ — by **xxx**, $\mathtt{case}:\gamma = \iota$, **1**

xxxii. done — by **xxxi**

Case: T-Invk

1. $\texttt{e} = \gamma\texttt{.m(e}'')$ } by the definition of T-INVK on **a**

2. $N = [\gamma/\texttt{this}]N'$

3. $\mathcal{H} \vdash \gamma : N''$ } by the premise of T-INVK on **a**

4. $\mathcal{H} \vdash \texttt{e}'' : N^3$

5. $mType(\texttt{m}, N'') = N^* \rightarrow N'$

6. $\mathcal{H} \vdash N^3 <: [\gamma/\texttt{this}]N^*$

7. $\mathcal{H}(\gamma) = \{N''; \ \ldots\}$ by the definition of T-VAR on **3**

8. $\gamma \neq \texttt{this}$ } by syntax convention on **7**, **a**, **b**

9. $\gamma \neq \texttt{x}$

Case analysis on $\gamma = \texttt{null}$

x. $\texttt{e} = \texttt{null.m(e}'')$ by `case`, **1**

xi. $\texttt{null.m(e}'')\texttt{;}\ \mathcal{H} \rightsquigarrow \texttt{err;}\ \mathcal{H}$ by R-INVK-NULL on **x**

xii. $\texttt{e;}\ \mathcal{H} \rightsquigarrow \texttt{err;}\ \mathcal{H}$ by **xi**, **x**

xiii. done by **xii**

Case analysis on $\gamma = \iota$

xiv. $\texttt{e}''\texttt{;}\ \mathcal{H} \rightsquigarrow \texttt{e}'''\texttt{;}\ \mathcal{H}''$ } by the inductive hyp on **4**, **b**

xv. $\texttt{e}'' = v'$

xvi. $\texttt{e}''\texttt{;}\ \mathcal{H} \rightsquigarrow \texttt{err;}\ \mathcal{H}$

Case analysis on $\texttt{e}''\texttt{;}\ \mathcal{H} \rightsquigarrow \texttt{e}'''\texttt{;}\ \mathcal{H}''$

xvii. $\texttt{e}'' \neq \texttt{err}$ by `case`, **xvi**

xviii. $\iota\texttt{.m(e}'')\texttt{;}\ \mathcal{H} \rightsquigarrow \iota\texttt{.m(e}''')\texttt{;}\ \mathcal{H}''$ by RC-INVK on **xvii**, `case`, **1**, $\texttt{case}:\gamma = \iota$

xix. **let** $\texttt{e}' = \iota\texttt{.m(e}''')$

xx. **let** $\mathcal{H}' = \mathcal{H}''$

xxi. $\texttt{e;}\ \mathcal{H} \rightsquigarrow \texttt{e}'\texttt{;}\ \mathcal{H}'$ by **xviii**, **xix**, **xx**, **1**, $\texttt{case}:\gamma = \iota$

xxii. done by **xxi**

Case analysis on $\texttt{e}'' = v'$

xxiii. $\texttt{e} = \iota\texttt{.m(}v'\texttt{)}$ by **1**, `case`, $\texttt{case}:\gamma = \iota$

xxiv. $\mathcal{H} \vdash v' : N^3$ by **4**, **xxiii**

xxv. **let** $N'' = \texttt{o}:\texttt{C<}\overline{\texttt{o}}\texttt{>}$

xxvi. **let** $mBody(\texttt{m}, \texttt{o}:\texttt{C<}\overline{\texttt{o}}\texttt{>}) = (\texttt{x;}\ \ \texttt{e}''')$ by **xxv**, **3**, **7**

xxvii. $\mathcal{H}(\iota) = \{\texttt{o:C<}\overline{\texttt{o}}\texttt{>; ...}\}$ by **7**, $\texttt{case:}\gamma = \iota$, **xxv**

xxviii. $\iota\texttt{.m(}v'\texttt{);}\mathcal{H} \rightsquigarrow \texttt{[}v'\texttt{/x,}\iota\texttt{/this,o/owner]e'''};\mathcal{H}$ by R-INVK on **xxvi**, **xxvii**, **xxiii**

xxix. **let** $\texttt{e}' = \texttt{[}v'\texttt{/x,}\iota\texttt{/this,o/owner]e'''}$

xxx. **let** $\mathcal{H}' = \mathcal{H}$

xxxi. $\texttt{e;}\ \mathcal{H} \rightsquigarrow \texttt{e';}\ \mathcal{H}'$ by **xxviii**, **xxiii**, **xxix**, **xxx**

xxxii. done by **xxxi**

Case analysis on $\texttt{e''};\ \mathcal{H} \rightsquigarrow \texttt{err;}\ \mathcal{H}$

xxxiii. $\iota\texttt{.m(e'');}\mathcal{H} \rightsquigarrow \texttt{err;}\ \mathcal{H}$ by RC-INVK-ERR on `case`, **1**, $\texttt{case:}\gamma = \iota$

xxxiv. $\texttt{e;}\ \mathcal{H} \rightsquigarrow \texttt{err;}\ \mathcal{H}$ by **xxxiii**, **1**, $\texttt{case:}\gamma = \iota$

xxxv. done by **xxxiv**

Case: T-Sheep

1. $\texttt{e} = \texttt{sheep(e'')}$ by the definition of T-SHEEP on **a**

2. $\mathcal{H} \vdash \texttt{e''} : N$ by the premise of T-SHEEP on **1**

3. $\texttt{e''};\ \mathcal{H} \rightsquigarrow \texttt{e'''};\ \mathcal{H}''$

4. $\texttt{e''} = v'$

5. $\texttt{e''};\ \mathcal{H} \rightsquigarrow \texttt{err;}\ \mathcal{H}$

} by the inductive hyp on **2**, **b**

Case analysis on $\texttt{e''};\ \mathcal{H} \rightsquigarrow \texttt{e'''};\ \mathcal{H}''$

vi. $\texttt{e'''} \neq \texttt{err}$ by `case`, **5**

vii. $\texttt{sheep(e'');}\ \mathcal{H} \rightsquigarrow \texttt{sheep(e''');}\ \mathcal{H}''$ by RC-ASSIGN on `case`, **vi**, **1**

viii. **let** $\texttt{e}' = \texttt{sheep(e''')}$

ix. **let** $\mathcal{H}' = \mathcal{H}''$

x. $\texttt{e;}\ \mathcal{H} \rightsquigarrow \texttt{e';}\ \mathcal{H}'$ by **vii**, **1**, **viii**, **ix**

xi. done by **x**

Case analysis on $\texttt{e''} = v'$

xii. $\mathcal{H} \vdash v' : N$ by `case`, **2**

Case analysis on $v' = \texttt{null}$

xiii. $\texttt{e} = \texttt{sheep(null)}$ by case, **1**

xiv. $\texttt{sheep(null)};\ \mathcal{H} \rightsquigarrow\ \texttt{null};\ \mathcal{H}$ by R-SHEEP-NULL on **xiii**, **2**

xv. **let** $\texttt{e}' = \texttt{null}$

xvi. **let** $\mathcal{H}' = \mathcal{H}$

xvii. $\texttt{e};\ \mathcal{H} \rightsquigarrow \texttt{e}';\ \mathcal{H}'$ by **xiv**, **xiii**, **xv**, **xvi**

