A Type System With Containers

by

Michael Thode

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2021

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In this thesis, we will introduce the concept of containers as they apply to programming languages. Encapsulation is a common topic in programming languages with well understood benefits. Here, we will investigate its converse, namely containment. This includes a demonstration of how containers can be integrated into a programming language and what benefits they can bring.

To support containment, a dependent type system is developed to enforce container rules. We add the notion of a container label to our types to indicate the container of the referred object. Around this type system we develop a language enhanced with container syntax. We use this language to show how containers can enable pass-by-value semantics, copying of complex objects and object serialization. An interpreter is implemented for this language to demonstrate its capabilities. Included is a container inferencing algorithm intended to minimize the extra syntax needed for container specification.

A second formal system is also defined. This includes type rules, operational semantics and a proof of soundness. We show that correctly-typed programs will obey all container restrictions at run-time. We fully type the configuration used by the semantics; this includes concrete containers as run-time constructs which allow us to verify correct containment. Mappings are maintained from the container labels of the language to physical run-time containers. We show that as container labels are translated across scopes (e.g. a function call), the physical containers remain consistent.

We conclude with a discussion on ways this system can be enhanced in the future to make containers easier to use, as well as describe additional capabilities such as version control of objects.

Acknowledgements

A big thanks to my supervisor Ondřej Lhoták who challenged me throughout the process of this thesis. It's a mystery to me how Ondřej can be so consistently critical and yet equally positive and encouraging. The constructive criticism is the foundation of what I've learned and the encouragement kept me moving forward.

To Gregor Richards and Yizhou Zhang, thank you for agreeing to review this thesis and the work you invested into making it better.

Thank you to Werner Dietl, for identifying key related works in the area of ownership types.

To all members of the PLG group, the informal daily conversations prior to the Covid-19 lock-down were often amusing, but just as frequently full of random insights into complex topics. I miss those days and hope all of you are doing well.

To my family who put up with me through this process. Lockdown has affected everyone, being stuck at home, being cut off from friends, it all takes a heavy toll. This is a time when we need extra support from each other. Unfortunately a thesis also demands time and balance is difficult to achieve. We have fortunately persevered and its time for me to pay back some debts.

Table of Contents

List of Figures

Chapter 1

Introduction

A container's essential property is the ability to prevent its contents from escaping. In the context of object oriented-programming languages, a container would prevent objects within the container from referencing an object outside the container. Within a container, there could exist an arbitrarily complex graph of objects, but there is no escape. Objects themselves can be containers, and we'll call an object which is a container a self-contained object.

With a container mechanism, there is an opportunity to simplify reasoning about complex programs. In addition to a container mechanism, if there were also restrictions on IO, then when calling a method of a self-contained object with read-only parameters, there would be absolute certainty that all side effects will be limited to inside that object. In complex applications, the relationships and interactions amongst objects may not be readily apparent, and by using containers, operations to an object can be typed such that there is no possibility of unexpected side effects. Complexity will still exist, but if the simple self-contained objects are typed as such, it will discourage software developers from creating undesirable entanglements.

Containment can be thought of as the converse of encapsulation. Encapsulation isolates private fields of an object from external meddling. It becomes easier to reason about an object's internal implementation when you know that there are no external readers or writers of the fields you are manipulating. The inside is protected from the outside with no incoming references. Conversely, with containment you can invoke a method on a contained object free from concern that this method could affect another critical object. The outside is protected from the inside with no outgoing references.

Although encapsulation features are far more common in programming languages than containment, containment is heavily used in computer systems. Perhaps the most well-known use of containers is the virtual machine. A single physical computer can run multiple virtual machines with complete isolation, as if they were running on separate hardware. If an administrator were to attempt to run many services without this separation, there would be significant risk of downtime due to simple things like a patch to one application requiring that the system reboot. Processes can also have issues with contention for operating system resources, e.g., attempting to listen on the same network port number. These issues disappear when services can be isolated into their own virtual machines.

However, virtual machines have other benefits beyond just isolation. With a defined container, it becomes easy to clone a virtual machine. It is common practice to take snapshots of virtual machines at regular intervals. If something catastrophic occurs, the machine can quickly be restored. Physical hardware, on the other hand, is more difficult to back up with the same precision. The virtual machine clearly defines what the machine is. For example, hardware BIOS settings are part of the virtual machine's definition and this is something that is not typically saved when a backup is done for a physical machine. Now, thinking in terms of a programming language, our goal is to show that the simple act of defining an enforced container boundary makes it easy to copy, serialize and restore objects.

Drawing once more from our analogy with virtual machines, we can see the role virtual machines play with scaleable concurrency. Elastic scalability is the ability to quickly add and remove resources for an application depending on present demand. Here, the virtual machine is a unit of deployment. Because of its self-contained nature, virtual machines can be replicated on-demand to add servers and likewise instances can be shutdown when needed. Again, we can see parallels in programming languages with significant research efforts working with concurrency models that do not use shared-mutable memory. Isolating threads from each other can greatly simplify a concurrent program.

Some examples of concurrency models that avoid shared memory are message passing models, actor models and map-reduce frameworks. One specific example is the use of channels and go-routines in the Go programming language. These features of Go provide a convenient way to coordinate threads. However, nothing in this system prevents the sharing of mutable memory across a channel. Of course a good programmer will know not to do that, but similarly a good programmer also knows to pass parameters with the correct types. The latter mistake will be caught by static typing. This can help to catch the errors that humans inevitably make. But, if we had a language that could enforce container boundaries, then we could use the type system to prevent a message from containing external references. The type system could prevent the sharing of memory across threads.

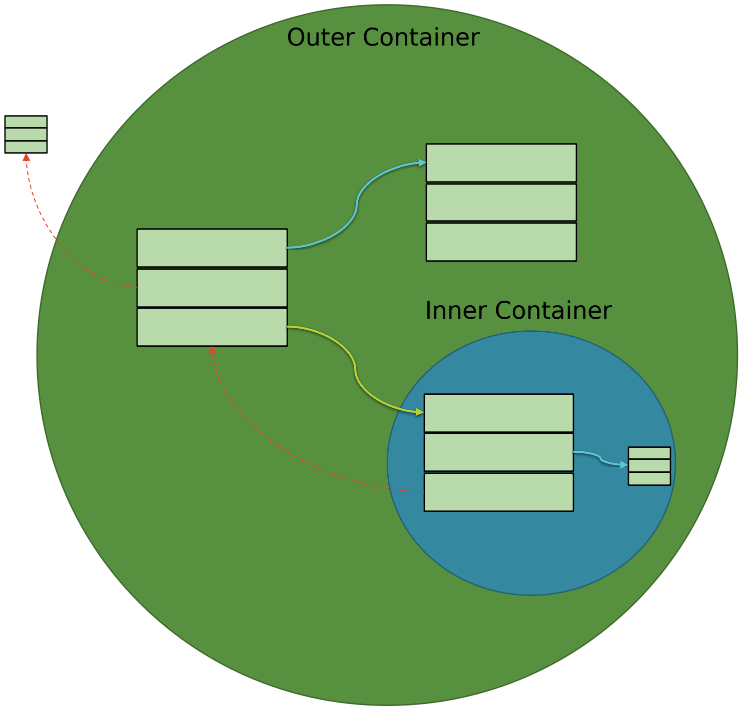


Figure 1.1: Containers: The diagram shows objects within containers (circles). The dotted red lines represent illegal references, which escape their container. Inward references (in yellow) are legal.

The goal of this thesis is to demonstrate how a container concept can be added to an imperative object-oriented language and to further show how containers can help solve issues such as the deep-vs-shallow-copy problem, serialization and deep comparison. We can also leverage containers to encourage pass-by-value semantics, reducing the unwanted aliasing that commonly appears in languages such as Java and Python where pass-by-reference is the norm.

1.1 Containers

For an object to be within a container c means that any field of the object must only reference objects which are also contained within c. Typically, a container is an object, but we also propose that a container could be a scope. For example, the local scope of a function invocation could be a useful container for temporary objects. When an object is constructed, it is placed into a specific container that can never change.

In addition to objects existing in containers, every reference will be typed with a container constraint which we call its container label. This indicates the container of the object being referenced. For example, a container label could specify that a reference is constrained to only point to objects contained within local variable `x`. We can see that container labels lead to a dependent type system. Every label requires its environment in order to be properly interpreted. For references that are fields in an object, the container label must not violate the container boundary of the object. It can be narrower than the container of the object, but never outside of the container.

We differentiate two kinds of objects in our system. Data objects are self-contained, meaning that no matter which container the object is placed into, fields of that object must not reference outside of the object. Entity objects are less strict; here, the container the object is placed in is the outer bound for references. We'll show that self-contained objects have useful properties, making the distinction worthwhile.

In order to enforce container rules, the typing system much check the container labels on every assignment and passing of parameters. Passing parameters to functions requires a careful consideration of scoping, as method parameters are declared in a different scope from the invocation of the method. Typing a function call is where most of the complexity of this system exists.

In the system developed in this thesis, there is significant tracking of containers at run-time. However, this exists for the purpose of verification of the type system. The goal of the type system is to statically check all references for proper containment. In doing so, the run-time overhead of containers can be zero.

1.2 Ownership Types

Earlier, we compared containment with encapsulation. Continuing this comparison, we highlight an area of research called ownership types. Chapter 6 discusses specific related work, but for now, we will outline key similarities and differences. With ownership types, objects are assigned an owner, which directly corresponds with our placing an object into a container. Where these systems differ is in their purpose. Ownership types have the concept of a dominator, where the owning object has exclusive control over the owned object. In our work with containers, we do not have this concept. Containers permit outside aliasing of contained objects. We lose the encapsulation benefits of ownership types, but we are free to focus on the containment concept in isolation.

```
class Demo {
    // The ’:: self’ notation indicates that reference r refers to an object
    // that is ’within’ the instance.
    // In an ownership type system, we would say that r is owned by the object
    // In this thesis, we say r is contained by the object
    ref r : Node :: self;
}

method Main.demo() {
    ref demo = Demo();   // Construct a new instance of Demo

    ref inner = demo.r;  // This would be a compile error in an ownership-type system.
                         // The field r is owned by demo, which encapsulates it.
                         // However, in our containment system this line is legal.
                         // We do not enforce encapsulation.

    ref demo2 = Demo();   // Construct another new instance of Demo

    demo.r =^ demo2.r;    // Reference assignment of the field r from demo2 to demo
                          // This line is illegal in our containment system, but
                          // encapsulation is not the reason.
                          // This is a containment violation.
                          // The field r in the demo container is not permitted to
                          // reference an object in the demo2 container.
}
```

Both ownership and containment use dependent type systems. In terms of type system mechanics, these two systems are very similar. Having a type be dependant on an owner versus a container makes little difference from a dependent-types perspective. The primary distinction between these two systems are the goals. We allow encapsulation violations in order to examine the potential of containment without the restrictions of ownership types.

1.3 Outline

We begin in chapter 2 by describing a language with container types. The grammar is introduced and code examples help explain the various behaviors. The container labels are fully specified and explained. Special attention is given to the handling of method calls and the use of generic container labels. Also detailed is a container label inferencing system which allows explicit specification of labels to be omitted in many cases. The language described has a working interpreter written in Haskell. The chapter ends with notes on the usage of this interpreter.

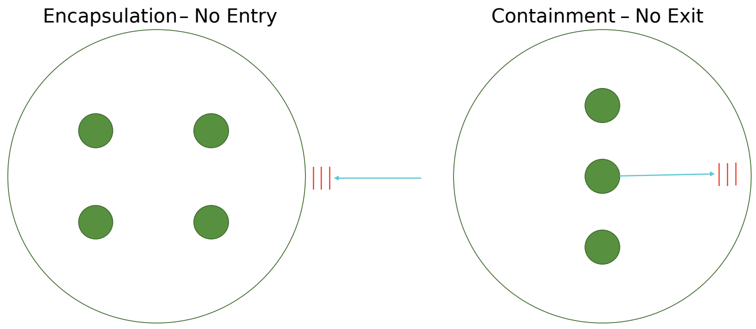


Figure 1.2: Containment vs. Encapsulation

Chapter 3 describes a formalized version of the language. The formal version is reduced in scope, but retains the key complication of managing container labels across scope boundaries. A full description of how types are composed is given, including specifying immutability and the container label. The grammar for this language is abstract and there is no container inferencing in this version. Finally, a typing environment is defined as well as a number of type rules to type the language in general, but also many specifics of working with container labels.

Chapter 4 defines the operational semantics of the formal container language. A system of small-step operational semantics is defined running in a configuration that tracks container properties. The semantics extend the language with a number of additional frames needed to execute more complex operations such as tuple initialization, parameter passing and copying objects. Type rules are provided for the additional frames. Additionally, type rules are provided to fully type the entire configuration. This includes the heap, frame stack and variable stack.

Chapter 5 takes the typings defined in chapter 4 and proves their soundness with the traditional progress and preservation theorems. There are a set of lemmas establishing that in this dependent type system, the elements that are depended upon are immutable, therefore typing will be preserved. The most important lemmas work with passing symbols across scopes to make sure that two distinct but corresponding container labels refer to the same physical object in the configuration.

Chapter 6 outlines how this thesis relates to other areas of research. And finally, chapter 7 describes areas where this work can be improved and extended.

Chapter 2

A Language with Containers

This chapter describes a demonstration language that implements a simple object oriented language which uses containers. We'll begin introducing the language by example. In the first example, we declare a reference to an instance of a class called demo. The extra syntax "`:: local`" constrains the references to only point at objects inside the container indicated by `local`, which refers to the scope of the current function. We call the container specification a *container label*.

```
ref r : DemoLL :: local =^ null;    // Operator =^ means assign reference
```

Objects can be containers as well and in most cases are more useful containers than the local scope. A container groups a set of objects that can work together (reference each other) and if you need to temporarily create such a group, then a `local` container label is useful. In our second code snippet, we define a class that we'll use to show how objects can be containers.

```
entity class DemoLL( id : Int, ref next : DemoLL :: container )

constructor DemoLL( id : Int ) {
    self.id = id;
    self.next =^ null; // For clarity, redundant since null is default.
}
```

Here, we define the class `DemoLL` which is a simple node of a linked list with two fields. The `next` field is declared with the label `container`. So far, we've seen a container label associated with a reference, but when an instance of `DemoLL` is created, it is also placed in a container. The reference field `next` is given the container label of `container` which refers to the container that the object was placed into. This means that the field `next` can only reference objects in the same container as the object. Also, note the `entity` keyword in the class declaration. This modifier indicates that the class is permitted to have references to other objects within a common container. By default, a class is a `data` class, which means it is self-contained. The label "`:: container`" is illegal in a data class because a reference with that label would violate self-containment. Now, we'll continue the example and make use of the `DemoLL` class in the next code snippet.

```
class Main ( fixed ref list : DemoLL :: self )
constructor Main() {
    self.list =^ DemoLL(1);
    self.start.next =^ DemoLL(2);
}
```

We have a program which begins by instantiating the class `Main`, invoking its constructor. Main is not declared with `entity`, so it is a self-contained class. Accordingly, its field `list` is declared with the container label `self`. Two nodes are created for our list and both are placed in the container `self`, which is the instance of class `Main`. When the constructor call `DemoLL`(1) was made, how did it know which container to place the new object into?

Container label inferencing allows the container label on the right-hand side of an assignment to be inferred based on the expectations of the left-hand side. In the example, `self.list` is declared with `:: self` and with this label on the left-hand side, there is only one correct label choice for placing `DemoLL`(1) into a container. Similarly, on the next line, `self.start.next` also resolves to the label `self`. The field `next` is typed with label `container`, which in this context refers to the container of `self.start`.

For containers to be a useful feature, the type checking must catch all container violations at compile time. We'll look at a simple error example next.

```
class Main ( ref list : DemoLL :: self )
constructor Main() {
    self.list =^ DemoLL(1);
    ref otherList :: local =^ DemoLL(2);

    self.list.next = otherList;    // COMPILER ERROR!
}
```

By placing the second instance of `DemoLL` into container `:: local`, it is segregated from objects in `:: self`. The attempt to assign to `self.list.next` is detected as a container violation. This type checking can prevent accidental mixing of objects that have the same class, but aren't intended to be used together. As an example, if you were assigning a partner to a police officer, you would want them both to be based in the same precinct. All data for a precinct could be in its own container object and the type system would ensure that any partner assignment operation respects the container boundaries. If a police management application were structured this way, it would be impossible to assign an officer a partner from a different precinct.

So far, we have been focusing on references, but there are also value types and indirect reference types. Indirect references are denoted by `iref`.

```
class Main ()
constructor Main() {
    var x = 1;
    iref r =^ x;  // r's type is inferred from its initializer
    r = 2;        // Assign through the indirect reference
    print x;      // Outputs 2, x has been modified
    x = 3
    print r;      // Outputs 3

    var y = x;    // Non-references (val & var) are always assign-by-value
    x = 4;
    print y;      // Outputs 3, y is fully indepedent from x
}
```

Value types are declared using `val` or `var`. Only self-contained data classes are permitted to be values. Entity classes are entangled with their environment and considered unsuitable for pass-by-value semantics, so they must be handled through references. Although the above example used the primitive type `Int`, the assign-by-value rules are the same for any self-contained class. When the variable `y` is initialized, a full deep copy of the initializer object is made.

With that quick introduction to the language, we will now take a deeper look at the language specification.

2.1 Grammar

Figures 2.1 and 2.2 detail the grammar of the language. Starting with figure 2.1, a number of binary operators are implemented. Most are self explanatory, but for `==` and `==^`, the former tests value equality and the latter tests reference equality. Instances of any self-contained classes can be compared for equality, no matter how complex. The algorithm for this comparison is explained in section 2.4.

Expressions include field accesses using a dot as in Java and other languages. However, we have both value and reference types in our language. In general, the semantics of the language is to automatically de-reference when needed. Field access, passing a value to a method and value assignment are all value contexts. Any time a value is expected, if the input is a reference, then an automatic de-reference is done. In contrast with C++, we do away with the `->` and `*` operators, but add `=^` to distinguish reference assignment from value assignment. Note that if the left-hand side of an assignment is an `iref`, then *bin-op* → `+` | `-` | `*` | `/` | `or` | `and` | `xor` | `>` | `==` | `==^`

e → *Bool* | *Integer* | *String*
e → *e bin-op e*
e → *e* `.` *field-name*
e → *e* `.` *method-name* (*e*, …, *e*)
e → *class-name* (*e*, …, *e*)
e → *var-name*
e → `self`
e → `null`

s → `if` *e* { s^* } `;`
s → `while` *e* { s^* } `;`
s → `return` *e* `;`
s → *e* `=` *e* `;`
s → *e* `=^` *e* `;`
s → `val` *var-name* = *e* `;`
s → `var` *var-name* = *e* `;`
s → [`read-only`][`fixed`] `ref` *var-name* [`::` *container-label*] = *e* `;`
s → [`read-only`][`fixed`] `iref` *var-name* [`::` *container-label*] = *e* `;`
s → `save` *e e* `;`
s → `load` *e e* `;`
s → `print` *e* `;`

path → `self`
path → *var-name*
path → *path* `.` *field-name*

container-label → *path*
container-label → `container`
container-label → `local`
container-label → ‘ *generic-name*
container-label → `unknown`

Figure 2.1: Expression, Statement and Container Label Grammar

program → *decl**

decl → [`entity`] `class` *class-name* (*field-decl*, . . . , *field-decl*)
decl → [`read-only`] `method` *class-name* `.` *method-name*(*parm-decl*, . . . , *parm-decl*) *ret-decl*
decl → `constructor` *class-name* (*parm-decl*, . . . , *parm-decl*)

parm-decl → [`val`] *field-name* `:` *class-name*
parm-decl → `var` *field-name* `:` *class-name*
parm-decl → [`fixed`] [`read-only`] `ref` *field-name* `:` *class-name* [`::` *container-label*]
parm-decl → [`fixed`] [`read-only`] `iref` *field-name* `:` *class-name* [`::` *container-label*]

field-decl → `val` *field-name* `:` *class-name*
field-decl → [`var`] *field-name* `:` *class-name*
field-decl → [`fixed`] [`read-only`] `ref` *field-name* `:` *class-name* [`::` *container-label*]
field-decl → [`fixed`] [`read-only`] `iref` *field-name* `:` *class-name* [`::` *container-label*]

ret-decl → `:` *class-name*
ret-decl → [`read-only`] `ref` `:` *class-name* [`::` *container-label*]
ret-decl → [`fixed`] [`read-only`] `iref` `:` *class-name* [`::` *container-label*]

Figure 2.2: Declaration Grammar

both reference assignment and value assignment are possible. With this extra operator, it is always clear whether a value or reference is needed.

Method/constructor calls and return statements behave as one would expect, but with additional consideration of containers. Section 2.3 will detail the handling of containers as parameters are passed and return values are received. The expression `self` functions like the `this` keyword in familiar languages like C++ and Java. We use the term self-contained extensively and it was natural for the language to match.

Like many other languages, statements include variable declarations, assignment, and flow control statements like `if, while, return`. In the examples, we've seen variable declarations of the various kinds. Also included is a `print` statement to display an object's contents and `save/load` statements which can save a self-contained object to a file and then load it back into a variable that will compare as equivalent to the original. These three IO statements are discussed further in section 2.4.

Also in figure 2.1 is the full definition of the container labels. We have already seen examples of `local, self, container`, and the remaining labels will be discussed in the next section 2.2.

In figure 2.2, the remainder of the grammar is defined. A program is a series of declarations which can be classes, methods or constructors. Nested within the top-level declarations are individual parameter declarations and in the case of methods a return type declaration.

Classes are data classes by default, meaning they must be self-contained. The optional `entity` keyword enables the declaration of references that point outside of the object. When fields are declared, the default kind is `var` and that keyword can be omitted. The container label of references can be omitted as well with the default being `self`. Reference fields can also be typed as `fixed` and/or `read-only`. The meaning of `fixed` is that the identity of the referred object cannot change and reference assignment is not allowed with this symbol. For a `read-only` reference, you are not permitted to modify the referred object and value-assignment statements are not allowed. Note that `read-only` doesn't guarantee immutability, as another alias could be able to modify the object.

When declaring parameters of methods and constructor calls there are also defaults, but they are different. If a container label is omitted from a parameter or return value then the default is `unknown`, which is useful when the method doesn't care what container the object is in. If no kind is specified for a parameter, then `val` is the default.

When values are returned, they are always immutable. Since value assignment requires a copy, the mutability of an expression result is of no consequence. Attempting to reference-assign an expression result value (r-value) to a reference is illegal.

2.2 Container Labels

In this section, we explain in more detail the full set of container labels. The grammar is repeated in figure 2.3 for convenience. The simplest form of a *path* container label is a local variable or the special keyword `self`. These symbols directly indicate a container object. Since the container is an object and we know its type, a path can be extended through field accesses to indicate a more precise container. Section 2.2.1 will explain paths further.

Each of the container labels we've seen so far specify a container relative to the current scope. A mechanism is needed to relate containers as they are passed to a function. The generic container labels provide the solution. A function can declare its reference parameters as generic, which enables its implementation to enforce abstract container constraints without knowing the identity of the containers. Section 2.3 provides a deeper look into generic container labels.

We've seen the label `container` in our previous examples. When this label is used in the type of a field, it refers to the container of the object holding the field. When used in a method, it refers to the container of the implicit self parameter. When used with parameters, the label `container` behaves just like a generic label. It can be modeled as an implicit `self :: ‘container-of-self` declaration. The label `self` has a similar dual meaning depending on context. For fields, it represents the object holding the field and for parameters, it represents the implicit `self` parameter.

The label `unknown` is typically used implicitly by omitting the `::` for a parameter. The normal use case would be a reference parameter of a method. Container labels are only needed when two or more objects are related to each other by a common container. A method may call methods on another object without knowing its container as long as the called method doesn't have a parameter typed with `:: container`. If the called method did have such a parameter, then our outer method would be unable to pass a parameter with the correct label.

container-label → *path*
container-label → `container`
container-label → `local`
container-label → ‘*generic-name*
container-label → `unknown`

path → `self`
path → *var-name*
path → *path*.*field-name*

Figure 2.3: Container Labels

2.2.1 Paths

Figure 2.4 shows an example of a local variable as a container label. Notice that the variables `x` and `y` are declared as `fixed`, meaning that they can't be re-assigned. The code of this example would not compile otherwise and you would receive a compile error at `r1 =^ r2`. The reason the error would be raised is that `x` could have been re-assigned between the initialization of `r1` and `r2`. Without full data-flow analysis, the compiler doesn't know that `r1` and `r2` reference the same container. With *path* container labels, every component of the path must be `fixed`. With explicit container labels, you will receive an error if you attempt to declare an unfixed label. If you declare a reference without an explicit label and the inferred label is unfixed, then the reference is implicitly declared with label `unknown`. This stability requirement means that in the dependent type system, all dependencies are immutable and types are preserved.

The example of figure 2.4 continues and declares a variable `y` and then attempts to mismatch containers. The compiler will not allow `r1 =^ y.v1` because `r1` can only reference objects contained with `x`.

```
class Info( id : Int, v1 : Int, v2 : Int )
constructor Info( id : Int, v1 : Int, v2 : Int ) {
    self.id = id;
    self.v1 = v1;
    self.v2 = v2;
}

class Main ()
constructor Main() {
    fixed var x = Info( 1, 10, 100 );

    iref r1 :: x =^ x.v1;  // The local variable x is the container
    iref r2 =^ x.v2;       // The container is inferred as x based on rhs
    r1 =^ r2;              // Legal because the containers match

    fixed var y = Info( 2, 20, 200 );
    r1 =^ y.v1;            // COMPILER ERROR!
}
```

Figure 2.4: Variables as Container Labels

Figure 2.5 contains a more complex example of paths using field accesses. References `r1` and `r2` have matching containers as before, but this time the path to the container has been extended to include a field access. Container labels can be declared where the container isn't directly in the current scope. In this case, we reach inside the variable `x` to find the container.

As mentioned before, every element of a path must be immutable. The local variable x is declared as fixed as well as the fields `i1` and `i2`. This means that the expression `x.i1` always evaluates to the same container and we can rely on it as a type. Similar to the last example, figure 2.5 also shows a compile error where the there is an attempt to reference-assign an object contained in `x.i2` to a reference typed with label `x.i1`.

Generally, container labels must be an exact match when doing a reference assignment. One exception is that `null` can be assigned to any reference. A second exception is when a reference is declared with label `unknown`, which allows for the reference to point within any container. There is no reflection mechanism to narrow a reference typed with an `unknown` label into a specific container label and this means that you are limited in what you can do with an `unknown` container label. Fields are not permitted to use the `unknown` label because a field is always restricted to the container of its object.

```
class Inner( id : Int, v1 : Int, v2 : Int )
constructor Info( id : Int, v1 : Int, v2 : Int ) {
    self.id = id;
    self.v1 = v1;
    self.v2 = v2;
}

class Outer( fixed ref i1 : Inner, fixed ref i2 : Inner )
constructor Info( id1 : Int, id2 : Int ) {
    self.i1 =^ Inner( id1, id1 * 10, id1 * 100 );
    self.i2 =^ Inner( id2, id2 * 10, id2 * 100 );
}

class Main ()
constructor Main() {
    fixed var x = Outer( 1, 2 );

    ref r1 :: x.i1 =^ x.i1.v1; // The field i1 of local variable x is the container
    iref r2 =^ x.i1.v2;        // The container is inferred as x.i1 based on rhs
    r1 =^ r2;                  // Legal because the containers match

    r1 =^ x.i2.v1;             // COMPILER ERROR! x.i2 is distinct from x.i1
}
```

Figure 2.5: Paths as Container Labels

2.3 Methods and Constructors

The checking of container labels for parameters and return values of methods requires a correspondence between symbols in two different scopes. We'll begin with an example shown in figure 2.6. We have a method called setTest() which takes a fixed reference r1 as its first parameter and this serves as the container we'll be focusing on. The second parameter r2 has r1 as its container label so we know that r2 is within r1 and if we modify r2, then part of r1 will change as well. The method updates the value referenced by r2, then dumps the entirety of r1 to reveal what has changed.

Now, looking at each of the calls to setTest(), we can see that the first two correct cases pass a second parameter which is contained within the first parameter. And, to demonstrate the error case, the third example mismatches x and y.

When type checking a method call like this, a mapping is built for all of the labels in the function call's parameters. First, when processing the first parameter, the container label r1 from the method's scope is associated with the container label x in the caller's scope. Then, when type checking the second parameter, we can use the mapping established by the first parameter to take the declared container label r1 of the second parameter and

```
class Info( v1 : Int, v2 : Int )
constructor Info( v1 : Int, v2 : Int ) {
    self.v1 = v1;
    self.v2 = v2;
}

class Main ()
method Main.setTest( fixed ref r1 : Info, iref r2 : Int :: r1, v : Int ) : Int {
    r2 = v;      // Assign through the indirect reference
    print r1.v1;
    print r1.v2;

    return 0;  // Currently the language doesn't permit returning void
}

constructor Main() {
    fixed var x = Info( 1, 10 );

    self.setTest( x, x.v1, 2 );    // Outputs 2, 10
    self.setTest( x, x.v2, 20 );   // Outputs 2, 20

    fixed var y = Info( 5, 50 );
    self.setTest( x, y.v1, 6 );   // COMPILER ERROR! y.v1 is not in x
}
```

Figure 2.6: Method Calls with Containers

map it into the caller's scope. With this mapping complete, we can type check the second parameter, which must have label `x`.

The first call to `setTest`() modifies v1 and the second call modifies v2. Because of the explicit specification of container, the side effect of `r1` being changed when `r2` is modified should not be a surprise. Perhaps more important than knowing what will change after an operation is certainty about what cannot change. It is impossible for the line `r2 = v` to modify anything other than the object `r1`.

When returning references, the same mapping process occurs. A method could return a reference contained within any of its parameters or the implicit `self` parameter. You can also pass and return references contained within the container of any parameter. We'll see how this can be done with generic container labels.

2.3.1 Generic Containers

In the previous example in figure 2.6, we declared a container label to be a previously declared parameter, but in many cases there is no need to pass the container itself, just the pieces we intend to interact with. With generic container labels, we can pass a parameter with an abstract container label. Figure 2.7 demonstrates this capability. Here, we've implemented a linked list insert method twice, and each implementation has a distinct container label. Now, in `Main`, we create a linked list with 3 elements in it. First, notice that when we call the constructor of `DemoLL`, we never explicitly need to place it in a container; it happens automatically. In the first construction, the container label is inferred from the left-hand side of the assignment as we've seen before. However, for the the inserted elements, the containers for the newly constructed nodes are inferred based on the expected parameter types of the method calls. This is interesting because the parameter types are generic and also must be resolved.

The next section will outline the inferencing algorithm, but for now we will reason through the inferencing steps specific to this example. With the method `self.insert`(), the generic container label `‘c` appears three times. As long as one of those instances has a definite container label in the method call, then the other two instances become fixed and their definite labels can be used for further inferencing.

A constructor's return type doesn't appear visually in the code, but to the inferencing system, every constructor returns a reference with a container label of `:: container`. This label becomes a concrete container at the call site by the same inferencing techniques used to resolve generics in the parameter labels.

Returning to the call to `self.insert`(), we see that its first parameter has the container label `:: self` which then becomes the label of the second parameter. Next, the constructor call to `DemoLL(2)` has an expected return type with label `:: self` which fully resolves the labels for the construction. Last, the return value of `self.insert`() is also contained within `:: self`.

The insertion of the third list element uses a different insert method, but the method's container labels map to the same containers in the caller's scope, allowing it to be compatible. The final call to `DemoLL.insert`() is a method call on the return value from `Main.insert`(), which in this example is the second node of the list with the container label `:: self` which refers to the instance of `Main`. The parameter `newNode` of `DemoLL.insert`() is typed with container label `:: container` and the mapping of this label into the caller's scope is inferenced using knowledge of the implicit `self` parameter's container label `:: self`. As before, the return container label for the call to `DemoLL`(3) can be determined since we now know the label for the parameter to `DemoLL.insert`().

This is a contrived example and the inferencing process for generics is complex. However, from a programmer's perspective when implementing a method, the parameter labels are abstract and there is no need to be concerned about how they will later resolve. As a

```
entity class DemoLL( id : Int, ref next : DemoLL :: container )

constructor DemoLL( id : Int ) {
    self.id = id;
}

method DemoLL.insert( ref newNode : DemoLL :: container ) ref : DemoLL :: container {
    newNode.next =^ self.next;
    self.next =^ newNode;
    return newNode;
}

method Main.insert(ref node : DemoLL :: ‘c, ref newNode : DemoLL :: ‘c) ref : DemoLL :: ‘c
{
    newNode.next =^ node.next;
    node.next =^ newNode;
    return newNode;
}

class Main ( fixed ref list : DemoLL :: self )
constructor Main() {
    self.list = DemoLL(1);
    self.insert( self.list, DemoLL(2) ).insert( DemoLL(3) );
}
```

Figure 2.7: Generic Container Labels

programmer calling a method, some care needs to be taken to ensure that the parameters have matching container labels, but there is no need to understand every detail of how the mapping between caller labels and callee labels is established.

2.3.2 Container Inferencing Algorithm

A goal of this project was to minimize the syntactic burden of specifying container labels and an inferencing algorithm was developed to allow the container label to be omitted in many cases. A bi-directional typing algorithm was developed to infer labels for variable initialization, assignment statements and method/constructor calls. Note that inferencing only occurs within a single statement. When each expression is typed, an expected container label is provided and this assists with the typing of constructor calls and methods that return generic labels. We'll look at the method call reasoning in detail.

When a method call is typed, we have multiple sources of information; the expected return container label is provided by higher-level code, typing information is in the method declaration. We also have the expressions passed to the method from the program's abstract syntax tree. Specifically, we have an expression for the implicit `self` parameter and each of the explicit parameters.

For each expression passed to the method call, we designate a fresh generic container label symbol. Then, we type each of the parameter expressions using the respective generated generic labels as the *expected* container label for the expression. E.g., `typeExpr`(*expr, generated-generic-label*). Often, expressions will immediately be typed with a concrete container label, but if an expression type comes back with the same generic type we passed in, then we know that the expression is free in its container label. At this point in the process, we have a list of expressions as well as preliminary typings for each of them. Note that after the inferencing process completes, we will re-type these expressions.

Next, we need to deal with the fact that the method is declared with container labels relative to the method's own scope. We need to take the method signature and associate the labels declared with each parameter with labels in the calling scope. To do this, we first use a function called `expressionToContainerLabel`(), which can take each of the parameter expressions and determine what the container label should be if that expression result is used as a container. Recall the example in figure 2.6, where the second parameter was contained within the first parameter. If a situation like that occurs, then `expressionToContainerLabel`() tells us what the appropriate label should be in the caller's scope. We now have a mapping from parameter names to caller-centric labels. Using this mapping, we map the method's declared container labels into the scope of the caller.

The inferencing can now be done in the container label space of the caller. Figure 2.8 presents pseudo code for this operation. For simplicity, we've flattened the inputs to just two lists of labels: *decl-labels* and *passed-labels*. These represent the labels for each passed parameter, based on their preliminary typing, as well as the expected return label. The two lists are aligned to each other, meaning index i in each list represents the ith parameter.

The inferencing consists of three steps. First, we create a map with the domain being the set of generic container labels declared in the method signature. These generic container labels should not be confused with the temporary unique *free-generics* that we injected when we typed the parameter expressions; these generic labels are from the method signature. Each generic label may appear in the method signature more than once, and for each generic, we record the passed-in container labels corresponding to the generic label. The function `findGenericParameters` implements this operation in figure 2.8. The result is a map from generic labels to a list of the passed container labels.