xviii. done by **xvii**

Case analysis on $v' \neq \texttt{null}$

xix. $v' = \iota$ by case

xx. **let** $\overline{\iota' \rightarrow \{N;\ \overline{\texttt{f} \rightarrow v}\}} = \{\iota' \rightarrow \{N;\ \overline{\texttt{f} \rightarrow v}\} \in \mathcal{H} \mid \mathcal{H} \vdash \iota' \preceq \iota\ \}$

xxi. **let** $\mathcal{H}(\overline{\iota''})\ undefined$

xxii. **let** $\mathcal{H}'' = \overline{\iota'' \rightarrow [\overline{\iota''/\iota'}]\{N;\ \overline{\texttt{f} \rightarrow v}\}}$

xxiii. $\texttt{sheep}(\iota); \mathcal{H} \rightsquigarrow [\overline{\iota''/\iota'}]\iota; \mathcal{H}, \mathcal{H}''$ by R-SHEEP on **xx**, **xxi**, **xxii**

xxiv. **let** $\texttt{e}' = [\overline{\iota''/\iota'}]\iota$

xxv. **let** $\mathcal{H}' = \mathcal{H}, \mathcal{H}''$

xxvi. $\texttt{e};\ \mathcal{H} \rightsquigarrow \texttt{e}';\ \mathcal{H}'$ by **xxiii**, **xxiv**, **xxv**, **1**, **xix**, $\texttt{case:}\ \texttt{e}'' = v'$

xxvii. done by **xxvi**

Case analysis on $\texttt{e}'';\ \mathcal{H} \rightsquigarrow \texttt{err};\ \mathcal{H}$

xxviii. $\texttt{sheep(e}'');\mathcal{H} \rightsquigarrow \texttt{err};\ \mathcal{H}$ by RC-SHEEP-ERR on case, **1**

xxix. $\texttt{e};\ \mathcal{H} \rightsquigarrow \texttt{err};\ \mathcal{H}$ by **xxviii**, **1**

xxx. done by **xxix**

Miscellaneous lemmas

Lemma 17 : Address Typing is Subtype the Address's Type in the Heap

If:

a. $\mathcal{H} \vdash \iota : N$

b. $\mathcal{H}(\iota) = \{N'\texttt{;}\ \overline{\texttt{f}\rightarrow v}\}$

c. $\vdash \mathcal{H}$ OK

then:

$\mathcal{H} \vdash N' <: N$

Proof by induction on the derivation of **a**, with a case analysis on the last step:

Case: T-Var

1.	$N = \mathcal{H}(\iota)_{\downarrow 1}$	by the definition of T-VAR on **a**
2.	$\mathcal{H}(\iota)_{\downarrow 1} = N'$	by **b**
3.	$\mathcal{H} \vdash \mathcal{H}(\iota)_{\downarrow 1} \preceq \mathcal{H}(\iota)_{\downarrow 1}$	by SR-REFL on $\mathcal{H}(\iota)_{\downarrow 1}$
4.	$\mathcal{H} \vdash N' \preceq N$	by **3**, **2**, **1**
5.	done	by **4**

Case: T-Subs

1.	$\mathcal{H} \vdash \mathcal{H}(\iota)_{\downarrow 1} \preceq N$	by LEMMA 15 on **a**
2.	$\mathcal{H}(\iota)_{\downarrow 1} = N'$	by **b**
3.	$\mathcal{H} \vdash N' \preceq N$	by **1**, **2**
4.	done	by **3**

Lemma 18 : Well-formed Types have no Free Variables

If:

a. $\mathcal{H} \vdash N$ OK

then:

$fv(N) = \emptyset$

Proof by natural deduction.

Lemma 19 : $fType$ is Invariant of Subtyping

If:

a. $\mathcal{H} \vdash N <: N^3$

b. $fType(\mathtt{f}, N) = N'$

c. $fType(\mathtt{f}, N^3) = N^4$

then:

$N' = N^4$

Proof by induction on the derivation of **a**, with a case analysis on the last step:

Lemma 20 : $fType$ gives Well-formed Type

If:

a. $\vdash \mathcal{H}$ OK

b. $\mathcal{H} \vdash N$ OK

c. $fType(\mathtt{f}, N) = N'$

then:

$\mathcal{H} \vdash N'$ OK

Proof by natural deduction.

Lemma 21 : $mType$ is preserved under Subtyping

If:

a. $\mathcal{H} \vdash N <: N'$

b. $mType(\mathtt{m}, N') = N_1 \to N_2$

then:

$mType(\mathtt{m}, N) = N_1 \to N_2$

Proof by induction on the derivation of **a**, with a case analysis on the last

step:

Lemma 22 : $mType$ preserves Type Well-formedness

If:

a. $\mathcal{H} \vdash N$ OK

b. $mType(\texttt{m}, N) = N_1 \rightarrow N_2$

then:

$\mathcal{H}$, `this`$:N \vdash N_1$ OK

$\mathcal{H}$, `this`$:N \vdash N_2$ OK

Proof by natural deduction.

Lemma 23 : Typing of the Expression in Methods through $mType$ and $mBody$

If:

a. $mType(\texttt{m}, \texttt{o:C<}\overline{\texttt{o}}\texttt{>}) = N' \rightarrow N$

b. $mBody(\texttt{m}, \texttt{o:C<}\overline{\texttt{o}}\texttt{>}) = \texttt{(x; e)}$

c. $\mathcal{H} \vdash \texttt{o:C<}\overline{\texttt{o}}\texttt{>}$ OK

then:

$\mathcal{H}$, `this:o:C<`$\overline{\texttt{o}}$`>, x:`$N' \vdash \texttt{e} : N$

Proof by natural deduction.

Lemma 24 : Expression Typing have Well-formed Type

If:

a. $\mathcal{H} \vdash \texttt{e} : N$

b. $\vdash \mathcal{H}$ OK

then:

$\mathcal{H} \vdash N$ OK

Proof by induction on the derivation of **a**, with a case analysis on the last step:

Lemma 25 : Expression Reduction preserves Expression Typing over New Heap

If:

a. $\mathcal{H} \vdash \mathtt{e} : N$

b. $\vdash \mathcal{H}$ OK

c. $\mathtt{e}'\mathtt{;}\ \mathcal{H} \rightsquigarrow \mathtt{e}''\mathtt{;}\ \mathcal{H}'$

then:

$\mathcal{H}' \vdash \mathtt{e} : N$

Proof by induction on the derivation of **c**, with a case analysis on the last step:

Lemma 26 : Expression Reduction preserves Well-formed Types over New Heap

If:

a. $\mathcal{H} \vdash N$ OK

b. $\vdash \mathcal{H}$ OK

c. $\mathtt{e}\mathtt{;}\ \mathcal{H} \rightsquigarrow \mathtt{e}'\mathtt{;}\ \mathcal{H}'$

then:

$\mathcal{H}' \vdash N$ OK

Proof by induction on the derivation of **c**, with a case analysis on the last step:

Lemma 27 : Expression Reduction preserves Subtyping New Heap

If:

a. $\mathcal{H} \vdash N <: N'$

b. $\vdash \mathcal{H}$ OK

c. $\texttt{e; } \mathcal{H} \rightsquigarrow \texttt{e}'\texttt{; } \mathcal{H}'$

then:

$\mathcal{H}' \vdash N <: N'$

Proof by induction on the derivation of **c**, with a case analysis on the last step:

Lemma 28 : Objects are Inside the Owner of Their Fields

If:

a. $\mathcal{H} \vdash \iota : N$

b. $fType(\texttt{f, } N) = N'$

c. $\mathcal{H} \vdash v : \texttt{[}\iota\texttt{/this]}N'$

then:

$\mathcal{H} \vdash \iota \preceq own_{\mathcal{H}}(v)$

Proof by natural deduction.