For each generic mapping, we merge the list of labels into a single label. The function `mergeAllLabels` does the merging. In this process, the injected *free-generics* are replaced

```
inferGenericParameters( free-generics, decl-labels, passed-labels ) =
    let
        // Map each declared generic label to a list of passed labels
        generic-map = findGenericParameters( zip(decl-labels, passed-labels) )

        // For all generics, merge the occurrences
        mergedGenericMap = genericMap.mapRange( mergeAllLabels )

        // If any free generics remain after merging then the program
        // has no concern for which label is chosen so we use - local
        mergedGenericMap = mergedGenericMap.substRange[label ∈ free-generics -> local]


findGenericParameters [] = {}
findGenericParameters( ⟨decl-label, passed-label⟩ ∘ tail) =
    let
        generic-map = findGenericParameters( tail )
    in
        if decl-label is generic then
            if decl-label ∈ dom(generic-map) then
                generic-map.set( decl-label ↦ passed-label ∘ generic-map.get(decl-label) )
            else
                generic-map.set( decl-label ↦ [passed-label] )

mergeLabels( unknown-cont, label2 ) = error: unknown-cont is incompatible with label2
mergeLabels( label1, unknown-cont ) = error: unknown-cont is incompatible with label1
mergeLabels( null-cont, label2 ) = label2
mergeLabels( label1, null-cont ) = label1
mergeLabels( label1, label2 ) =
    if label1 ∈ free-generics
        if label2 ∈ free-generics
            label1   // Both are free, keep label1
        else
            label2   // Use the non-free label
    else if label2 ∈ free-generics
        label1   // Use the non-free label
    else if label1 == label2 then
        label1
    else
        error Labels label1 and label2 are incompatible

mergeAllLabels( [ label1 ] ) = label1
mergeAllLabels( label1 ∘ tail-labels ) =
    mergeLabel( label1, mergeAllLabels(tail-labels) )
```

Figure 2.8: Pseudo-code for Container Label Inferencing

with container labels from the caller's scope. Errors are raised if there are inconsistencies. The final result is that we have a concrete container label for every generic container label in the method signature.

With this map in place, the typing of the method call can be completed. As a final step, each parameter expression is re-typed now that we know the exact container label needed for each expression. This is how we can call a method and pass a constructor call as one of the parameters. Once the specific parameter knows its container label, then the constructor call is re-typed, which sets the container for the newly constructed object.

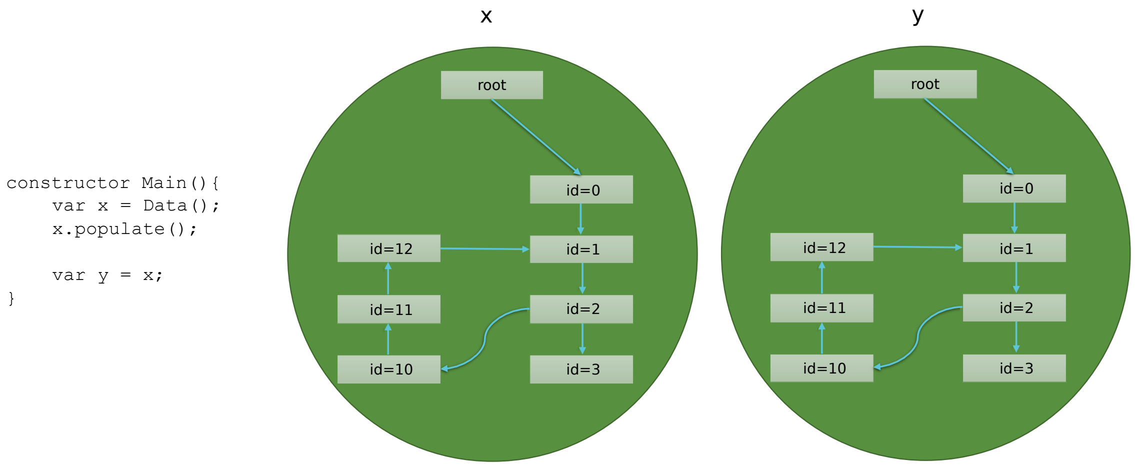
With this inferencing logic in place, most container specification can be eliminated from code. Declarations of methods and fields will need explicit syntax, but there are defaults in place that can help there as well. Any proposed language feature faces a cost/benefit analysis and the inferencing logic was developed to reduce the overhead cost of using container types.

2.4 Self Containment

In this section, we look at four capabilities of self-contained objects. The language allows you to declare data classes and entity classes, which mean self-contained and non-self-contained. This distinction allows for extra capabilities to be added only for self-contained objects.

An entity is entangled with its environment. Imagine a machine that could clone a person. After one person becomes two, then which one will receive a pay check from their employer? Which one gets to live with their spouse? The point of this example is that copying an object that has relationships to its environment is complicated. Logic to copy an entity class needs to be customized for that particular class. Our language takes the opinion that entities should never be copied automatically. Similar to Java, you can write a clone method if you desire, but there is no language support. This can be seen by the rule that entity objects are always managed by reference, never by value.

Self-contained objects, on the other hand, are pure data. It is natural to copy them because they have no entanglements with their environment. One novel aspect of this system is that inside a data object, there can exist entity objects with complex relationships. Entity objects are copied as part of copying the outer data object, but an entity is never copied by itself. The well-defined boundary means we can recreate an identical graph of the inner entities.



```
constructor Main(){
    var x = Data();
    x.populate();

    var y = x;
}
```

Figure 2.9: Deep Copy On Assignment: Assigning by value creates a new disjoint object.

Contrast this to a language like C++. If you built a class using shared smart pointers, you will get a shallow copy from the default copy constructor. If you use direct containment, then then you get a deep copy. There is no in-between. If you have a nested object with multiple aliases, then there is no immediate way to copy the nested object only once and have every alias in the copy point at the new nested object. Of course, you can implement this yourself, but there is no automatic support. Next, we'll explain how we can solve the deep copy vs. shallow copy problem for self-contained objects.

2.4.1 Object Copying

By declaring a class as self-contained and having the type system enforce that containment, you can be certain that every object reachable from the self-contained object belongs to the object and should also be copied. After a deep-copy operation, the resulting object is completely disjoint from the original and modifications to the original will have no effect on the copy. If the original object had multiple aliases that could modify the object unexpectedly, then a local, unaliased copy can be created, which is immune to outside mutation.

```
copySelfContained( heap, src-location, class )
    // Begin the copy process with an empty copy-map
    ⟨copy-map, dst-location⟩ = copy(heap, ∅, src-location, class)
    return dst-location

// Recursively copy a tuple within an accumulated copy-map
// Tuples that have already been copied simply return the heap location
// of the copy.
copy( heap, copy-map, src-location, class )
    if src-location ∈ copy-map then
        return ⟨copy-map, copy-map(src-location)⟩

    dst-location = heap.alloc( class )
    copy-map = copy-map[src-location ↦ dst-location]
    src-fields = heap(src-location)

    for( i = 0; i < len(src-fields); i++ )
        if src-fields[i] is primitive then
            heap(dst-location)[i] = src-fields[i]
        else
            ⟨copy-map, dst-field⟩ =
                copy(heap, copy-map, src-fields[i], classOf(src-fields[i]))
            heap(dst-location)[i] = dst-field

    return (copy-map, dst-location)
```

Figure 2.10: Pseudo-code for Deep-Copying a Self-Contained Tuple

Copying objects in the container language is automatic and there is no `copy` function or operator. Copying occurs automatically when assigning by value or passing a parameter by value. As an example, once you can treat an object like data, then a feature like an undo button in a word processor could simply be implemented by taking periodic snapshots of the document and simply reverting to an earlier snapshot when needed.

Figure 2.10 outlines the algorithm to deep-copy tuples. The key component is the *copy-map*, which records a mapping of old source tuples to their copied counterparts. The copy process is recursive, and when a tuple is found for the first time, it is copied and added to the map. If there is a second alias to the tuple, then a map lookup is used to find the matching copied tuple and avoid copying the same tuple multiple times. This ensures that cycles of references are handled correctly and the code does not get stuck in an infinite loop.

```
compareTuple( heap, compare-map, lhs-location, rhs-location, class )
    if lhs-location $\in$ compare-map then
        // If we are comparing a tuple we compared before, then the lhs must also be the
            same tuple
        return $\langle$compare-map, compare-map(lhs-location) == rhs-location$\rangle$
    else
        compare-map = compare-map[lhs-location $\mapsto$ lhs-location]
        lhs-fields = heap(lhs-location)
        rhs-fields = heap(rhs-location)
        field-types = getFieldTypes( class )

        // Process each field
        for( i = 0; i < len( lhs-fields ); i++ )
            if field-types[i] is primitive then
                if lhs-fields[i] != rhs-fields[i] then
                    return $\langle$compare-map, false$\rangle$
            else if (lhs-fields[i] == null and rhs-fields[i] != null) or
                        (lhs-fields[i] != null and rhs-fields[i] == null) then
                    return $\langle$compare-map, false$\rangle$
            else
                $\langle$compare-map, same$\rangle$ = compareTuple( heap, compare-map, lhs-fields[i],
                    rhs-fields[i], field-types[i].class )
                if not same then
                    return $\langle$compare-map, false$\rangle$
        return $\langle$compare-map, true$\rangle$
```

Figure 2.11: Pseudo-code for Deep Object Comparison

2.4.2 Equality Comparison

Another benefit of self-containment is a well-defined equality comparison. Comparison by reference equality is common practice in object oriented-languages. Here, we have the ability to compare complex objects by their content using the common == operator. The process works very much like the copy algorithm. When two tuples are compared, a map is kept to associate nested tuples with the corresponding tuples in the other object. Figure 2.11 outlines a comparison algorithm. The algorithm is recursive and it short-circuits when a tuple is encountered a second time, just like the copy algorithm.

This algorithm would work on a non-self-contained object as well, however, we disallow it based on the opinion that entities are better compared by reference. As with copying objects, custom comparison code can be written for entities. Custom code has the advantage of knowing the purpose of an entity's outside relationships and can make better decisions than a generic comparison.

2.4.3 Serialization

The last benefit of self-contained objects we'll discuss is serialization. The language supports the statements `load` and `save` that can be used to persist and reload any self-contained object. Serialization can be done using the similar recursive logic with a map tracking previously encountered objects as was used to copy objects. By using containers, we can guarantee that an object serialized to a file and then de-serialized back into a new object will compare as equal to the original using the `==` operator.

Serialization of data covers a wide range of use cases such as persistence and messaging between threads or processes across a network. By requiring self-containment, we don't require additional meta-data in order to marshal data correctly. Because of container enforcement, there is also nothing a programmer could do to make a self-contained class un-serializable.

2.5 Demonstration System

The demonstration language was implemented in Haskell and implemented as an interpreter using a style that loosely resembles small-step semantics. This design was chosen so that this code base could inform the subsequent work to formalize the language. The source code for the project can be found on gitlab at https://git.uwaterloo.ca/mthode/woven-c.

There are a large number of test cases in the `test/` folder and although they were written for testing, they would also be useful to a human looking to get a feel for how the language works. On a Linux system, the interpreter can be built using the `build` command in the root directory. The `ghc` Haskell compiler must be installed beforehand. In particular, the test `test/stmt29.woven` would be a good example to start with. It implements a linked list and implements a sort method on the list. Each test also includes a `.chk` file which verifies the output when the `-t` option is used.

```
Usage: woven [args] (<source-file> | <test-name>)
-t      Testing Mode.  Runs a test if specified, otherwise runs all tests
-i      Interactive execution
-vt     Verbose output of scanned tokens
-va     Verbose output of abstract syntax tree
-vta    Verbose output of annotated/typed abstract syntax tree
-ve     Verbose output of the type environment
-vx     Verbose output of each step of execution
-vg     Write graph file graph.dot after completion
```

The interpreter breaks execution down into steps, which can be observed using the `-i` command line option. When this option is used, the heap and stack are displayed after each step is executed. Also of interest is the -vg option, which will output a diagram of the heap into a file called `graph.dot` which can be interpreted by the Graphviz appliction. This was a useful tool to verify object copying semantics were working correctly.

```
dot -T pdf graph.dot > graph.pdf
```

Chapter 3

Containers Formalized

In this chapter, we will formalize a type system and language for containers. The language is reduced in scope for the system described in chapter 2 to reduce the complexity. Methods have been replaced with functions. At the beginning of development, it was assumed that methods would need special treatment in order to manage their containers. However, through the development of generic containers, it was found that the special containers `self` and `container-of-self` could be supported by the same generic container mechanism used for other parameters. Since no special treatment was necessary, the formal system was simplified to plain functions and constructors were replaced with a tuple initializer command.

Indirect references were also removed for two reasons. Firstly, in the demonstration system, the implementation of `iref` didn't add any significant challenges beyond what was needed for plain `ref` variables. Secondly, this work in its present state hasn't properly motivated `iref`'s existence. In chapter 7 there is a brief discussion about adding a concept of a *role* to the language, and `iref` was meant to be a building block for that work.

Primitive types and operators have also been removed, as these can be modeled as special tuples and functions. With these changes in mind, significant complexity still remains and we are left with many details that are often omitted in formal works. Types are defined as a 5-tuple in order to include container information as well as carefully tracking properties like mutability which are critical to the type rules.

Although methods are replaced with functions in the formalism, tuples remain a key ingredient and the intent is still to consider these tuples as objects even though an abstraction mechanism and polymorphism are not present. We will refer to a tuple's definition as a class, remaining consistent with object-oriented terminology.

In addition to the planned changes for the abstract language, there were many incremental changes made as the formalism was developed. Even though the interpreter was implemented in a style similar to the formalism, modifications were needed in order to facilitate proving the soundness of the semantics. There is no claim that the soundness of the formalism implies soundness of the demonstration system as there are many differences. No doubt, some of the changes that were needed in the abstract language represent bugs in the demonstration system.

The type system defined in this chapter is a dependent type system which introduces significant complexity. The types defined in this chapter include a container label which refers to a another in-scope construct (object or scope) to indicate the container of an object or the container constraint of a reference. One of the primary concerns of this formalism is stability and the importance of types depending solely on immutable symbols.

3.1 Types

This section details the structure of a container-type. Figure 3.1 defines the formal representation of a type including a list of the possible container labels. We'll discuss each of the components in turn.

Variables and fields have a *kind*, which can be either a `ref` or a `value`, and each variable also has a mobility and mutability component, which are represented with the symbols δ and γ, respectively. Mobility is similar to Java's `final` keyword. Here, a reference is considered `movable` if the identity of the object referred to can change. Mutability indicates whether the destination of a reference can be modified. We also need to distinguish between an object being mutable and a reference that is able to mutate the object it refers to; the former being a guarantee that a referenced object will never change, and the latter simply a permission granted to the reference. These distinctions are essential because the container language uses dependent types which require certainty that all dependent properties are preserved.

One option considered was representing mutability as two properties: one describing the reference (*this-alias-has-mutation-permission*) and the other describing the object (*this-object-is-mutable*). One issue is that they are not orthogonal, because you can't have permission to modify something that is immutable. This scheme was rejected in favor of a single property with three possible values `mutable, read-only, immutable`. The `read-only` state disallows a reference from modifying its object, but it does not guarantee that the object can't be modified by another alias. Therefore the type rules will be

Identifiers

C	Class name
F	Function name
f	Field name
v	Variable name (parameter or local variable)
g	Generic container label name

Container Labels

$$\begin{array}{lll}
\mathit{Path} ::= \texttt{tuple}\ \texttt{|}\ \texttt{var(}v\texttt{)}\ \texttt{|}\ \texttt{step(}\mathit{path}, f\texttt{)} & & \mathit{path} \in \mathit{Path} \\
\mathit{ContainerLabel} ::= \texttt{unknown-label} & & \theta \in \mathit{ContainerLabel} \\
\qquad\qquad\quad\ \texttt{|}\ \texttt{null-label} & & \\
\qquad\qquad\quad\ \texttt{|}\ \texttt{local} & & \\
\qquad\qquad\quad\ \texttt{|}\ \texttt{container-of-tuple} & & \\
\qquad\qquad\quad\ \texttt{|}\ \texttt{generic(}g\texttt{)} & & \\
\qquad\qquad\quad\ \texttt{|}\ \mathit{path} & &
\end{array}$$

Types

$$\begin{array}{lll}
\mathit{Kind} ::= \texttt{value}\ \texttt{|}\ \texttt{ref} & & \mathit{kind} \in \mathit{Kind} \\
\mathit{Mutability} ::= \texttt{mutable}\ \texttt{|}\ \texttt{read-only}\ \texttt{|}\ \texttt{immutable} & & \delta \in \mathit{Mutability} \\
\mathit{Mobility} ::= \texttt{movable}\ \texttt{|}\ \texttt{unfixed}\ \texttt{|}\ \texttt{fixed} & & \gamma \in \mathit{Mobility} \\
\mathit{Type} ::= \langle \mathit{kind}, C, \theta, \delta, \gamma \rangle & & T \in \mathit{Type}
\end{array}$$

Type Deconstructors

With $T = \langle \mathit{kind}, C, \theta, \delta, \gamma \rangle$

$$T_{\texttt{(kind)}} = \mathit{kind} \quad T_{\texttt{(class)}} = C \quad T_{\texttt{(label)}} = \theta \quad T_{\texttt{(mut)}} = \delta \quad T_{\texttt{(mob)}} = \gamma$$

Figure 3.1: Representation of Container Types

pessimistic, and read-only will disqualify an expression from being used to indicate the container of a symbol.

Having a read-only state simplified the type rules because it allows uncertainty about the true mutability of the object. Precise knowledge of every object's mutability would be a nice property to have, but functions would lose generality if immutable parameters were distinct from read-only-mutable parameters.

References can't be declared as immutable following the principle above about the generality of passed parameters. It's not permitted for values to be declared as read-only; this wouldn't make sense because there is no ambiguity at the original declaration site. Ambiguity of the mutability of an object only arises after aliases are created and passed around.

The story is similar for mobility, where movable means that a symbol can be *moved* to reference a different object. Only when an expression types with $\gamma =$ movable can that expression be used on the left-hand side of an assignment statement. The other use of mobility is to help determine when an expression can be converted into a container label. This is important for typing field access expressions, which will be discussed further in section 3.4.1. Only when $\gamma =$ fixed can an expression be converted into a container label. Similar to read-only, unfixed is a middle ground where stability is not guaranteed and assignment is not possible.

The final component of a type is the container label, which indicates the container relative to symbols in the current scope or to the scope itself. This is the key feature of this system which enables the typing of containers.

```
ref r : Foo :: local =^ null;           // <ref, Foo, local, mutable, movable>
ref r : Foo :: x.y =^ null;             // <ref, Foo, step(var(x), y), mutable, movable>
readonly ref r : Foo :: local =^ null;  // <ref, Foo, local, read-only, movable>
fixed ref r : Foo :: local =^ null;     // <ref, Foo, local, mutable, fixed>

val x : Bar = Bar();                    // <value, Bar, local, immutable, fixed>
var x : Bar = Bar();                    // <value, Bar, local, mutable, movable>
fixed var x : Bar = Bar();              // <value, Bar, local, mutable, fixed>
```

Figure 3.2: Formal Type Examples: Type declarations mapped to their formal representation.

When references are typed with container labels, they describe the container of the referred object. Value types behave differently, and as we'll see in the next section, the type rules require that value symbols must be typed with self-contained classes. This means that unlike references, the container for values is always the scope in which they are declared. For example, a local variable of a self-contained class will always have container label `local` and a field value will always have container label `tuple`. In chapter 2, details like this were implicit, whereas in the formalism everything is made explicit.

The label `unknown-label` can be used whenever the code has no concern for which container a referenced object belongs in. As mentioned in chapter 2, a goal is to minimize the burden of writing code with containers, and `unknown-label` allows for the omission of a definite label. Fields are an exception: `unknown-label` is not permitted as the label for a field. Since objects exist within containers, a field cannot be `unknown-label`, as that would allow for container violations, because all fields in an object must obey the object's container boundaries.

A special label exists solely for typing the expression `null`, namely `null-label`. It's not permitted to declare a symbol with this label. The label `local` represents the scope of a function invocation, and all references constrained by label `local` can only reference local variables. The label `var`(v) indicates that the container is a parameter or variable within the current typing environment.

The labels `local` and `var`(v) are dependent on their scope, and this raises the question of how parameters can be passed across scopes and retain the correct container label. For example, how can a reference to a local variable in the caller scope be passed to the callee? This issue is solved by the use of generic container labels. Functions that use parameters typed with generic labels are generic functions parameterized by the set of generic labels appearing in the parameter types. When calling such a function, the generic substitutions are made based on the container labels of the passed parameters. Parameters, return values and local references with generic container labels are indicated by the label `generic`(g). Note that function parameters can only be generic in their labels; no mechanism is provided for parameters to be generic in their class.

For reference fields, the labels `container-of-tuple` and `tuple` are used, with `container-of-tuple` meaning the container of the tuple that the field belongs to, and `tuple` meaning the tuple itself is the container.

The last container label to discuss is *path*. A path begins with a root container label that is also an object; this excludes `local` and `container-of-tuple`. Starting with the root, a series of field accesses can be taken to reach the final container. This allows you to declare a reference as contained within a series of nested fields.

In all, there are 5 components to a type, with each playing a role in the correct enforcement of containment. The type rules are defined in section 3.4, but first comes the definition the abstract language.

$$
\begin{array}{ll}
\mathit{ClassContainment} ::= \texttt{self-cont} \mid \texttt{by-container} & cc \in \mathit{ClassContainment} \\
\mathit{ClassDefinition} ::= \langle \overline{f : T}, cc \rangle & \mathit{class\text{-}def} \in \mathit{ClassDefinition} \\
\mathit{FunctionSignature} ::= \langle \overline{p : T}, T_{ret} \rangle & S \in \mathit{FunctionSignature} \\
\mathit{FunctionDefinition} ::= \langle S, s \rangle & \\
\mathit{LabelMap} ::= \{\theta_{callee} \rightarrow \theta_{caller}\} & \phi \in \mathit{LabelMap} \\
\mathit{Expression} ::= \texttt{var}(v) & e \in \mathit{Expression} \\
\qquad\qquad \mid \texttt{field}(e, f) & \\
\qquad\qquad \mid \texttt{call}(F, \phi, \overline{e}) & \\
\qquad\qquad \mid \texttt{init}(C, \phi, \overline{e}) & \\
\qquad\qquad \mid \texttt{null} & \\
\mathit{Statement} ::= \texttt{seq}(s_1, s_2) & s \in \mathit{Statement} \\
\qquad\qquad \mid \texttt{let}(T, v, e, s) & \\
\qquad\qquad \mid \texttt{assign-value}(T, e_{lhs}, e_{rhs}) & \\
\qquad\qquad \mid \texttt{assign-ref}(e_{lhs}, e_{rhs}) & \\
\qquad\qquad \mid \texttt{return}(e) &
\end{array}
$$

Figure 3.3: Abstract Language Definition

3.2 Language

The grammar of our abstract container language is in figure 3.3. Class definitions consist of a list of fields and their types as well as a *ClassContainment* setting, which determines if instances of this class are self-contained, meaning no reference can point to an external object, or contained by the container that the object is placed into. Function signatures contain a list of parameter types and the return type.

A *LabelMap* holds a mapping of container labels mapping labels in a callee scope back to labels in the caller's scope. Chapter 2 described an algorithm which could calculate this mapping, however we omit this complication in this abstract language. The specification of the *LabelMap* ϕ is part of the program.

There are five forms of expressions: variable access, field access, function call, tuple initialization and the null literal. Both `call()` and `init()` take a ϕ parameter. For functions, it maps parameter labels, and for tuple initialization, ϕ provides a mapping for the `container-of-tuple` label.

For statements we have `seq`() that simply allows multiple statements to exist in a function body, which by definition is a single statement. The statement `let` defines a new local variable and initializes it. The final parameter to `let` is a statement which runs with the new variable defined. Once this statement completes, the symbol is out of scope and

$\Gamma ::= \{C \rightarrow \mathit{ClassDefinition} \mid F \rightarrow \mathit{FunctionDefinition}\}$
$\Delta ::= \langle \{v \rightarrow T\}, T_{ret} \rangle$

With $\Delta = \langle \{\dots, v_i \mapsto T_i, \dots\}, T_{ret} \rangle$
$\Delta(v_i) = T_i$
$\Delta_{(\texttt{ret})} = T_{ret}$
$\Delta[v_{new} \mapsto T_{new}] = \langle \{\dots, v_i \mapsto T_i, \dots, v_{new} \mapsto T_{new}\}, T_{ret} \rangle$

Figure 3.4: Global and Local Typing Environments

not available to subsequent statements. There are two assignment statements to assign values and references. The distinction between these two statements isn't as important as it was in the demonstration language in chapter 2 because there is no indirect reference *kind* where you could assign a value through a reference. In this formalism, the left-hand side of `assign-ref` must always be a reference and the left-hand-side of `assign-value` must always be a value. Finally, the return statement returns a value from a function as one would expect.

3.3 Typing Environments

For typing environments, we use the symbol Γ for global definitions of classes and functions. For typing of local environments for statements and expressions, Δ is used. Both are defined in figure 3.4 as well as notation for deconstructing components from Δ and adding new symbols to Δ.

3.4 Type Rules

In this section, we define the typing rules for the abstract container language. To determine if a program is valid, all classes in Γ must satisfy $\Gamma \vdash$ `class-ok` (figure 3.13) and all functions must satisfy $\Gamma \vdash \langle S, s \rangle$ `func-ok` (figure 3.18). However, there are many components of classes and functions that need individual validation and we will begin by defining a set of functions needed to support the type rules.

```
// Find the set of container labels use by the parameters of F
// that need to be in the domain of phi.
distinctParameterLabels($\Gamma, F$) =
    let
        function(_, $\langle\overline{p : T}, T_{ret}\rangle$, _) = $\Gamma(F)$
    in
        distinctParameterLabels2($T_{ret} \circ \overline{T}$)

distinctParameterLabels2($T_1 \circ \overline{T}$) =
    let
        $distinct$ = distinctParameterLabels2($\overline{T}$)
        $\theta$ = requiredLabelMapping($T_{1(\texttt{label})}$)
    in
        if $\theta \neq$ unknown-label $\wedge$ $\theta \notin distinct$
            then $\theta \circ distinct$ else $distinct$

requiredLabelMapping($\theta$)
    case $\theta$ of
        var($v$) $\rightarrow$ var($v$)
        generic($g$) $\rightarrow$ generic($g$)
        step($\theta_{base}$, $f$) $\rightarrow$ requiredLabelMapping($\theta_{base}$)
        otherwise $\rightarrow$ unknown-label

// Take a container label relative to a scope and compose a
// container label relative to the callers scope based on phi.
mapLabel($\phi$, $\theta$) =
    case $\theta$ of
        step($base, f$) $\rightarrow$
            step(mapLabel($\phi, base$), $f$)
        otherwise $\rightarrow$
            $\phi$($\theta$)

// Take a type relative to the current and compose a type relative to
// the calling scope based
exportType($\phi$, $T$) = $\langle T_{(\texttt{kind})}, T_{(\texttt{class})},$ mapLabel$(\phi, T_{(\texttt{label})}), T_{(\texttt{mut})}, T_{(\texttt{mob})}\rangle$

// Construct a new type environment for the body of a function.
// The function's parameters populate the scope.
funcEnv($\Gamma, F$)
  let
    $\langle\overline{p : T_p}, T_{ret}\rangle = \Gamma(F)$
  in
    $\langle\overline{p \rightarrow T_p}, T_{ret}\rangle$
```

Figure 3.5: `call`() Related Typing Functions

```
expressionToLabel($\Gamma, \Delta, e$) =
    $\Gamma; \Delta \vdash e : T$
    if $T_{\texttt{(mob)}} \neq$ fixed then
        // The expression e does not evaluate to a fixed container
        unknown-label
    else
        case $e$ of
            var($v$) $\rightarrow$ var($v$)
            field($e_{obj}, f$) $\rightarrow$
                case expressionToLabel($\Gamma, \Delta, e_{obj}$) of
                    $path$ $\rightarrow$ step($path, f$)
                    otherwise $\rightarrow$ unknown-label
            otherwise $\rightarrow$ unknown
```

Figure 3.6: Field Access Related Typing Functions (1)

3.4.1 Type Rule Helper Functions

The first set of functions in figure 3.5 help with typing function calls. In order to determine if the ϕ label mapping parameter to `call`() is correct, we need to know what the domain of ϕ should be. The `distinctParameterLabels`() function builds the list of labels that must be in the domain of ϕ. This list includes all of the `generic` and `var` container labels appearing in the function parameters.

The function `mapLabel`() takes care of applying the ϕ mapping. Its main behavior is to look up $\phi(\theta_{callee})$, but when paths are present, it recursively processes the paths until it reaches the root, then the root is replaced according to ϕ and the path is reconstructed using the new root. The `exportType`() function is essentially a wrapper around `mapLabel`() to map the entire type tuple. All components other than the container label are directly mapped.

The function `funcEnv`() builds the initial Δ typing environment for the function containing the types of each parameter and T_{ret} for the return value.

The functions defined in figures 3.6 and 3.7 all relate to the typing of the field access expression. When accessing a field we immediately know the type of the field with respect to the tuple it is defined in, but type we need must be relative to the local scope and a translation is needed.

Fields are typed with container labels relative to either `tuple` or `container-of-tuple` and these must be converted to labels relative to the current scope. For example, using the field access notation of the demonstration language, in expression `a.b.c`, we begin with the tuple-relative type of `c`. If `c` is typed with a container label rooted in `container-of-tuple`, then the container label of `a.b.c` is the same as the container label of `a.b`, which can be

```
// Take a field's declared container label and compose a new label
// relative to the local scope
mapFieldLabel($\Gamma, \Delta, \theta_{cont}, \theta_{tuple}, \theta_{field}$) =
  case $\theta_{field}$ of
      container-of-tuple → $\theta_{cont}$
      tuple → $\theta_{tuple}$
      step($\theta_{base}$, $f$) → case mapFieldLabel($\Gamma, \Delta, \theta_{cont}, \theta_{tuple}, \theta_{base}$) of
          unknown-label → unknown-label
          $\theta'_{base}$ → step($\theta'_{base}$, $f$)

// Determine the type of a field access expression
fieldAccessType($\Gamma, \Delta, e_{obj}, f$)
  let
    $\Gamma; \Delta \vdash e_{obj} : T_{obj}$
    $T_{field}$ = $\Gamma(T_{\texttt{(class)}})(f)$
    $\theta_{tuple}$ = expressionToLabel($e_{obj}$)
    $\theta$ = mapFieldLabel($\Gamma, \Delta, T_{obj\texttt{(cont)}}, \theta_{tuple}, T_{field\texttt{(cont)}}$)

    $\delta$ = if $T_{field\texttt{(kind)}}$ == value then
            // When a value field is mutable, the expressions mutability
            // is inherited from the base object which might not be mutable
            if $T_{field\texttt{(mut)}}$ == mutable then
                $T_{obj\texttt{(mut)}}$
            else
                immutable
        else
            // Field is a reference
            if $T_{obj\texttt{(mut)}}$ == mutable $\wedge$ $T_{field\texttt{(mut)}}$ == mutable then
                mutable
            else
                read-only

    $\gamma$ = if $T_{field\texttt{(kind)}}$ == ref $\wedge$ $\theta$ == unknown-label then
            // If we can't determine a precise container then no updates
            // to references are allowed
            unfixed
        else if $T_{field\texttt{(mob)}}$ = movable then
            if $T_{obj\texttt{(mut)}}$ = mutable then
                movable
            else
                unfixed  // Object immutability overrides a movable field
        else
            // Field is fixed
            if $T_{obj\texttt{(mob)}}$ == fixed then
                fixed
            else
                // When the base expression is unstable then everything
                // following is also unstable
                unfixed
  in
    $\langle T_{field\texttt{(kind)}}, T_{field\texttt{(class)}}, \theta, \delta, \gamma \rangle$
```

Figure 3.7: Field Access Related Typing Functions (2)

determined by recursion.

Otherwise, if field `c` had the container label `tuple`, then the container is the object that the expression `a.b` evaluates to. The function `expressionToLabel`() shown in figure 3.6 is used to make the conversion from an expression to a container label. It builds a container label path from a root variable and a series of field-access steps. There are many expressions that can't be converted to a specific container label. In these cases, `unknown-label` will be returned. Container labels need to be stable, so all dependencies present in them must be immutable. For an expression to be converted, it must only contain `fixed` variables and fields.

The final two functions in figure 3.7 determine the type of a field access expression. There is special logic for the container label, mutability and mobility of the type. The function `fieldAccessType`() makes use of `expressionToLabel`() to determine θ_{tuple} which is the container label representing the object e_{obj} evaluates to. With θ_{tuple} computed, `mapFieldLabel`() does the appropriate substitutions to create a label for the accessed field which is relative to the symbols in the local scope.

The remainder of `fieldAccessType`() computes the mutability and mobility components of the type. For δ, value fields can have their mutability overridden by the object. For example, if a field is mutable but the object is immutable, then you should not be able to modify the field. References are treated similarly, but `read-only` is always used instead of `immutable`.

Figure 3.8 provides two examples of container label determination for field access expressions. First all the relevant information is gathered, then `mapFieldLabel`() selects the appropriate container label. In figure 3.9, the if-then-else logic of figure 3.7 is unraveled. The tables show the full set of possibilities for determination of δ and γ for a field access expression. The final column shows the the result that `fieldAccessType`() will return, based on the values in the proceeding columns.

For the mobility component, the determination of γ also takes into account the previously computed θ. The reason is that `unknown-label` is overloaded in meaning. When a reference parameter or local variable is declared with container label `unknown-label`, it indicates that it doesn't matter which container it references. This means that the type rule for reference assignments permits any right-hand-side container label to be assigned to `unknown-label`. However, in all other areas, `unknown-label` should be taken literally, as there is no knowledge available about what container the referred object is inside. To resolve this ambiguity, expressions typed with `unknown-label` that are not actually *don't-care* are also typed with $\gamma =$ `unfixed` to prevent assignments that would break the containment system. The expression `var`(v) is the only expression that can be typed with

```
entity class X {
    ref y1 : Y :: self;
    ref y2 : Y :: container;
}

ref x : X :: local =^ ...
ref y1 =^ x.y1;   // Example A
ref y2 =^ x.y2;   // Example B
```

Example A: Determine the container label for the expression `x.y1`

- $\theta_{cont} = \texttt{local}$ — Determined from the type of expression x
- $\theta_{tuple} = \texttt{var}(\texttt{x})$ — Determined by $\texttt{expressionToLabel}(\Gamma, \Delta, \texttt{x})$
- $\theta_{field} = \texttt{self}$ — From the declaration of the field y1
- $\texttt{mapFieldLabel}(\Gamma, \Delta, \texttt{local}, \texttt{var}(\texttt{x}), \texttt{self})$ returns $\texttt{var}(\texttt{x})$

Example B: Determine the container label for the expression `x.y2`

- $\theta_{cont} = \texttt{local}$ and $\theta_{tuple} = \texttt{var}(\texttt{x})$ as before
- $\theta_{field} = \texttt{container}$ — From the declaration of the field y2
- $\texttt{mapFieldLabel}(\Gamma, \Delta, \texttt{local}, \texttt{var}(\texttt{x}), \texttt{container})$ returns `local`

Figure 3.8: Container Labels for Field Access Expressions

Field Kind	Field δ	Base Expr. δ	Expression δ
`value`	`mutable`	θ	θ
`value`	`immutable`	any	`immutable`
`ref`	`mutable`	`mutable`	`mutable`
`ref`	`read-only`	any	`read-only`
`ref`	any	not `mutable`	`read-only`

Field Kind	Expr θ	Field γ	Base Expr. γ	Base Expr. δ	Expr. γ
`ref`	`unknown`	any	any	any	`unfixed`
any	any	`movable`	any	`mutable`	`movable`
any	any	`movable`	any	not `mutable`	`unfixed`
any	any	`fixed`	`fixed`	any	`fixed`
any	any	`fixed`	not `fixed`	any	`unfixed`

Figure 3.9: Mutability and Mobility for Field Access Expressions

both `movable` and `unknown-label`.

The remainder of the logic for γ determination is less subtle. An immutable object prevents assignments to a field which is `movable`, and `fixed` fields become `unfixed` if the object is not also `fixed`. This ensures that any attempt to convert this expression's type into a container label only succeeds when both the object and field are `fixed`.