Substitution lemmas

Lemma 29 : Address Substitution on Well-formed Types

If:

a. $\mathcal{H} \vdash \iota : N$

b. $\mathcal{H}$, this:$N \vdash N'$ OK

then:

$\mathcal{H} \vdash [\iota/\texttt{this}]N'$ OK

Proofby natural deduction.

Lemma 30 : Address Substitution on Expression Typing

If:

a. $\mathcal{H} \vdash \iota : N$

b. $\mathcal{H}$, this:$N \vdash \texttt{e} : N'$

then:

$\mathcal{H} \vdash [\iota/\texttt{this}]\texttt{e} : [\iota/\texttt{this}]N'$

Proofby natural deduction.

Lemma 31 : Value Substitution on Expression Typing

If:

a. $\mathcal{H} \vdash v : N$

b. $\mathcal{H}$, x:$N \vdash \texttt{e} : N'$

then:

$\mathcal{H} \vdash [v/\texttt{x}]\texttt{e} : [v/\texttt{x}]N'$

Proofby natural deduction.

Lemma 32 : Context Substitution on Inside Relation

If:

a. $\mathcal{H} \vdash \circ \text{ OK}$

b. $\mathcal{H} \vdash \iota \preceq \iota'$

then:

$\mathcal{H} \vdash [\circ/\mathtt{x}](\iota \preceq \iota')$

Proofby natural deduction.

Weakening Lemmas

The weakening lemmas are trivial, and therefore omitted in the digital version.

Lemma 33 : Weakening on Well-formed Context

Lemma 34 : Weakening on Well-formed Heap

Lemma 35 : Weakening on Well-formed Address

Lemma 36 : Weakening on Dynamic Value Typing

Lemma 37 : Weakening on Expression Typing

Appendix B

Proofs for Mojo-jojo

In this appendix, we present our proofs for Mojo-jojo. We describe in detail the proof for subject reduction, however, the details of the lesser lemmas have not been presented. The complete proof of this system only exists in the form of hand-written hard copy.

Preservation

Subject Reduction

Theorem 3 : Subject Reduction

If:

a. $\vdash \mathcal{H}$ OK

b. $\mathcal{H} \vdash \mathtt{e} : T$

c. $\mathtt{e}\mathtt{;}\ \mathcal{H} \rightsquigarrow v\mathtt{;}\ \mathcal{H}'$

then:

$\vdash \mathcal{H}'$ OK

$\mathcal{H}' \vdash v : T$

Proof by induction on the derivation of **c**, with a case analysis on the last step:

Case: R-Assign

1.	$\mathtt{e} = \iota\,.\,\mathtt{f} = e'$	} by the definition of R-ASSIGN on **c**
2.	$\mathcal{H}(\iota) = \{R;\ \overline{\mathtt{f}\to v}\}$	
3.	$\mathtt{e}';\ \mathcal{H} \rightsquigarrow v;\ \mathcal{H}''$	
4.	$\mathcal{H}' = \mathcal{H}''[\iota \mapsto \{N';\ \overline{\mathtt{f}\to v}[\mathtt{f}_i \mapsto v]\}]$	
5.	$\mathcal{H} \vdash \iota : \exists\overline{\mathtt{o}};\overline{\mathcal{C}}.N$	} by LEMMA 44 on **1**, **b**
6.	$fType(\mathtt{f}, N) = T'$	
7.	$\mathcal{H} \vdash e : T''$	
8.	$\mathcal{H}, \overline{\mathtt{o} : \top}, \overline{\mathcal{C}} \vdash T'' <: [\iota/\mathtt{this}]T'$	
9.	$\mathcal{H} \vdash T'' <: T$	
10.	$\mathcal{H}'' \vdash v : T''$	} by the inductive hyp on **a**, **3**, **7**
11.	$\vdash \mathcal{H}''$ OK	
12.	$\mathcal{H}'' \vdash T'' <: T$	by LEMMA 63 on **9**, **3**
13.	$\mathcal{H} \vdash T$ OK	by LEMMA 55 on **b**, **a**
14.	$\mathcal{H} \vdash T''$ OK	by LEMMA 55 on **7**, **a**
15.	$\mathcal{H}'' \vdash v : T$	by T-SUBS on **12**, **14**, **10**
16.	$\mathcal{H}' \vdash v : T$	by T-RUNTIME on **15**, **4**,

and observing no changes to $\mathcal{H}$.

17.	$\forall\, \iota \to \{N, \overline{\mathtt{f}\to v}\} \in \mathcal{H}''$	by the definition of F-HEAP

on **11**

18.	**let** $fType(\mathtt{f}_i, R) = T'''$	
19.	$\mathcal{H}' \vdash v : T''$	by T-RUNTIME on **10**, **4**
20.	$\mathcal{H}', \overline{\mathtt{o} : \top}, \overline{\mathcal{C}} \vdash T'' <: [\iota/\mathtt{this}]T'$	by T-RUNTIME on **10**, **8**,

F-HEAP

21.	$\mathcal{H} \vdash R <: \exists\overline{\mathtt{o}};\overline{\mathcal{C}}.N$	by LEMMA 44 on **5**, **2**
22.	$\mathcal{H}' \vdash R <: \exists\overline{\mathtt{o}};\overline{\mathcal{C}}.N$	by T-RUNTIME on **21**, **4**
23.	$\exists\, \overline{b}$ st. $R = [\overline{b/\mathtt{o}}]N$	} by S-ENV on **22**
24.	$\mathcal{H}' \vdash [\overline{b/\mathtt{o}}]\overline{\mathcal{C}}$	
25.	**let** $T''' = [\overline{b/\mathtt{o}}]T'$	by **23**, **6**, **18**
26.	$\mathcal{H}' \vdash [\overline{b/\mathtt{o}}]T'' <: [\overline{b/\mathtt{o}}][\iota/\mathtt{this}]T'$	by LEMMA 61 on **24**, **20**

27. $\mathcal{H}' \vdash T'' <: \texttt{[}\iota\texttt{/this]}T'''$ by **26**, **25** as $\iota \notin \overline{o}$ by syntax, $\texttt{this} \notin \overline{b}$ by **23**, R ok by $\mathcal{H}'$ ok, and $\overline{o} \in fv(T'')$ by **14**

28. $\mathcal{H}' \vdash T'''$ OK by LEMMA 49 on **14**, **18**

29. $\mathcal{H}' \vdash \texttt{[}\iota\texttt{/this]}T'''$ OK by LEMMA 58 on **28**, **2**, F-VAR

30. $\mathcal{H}' \vdash v' : \texttt{[}\iota\texttt{/this]}T'''$ by **27**, **29**, **19**

31. **let** $v = \iota'$ by assume without loss of generality $v \neq \texttt{null}$

32. $v \in \texttt{dom}(\mathcal{H}')$ by LEMMA 47 on **16**, **31**

33. $\vdash \mathcal{H}'$ OK by **17**, **4**, **18**, **30**, **32**

34. done by **33**

Case: R-Invk

1. $e = \iota\texttt{.m(}e'\texttt{)}$
2. $\mathcal{H}(\iota) = \{r\texttt{:C<}\overline{T}\texttt{>; ...}\}$
3. $e'\texttt{;}\ \mathcal{H} \rightsquigarrow \iota'\texttt{;}\ \mathcal{H}''$
4. $mBody(\texttt{m, }r\texttt{:C<}\overline{T}\texttt{>}) = (\texttt{x; e}'')$
5. $[\iota'\texttt{/x}, \iota\texttt{/this}]\texttt{e}''\texttt{;}\ \mathcal{H}'' \rightsquigarrow v\texttt{;}\ \mathcal{H}'$
} by the definition of R-INVK on **c**

6. $\mathcal{H} \vdash \iota : \exists\overline{o}\texttt{;}\overline{\mathcal{C}}.N$
7. $\mathcal{H} \vdash e' : T''$
8. $mType(\texttt{m, }N) = \overline{\mathcal{C}'}\texttt{.}\,T' \to T_0$
9. $\mathcal{H}, \overline{o : \top}, \overline{\mathcal{C}} \vdash T'' <: \texttt{[}\iota\texttt{/this]}T'$
10. $\mathcal{H}, \overline{o : \top}, \overline{\mathcal{C}} \vdash [\iota\texttt{/this}]\overline{\mathcal{C}'}$
11. $\mathcal{H} \vdash \Downarrow_{\overline{o};\overline{\mathcal{C}}} [\iota\texttt{/this}]T_0 <: T$
} by LEMMA 46 on **1**, **b**

12. $\mathcal{H}'' \vdash \iota' : T''$
13. $\vdash \mathcal{H}''$ OK
} by the inductive hyp on **3**, **7**, **a**

14. $\mathcal{H} \vdash r\texttt{:C<}\overline{T}\texttt{>} <: \exists\overline{o}\texttt{;}\overline{\mathcal{C}}.N$ by LEMMA 47 on **6**, **2**

15. $\mathcal{H} \vdash [\overline{b/o}]\overline{\mathcal{C}}$
16. $r\texttt{:C<}\overline{T}\texttt{>} = [\overline{b/o}]N$
} by S-ENV on **14**

17. $mType(\texttt{m, }r\texttt{:C<}\overline{T}\texttt{>}) = [\overline{b/o}]\overline{\mathcal{C}'}\texttt{.}\,T' \to T_0$ by LEMMA 51 on **8**, **16**

18. $\mathcal{H}, [\overline{b/o}]\overline{\mathcal{C}'}, \texttt{this} : r\texttt{:C<}\overline{T}\texttt{>}, \texttt{x} : [\overline{b/o}]T' \vdash e'' : [\overline{b/o}]T_0$ by LEMMA 52 on **4**, **17**

19. $\mathcal{H} \vdash \iota : r\,\texttt{:C<}\overline{T}\texttt{>}$ by T-VAR on **2**

20. $\mathcal{H} \vdash r\,\texttt{:C<}\overline{T}\texttt{>}$ OK by LEMMA 55 on **19**, **a**

21. $\mathcal{H} \vdash T_0$ OK by LEMMA 50 on **17**, **20**, **a**

22. $\mathcal{H}'', [\iota/\texttt{this}][\overline{b/\texttt{o}}]\overline{\mathcal{C}'}, \texttt{x} : [\iota/\texttt{this}][\overline{b/\texttt{o}}]T' \vdash [\iota/\texttt{this}]e'' : [\iota/\texttt{this}][\overline{b/\texttt{o}}]T_0$ by LEMMA 59 on **18**, **19**, **16**

23. $\mathcal{H}'', \overline{\texttt{o} : \top}, \overline{\mathcal{C}}, \texttt{this} : N \vdash \iota' : T''$ by LEMMA 64 on **12**

24. $\mathcal{H} \vdash \exists\overline{\texttt{o}}\texttt{;}\overline{\mathcal{C}}.N$ OK by LEMMA 55 on **6**

25. $\mathcal{H}'', \overline{\texttt{o} : \top}, \overline{\mathcal{C}} \vdash N$ OK

26. $\mathcal{H}'', \overline{\texttt{o} : \top}, \overline{\mathcal{C}} \vdash \overline{\mathcal{C}}$ OK

} by F-EXIST on **24**

27. $\mathcal{H}'', \overline{\texttt{o} : \top}, \overline{\mathcal{C}}, \overline{\mathcal{C}'}, \texttt{this} : N \vdash T'$ OK by LEMMA 50 on **25**, **8**

28. $\mathcal{H}'', \overline{\texttt{o} : \top}, \mathcal{C}, \overline{\mathcal{C}'} \vdash \iota' : [\iota/\texttt{this}]T'$ by T-SUBS on **23**, **9**, **27**, LEMMA 64 on result, LEMMA 59 on result

29. $\mathcal{H}'', \mathcal{C}, [\overline{b/\texttt{o}}]\overline{\mathcal{C}'} \vdash [\overline{b/\texttt{o}}]\iota' : [\overline{b/\texttt{o}}][\iota/\texttt{this}]T'$ by LEMMA 62 on **28**, **15**

30. $\mathcal{H}'', [\overline{b/\texttt{o}}]\overline{\mathcal{C}'} \vdash \iota' : [\overline{b/\texttt{o}}][\iota/\texttt{this}]T'$ by **29**, $\iota' \notin \overline{\texttt{o}}$, $\texttt{this} \notin \overline{b}$ (by **16**, **19**)

31. $\mathcal{H}'', [\iota/\texttt{this}][\overline{b/\texttt{o}}]\overline{\mathcal{C}'} \vdash [\iota/\texttt{this}, \iota'/\texttt{x}]e'' : [\iota/\texttt{this}, \iota'/\texttt{x}][\overline{b/\texttt{o}}]T_0$ by LEMMA 59 on **30**, **22**

32. $\mathcal{H} \vdash [\overline{b/\texttt{o}}]\overline{\mathcal{C}'}$ by LEMMA 60 on **10**, **15**

33. $\mathcal{H}'' \vdash [\iota/\texttt{this}, \iota'/\texttt{x}]e'' : [\iota/\texttt{this}, \iota'/\texttt{x}][\overline{b/\texttt{o}}]T_0$ by LEMMA 41 on **32**, **31**

34. $\mathcal{H}' \vdash v : [\iota/\texttt{this}, \iota'/\texttt{x}][\overline{b/\texttt{o}}]T_0$ by the inductive hyp on **33**, **5**, **13**

35. $\mathcal{H}'' \vdash [\overline{b/\texttt{o}}][\iota/\texttt{this}]T_0 <: \Downarrow_{\overline{\texttt{o}};\overline{\mathcal{C}}} [\iota/\texttt{this}]T_0$ by LEMMA 54 on **21**, **15**

36. $\mathcal{H} \vdash [\overline{b/\texttt{o}}][\iota/\texttt{this}]T_0 <: T$ by S-TRANS on **35**, **11**

37. $\mathcal{H}' \vdash v : [\overline{b/\texttt{o}}][\iota/\texttt{this}]T_0$ by **34**, **27**, **16**, **2**, **b**

38. $\mathcal{H}' \vdash v : T$ by T-SUBS on **37**, **36**, Well-Formed Types on **b**.