3.4.2 Type Validation

$$\textsc{T-SP-Step}\quad \frac{\Gamma; \Delta; \langle C, \overline{f : T}\rangle \vdash \mathit{path} : C_{\mathit{base}} \qquad C_{\mathit{base}} \neq C \qquad T = \Gamma(C_{\mathit{base}})(f) \qquad T_{\texttt{(mobility)}} = \texttt{fixed}}{\Gamma; \Delta; \langle C, \overline{f : T}\rangle \vdash \texttt{step}(\mathit{path}, f) : T_{\texttt{(class)}}}$$

$$\textsc{T-SP-Self-Step}\quad \frac{\Gamma; \Delta; \langle C, \overline{f : T}\rangle \vdash \mathit{path} : C_{\mathit{base}} \qquad C_{\mathit{base}} = C \qquad f = f_i \in \overline{f : T} \qquad T_{i\texttt{(mobility)}} = \texttt{fixed}}{\Gamma; \Delta; \langle C, \overline{f : T}\rangle \vdash \texttt{step}(\mathit{path}, f) : T_{i\texttt{(class)}}}$$

$$\textsc{T-SP-Tuple}\quad \frac{C \in \Gamma}{\Gamma; \Delta; \langle C, \overline{f : T}\rangle \vdash \texttt{tuple} : C} \qquad\qquad \textsc{T-SP-Var}\quad \frac{T = \Delta(v) \qquad T_{\texttt{(mobility)}} = \texttt{fixed}}{\Gamma; \Delta; \langle \varnothing, \varnothing\rangle \vdash \texttt{var}(v) : T_{\texttt{(class)}}}$$

$$\textsc{T-SP-Parm}\quad \frac{v = v_i \in \overline{v : T} \qquad T_{i\texttt{(mobility)}} = \texttt{fixed} \qquad T_{i\texttt{(kind)}} = \texttt{ref}}{\Gamma; \varnothing; \langle \varnothing, \overline{v : T}\rangle \vdash \texttt{var}(v) : T_{i\texttt{(class)}}}$$

Figure 3.10: Path Validation

With our helper functions defined, we can begin defining the type rules, beginning with container label path validation. In figure 3.10, the judgment $\Gamma; \Delta; \langle C, \overline{f : T}\rangle \vdash \texttt{step}(\mathit{path}, f) : T_{\texttt{(class)}}$ determines if a container label path is valid. Paths are distinguished from other container labels in that they refer to containers that must be objects and type as a class instance. This isn't true for container labels `local`, `container-of-tuple` and `generic`(g).

The class name C indicates that the path exists in a class declaration context in the container label of a field. A field's type must only reference preceding fields so that dependencies can be initialized beforehand, and the specification of C enables this logic. Otherwise, $\varnothing$ is used in place of C when not in a class declaration context.

The $\overline{v : T}$ part of the judgment is also for managing the order of dependencies. Both parameter declarations and field declarations populate a list of previous declarations. When local variables are typed, $\varnothing$ is used and variables are found in Δ by the rule T-SP-Var. When field and parameter declarations are typed, Δ will be empty.

The critical feature that all four path typing rules share is $T_{\texttt{(mobility)}} = \texttt{fixed}$. This ensures that all steps along a path are stable and that a container label will continue to specify the same container throughout its lifetime. Parameters have an additional restriction that any dependency must be a reference parameter. Since value parameters are passed by value, a copy is made when the function is invoked. Therefore, no other passed parameter could reference this fresh object.

V-Field-Label

$$\frac{\mathit{kind} = \texttt{ref}}{\Gamma; \langle C, \overline{f : T} \rangle; \mathit{kind} \vdash \texttt{container-of-tuple}\ \ \texttt{label-ok-f}}$$

V-Field-Label-Path

$$\frac{\Gamma; \varnothing; \langle C, \overline{f : T} \rangle \vdash \texttt{step}(\mathit{path}, f) : C}{\Gamma; \langle C, \overline{f : T} \rangle; \mathit{kind} \vdash \texttt{step}(\mathit{path}, f)\ \ \texttt{label-ok-f}}$$

V-Parameter-Label-Local

$$\frac{\theta = \texttt{local} \qquad \mathit{kind} = \texttt{value}}{\Gamma; \overline{v : T}; \mathit{kind} \vdash \theta\ \ \texttt{label-ok-p}}$$

V-Parameter-Label

$$\frac{\theta \in \{\texttt{unknown-label, generic(\_)}\} \qquad \mathit{kind} = \texttt{ref}}{\Gamma; \overline{v : T}; \mathit{kind} \vdash \theta\ \ \texttt{label-ok-p}}$$

V-Parameter-Label-Path

$$\frac{\Gamma; \Delta; \langle \varnothing, \overline{v : T} \rangle \vdash \texttt{step}(\mathit{path}, f) : C \qquad \mathit{kind} = \texttt{ref}}{\Gamma; \overline{v : T}; \mathit{kind} \vdash \texttt{step}(\mathit{path}, f)\ \ \texttt{label-ok-p}}$$

V-Local-Label-Local

$$\frac{\theta = \texttt{local}}{\Gamma; \Delta; \mathit{kind} \vdash \theta\ \ \texttt{label-ok-l}}$$

V-Local-Label

$$\frac{\theta \in \{\texttt{unknown-label, generic(\_)}\} \qquad \mathit{kind} = \texttt{ref}}{\Gamma; \Delta; \mathit{kind} \vdash \theta\ \ \texttt{label-ok-l}}$$

V-Local-Label-Path

$$\frac{\Gamma; \Delta; \langle \varnothing, \varnothing \rangle \vdash \texttt{step}(\mathit{path}, f) : C \qquad \mathit{kind} = \texttt{ref}}{\Gamma; \Delta; \mathit{kind} \vdash \texttt{step}(\mathit{path}, f)\ \ \texttt{label-ok-l}}$$

Figure 3.11: Container Label Validation

The remainder of the container label validation rules are in figure 3.11. Here, there are different judgments for fields, parameters and local variables. Different rules are applied for these contexts. For example, a field is not permitted to be labeled as `unknown-label`. Only `container-of-tuple` and a path are valid for fields. This is because objects are placed in containers and a reference field can never be more permissive than its object.

For typing parameter labels, there are restrictions where parameters can only depend on parameters declared before them. The label `unknown-label` is permitted, so function parameters can be declared without needing to specify a generic container label when there is no need to know the container. Reference parameters are not allowed to be contained within value parameters. This is because the pass-by-value semantics make a fresh copy of value parameters, so no passed reference parameter could refer to the fresh object.

The typing of values is consistent in that their container is the scope they are declared in, meaning `tuple` for fields and `local` for parameters and local variables. Other labels are only used for references.

$$\textsc{V-Field-Type}\quad \frac{\Gamma \vdash T\ \texttt{type-ok-common} \qquad \Gamma; T_{(\texttt{kind})} \vdash T_{(\texttt{label})}\ \texttt{label-ok-f}}{\Gamma; \langle C, \overline{f : T}\rangle \vdash T\ \texttt{type-ok-f}}$$

$$\textsc{V-Parameter-Type}\quad \frac{\Gamma \vdash T\ \texttt{type-ok-common} \qquad \Gamma; \Delta; T_{(\texttt{kind})} \vdash T_{(\texttt{label})}\ \texttt{label-ok-p}}{\Gamma; \overline{v : T} \vdash T\ \texttt{type-ok-p}}$$

$$\textsc{V-Local-Type}\quad \frac{\Gamma \vdash T\ \texttt{type-ok-common} \qquad \Gamma; \Delta; T_{(\texttt{kind})} \vdash T_{(\texttt{label})}\ \texttt{label-ok-l}}{\Gamma; \Delta \vdash T\ \texttt{type-ok-l}}$$

$$\textsc{V-Common-Type}\quad \frac{\Gamma \vdash T_{(\texttt{class})}\ \texttt{class-ok} \qquad T_{(\texttt{kind})} = \texttt{ref} \vee (T_{(\texttt{class})})_{(\texttt{containment})} = \texttt{self-cont}}{\Gamma \vdash T\ \texttt{type-ok-common}}$$

Figure 3.12: Type Validation

The verification of types is also broken into separate judgments for the three contexts that symbols can appear in. The rules defined in figure 3.12 are straightforward as the complexities are all handled in the `label-ok` judgments. The rule V-Common-Type checks that the class is well defined and that values can only be declared for self-contained classes. The reason for the self-contained restriction is that value types have pass-by-value and assign-by-value semantics and non-self-contained objects can't be copied.

$$\textsc{V-Fields}\quad \frac{\begin{array}{c} f_1 : T_1 = \texttt{head}(\overline{f : T}) \qquad \Gamma; \langle C, \overline{f_{dep} : T_{dep}} \rangle \vdash T_1 \ \texttt{type-ok-f} \\ \Gamma; \langle C, (f_1 : T_1) \circ \overline{f_{dep} : T_{dep}} \rangle \vdash \texttt{tail}(\overline{f : T}) \ \texttt{fields-ok} \end{array}}{\Gamma; \langle C, \overline{f_{dep} : T_{dep}} \rangle \vdash \overline{f : T} \ \texttt{fields-ok}}$$

$$\textsc{V-Class}\quad \frac{\Gamma; \langle C, \varnothing \rangle \vdash \overline{f : T} \ \texttt{fields-ok}}{\Gamma \vdash C \ \texttt{class-ok}}$$

Figure 3.13: Class Validation

To validate a class, the judgment $\Gamma \vdash C$ `class-ok` is defined in figure 3.13. Each field must satisfy `type-ok-f` where the context consists of the fields declared before it.

3.4.3 Language Typing

$$\textsc{V-Initializer-Label-Map}\quad \frac{\Gamma; \Delta_{caller} \vdash \phi(\texttt{container-of-tuple}) \ \texttt{label-ok-l}}{\Gamma; \Delta_{caller} \vdash \langle \texttt{tuple}, \phi \rangle \ \texttt{lmap-ok}}$$

$$\textsc{V-Function-Label-Map}\quad \frac{\begin{array}{c} \Delta = \texttt{funcEnv}(\Gamma, F) \qquad \forall \theta \in \texttt{distinctParameterLabels}(\Gamma, F).\ \theta \in \texttt{dom}(\phi) \\ \forall \theta \in \texttt{dom}(\phi).\ \Gamma; \Delta \vdash \theta \ \texttt{label-ok-l} \qquad \forall \theta \in \texttt{range}(\phi).\ \Gamma; \Delta_{caller} \vdash \theta \ \texttt{label-ok-l} \end{array}}{\Gamma; \Delta_{caller} \vdash \langle F, \phi \rangle \ \texttt{lmap-ok}}$$

Figure 3.14: Label Map Validation

Label maps play an important role in managing containers. Before a function call can be type checked, there needs to be a way to compare labels in the caller's scope with labels in the callee. The ϕ mapping takes labels as they were declared in the function and maps them back to the caller's symbol space. For example, if the function contains a reference parameter with label $\texttt{generic}(g)$, then the mapping must contain that label in its domain. The function `distinctParameterLabels()` establishes the set of labels that must appear in the domain of ϕ.

The `lmap-ok` judgment verifies that the domain of ϕ is valid with respect to function F, and then validates each label in the domain in the scope of the function and each label in the range of ϕ in the caller's scope.

Value-Initializable

$$\frac{T_{lhs(\texttt{kind})} = \texttt{value} \qquad T_{rhs(\texttt{class})} = T_{lhs(\texttt{class})}}{\Gamma; \Delta \vdash T_{lhs} \widetilde{\succ} T_{rhs}}$$

Ref-Initializable

$$\frac{\begin{array}{c} T_{lhs} = \langle \texttt{ref}, C, \theta_{lhs}, \delta_{lhs}, \_ \rangle \qquad T_{rhs} = \langle \_, C, \theta_{rhs}, \delta_{rhs}, \_ \rangle \\ \delta_{lhs} = \delta_{rhs} \lor \delta_{lhs} = \texttt{read-only} \qquad \Gamma; \Delta \vdash \theta_{lhs} \ \texttt{l-match} \ \theta_{rhs} \end{array}}{\Gamma; \Delta \vdash T_{lhs} \widetilde{\succ} T_{rhs}}$$

Value-Assignable

$$\frac{T_{lhs} = \langle \texttt{value}, C, \_, \_, \texttt{movable} \rangle \qquad T_{rhs(\texttt{class})} = C}{\Gamma; \Delta \vdash T_{lhs} \succ T_{rhs}}$$

Ref-Assignable

$$\frac{\begin{array}{c} T_{lhs} = \langle \texttt{ref}, C, \theta_{lhs}, \delta_{lhs}, \texttt{movable} \rangle \qquad T_{rhs} = \langle \_, C, \theta_{rhs}, \delta_{rhs}, \_ \rangle \\ \delta_{lhs} = \delta_{rhs} \lor \delta_{lhs} = \texttt{read-only} \qquad \Gamma; \Delta \vdash \theta_{lhs} \ \texttt{l-match} \ \theta_{rhs} \end{array}}{\Gamma; \Delta \vdash T_{lhs} \widehat{\succ} T_{rhs}}$$

Container-Label-Match

$$\frac{\theta_{lhs} = \theta_{rhs} \lor \theta_{lhs} = \texttt{unknown-label} \lor \theta_{rhs} = \texttt{null-label}}{\Gamma; \Delta \vdash \theta_{lhs} \ \texttt{l-match} \ \theta_{rhs}}$$

Figure 3.15: Assignment and Initialization

Next, in figure 3.15, we define a set of rules to determine which types are assignable to which destination types. The operator $\widetilde{\succ}$ is used for initialization where the mobility of the type is ignored. For regular assignments, $\succ$ is used for value assignment, and $\widehat{\succ}$ is used for reference assignment. Value assignment has fewer conditions than reference assignment, because a copy of the right-hand side is made and the mutability of the copy is independent of the original. The left-hand side type must be a `value` and the classes must match in order to be value-assignable. No subtyping is implemented in this system.

For reference assignment, the mutability of the left-hand side must match the right-hand side, unless the left-hand side is typed as `read-only`, which is universally compatible.

The container labels must also match using the `l-match` judgment.

The normal case for labels to be compatible by `l-match` is if the left-hand side has the same label as the right-hand side. There are two special cases. First, if the left-hand side has label `unknown-label`, then any right-hand side label is permitted. Second, if the right-hand side is `null-label` (can only occur with the `null` expression) then the assignment is permitted.

T-Field-Access

$$\frac{\begin{array}{c} \Gamma; \Delta \vdash e : T_e \\ T_{e(\texttt{class})} = C \qquad \Gamma(C) = \langle \overline{f : T}, \_ \rangle \qquad f \in \overline{f} \qquad T = \texttt{fieldAccessType}(\Gamma, \Delta, e, f) \end{array}}{\Gamma; \Delta \vdash \texttt{field}(e, f) : T}$$

T-Call-Function

$$\frac{\begin{array}{c} \Gamma(F) = \langle \overline{p : T_p}, T_{ret} \rangle \qquad \Gamma; \Delta \vdash \langle F, \phi \rangle \;\; \texttt{lmap-ok} \\ \Delta \vdash F \;\; \texttt{sig-ok} \qquad \overline{\Gamma; \Delta \vdash e : T_e} \qquad \overline{\Delta \vdash \texttt{exportType}(\phi, T_p) \mathrel{\widetilde{\succ}} T_e} \qquad \overline{g \in T_p \cup T_{ret}} \end{array}}{\Gamma; \Delta \vdash \texttt{call}(F, \phi, \overline{e}) : \texttt{exportType}(\phi, T_{ret})}$$

T-Init

$$\frac{\begin{array}{c} \Gamma \vdash C \;\; \texttt{class-ok} \qquad \Gamma; \Delta \vdash \langle \texttt{tuple}, \phi \rangle \;\; \texttt{lmap-ok} \qquad \Gamma(C) = \langle \overline{f : T_f}, \_ \rangle \qquad \overline{\Gamma; \Delta \vdash e : T_e} \\ \theta = \texttt{mapLabel}(\phi, \texttt{container-of-tuple}) \qquad \overline{\Gamma; \Delta \vdash \texttt{exportType}(\phi, T_f) \mathrel{\widetilde{\succ}} T_e} \end{array}}{\Gamma; \Delta \vdash \texttt{init}(C, \phi, \overline{e}) : \langle \texttt{ref}, C, \theta, \texttt{mutable}, \texttt{fixed} \rangle}$$

T-Variable

$$\frac{\Delta(v) = T}{\Gamma; \Delta \vdash \texttt{var}(v) : T}$$

T-Null

$$\frac{\Gamma \vdash C \;\; \texttt{class-ok}}{\Gamma; \Delta \vdash \texttt{null} : \langle \texttt{ref}, C, \texttt{null-cont}, \texttt{immutable}, \texttt{unfixed} \rangle}$$

Figure 3.16: Expression Typing

With the building block rules defined, we can now type the expressions and statements of the language. Figure 3.16 contains the type rules for expressions. For typing field accesses, most of the work is done in the function `fieldAccessType()` which was described in section 3.4.1. Otherwise, the base expression must type correctly as T_e and field f must be defined in the class of T_e.

In the rule T-Call-Function, we see how label maps are used in practice. The `exportType()` function (figure 3.5) applies the map ϕ to the parameter types and return type of the `call()` to map them into the container label space of the caller. Once they are mapped, the type of each passed parameter is checked using the $\widetilde{\succ}$ relation. Finally, the return type is also mapped and becomes the type of the `call`() expression.

The rule T-Init behaves in a very similar way to a function call. The semantics of `init`() is simply to allocate a new tuple in the heap and initialize each of the fields using the passed parameters. Even though there is no function code to run in a new scope, the new tuple is a scope with container labels relative to the tuple. Each parameter's type is checked against the respective field's type after it has been mapped into the space of the caller using ϕ. Here, ϕ is much simpler than `call`() and only `container-of-tuple` is in its domain.

The rules for `var`(v) and `null` are straightforward. The C in T-Null is a free variable.