39. done by **38**, $\vdash \mathcal{H}'$ OK on **34**, **31**

Case: R-Field

1.	$\mathtt{e} = \iota.\mathtt{f}_i$	} by the definition of R-FIELD on **c**
2.	$v = v_i$	
3.	$\mathcal{H}(\iota) = \{R;\ \overline{\mathtt{f} \rightarrow v}\}$	
4.	$\mathcal{H}' = \mathcal{H}$	
5.	$\mathcal{H} \vdash \iota : \exists \overline{o}; \overline{\mathcal{C}}.N$	} by LEMMA 43 on **1**, **b**
6.	$fType(\mathtt{f}_i, N) = T'$	
7.	$\mathcal{H}' \vdash \Downarrow_{\overline{o};\overline{\mathcal{C}}} [\iota/\mathtt{this}]T' <: T$	
8.	$fType(\mathtt{f}_i, N') = T''$	} by the premise of F-HEAP on **3**, **4**, **a**
9.	$\mathcal{H} \vdash N'$ OK	
10.	$\mathcal{H} \vdash v_i : [\iota/\mathtt{this}]T''$	
11.	$\mathcal{H} \vdash \exists \emptyset; \emptyset.N' <: \exists \overline{o}; \overline{\mathcal{C}}.N$	by LEMMA 47 on **5**
12.	$\exists\, \overline{\mathtt{a}}\ \mathtt{st}: \mathcal{H} \vdash [\overline{\mathtt{a}/\mathtt{o}}]\overline{\mathcal{C}}$	by LEMMA 48 on **11**, **9**
13.	$N' = [\overline{\mathtt{a}/\mathtt{o}}]N$	by **11**, **12**, **9**
14.	$T'' = [\overline{\mathtt{a}/\mathtt{o}}]T'$	by LEMMA 53 on **8**, **6**, **13**, **12**
15.	$\mathcal{H}' \vdash v_i : [\iota/\mathtt{this}]([\overline{\mathtt{a}/\mathtt{o}}]T')$	by **10**, **14**, **4**
16.	$\mathtt{this} \notin \overline{\mathtt{a}}$	by syntax convention on this
17.	$\mathcal{H}' \vdash v_i : [\overline{\mathtt{a}/\mathtt{o}}, \iota/\mathtt{this}]T'$	by substitution convention on **15**, **16**
18.	$\iota \notin \overline{o}$	by deep ownership
19.	$\mathcal{H}' \vdash v_i : [\overline{\mathtt{a}/\mathtt{o}}]([\iota/\mathtt{this}]T')$	by substitution convention on **17**, **18**
20.	$\mathcal{H}' \vdash v_i : \Downarrow_{\overline{o};\overline{\mathcal{C}}} [\iota/\mathtt{this}]T'$	by **19**, **12**
21.	$\mathcal{H} \vdash T$ OK	by LEMMA 55 on **b**
22.	$\mathcal{H}' \vdash T$ OK	by **21**, **4**
23.	$\mathcal{H}' \vdash v : T'$	by T-SUBS on **20**, **7**, **22**
24.	done	by **23**, **4**, **a**

Case: R-New

1.	$\texttt{e} = \texttt{new}\ r\texttt{:C<}\overline{T}\texttt{>}$	by the definition of R-NEW on **c**
2.	$v = \iota$	
3.	$\mathcal{H} \vdash r\texttt{:C<}\overline{T}\texttt{>}$ OK	by LEMMA 45 on **1**, **b**
4.	$\mathcal{H} \vdash r\texttt{:C<}\overline{T}\texttt{>} <: T$	
5.	$\mathcal{H}(\iota)\ undefined$	by the premise of R-NEW on **1**, **b**
6.	$fields(\texttt{C}) = \overline{\texttt{f}}$	
7.	$decl(r\texttt{:C<}\overline{T}\texttt{>}) = \overline{\mathcal{C}}$	
8.	$\mathcal{H}' = \mathcal{H}, \iota \rightarrow \{r\texttt{:C<}\overline{T}\texttt{>; }\overline{\texttt{f}\rightarrow\texttt{null}}\}$	
9.	$\mathcal{H} \vdash r$ OK	by the definition of F-CLASS on **3**
10.	$\mathcal{H} \vdash \overline{T}$ OK	
11.	$\mathcal{H} \vdash [r/\texttt{owner}, \overline{T/\texttt{Y}}]\overline{\mathcal{C}'}$	
12.	$\texttt{class C<}\overline{\texttt{Y}}\ \overline{\mathcal{C}}\texttt{> \{ ... \}}$	
13.	$\mathcal{H}'(\iota) = r\texttt{:C<}\overline{T}\texttt{>}$	by the definition of $\mathcal{H}$ on **8**
14.	$\mathcal{H}' \vdash \iota : r\texttt{:C<}\overline{T}\texttt{>}$	by T-VAR on **13**
15.	$\mathcal{H}' \vdash r\texttt{:C<}\overline{T}\texttt{>}$ OK	by LEMMA 55 on **14**
16.	$\mathcal{H} \vdash T$ OK	by LEMMA 55 on **b**
17.	$\mathcal{H}' \vdash T$ OK	by LEMMA 57 on **16**
18.	$\mathcal{H}' \vdash \iota : T$	by T-SUBS on **14**, **4**, **17**
19.	$\mathcal{H}' \vdash v : T$	by **18**, **2**
20.	$\vdash \mathcal{H}'$ OK	by F-HEAP on **8**, **14**, **15**, **a**
21.	done	by **19**, **20**

Case: R-Field-Null/R-Assign-Null/R-Invk-Null

Trivial as $\texttt{e} = \texttt{err}$ by definition of these cases.

Case: RC-Invk/RC-Assign

Trivial by inductive hypothesis over the premise of these cases.

Strengthening Lemmas

Lemma 38 : Strengthening: Well-Formed Types

If:

a. $\mathcal{H} \vdash \overline{\mathcal{C}}$

b. $\overline{\mathcal{C}}, \mathcal{H} \vdash T$ OK

then:

$\mathcal{H} \vdash T$ OK

Proof by induction on the derivation of **b**, with a case analysis on the last step.

Lemma 39 : Strengthening: Well-Formed Constraints

If:

a. $\mathcal{H} \vdash \overline{\mathcal{C}}$

b. $\overline{\mathcal{C}}, \mathcal{H} \vdash \mathcal{C}$ OK

then:

$\mathcal{H} \vdash \mathcal{C}$ OK

Proof by induction on the derivation of **b**, with a case analysis on the last step.

Lemma 40 : Strengthening: Well-Formed Boxes

If:

a. $\mathcal{H} \vdash \overline{\mathcal{C}}$

b. $\overline{\mathcal{C}}, \mathcal{H} \vdash b$ OK

then:

$\mathcal{H} \vdash b$ OK

Proof by induction on the derivation of **b**, with a case analysis on the last step.

Lemma 41 : Strengthening: Type Checking

If:

a. $\mathcal{H} \vdash \overline{\mathcal{C}}$

b. $\overline{\mathcal{C}}, \mathcal{H} \vdash e : T$

then:

$\mathcal{H} \vdash e : T$

Proof by induction on the derivation of **b**, with a case analysis on the last step.

Lemma 42 : Strengthening: Subtyping

If:

a. $\mathcal{H} \vdash \overline{\mathcal{C}}$

b. $\overline{\mathcal{C}}, \mathcal{H} \vdash T <: T'$

then:

$\mathcal{H} \vdash T <: T'$

Proof by induction on the derivation of **b**, with a case analysis on the last step.

Inversion Lemmas

Lemma 43 : Inversion for Field Lookup

If:

a. $\Delta; \Gamma; \overline{\mathtt{X}} \vdash \gamma\mathtt{.f} : T$

then:

$\Delta; \Gamma; \overline{\mathtt{X}} \vdash \gamma : \exists\overline{\mathtt{o}}\mathtt{;}\overline{\mathcal{C}}.N$

$fType(\mathtt{f,}\ N) = T'$

$\Delta; \Gamma; \overline{\mathtt{X}} \vdash \Downarrow_{\overline{\mathtt{o}};\overline{\mathcal{C}}} T' <: T$

Proof by induction on the derivation of **a**, with a case analysis on the last step.

Lemma 44 : Inversion for Field Assignment

If:

a. $\Delta; \Gamma; \overline{\mathtt{X}} \vdash \gamma\mathtt{.f = e} : T$

then:

$\Delta; \Gamma; \overline{\mathtt{X}} \vdash \gamma : \exists\overline{\mathtt{o}}\mathtt{;}\overline{\mathcal{C}}.N$

$\Delta; \Gamma; \overline{\mathtt{X}} \vdash \mathtt{e} : T''$

$fType(\mathtt{f,}\ N) = T'$

$\Delta; \Gamma, \overline{\mathtt{o} : \top}; \overline{\mathtt{X}} \vdash T'' <: \mathtt{[}\gamma\mathtt{/this]}T'$

$\Delta; \Gamma; \overline{\mathtt{X}} \vdash T'' <: T$

Proof by induction on the derivation of **a**, with a case analysis on the last step.

Lemma 45 : Inversion for Object Creation

If:

a. $\Delta; \Gamma; \overline{\mathtt{X}} \vdash \mathtt{new}\, N : T$

then:

$\Delta; \Gamma; \overline{\mathtt{X}} \vdash N\ \textsc{ok}$

$\Delta; \Gamma; \overline{\mathtt{X}} \vdash N <: T$

Proof by induction on the derivation of **a**, with a case analysis on the last step:

Lemma 46 : Inversion for Method Invocation

If:

a. $\Delta; \Gamma; \overline{\mathrm{X}} \vdash \gamma\texttt{.m(}e\texttt{)} : T$

then:

$\Delta; \Gamma; \overline{\mathrm{X}} \vdash \gamma : \exists \overline{\mathrm{o}}\texttt{;}\overline{\mathcal{C}}.N$

$\Delta; \Gamma; \overline{\mathrm{X}} \vdash \mathtt{e} : T'$

$mType(\mathtt{m}, N) = \overline{T'} \to T'$

$\Delta; \Gamma, \overline{\mathrm{o} : \top}; \overline{\mathrm{X}} \vdash T' <: [\gamma/\mathtt{this}]T''$

$\Delta; \Gamma; \overline{\mathrm{X}} \vdash \Downarrow_{\overline{\mathrm{o}};\overline{\mathcal{C}}} T'' <: T$

Proof by induction on the derivation of **a**, with a case analysis on the last step:

Lemma 47 : Inversion for Variable

If:

a. $\Delta; \Gamma; \overline{\mathrm{X}} \vdash \gamma : T$

then:

$\Delta; \Gamma; \overline{\mathrm{X}} \vdash \Gamma(\gamma) <: T$

Proof by induction on the derivation of **a**, with a case analysis on the last step:

Lemma 48 : Inversion for Subtyping

If:

a. $\Delta; \Gamma; \overline{\mathrm{X}} \vdash \exists \overline{\mathrm{o}}\texttt{;}\overline{\mathcal{C}}.N <: \exists \overline{\mathrm{o}'}\texttt{;}\overline{\mathcal{C}'}.N'$

b. $\Delta; \Gamma; \overline{\mathtt{X}} \vdash N$ OK

then: $\exists\, \overline{\mathtt{a}}$ such that :

$\Delta, \overline{\mathcal{C}}; \Gamma, \overline{\mathtt{o} : \top}; \overline{\mathtt{X}} \vdash \overline{[\overline{\mathtt{a}/\mathtt{o}'}]\mathcal{C}'}$

$N = [\overline{\mathtt{a}/\mathtt{o}'}]N'$

Proof by induction on the derivation of **a**, with a case analysis on the last step:

Auxiliary Function Lemmas

Lemma 49 : *fType* gives Well-Formed Types

If:

a. $fType(\mathtt{f},\ N) = T$

b. $\mathcal{H} \vdash N$ OK

c. $\vdash \mathcal{H}$ OK

then:

$\mathcal{H}; \overline{\mathcal{C}}; \mathtt{this} : N \vdash T$ OK

Proof by natural deduction on **a**, the definition of *fType*, and the assumption Ncontains only valid fields.

Lemma 50 : *mType* gives Well-Formed Types

If:

a. $\Delta; \Gamma; \overline{\mathtt{X}} \vdash r : \mathtt{C}\langle\overline{T}\rangle$ OK

b. $mType(\mathtt{m},\ r : \mathtt{C}\langle\overline{T}\rangle) = \overline{\mathcal{C}}.T' \rightarrow T$

then:

$\Delta, \overline{\mathcal{C}}; \Gamma, \mathtt{this} : r : \mathtt{C}\langle\overline{T}\rangle; \overline{\mathtt{X}} \vdash T$ OK

$\Delta, \overline{\mathcal{C}}; \Gamma, \mathtt{this} : r : \mathtt{C}\langle\overline{T}\rangle; \overline{\mathtt{X}} \vdash T'$ OK

Proof by natural deduction on **b**, and the definition of *mType*.

Lemma 51 : Substitution preserves *mType*

If:

a. $mType(\texttt{m}, r:\texttt{C}\overline{<T>}) = \overline{\mathcal{C}}.T' \rightarrow T$

b. $mType(\texttt{m}, [\overline{b'/\circ}](r:\texttt{C}\overline{<T>})) = \overline{\mathcal{C}_1}.T'_1 \rightarrow T_1$

then:

$\overline{\mathcal{C}_1}.T'_1 \rightarrow T_1 = [\overline{b'/\circ}](\overline{\mathcal{C}}.T' \rightarrow T)$

Proof by natural deduction.

Lemma 52 : *mType* and *mBody*

If:

a. $mType(\texttt{m}, r:\texttt{C}\overline{<T>}) = \overline{\mathcal{C}}.T' \rightarrow T$

b. $mBody(\texttt{m}, r:\texttt{C}\overline{<T>}) = \overline{\texttt{x}};e$

c. $\Delta; \Gamma; \overline{\texttt{X}} \vdash r:\texttt{C}\overline{<T>}$ OK

then:

$\Delta, \overline{\mathcal{C}}; \Gamma, \texttt{this} : r:\texttt{C}\overline{<T>}; \overline{\texttt{X}}, \overline{\texttt{x}:T'} \vdash e : T$

Proof by natural deduction.

Lemma 53 : Substitution preserves *fType*

If:

a. $mType(\texttt{f}, N) = T$

b. $mType(\texttt{f}, [\overline{b'/\circ}]N) = T'$

then:

$T' = [\overline{b'/\circ}]T$

Proof by natural deduction.