T-Let

$$\frac{\Delta \vdash e : T_{init} \qquad \Delta \vdash T \widetilde{\succ} T_{init} \qquad \begin{array}{c}\Gamma \vdash T \ \texttt{ok-l} \\ \Gamma; \Delta[\texttt{vars} \mapsto \Delta_{(\texttt{vars})} \cup \langle v, T \rangle] \vdash s : \texttt{Void}\end{array}}{\Gamma; \Delta \vdash \texttt{let}(T, v, e, s) : \texttt{Void}}$$

T-Statement-Sequence

$$\frac{\Gamma; \Delta \vdash s_1 : \texttt{Void} \qquad \Gamma; \Delta \vdash s_2 : T}{\Gamma; \Delta \vdash \texttt{seq}(s_1, s_2) : T}$$

T-Assign-Value

$$\frac{\Gamma; \Delta \vdash e_{rhs} : T_{rhs} \qquad \begin{array}{c}\Gamma; \Delta \vdash e_{lhs} : T_{lhs} \\ T_{lhs} = \langle \texttt{value}, \_, \_, \_, \texttt{movable} \rangle\end{array} \qquad \Delta \vdash T_{lhs} \succ T_{rhs}}{\Gamma; \Delta \vdash \texttt{assign-value}(T_{lhs}, e_{lhs}, e_{rhs}) : \texttt{Void}}$$

T-Assign-Reference

$$\frac{\Gamma; \Delta \vdash e_{rhs} : T_{rhs} \qquad \begin{array}{c}\Gamma; \Delta \vdash e_{lhs} : T_{lhs} \\ T_{lhs} = \langle \texttt{ref}, \_, \_, \_, \texttt{movable} \rangle\end{array} \qquad \Delta \vdash T_{lhs} \widehat{\succ} T_{rhs}}{\Gamma; \Delta \vdash \texttt{assign-ref}(e_{lhs}, e_{rhs}) : \texttt{Void}}$$

T-Return

$$\frac{\Delta_{(\texttt{ret})} = T_{ret} \qquad \Delta \vdash e : T_e \qquad \Delta \vdash T_{ret} \widetilde{\succ} T_e}{\Gamma; \Delta \vdash \texttt{return}(e) : T_e}$$

Figure 3.17: Statement Typing

The rules for statements are defined in figure 3.17. The first rule T-Let types statement

s with variable $v : T$ added to its environment. For the assignment statements, the left-hand-side expression must be `movable` and the details of what types can be assigned are specified in figure 3.15. The last statement is `return` where the return expression $e : T$ must be initialization-assignable to the return type.

V-Parameters

$$\frac{\begin{array}{c} v_1 : T_1 = \texttt{head}(\overline{p : T}) \\ \Gamma; \overline{p_{dep} : T_{dep}} \vdash T_1 \;\; \texttt{type-ok-p} \qquad \Gamma; (p_1 : T_1) \circ \overline{p_{dep} : T_{dep}} \vdash \texttt{tail}(\overline{p : T}) \;\; \texttt{parms-ok} \end{array}}{\Gamma; \overline{p_{dep} : T_{dep}} \vdash \overline{p : T} \;\; \texttt{parms-ok}}$$

V-Signature

$$\frac{\begin{array}{c} S = \langle \overline{p : T_p}, T_{ret} \rangle \\ \Gamma; \varnothing \vdash \overline{p : T} \;\; \texttt{parms-ok} \qquad \Gamma \vdash T_{ret} \;\; \texttt{type-ok-p} \qquad T_{ret(\texttt{mob})} = \texttt{unfixed} \end{array}}{\Gamma \vdash S \;\; \texttt{sig-ok}}$$

V-Function

$$\frac{S = \langle \overline{p : T_p}, T_{ret} \rangle \qquad \Gamma \vdash S \;\; \texttt{sig-ok} \qquad \Delta = \texttt{funcEnv}(\Gamma, F) \qquad \Gamma; \Delta \vdash s : T_{ret}}{\Gamma \vdash \langle S, s \rangle \;\; \texttt{func-ok}}$$

Figure 3.18: Function Validation

The final set of rules complete the typing of the container language and are defined in figure 3.18. These rules type function definitions, which are a pair of a function signature and a function body statement: $\langle S, s \rangle$. All parameters and the return value must satisfy `type-ok-p`. The parameter labels can only be dependent on parameters that come before and the return value must have the additional constraint that its mobility is `unfixed`

Chapter 4

Operational Semantics

In this chapter, we define a small-step operational semantics for the language defined in chapter 3. We follow the methodology of structural operational semantics developed by Gordon Plotkin [17]. More directly, the style of our semantics were inspired by the MJ system by G.M. Bierman et al. [4].

4.1 Overview

The semantics of the language operate within a configuration which contains the program heap, stack and all other constructs needed to evaluate a program. This is illustrated in figure 4.1. The semantic rules relate a configuration to its subsequent configuration $\mathcal{C} \rightarrow \mathcal{C}'$.

The heap consists of references, boxes and tuples stored at abstract locations and the program is expressed by frames. A frame encodes an operation to be evaluated by the abstract machine. Frames include all of the expressions and statements of the language, but in addition, there are extra frames to support complex operations, which need to be decomposed into multiple scalar steps.

The configuration holds a current frame, which represents the next operation to be evaluated. In addition, the configuration contains a stack of stacks. The outer stack holds local configurations, which represent the state for a function invocation. Within a local configuration, there is a stack of frames, which we call the frame-stack. During the evaluation of a function, frames will be pushed and popped from the frame-stack as needed.

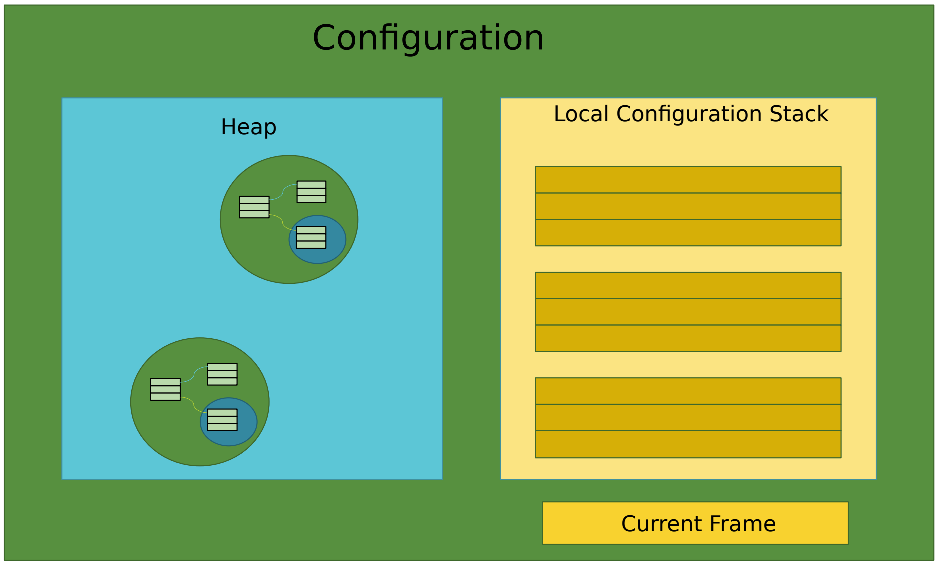


Figure 4.1: Configuration Overview

There are many definitions needed before we can present the semantic rules. In the next section, we will formally define the structure of the heap.

4.2 Heap

The heap structure shown in figure 4.2 is a mapping from an abstract location to a heap construct. There are three different constructs in the heap: tuples, boxes and references. We distinguish heap locations by which construct they refer to. A heap location is either a *TupleLocation* or a *SymbolLocation*, with *SymbolLocation* further subdivided into a *BoxLocation* or a *RefLocation*. The mappings for tuple locations are immutable once a tuple's fields are fully initialized. Mappings for symbol locations are mutable.

A tuple construct itself is an immutable mapping from field names to *SymbolLocations*. Later, we will see that local variables are also an immutable mapping from symbols to *SymbolLocations*. All mutation of the fields of a tuple occurs by modifying the individual *Box* and *Ref* constructs.

The distinction between *Box* and *Ref* is that a *Box* maps to a value tuple (self-contained) which is assigned by value, meaning the source tuple must be copied and the

$$
\begin{array}{ll}
\mathit{Location} ::= \mathit{TupleLocation} \;\mid\; \mathit{SymbolLocation} & \ell \in \mathit{Location} \\
\mathit{SymbolLocation} ::= \mathit{BoxLocation} \;\mid\; \mathit{RefLocation} & \\
\mathit{Tuple} ::= \overline{\mathit{FieldName} \rightarrow \mathit{SymbolLocation}} & \\
\mathit{Box} ::= \mathit{TupleLocation} & \\
\mathit{Ref} ::= \mathit{TupleLocation} \;\mid\; \texttt{null-ref} & \\
\mathit{Heap} ::= \{\, \mathit{TupleLocation} \rightarrow \mathit{Tuple} & \mathcal{H} \in \mathit{Heap} \\
\qquad\quad \mid \mathit{BoxLocation} \rightarrow \mathit{Box} & \\
\qquad\quad \mid \mathit{RefLocation} \rightarrow \mathit{Ref} \,\} &
\end{array}
$$

Figure 4.2: Heap Structure. *TupleLocation*, *BoxLocation* and *RefLocation* are abstract addresses in the heap. The tuple construct is immutable once fully initialized, but boxes and references can be updated.

box's mapping is set to a fresh tuple. The *Ref* construct, on the other hand, may reference objects that are not self-contained and is also permitted to be `null-ref`. No copy is made when a reference is assigned.

4.3 Frames

Our language consists of statements and expressions, but in order to implement a small-step semantics for this language, we need to introduce a number of additional frames to the language. Frames are the unit of computation, and each step of the machine applies a rule to the current frame to produce the next state of the configuration.

In addition to the statement and expression frames, figure 4.3 introduces a number of new frames. Expressions evaluate to a *Result* frame, which is a pair of a heap location and the type of that heap location. The frame `void` is simply the result of executing a statement. The statement `let` introduces a symbol that is only in scope for the `let` body statement and `pop-local`(v) is responsible for removing the variable when it goes out of scope.

The remainder of the frames relate to initialization of symbols and copying tuples. The frame `init-symbols` is used to marshal parameters for a function call, with `init-symbol` handling individual parameters. Similarly, `init-fields` and `init-field` are responsible for initializing a new tuple's fields.

There are a number of rules associated with copying an object. Although this process is similar to initializing a new tuple, the addition of a copy environment is required to track

$$
\begin{array}{l}
\mathit{Result} ::= \langle \mathit{Location}, \mathit{Type} \rangle \\
\mathit{CopyResult} ::= \mathit{CopyEnv} \;\mid\; \langle \mathit{Result}, \mathit{CopyEnv} \rangle \\
\mathit{AnyResult} ::= \mathit{Result} \;\mid\; \mathit{CopyResult} \\
\mathit{CopyEnv} ::= \{ \mathit{Location} \rightarrow \mathit{Location} \} \\
\mathit{Frame} ::= \mathit{ClosedFrame} \;\mid\; \mathit{OpenFrame}
\end{array}
$$

$$
\begin{array}{rl}
\mathit{ClosedFrame} ::= & \mathit{Statement} \\
\mid & \mathit{Expression} \\
\mid & \mathit{AnyResult} \\
\mid & \texttt{void} \\
\mid & \texttt{pop-local}(\mathit{VarName}) \\
\mid & \texttt{init-symbols}(\ \overline{\langle \mathit{Name}, \mathit{Result} \rangle}\ ) \\
\mid & \texttt{init-symbol}(\ \mathit{Name}, \mathit{Result}\ ) \\
\mid & \texttt{init-fields}\ (\mathit{Result}, \overline{\langle \mathit{Name}, \mathit{Result} \rangle}\ ) \\
\mid & \texttt{init-field}\ (\mathit{Result}, \mathit{Name}, \mathit{Result}\ ) \\
\mid & \texttt{assign-copied}(\ \mathit{Result}, \mathit{Result}\ ) \\
\mid & \texttt{copy-tuple}(\ \mathit{Type}, \mathit{Result}\ ) \\
\mid & \texttt{copy-tuple2}(\ \mathit{Type}, \mathit{Result}, \mathit{CopyEnv}\ ) \\
\mid & \texttt{copy-fields}(\ \mathit{Result}, \overline{\langle \mathit{Name}, \mathit{Result} \rangle}, \mathit{CopyEnv}\ ) \\
\mid & \texttt{copy-init}(\ \mathit{Result}, \text{Name}, \langle \mathit{Result}, \mathit{CopyEnv} \rangle\ ) \\
\mid & \texttt{discard-copy-env}(\ \langle \mathit{Result}, \mathit{CopyEnv} \rangle\ )
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{R} \in \mathit{Result} & \mathcal{R}_{\texttt{(loc)}} = \ell \text{ where } \mathcal{R} = \langle \ell, T \rangle \\
\mathcal{R}^{*} \in \mathit{AnyResult} & \\
\mathcal{A} \in \mathit{CopyEnv} & \\
\mathcal{F} \in \mathit{Frame} & \\
\mathcal{CF} \in \mathit{ClosedFrame} &
\end{array}
$$

Figure 4.3: Closed Frames

$$
\begin{array}{rll}
\mathit{OpenFrame} & ::= & \texttt{let}(T,\ l,\ \Box,\ s) \\
 & | & \texttt{assign-value}(T,\ \Box,\ e) \\
 & | & \texttt{assign-value}(T,\ l,\ \Box) \\
 & | & \texttt{assign-ref}(\Box,\ e) \\
 & | & \texttt{assign-ref}(l,\ \Box) \\
 & | & \texttt{return}(\Box) \\
 & | & \texttt{field}(\Box,\ f) \\
 & | & \texttt{call}(F,\ \phi,\ \mathcal{R}_1 \ldots \mathcal{R}_{i-1}, \Box, e_{i+1} \ldots e_n) \\
 & | & \texttt{init}(C,\ \phi,\ \mathcal{R}_1 \ldots \mathcal{R}_{i-1}, \Box, e_{i+1} \ldots e_n) \\
 & | & \texttt{init-symbol}(v, \Box) \\
 & | & \texttt{init-field}\ (\mathcal{R}_{tuple}, f, \Box) \\
 & | & \texttt{copy-fields}(\mathcal{R}_{tuple}, \overline{\langle f, \mathcal{R} \rangle},\ \Box\ ) \\
 & | & \texttt{copy-init}(\mathcal{R}_{tuple}, f, \Box) \\
 & | & \texttt{discard-copy-env}(\Box)
\end{array}
$$

$\mathcal{OF} \in \mathit{OpenFrame}$

Figure 4.4: Open Frames

which tuples have already been copied. *CopyEnv* maps tuples contained within the source tuple to new tuples constructed within the destination tuple. When recursively processing the source tuple, a nested tuple could be aliased multiple times and cycles could also exist. The mapping ensures that each tuple in the source is only copied once, and when an alias is encountered, the destination reference is set to $\mathcal{A}(\ell_{\textit{src-tuple}})$.

The frame `copy-tuple` is the entry point into the copy environment. An empty *CopyEnv* is created and a `discard-copy-env` frame is pushed to the frame-stack to strip the *CopyEnv* from the resulting copied tuple. The frame `copy-tuple2` is then set as the current frame. Any recursive copies bypass `copy-tuple` and directly use the frame `copy-tuple2` so they can continue to use the same environment. As copies are made, the environment $\mathcal{A}$ accumulates the tuple mappings. The frames `copy-fields` and `copy-init` behave much like `init-fields` and `init-field` except that they contain the copy environment as an extra parameter. The last frame associated with copying tuples is `assign-copied`, which completes an `assign-value` operation after the copy has been done.

Complex statements and expression must be evaluated in multiple steps. To facilitate this, there are decomposition rules which will extract the next sub-expression from a complex frame and replace it with a hole which is written as $\Box$. A frame is *closed* if it has no

$$\begin{array}{ll}
\mathit{Variables} ::= \overline{\mathit{Name} \rightarrow \mathit{SymbolLocation}} & \mathcal{V} \in \mathit{Variables} \\
\mathit{FrameStack} ::= \overline{\mathit{Frame}} & \overline{\mathcal{F}} \in \mathit{FrameStack} \\
\mathit{ScopeID} ::= \mathbb{Z}^{+} & \mathcal{ID} \in \mathit{ScopeID} \\
\mathit{Context} ::= \texttt{initial} \mid \texttt{tuple} \mid \mathit{FunctionName} & \mathit{ctx} \in \mathit{Context} \\
\mathit{LocalConfig} ::= \langle \mathcal{V}, \overline{\mathcal{F}}, \langle \mathit{ctx}, \phi \rangle, \mathcal{ID} \rangle & \mathcal{L} \in \mathit{LocalConfig}
\end{array}$$

With $\mathcal{L} = \langle \mathcal{V}, \overline{\mathcal{F}}, \langle \mathit{ctx}, \phi \rangle, \mathcal{ID} \rangle$

$$\begin{array}{ll}
\mathcal{L}_{(\texttt{vars})} = \mathcal{V} & \mathcal{L}[\texttt{vars} \mapsto \mathcal{V}'] = \langle \mathcal{V}', \overline{\mathcal{F}}, \langle \mathit{ctx}, \phi \rangle, \mathcal{ID} \rangle \\
\mathcal{L}_{(\texttt{fstack})} = \overline{F} & \mathcal{L}[\texttt{decl}\ \ v \mapsto \ell] = \langle (v \mapsto \ell) \circ \mathcal{V}, \overline{\mathcal{F}}, \langle \mathit{ctx}, \phi \rangle, \mathcal{ID} \rangle \\
\mathcal{L}_{(\texttt{ctx,lmap})} = \langle \mathit{ctx}, \phi \rangle & \mathcal{L}[\texttt{fstack} \mapsto \overline{F}'] = \langle \mathcal{V}, \overline{\mathcal{F}}', \langle \mathit{ctx}, \phi \rangle, \mathcal{ID} \rangle \\
\mathcal{L}_{(\texttt{lmap})} = \phi & \mathcal{L}[\texttt{push}\ \ \mathcal{F}] = \langle \mathcal{V}, \mathcal{F} \circ \overline{\mathcal{F}}, \langle \mathit{ctx}, \phi \rangle, \mathcal{ID} \rangle \\
\mathcal{L}_{(\texttt{id})} = \mathcal{ID} & \mathcal{L}[\texttt{push}\ \ \overline{\mathcal{F}_{next}}] = \langle \mathcal{V}, \overline{\mathcal{F}_{next}} \cdot \overline{\mathcal{F}}, \langle \mathit{ctx}, \phi \rangle, \mathcal{ID} \rangle
\end{array}$$

$$\mathit{Config} ::= \begin{pmatrix} \mathit{Heap} \\ \overline{\mathit{LocalConfig}} \\ \mathit{ClosedFrame} \end{pmatrix} \qquad \mathcal{C} \in \mathit{Config} \qquad \mathcal{C} = \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \mathcal{CF} \end{pmatrix}$$

Figure 4.5: Configuration

holes and *open* if there is a hole. After a sub-expression has completed, the result will be used to fill the hole and execution can continue. Figure 4.4 lists each of the open frames. Note that some frames will contain lists of expressions that will progress through a series of nested evaluations and hole-filling before the complete set of results can be operated upon.

4.4 Configuration

The state of the abstract machine as well as its component parts are defined in figure 4.5. The heap, stack and current frame are the three components of the configuration, which is written as a vertical vector. We've previously defined the heap and closed frames, but the stack is new and we'll introduce it now. The configuration contains a stack of local configurations, with each local configuration representing a scope. When a function call is made, a new local configuration is created and pushed onto the stack. When tuples are initialized, a new local configuration is also created.

Within a local configuration, variables and a frame-stack are stored. Variables map local symbols and function parameter symbols to locations in the heap. The frame-stack contains frames that represent the remainder of the code that needs to run within a function. There are two common patterns of usage with the frame-stack. The first is when the current frame evaluates to a result with an open frame at the top of the stack. The open frame will have a hole (□) in it and the result will fill in the hole, making a closed frame which becomes the new current frame. The other main pattern is when the current frame is `void`, indicating that a statement has completed. In this case, a frame is popped off the frame-stack to become the new current frame.

With the stack of local configurations and a nested frame-stack within each local configuration, we have a stack of stacks. This decision was originally made to support an early return in the middle of a function, as is commonly supported in mainstream imperative languages. When a function returned early, the remaining inner frame-stack in the local configuration could be immediately discarded and control would pop directly back to the caller using the outer stack. However, other features required for a short-circuit return did not make it into this formalism so we are left with the complexity without the payoff. Despite this limitation, the stack of stacks complication was manageable and could be used in future research. Perhaps exceptions and break statements could be supported as well using this technique.

There are two more bookkeeping components of a local configuration. First is *LabelMap*, linking the container labels of the current scope with the caller's scope. Along with the mapping is the context for the mapping so that the stack typing rules can validate the map. The context in the normal case is the name of the current function. Otherwise, it is the context marked as `tuple`, which indicates a tuple initialization context, or `initial`, which is a special context which calls `main`() to bootstrap the machine.

The last component of the local configuration is a unique identifier for the scope. This identifier is used for configuration typing which will be explained in the next section.

Also in figure 4.5 are notations for extracting sub-components of a local configuration $\mathcal{L}$ and also notations for updating $\mathcal{L}$. These are used heavily in the rules for the operational semantics.

4.4.1 Configuration Typing Environment

In order to show soundness for the operational semantics, we need to type the configuration including all of our new frames. But before we address the typing, we need to introduce the configuration typing environment and basic definitions of physical types in contrast with the logical types that we have discussed so far. The new definitions are in figure 4.6.

$Container ::= TupleLocation \mid ScopeID \mid \texttt{unknown-cont} \mid \texttt{null-cont}$
$PhysicalType ::= \langle Class, Container, Mutability, Mobility \rangle$
$ContainerMap ::= \{ContainerLabel \rightarrow Container\}$
$Uninitialized ::= \{TupleLocation\}$
$ConfigTyping ::= \langle \{Location \rightarrow PhysicalType\}, \{ScopeID \mapsto ContainerMap\} \rangle$

$\Theta \in Container \qquad \mathcal{T} \in PhysicalType$
$\mathcal{M} \in ContainerMap \qquad \Sigma \in ConfigTyping$

With $\Sigma = \langle \textit{physical-types}, \textit{container-maps} \rangle$
$\Sigma(\ell) = \textit{physical-types}(\ell)$
$\Sigma_{(\texttt{cmap})}(\mathcal{ID}) = \textit{container-maps}(\mathcal{ID})$

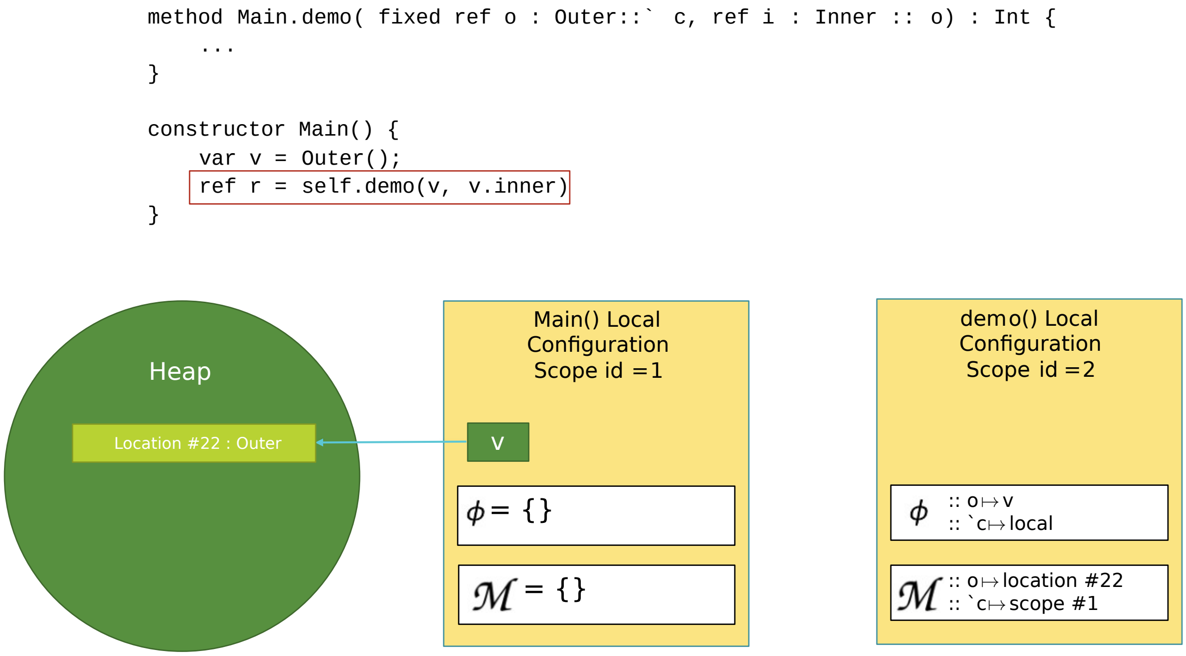
Figure 4.6: Configuration Typing Environment

Container labels are relative to their scope, and the same container can be represented with different container labels in different scopes. When a reference parameter is passed to or returned from a function, we need to know that the container indicated in both scopes is indeed the same physical construct in the configuration.

We used lowercase θ as a container label and the physical configuration construct will be represented with uppercase Θ. A physical container can be a tuple location in the heap or the unique identifier of a local configuration. Where container labels can only be interpreted relative to their scope, physical containers are global and can be compared independently of their origin.

Along with a physical container, we need a physical type $\mathcal{T}$, which is similar to our logical types T except the container label is replaced by a physical container. With these physical constructs defined, any typing we do will need to be able to convert logical types to physical types. We'll soon define the `ptype()` function in figure 4.8, which performs these conversions, but for now we introduce the mapping $\mathcal{M}$. We saw in chapter 3 the ϕ mapping which maps labels to labels in the calling scope. Here, $\mathcal{M}$ maps container labels to physical labels for a specific scope. Notation for accessing Σ is also presented in figure 4.6. An example is provided in figure 4.7, with two scopes shown, a caller and callee. Concrete mappings are shown to demonstrate how the container labels and physical containers are connected.

With these definitions in place, Σ is the typing environment for the configuration. Every location in the heap is mapped to a physical type. The environment Σ also contains a container-map for every local configuration, which allows container labels to be mapped



```
method Main.demo( fixed ref o : Outer::` c, ref i : Inner :: o) : Int {
    ...
}

constructor Main() {
    var v = Outer();
    ref r = self.demo(v, v.inner)
}
```

Figure 4.7: Container Mappings Example

```
// Build a PhysicalType for a type with respect to a configuration
ptype($\Sigma$, $\mathcal{H}$, $\mathcal{L}$, $T$) =
    $\langle T_{(\texttt{class})}$, labelToCont($\Sigma$, $\mathcal{H}$, $\mathcal{L}$, $T_{(\texttt{label})}$), $T_{(\texttt{mut})}$, $T_{(\texttt{mob})} \rangle$

// Determine the physical container for a container label within a
// local environment
labelToCont($\Sigma$, $\mathcal{H}$, $\mathcal{L}$, $\theta$) =
    case $\theta$ of
        unknown-label -> unknown-cont
        null-label -> null-cont
        variable($v$) -> locToCont($\mathcal{H}$,$\mathcal{L}_{(\texttt{vars})}$($v$))
        step(base, f)) ->
            let $\ell_{tuple}$ = labelToCont($\Sigma$, $\mathcal{H}$, $\mathcal{L}$, base) in
                locToCont($\mathcal{H}$, $\mathcal{H}$($\ell_{tuple}$)(f))
        $\theta$ -> $\Sigma_{(\texttt{cmap})}$($\mathcal{L}_{(\texttt{id})}$)($\theta$)

// De-reference references and boxes
locToCont($\mathcal{H}$, l) =
  case l of
    TupleLocation -> l
    BoxLocation -> $\mathcal{H}$(l)
    RefLocation -> locToCont($\mathcal{H}$, $\mathcal{H}$(l))

// De-reference references and boxes
toTuple($\mathcal{H}$, l) =
  case l of
    TupleLocation -> l
    BoxLocation -> $\mathcal{H}$(l)
    RefLocation -> $\mathcal{H}$(l)
```

Figure 4.8: Operational Rules Support Functions (1)

to their physical container. Finally, we also introduce $\mathcal{U}$ which is a set of uninitialized tuples. Combined, $\Sigma; \mathcal{U}$ comprise the entire typing environment for a configuration. The operational semantics do not depend on Σ or $\mathcal{U}$ in any way; they exist solely to enable the type safety proofs.

4.5 Supporting Functions

Similar to chapter 3, there are a number of supporting functions needed to type the configuration. Two important functions are reused from chapter 3, namely `exportType`() and `mapLabel`(), defined in figure 3.5.

```
// Create an initial local configuration for tuple initialization
tupleLocalConfig($\phi$,$\mathcal{ID}$) = $\langle \{\}, \varnothing, \langle \texttt{tuple}, \phi \rangle, \mathcal{ID} \rangle$

// Create an initial local configuration for function F
// with the function body s placed on the frame-stack
funcLocalConfig($F, \phi, s^{\Delta}$) = $\langle \{\}, s^{\Delta}, \langle F, \phi \rangle, \texttt{uniqueID()} \rangle$

// Create a mapping for container labels to physical containers for
// a function body scope
labelMappings($\Sigma$, $\mathcal{H}$, $\mathcal{L}_{caller}$, $\phi$) =
    $\{\theta_i \mapsto \texttt{labelToCont}(\Sigma, \mathcal{H}, \mathcal{L}_{caller}, \phi(\theta_i)) \mid \theta_i \in \texttt{dom}(\phi)\} \cup \{\texttt{local} \mapsto \mathcal{L}_{\texttt{(id)}}\}$

// Create a mapping for container labels to physical containers for a tuple scope
tupleMappings($\Sigma$, $\mathcal{H}$, $\mathcal{L}_{caller}$, $\phi$, $\Theta_{tuple}$) =
    let $\theta_{cont}$ = $\phi$(container-of-tuple) in
        $\{\texttt{container-of-tuple} \mapsto \texttt{labelToCont}(\Sigma, \mathcal{H}, \mathcal{L}_{caller}, \theta_{cont}),\ \texttt{tuple} \mapsto \Theta_{tuple}\}$

// Take a type relative to the caller's scope current and compose a type
// relative to the current scope.
importType($T_{caller}, T_{local}$) = $\langle T_{caller\texttt{(kind)}}, T_{caller\texttt{(class)}}, T_{local\texttt{(label)}}, T_{caller\texttt{(mut)}}, T_{caller\texttt{(mob)}} \rangle$
```

Figure 4.9: Operational Rules Support Functions (2)

Figure 4.8 defines four functions. The first, `ptype`(), is used extensively; it converts a logical type T to a physical type $\mathcal{T}$. The components for class, mutability and mobility are directly copied and the container label is converted to a physical container using `labelToCont`(). The conversion is handled by cases. When the container is a variable, the physical container is simply the heap location of that variable. To obtain that address, the function `locToCont`() automatically follows references.

When converting a label of `step`$(base, f)$, the base is recursively processed, yielding a tuple location. From that tuple, the field f is accessed, resulting in the desired container. For all other container labels, for example `generic`(g), the physical container is derived from the mapping $\mathcal{M}$, which is associated with every local configuration.

The fourth function of figure 4.8, `toTuple`(), is used in the operational semantics to automatically follow references if present and return the location of the desired tuple.

Next, there are 5 additional functions in figure 4.5, which are all related to establishing new local configurations and label-to-container mappings. The function `tupleLocalConfig`() creates a new scope for initialization of a new tuple. The $\mathcal{ID}$ parameter in this case must be the heap location of the tuple. The `uniqueID`() function is abstract and generates a globally unique identifier. We rely on global uniqueness to permit a tuple heap location to also serve a double duty as the key for $\Sigma_{\texttt{(cmap)}}$.

When initializing a new scope for a function call, `funcLocalConfig`() builds the initial local configuration to execute the body of a function. The statement body s^{Δ} is placed on the frame-stack, the mapping ϕ is recorded and a unique identifier is assigned.

The `labelMappings`() function takes care of building the mapping $\mathcal{M}$ for a new function body scope. We have the label-to-label mapping ϕ, which maps container labels in the callee's scope to the caller's scope. We also have the mapping $\mathcal{M}_{caller}$, so we can take a callee-relative label, convert it into a caller-relative label and finally use the caller's $\mathcal{M}_{caller}$ to obtain the appropriate physical container. These steps are taken for all container labels in ϕ to generate $\mathcal{M}$.

4.6 Frame Typing

Each of our frames requires typing. All statements and expressions of the language are frames, and these frames retain their typing as in figures 3.16 and 3.17. However, for the new frames, we need new rules, which in turn need an extended typing environment. Statements and expressions are typed in context $\Gamma; \Delta$ and the extended frames are typed within $\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L}$. However, for the purposes of frame typing, we consider these to be the same judgment with the original statement and expression typing simply ignoring the extra configuration context.

Figures 4.10 and 4.11 define the typing rules for our new frames. Most rules are straightforward in that they the validate their component parts. A few rules make use of judgments that have not been defined yet. For reference, the judgment `t-match` is defined in figure 4.24 and `copy-map-ok` is in figure 4.25. Although a hole ($\square$) is not a frame, it needs to be typed, as it represents a future result. To do so, we add its type to Δ and use the rule T-Hole to type it.

The rule T-Location in particular does more than just type a location. It also establishes consistency between the location's logical and physical types. This type of consistency requirement is discussed in full in section 4.9, but for now it suffices to understand that frames are typed in the configuration typing context Σ and must be consistent with their corresponding physical types.

4.6.1 Program Type Annotations

All frames are typed prior to applying any semantic rules. As part of the typing process, we create an annotated version of the program where each frame is tagged with the typing

T-Hole

$$\frac{\Box : T \in \Delta}{\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \Box : T}$$

T-Void

$$\frac{}{\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \texttt{void} : \texttt{Void}}$$

T-Location

$$\frac{\Sigma(\ell) = \mathcal{T} = \texttt{ptype}(\Sigma; \mathcal{H}, \mathcal{L}, T) \qquad \Sigma; \mathcal{H}; \mathcal{L} \vdash T \ \texttt{t-match} \ \mathcal{T}}{\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \langle \ell, T \rangle : T}$$

T-CopyEnv

$$\frac{\Sigma \vdash \mathcal{A} \ \texttt{copy-map-ok}}{\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \mathcal{A} : \mathit{CopyEnv}}$$

T-Location-CopyEnv

$$\frac{\Sigma \vdash \mathcal{A} \ \texttt{copy-map-ok} \qquad \Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \langle \ell, T \rangle : T}{\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \langle \langle \ell, T \rangle, \mathcal{A} \rangle : \langle T, \mathit{CopyEnv} \rangle}$$

T-Pop-Local

$$\frac{v \in \Delta}{\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \texttt{pop-local}(v) : \texttt{Void}}$$

T-Copy-Discard-Env

$$\frac{\Sigma \vdash \mathcal{A} \ \texttt{copy-map-ok} \qquad \Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \langle \ell, T \rangle : T}{\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \texttt{copy-discard-env}(\langle \langle \ell, T \rangle, \mathcal{A} \rangle) : T}$$

T-Init-Symbols

$$\frac{\overline{T_i = \Delta(v_i)} \qquad \overline{\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \ell_{\mathit{init}_i} : T_{\mathit{init}_i}} \qquad \overline{(T_{i\texttt{(kind)}} = \texttt{ref} \wedge T_i \overset{\sim}{\succ} T_{\mathit{init}_i}) \vee (T_i = T_{\mathit{init}_i})}}{\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \texttt{init-symbols}(\overline{\langle v, \langle \ell_{\mathit{init}}, T_{\mathit{init}} \rangle \rangle}) : \texttt{Void}}$$

T-Init-Symbol

$$\frac{\begin{array}{c} T = \Delta(v) \qquad \Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \ell_{\mathit{init}} : T_{\mathit{init}} \\ (T_{\texttt{(kind)}} = \texttt{ref} \wedge T \overset{\sim}{\succ} T_{\mathit{init}}) \vee (T = T_{\mathit{init}}) \qquad T_{\texttt{(kind)}} = \texttt{ref} \vee \ell_{\mathit{init}} \neq \texttt{null} \end{array}}{\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \texttt{init-symbol}(v, \langle \ell_{\mathit{init}}, T_{\mathit{init}} \rangle) : \texttt{Void}}$$