Close Operator Lemmas

Lemma 54 : Close Operator preserves Well-formedness

If:

a. $\mathcal{H}, \overline{\circ : \top}, \overline{\mathcal{C}} \vdash T$ OK

b. $\mathcal{H}, \overline{\circ : \top} \vdash \overline{\mathcal{C}}$ OK

then:

$\mathcal{H} \vdash \Downarrow_{\overline{\circ};\overline{\mathcal{C}}} T$ OK

Proof by induction on the derivation of **a**, with a case analysis on the last step.

Miscellaneous lemmas

Lemma 55 : Expression Typing has Well-formed Types

If:

a. $\mathcal{H} \vdash e : T$

b. $\vdash \mathcal{H}$ OK

then:

$\mathcal{H} \vdash T$ OK

Proof by induction on the derivation of **a**, with a case analysis on the last step.

Lemma 56 : Subtyping gives Equality

If:

a. $\Delta; \Gamma; \overline{\mathrm{x}} \vdash \exists \emptyset ; \emptyset . N <: \exists \emptyset ; \emptyset . N'$

then:

$N = N'$

Proof by induction on the derivation of **a**, with a case analysis on the last step.

Lemma 57 : Heap Reduction preserves Well-formed Types

If:

a. $\mathcal{H} \vdash T \ \mathrm{OK}$

b. $e ; \mathcal{H} \rightsquigarrow v ; \mathcal{H}'$

then:

$\mathcal{H}' \vdash T \ \mathrm{OK}$

Proof by induction on the derivation of **b**, with a case analysis on the last step.

Substitution and Weakening Lemmas

The substitution and weakening lemmas are trivial, and therefore omitted in the digital version.

Lemma 58 : Address Substitution for Well-Formed Types

Lemma 59 : Address Substitution for Expression Typings

Lemma 60 : Context Substitution for Constraints

Lemma 61 : Context Substitution for Subtyping

Lemma 62 : Context Substitution for Typing

Lemma 63 : Weakening for Subtyping

Lemma 64 : Weakening for Address Typing

Bibliography

[1] ECMAScript Lanaguage Specification, Standard ECMA-262, 5.1 Edition. Research report, ECMA International, 2011.

[2] ALDRICH, J., AND CHAMBERS, C. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP* (2004).

[3] ALMEIDA, P. S. Balloon Types: Controlling Sharing of State in Data Types. In *ECOOP* (1997).

[4] BLACK, A. P., BRUCE, K. B., HOMER, M., AND NOBLE, J. Grace: The absence of (inessential) difficulty. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (New York, NY, USA, 2012), Onward! 2012, ACM, pp. 85–98.

[5] BOYAPATI, LISKOV, AND SHRIRA. Ownership Types for Object Encapsulation. In *Principles of Programming Languages (POPL)* (2003).

[6] BOYAPATI, C., LEE, R., AND RINARD, M. C. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA* (2002).

[7] BOYAPATI, C., AND RINARD, M. A Parameterized Type System for Race-free Java Programs. In *OOPSLA* (2001).

[8] BOYAPATI, C., SALCIANU, A., BEEBEE, W., JR., AND RINARD, M. Ownership types for safe region-based memory management in real-

time java. In *Programming Language Design and Implementation (PLDI* (2003).

[9] BOYLAND, J. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience 31*, 6 (May 2001), 533–553.

[10] BRACHA, G. Generics in the Java Programming Language, 2004. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[11] BRACHA, G., ODERSKY, M., STOUTAMIRE, D., AND WADLER, P. Making the future safe for the past: adding genericity to the java programming language. *SIGPLAN Not. 33* (October 1998), 183–200.

[12] BREG, F., AND POLYCHRONOPOULOS, C. D. Java virtual machine support for object serialization. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande* (New York, NY, USA, 2001), JGI '01, ACM, pp. 173–180.

[13] CAMERON, N. *Existential Types for Variance - Java Wildcards and Ownership Types*. PhD thesis, Department of Computing, Imperial College London, 2008.

[14] CAMERON, N., AND DROSSOPOULOU, S. Existential Quantification for Variant Ownership. In *European Symposium on Programming Languages and Systems (ESOP)* (2009).

[15] CAMERON, N., DROSSOPOULOU, S., AND ERNST, E. A Model for Java with Wildcards. In *ECOOP* (2008).

[16] CAMERON, N., DROSSOPOULOU, S., NOBLE, J., AND SMITH, M. Multiple Ownership. In *OOPSLA* (2007).

[17] CAMERON, N., ERNST, E., AND DROSSOPOULOU, S. Towards an Existential Types Model for Java Wildcards. In *Formal Techniques for Java-like Programs (FTfJP)* (2007).

[18] CARDELLI, L., AND WEGNER, P. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys 17*, 4 (1985), 471–522.

[19] CAYLEY, A. *The collected mathematical papers of Arthur Cayley*, vol. 7. The University Press, 1894.

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

[21] CLARKE, D., DROSSOPOULOU, S., AND NOBLE, J. Roles of Owners. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)* (2011).

[22] CLARKE, D., AND WRIGSTAD, T. External Uniqueness. In *Foundations of Object-Oriented Languages (FOOL)* (2003).

[23] CLARKE, D., AND WRIGSTAD, T. External Uniqueness is Unique Enough. In *ECOOP* (2003).

[24] CLARKE, D., WRIGSTAD, T., ÖSTLUND, J., AND JOHNSEN, E. Minimal ownership for active objects. In *Programming Languages and Systems*. 2008.

[25] CLARKE, D. G., AND DROSSOPOULOU, S. Ownership, Encapsulation and the Disjointness of Type and Effect. In *OOPSLA* (2002).

[26] CLARKE, D. G., POTTER, J. M., AND NOBLE, J. Ownership Types for Flexible Alias Protection. In *OOPSLA* (1998).

[27] CUNNINGHAM, D., DIETL, W., DROSSOPOULOU, S., FRANCALANZA, A., MÜLLER, P., AND SUMMERS, A. Universe Types for Topology and Encapsulation. In *Formal Methods for Components and Objects (FMCO)* (2008).

[28] DE JONG, S., KAKIVAYA, G., AND ROXE, J. Architecture and method for serialization and deserialization of objects, Aug. 9 2005. US Patent 6,928,488.

[29] DIETL, W., DROSSOPOULOU, S., AND MÜLLER, P. Generic Universe Types. In *European Conference on Object Oriented Programming (ECOOP)* (2007).

[30] DIETL, W., DROSSOPOULOU, S., AND M"ULLER, P. Separating ownership topology and encapsulation with Generic Universe Types. *ACM Transactions on Programming Languages and Systems* (2011).

[31] DIETL, W., AND MÜLLER, P. Exceptions in Ownership Type Systems. In *Formal Techniques for Java-like Programs (FTfJP)* (2004).

[32] DIETL, W., AND MLLER, P. Universes: Lightweight ownership for jml. *JOURNAL OF OBJECT TECHNOLOGY 4* (2005), 5–32.

[33] DROSSOPOULOU, S., AND NOBLE, J. Trust the clones . In *Formal Verification of Object-Oriented Software (FoVEOOS)* (2011).

[34] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns*. Addison-Wesley, 1995.

[35] GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[36] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *The Java Language Specification Third Edition*. Addison-Wesley, Boston, Mass., 2005.

[37] GRAYDON HOARE, R. P. D. The Rust reference manual. Tech. rep., Mozilla Corporation, 2014. From `rust-lang.org`.

[38] GREENHOUSE, A., AND BOYLAND, J. An Object-Oriented Effects System. In *ECOOP* (1999).

[39] GROGONO, P., AND CHALIN, P. Copying, sharing, and aliasing. In *In Proceedings of the Colloquium on Object Orientation in Databases and Software Engineering (COODBSE'94)* (1994).

[40] GROGONO, P., AND SAKKINEN, M. Copying and comparing: Problems and solutions. In *ECOOP*. 2000.

[41] HARMS, D. E., AND WEIDE, B. W. Copying and swapping: Influences on the design of reusable software components. In *IEEE Transaction on Software Engineering* (1991).

[42] HOGG, J. Islands: Aliasing Protection in Object-Oriented Languages. In *OOPSLA* (1991).

[43] HOGG, J., LEA, D., WILLS, A., DECHAMPEAUX, D., AND HOLT, R. The geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess. 3* (April 1992), 11–16.

[44] HUTH, M., AND RYAN, M. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, New York, NY, USA, 2004.

[45] IGARASHI, A., PIERCE, B. C., AND WADLER, P. A Recipe for Raw Types. In *Foundations of Object-Oriented Languages (FOOL)* (2001).

[46] IGARASHI, A., PIERCE, B. C., AND WADLER, P. Featherweight Java: a Minimal Core Calculus For Java and GJ. *ACM Trans. Program. Lang. Syst.* (2001).

[47] JENSEN, T., KIRCHNER, F., AND PICHARDIE, D. Secure the clones: Static enforcement of policies for secure object copying. In *European Symposium on Programming (ESOP)* (2011).

[48] JONES, T., HOMER, M., AND NOBLE, J. Brand objects for nominal typing. In *European Conference on Object-Oriented Programming (ECOOP)* (2015).

[49] KRISHNASWAMI, N., AND ALDRICH, J. Permission-Based Ownership: Encapsulating State in Higher-Order Typed Languages. In *Programming Language Design and Implementation (PLDI)* (2005).

[50] LANDI, W., AND RYDER, B. G. Pointer-induced aliasing: a problem taxonomy. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1991), POPL '91, ACM, pp. 93–103.

[51] LI, P., CAMERON, N., AND NOBLE, J. Mojojojo - More Ownership for Multiple Owners . In *Foundations of Object-Oriented Languages (FOOL)* (2010).

[52] LI, P., CAMERON, N., AND NOBLE, J. Sheep cloning with ownership types. In *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages* (2012), Citeseer, p. 59.

[53] LI, P., CAMERON, N., AND NOBLE, J. Dynamic semantics of sheep cloning: Proving cloning. In *IWACO 2014: 6th International Workshop on Aliasing, Capabilities and Ownership* (2014).

[54] LU, Y., AND POTTER, J. A type system for reachability and acyclicity. In *ECOOP* (2005).

[55] LU, Y., AND POTTER, J. On Ownership and Accessibility. In *ECOOP* (2006).

[56] LU, Y., POTTER, J., AND XUE, J. Validity Invariants and Effects. In *European Conference on Object Oriented Programming (ECOOP)* (2007).

[57] LUCASSEN, J. M., AND GIFFORD, D. K. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1988), POPL '88, ACM, pp. 47–57.

[58] MADS TORGERSEN AND ERIK ERNST AND CHRISTIAN PLESNER HANSEN. Wild FJ. In *Foundations of Object-Oriented Languages (FOOL)* (2005).

[59] MCKAY, B. D., AND PIPERNO, A. Practical graph isomorphism, II. *Journal of Symbolic Computation 60* (2014), 94 – 112.

[60] MEYER, B. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[61] MITCHELL, J. C., AND PLOTKIN, G. D. Abstract Types have Existential Type. *Transactions on Programming Languages and Systems 10*, 3 (1988), 470–502.

[62] MITCHELL, N. The Runtime Structure of Object Ownership. In *ECOOP* (2006).

[63] MITCHELL, N., SCHONBERG, E., AND SEVITSKY, G. Making sense of large heaps. In *European Conference on Object-Oriented Programming (ECOOP)* (2009).

[64] MÜLLER, P., AND POETZSCH-HEFFTER, A. Universes: A Type System for Controlling Representation Exposure. In *Programming Languages and Fundamentals of Programming* (1999).

[65] MÜLLER, P., AND POETZSCH-HEFFTER, A. Universes: A Type System for Alias and Dependency Control. Tech. Rep. 279, Fernuniversität Hagen, 2001.

[66] MÜLLER, P., AND RUDICH, A. Ownership transfer in universe types. In *OOPSLA* (2007).

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

[68] NOBLE, J., CLARKE, D., AND POTTER, J. Object ownership for dynamic alias protection. In *Proceedings of the 32nd International Conference on Technology of Object-Oriented Languages (TOOLS)* (1999).

[69] NOBLE, J., AND POTANIN, A. On Owners-as-Accessors. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)* (2014).

[70] NOBLE, J., VITEK, J., AND POTTER, J. Flexible Alias Protection. In *European Conference on Object Oriented Programming (ECOOP)* (1998).

[71] O'HEARN, P. W., REYNOLDS, J. C., AND YANG, H. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic* (London, UK, UK, 2001), CSL '01, Springer-Verlag, pp. 1–19.

[72] OSTLUND, J., AND WRIGSTAD, T. Owners as Ombudsmen. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)* (2011).

[73] PIERCE, B. C. *Types and Programming Languages*. MIT Press, 2002.

[74] PIPERNO, A. Search space contraction in canonical labeling of graphs (preliminary version). *CoRR abs/0804.4881* (2008).

[75] POTANIN, A., NOBLE, J., CLARKE, D., AND BIDDLE, R. Generic Ownership for Generic Java. In *OOPSLA* (2006).

[76] POTANIN, A., NOBLE, J., FREAN, M., AND BIDDLE, R. Scale-free Geometry in Object-Oriented Programs. *Communications of the ACM* (2005).

[77] POTTER, J., NOBLE, J., AND CLARKE, D. The Ins and Outs of Objects. In *Australian Software Engineering Conference (ASWEC)* (1998).

[78] REYNOLDS, J. C. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science* (Washington, DC, USA, 2002), LICS '02, IEEE Computer Society, pp. 55–74.

[79] RIGGS, R., WALDO, J., WOLLRATH, A., AND INC, S. M. Pickling state in the java system. In *In Proceedings of the USENIX 1996 conference on Object-Oriented Technologies* (1996), USENIX Association.

[80] SMITH, M. *A Model of Effects with an Application to Ownership Types*. PhD thesis, Department of Computing, Imperial College London, 2007.

[81] STROUSTRUP, B. *The C++ Programming Language. Special Edition*. Addison-Wesley, 2000.

[82] SUNDARAM, G., AND SKIENA, S. S. Recognizing small subgraphs. *Networks 25* (1995), 183–191.

[83] UNGAR, D., AND SMITH, R. B. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications* (1987), OOPSLA '87.

[84] WEST, D. B. *Introduction to Graph Theory*, 2 ed. Prentice Hall, September 2000.

[85] WHITNEY, H. Congruent graphs and the connectivity of graphs. *American Journal of Mathematics 54* (1932), pp. 150–168.

[86] WINSKEL, G. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.