T-Init-Fields

$$\frac{\begin{array}{c} C = T_{\mathit{tuple}\texttt{(class)}} \qquad \overline{T_{\mathit{field}_i} = \Gamma(C)(f_i)} \\ \overline{\Gamma; \Delta; \Sigma \vdash \ell_{\mathit{init}_i} : T_{\mathit{init}_i}} \qquad \overline{(T_{\mathit{field}_i\texttt{(kind)}} = \texttt{ref} \wedge T_{\mathit{field}_i} \overset{\sim}{\succ} T_{\mathit{init}_i}) \vee (T_{\mathit{field}_i} = T_{\mathit{init}_i})} \end{array}}{\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \texttt{init-fields}(\langle \ell_{\mathit{tuple}}, T_{\mathit{tuple}} \rangle, \overline{\langle f, \langle \ell_{\mathit{init}}, T_{\mathit{init}} \rangle \rangle}) : T_{\mathit{tuple}}}$$

T-Init-Field

$$\frac{\begin{array}{c} C = T_{\mathit{tuple}\texttt{(class)}} \qquad T_{\mathit{field}} = \Gamma(C)(f) \\ \Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \ell_{\mathit{init}} : T_{\mathit{init}} \qquad (T_{\mathit{field}\texttt{(kind)}} = \texttt{ref} \wedge T_{\mathit{field}} \overset{\sim}{\succ} T_{\mathit{init}}) \vee (T_{\mathit{field}} = T_{\mathit{init}}) \\ T_{\mathit{field}\texttt{(kind)}} = \texttt{ref} \vee \ell_{\mathit{init}} \neq \texttt{null} \end{array}}{\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \texttt{init-field}(\langle \ell_{\mathit{tuple}}, T_{\mathit{tuple}} \rangle, f, \langle \ell_{\mathit{init}}, T_{\mathit{init}} \rangle) : \texttt{Void}}$$

Figure 4.10: Frame Typing (1)

T-Copy-Tuple

$$\frac{\Gamma; \Delta; \Sigma \vdash \ell_{src} : T_{src} \qquad T_{dst} \succ T_{src}}{\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \texttt{copy-tuple}(T_{dst}, \langle \ell_{src}, T_{src} \rangle) : T_{dst}}$$

T-Copy-Tuple-2

$$\frac{\begin{array}{c}\Gamma; \Delta; \Sigma \vdash \ell_{src} : T_{src} \\ \Gamma \vdash T_{dst}\ \texttt{type-ok-common} \qquad T_{dst} \succ T_{src} \qquad \Sigma \vdash \mathcal{A}\ \texttt{copy-map-ok}\end{array}}{\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \texttt{copy-tuple2}(T_{dst}, \langle \ell_{src}, T_{src} \rangle, \mathcal{A}) : \langle T_{dst}, \mathit{CopyEnv} \rangle}$$

T-Copy-Fields

$$\frac{\dfrac{\Gamma; \Delta; \Sigma \vdash \ell_{tuple} : T_{tuple} \qquad \dfrac{C = T_{tuple\texttt{(class)}}}{T_{field_i} = \Gamma(C)(f_i)}}{(T_{\texttt{(kind)}} = \texttt{ref} \wedge T_{field_i} \overset{\sim}{\succ} T_{init_i}) \vee (T_{field_i} = T_{init_i})} \qquad \begin{array}{c}\overline{\Gamma; \Delta; \Sigma \vdash \ell_{init_i} : T_{init_i}} \\ \Sigma \vdash \mathcal{A}\ \texttt{copy-map-ok}\end{array}}{\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \texttt{copy-fields}(\langle \ell_{tuple}, T_{tuple} \rangle, \overline{\langle f, \langle \ell_{init}, T_{init} \rangle \rangle}, \mathcal{A}) : \mathit{CopyMap}}$$

T-copy-init

$$\frac{\begin{array}{c}C = T_{tuple\texttt{(class)}} \qquad T_{field} = \Gamma(C)(f) \qquad \Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \ell_{tuple} : T_{tuple} \\ \Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \ell_{init} : T_{init} \qquad (T_{\texttt{(kind)}} = \texttt{ref} \wedge T_{field} \overset{\sim}{\succ} T_{init}) \vee (T_{field} = T_{init}) \\ T_{\texttt{(kind)}} = \texttt{ref} \vee \ell_{init} \neq \texttt{null} \qquad \Sigma \vdash \mathcal{A}\ \texttt{copy-map-ok}\end{array}}{\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \texttt{copy-init}(\langle \ell_{tuple}, T_{tuple} \rangle, f, \langle \langle \ell_{init}, T_{init} \rangle, \mathcal{A} \rangle) : \mathit{CopyEnv}}$$

T-Assign-Copied

$$\frac{\begin{array}{c}\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \langle \ell_{lhs}, T_{lhs} \rangle : T_{lhs} \\ \Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \langle \ell_{rhs}, T_{rhs} \rangle : T_{rhs} \qquad T_{lhs\texttt{(mobility)}} = \texttt{movable} \qquad T_{rhs} = T_{lhs}\end{array}}{\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \texttt{assign-copied}(\langle \ell_{lhs}, T_{lhs} \rangle, \langle \ell_{rhs}, T_{rhs} \rangle) : \texttt{Void}}$$

Figure 4.11: Frame Typing (2)

environment that it is typed in. For example, if we have the judgment $\Gamma; \Delta \vdash e : T$, then in the annotated program, we will write e^{Δ}, making both e and Δ available to the conditions of the rules. By binding Δ to each frame, we enable further configuration typing judgments to be made.

4.7 Special Configurations

Initial-Config

$$\mathcal{C}_{initial} = \begin{pmatrix} \varnothing \\ \{\langle \varnothing, \varnothing, \langle \texttt{initial}, \varnothing \rangle, 0 \rangle\} \\ \texttt{call}(\texttt{"main"}, \varnothing, \varnothing) \\ \boxed{\Sigma = \varnothing} \end{pmatrix}$$

Terminal-Result

$$\begin{pmatrix} \mathcal{H} \\ \{\langle \varnothing, \varnothing, \langle \texttt{initial}, \varnothing \rangle, 0 \rangle\} \\ \langle \ell, T \rangle \end{pmatrix}$$

Terminal-Exception

$$\begin{pmatrix} \mathcal{H} \\ \overline{\mathcal{L}} \\ \texttt{NPE} \end{pmatrix}$$

Figure 4.12: Initial and Terminal Configurations

Figure 4.12 defines $\mathcal{C}_{initial}$ which is the initial configuration. All programs are expected to have a `main`() function defined. There are no variables in this special `initial` scope and the heap is empty. Also shown in a boxed region is the initial configuration typing environment. The boxing is intended to separate distinct concerns. Primarily, we are defining the semantics (unboxed), which do not require Σ. However, as the rules of the operational semantics are defined, we also show how Σ should be updated to reflect any new typing needed after a rule is applied.

Also in 4.12 are two terminal state patterns. Terminal-Result represents normal termination of a program with ℓ as the heap address containing the return value of `main`(). Similarly, Terminal-Exception represents termination of a program after an attempt to de-reference `null`. No further rules can be applied if the configuration matches one of these patterns.

4.8 Transition Rules

In this section, we define the rules of the operational semantics. The rules are written with the current configuration in the lower left, related to the subsequent configuration in the lower right ($\mathcal{C} \to \mathcal{C}'$). When relevant, the typing environment Σ is shown boxed as a fourth element of the configuration's vertical vector and information about uninitialized tuples $\mathcal{U}$ appears boxed to the right of the heap component $\mathcal{H}$. Again, this is typing information distinct from the operational semantics. Most rules operate with $\mathcal{U} = \varnothing$, and in these cases, we omit any mention of $\mathcal{U}$ in the rules. However, for initialization-related frames that do operate with uninitialized tuples, there will be a box describing how the rule impacts $\mathcal{U}$.

Above the line are the conditions required for the rule. Some conditions are simply definitions of symbols used to compose the next configuration. In general, most rules have their conditions formatted into two columns with the more significant ones in the left column and simple definitions on the right. This convention is broken in a few cases for better layout with lengthy conditions.

4.8.1 Machinery

First we look at three simple foundational rules in figure 4.13. First, rule Next-Frame runs after a statement rule completes and leaves its `void` result as the current frame. A closed-frame is popped from the frame-stack and set as the new current frame.

For processing expression results, the rule Next-Frame takes a result, pops an open frame from the frame-stack and replaces the □ in the open frame with the result. The now-closed frame is set as the new current-frame.

The last of our utility rules is Pop-Local which removes a local variable from the local configuration after a `let` statement has completed its execution.

4.8.2 Statements

The next set of rules are for statements and are defined in figure 4.14. The rule for `let` statements pushes $s^{\Delta'}$ and a `pop-local` frame which removes the variable v after the body of the `let` has completed. The current frame is set to `init-symbol` which defines and initializes the new variable.

$$\textsc{Next-Frame}\quad \frac{\mathcal{L}_{(\texttt{fstack})} = \mathcal{CF} \circ \overline{F}}{\begin{pmatrix}\mathcal{H}\\ \mathcal{L} \circ \overline{\mathcal{L}}\\ \texttt{void}\end{pmatrix} \rightarrow \begin{pmatrix}\mathcal{H}\\ \mathcal{L}[\texttt{fstack} \mapsto \overline{F}] \circ \overline{\mathcal{L}}\\ \mathcal{CF}\end{pmatrix}}$$

$$\textsc{Consume-Result}\quad \frac{\mathcal{L}_{(\texttt{fstack})} = \mathcal{OF} \circ \overline{F}}{\begin{pmatrix}\mathcal{H}\\ \mathcal{L} \circ \overline{\mathcal{L}}\\ \mathcal{R}^*\end{pmatrix} \rightarrow \begin{pmatrix}\mathcal{H}\\ \mathcal{L}[\texttt{fstack} \mapsto \overline{F}] \circ \overline{\mathcal{L}}\\ \mathcal{OF}[\square \mapsto r]\end{pmatrix}}$$

$$\textsc{Pop-Local}\quad \frac{\mathcal{V}' = \mathcal{L}_{(\texttt{vars})} \setminus \{v\}}{\begin{pmatrix}\mathcal{H}\\ \mathcal{L} \circ \overline{\mathcal{L}}\\ \texttt{pop-local}^{\triangle}(v)\end{pmatrix} \rightarrow \begin{pmatrix}\mathcal{H}\\ \mathcal{L}[\texttt{vars} \mapsto \mathcal{V}'] \circ \overline{\mathcal{L}}\\ \texttt{void}\end{pmatrix}}$$

Figure 4.13: Transition Rules — Machinary

The rule for `assign-value` uses `copy-tuple` to implement assign-by-value semantics. After the copy is complete, `assign-copied` consumes the result and completes the assignment by updating the heap with the left-hand side *Box* updated to reference the new tuple. For reference assignment, `assign-ref` doesn't have assign-by-value, semantics so it directly updates the heap to reference the right-hand side tuple.

For `return` statements there are two rules, one for values and one for references. When returning a value, `copy-tuple` is invoked, but it runs in the scope of the calling function and the result of the copy is directly consumed by the calling function. Note the use of the `exportType()` function, which converts the return type's container labels to be relative to the function caller's scope. Again, returning references is similar, but skips the `copy-tuple` steps.

4.8.3 Expressions

Rules for expressions appear in figure 4.15. For field access expressions, FIELD-ACCESS first normalizes ℓ_{tuple} using `toTuple`(), which automatically de-references ℓ_{tuple} if it is a reference. The field is then found in the heap by looking up symbol f in the tuple structure. The logic for `var`(v) is also simple. The local configuration holds the local variables in $\mathcal{L}_{(\texttt{vars})}$.

In the rule CALL-FUNCTION, there are a number of critical steps. Recall that one of the most important aspects of this system is the management of container labels when passing parameters across scopes. The first mapping ϕ is provided by the program, and it maps labels in the new called function body back to the caller's scope. The ϕ map is stored in the newly created local configuration $\mathcal{L}_{fn}$. The second map $\mathcal{M}$ is built using the

Let

$$\frac{\Delta' = \Delta[v \mapsto T]}{\begin{pmatrix}\mathcal{H}\\ \mathcal{L}\circ\overline{\mathcal{L}}\\ \texttt{let}^{\Delta}(T, v, \mathcal{R}_{init}, s)\end{pmatrix} \rightarrow \begin{pmatrix}\mathcal{H}\\ \mathcal{L}[\texttt{push}\ s^{\Delta'}\circ\texttt{pop-local}^{\Delta'}(v)]\circ\overline{\mathcal{L}}\\ \texttt{init-symbol}^{\Delta}(v,\mathcal{R}_{init})\end{pmatrix}}$$

ASSIGN-VALUE

$$\frac{\begin{array}{cc}\ell_{rhs'} = \texttt{toTuple}(\mathcal{H}, \ell_{rhs}) & \mathcal{H}(\ell_{rhs}) \neq \texttt{null-ref}\\ \mathcal{F}_{next} = \texttt{assign-copied}^{\Delta}(\langle \ell_{lhs}, T_{lhs}\rangle, \square) & \ell_{lhs} \in \mathit{BoxLocation}\end{array}}{\begin{pmatrix}\mathcal{H}\\ \mathcal{L}\circ\overline{\mathcal{L}}\\ \texttt{assign-value}^{\Delta}(\langle \ell_{lhs}, T_{lhs}\rangle, \langle \ell_{rhs}, T_{rhs}\rangle)\end{pmatrix} \rightarrow \begin{pmatrix}\mathcal{H}\\ \mathcal{L}[\texttt{push}\ \mathcal{F}_{next}]\circ\overline{\mathcal{L}}\\ \texttt{copy-tuple}^{\Delta}(T_{lhs}, \langle \ell'_{rhs}, T_{rhs}\rangle)\end{pmatrix}}$$

Assign-Copied

$$\frac{\begin{array}{c}\ell_{lhs} \in \mathit{BoxLocation}\\ \ell_{rhs} \in \mathit{TupleLocation}\end{array}}{\begin{pmatrix}\mathcal{H}\\ \mathcal{L}\circ\overline{\mathcal{L}}\\ \texttt{assign-copied}^{\Delta}(\langle \ell_{lhs}, T_{lhs}\rangle, \langle \ell_{rhs}, T_{rhs}\rangle)\end{pmatrix} \rightarrow \begin{pmatrix}\mathcal{H}[\ell_{lhs} \mapsto \ell_{rhs}]\\ \mathcal{L}\circ\overline{\mathcal{L}}\\ \texttt{void}\end{pmatrix}}$$

ASSIGN-REF

$$\frac{\begin{array}{cc}\ell'_{rhs} = \texttt{toTuple}(\mathcal{H}, \ell_{rhs}) & \ell_{lhs} \in \mathit{RefLocation}\end{array}}{\begin{pmatrix}\mathcal{H}\\ \mathcal{L}\circ\overline{\mathcal{L}}\\ \texttt{assign-ref}^{\Delta}(\langle \ell_{lhs}, T_{lhs}\rangle, \langle \ell_{rhs}, T_{rhs}\rangle)\end{pmatrix} \rightarrow \begin{pmatrix}\mathcal{H}[\ell_{lhs} \mapsto \ell'_{rhs}]\\ \mathcal{L}\circ\overline{\mathcal{L}}\\ \texttt{void}\end{pmatrix}}$$

Return-Value

$$\frac{\begin{array}{cc} & \mathcal{H}(\ell_{ret}) \neq \texttt{null-ref}\\ \ell'_{ret} = \texttt{toTuple}(\mathcal{H}, l) & T_{(\texttt{kind})} = \texttt{value}\end{array}}{\begin{pmatrix}\mathcal{H}\\ \mathcal{L}\circ\overline{\mathcal{L}}\\ \texttt{return}^{\Delta}(T, \langle \ell_{ret}, T_{ret}\rangle)\end{pmatrix} \rightarrow \begin{pmatrix}\mathcal{H}\\ \overline{\mathcal{L}}\\ \texttt{copy-tuple}^{\Delta}(T, \langle \ell'_{ret}, \texttt{exportType}(\mathcal{L}_{(\texttt{lmap})}, T_{ret})\rangle)\end{pmatrix}}$$

Return-Ref

$$\frac{\begin{array}{cc}\ell'_{ret} = \texttt{toTuple}(\mathcal{H}, \ell) & T_{(\texttt{kind})} = \texttt{ref}\end{array}}{\begin{pmatrix}\mathcal{H}\\ \mathcal{L}\circ\overline{\mathcal{L}}\\ \texttt{return}^{\Delta}(T, \langle \ell_{ret}, T_{ret}\rangle)\end{pmatrix} \rightarrow \begin{pmatrix}\mathcal{H}\\ \overline{\mathcal{L}}\\ \langle \ell'_{ret}, \texttt{exportType}(\mathcal{L}_{(\texttt{lmap})}, T_{ret})\rangle\end{pmatrix}}$$

Figure 4.14: Transition Rules — Statements

FIELD-ACCESS

$$\frac{\begin{array}{ll} \ell'_{tuple} = \texttt{toTuple}(\mathcal{H}, \ell_{tuple}) & \mathcal{H}(\ell_{tuple}) \neq \texttt{null-ref} \\ \ell_{field} = \ell'_{tuple}(f) & \Gamma; \Delta \vdash \texttt{field}(\dots) : T_{field} \end{array}}{\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{field}^{\Delta}(\langle \ell_{tuple}, T_{tuple} \rangle, f) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \langle \ell_{field}, T_{field} \rangle \end{pmatrix}}$$

VAR

$$\frac{\begin{array}{c} T = \Delta(v) \\ \ell = \mathcal{L}_{(\texttt{vars})}(v) \end{array}}{\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{var}^{\Delta}(v) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \langle \ell, T \rangle \end{pmatrix}}$$

CALL-FUNCTION

$$\frac{\begin{array}{ll} & \Gamma\{F\} = \texttt{function}(\_, S, s) \\ \mathcal{L}_{fn} = \texttt{funcLocalConfig}(F, \phi, s^{\Delta_{fn}}) & \mathrm{S} = \langle \overline{p : T}, \_ \rangle \\ & \overline{T'_{init} = \texttt{importType}(T_{init}, T)} \end{array}}{\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{call}^{\Delta}(F, \phi, \overline{\langle \ell_{init}, T_{init} \rangle}, s^{\Delta_{fn}}) \\ \boxed{\Sigma} \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L}_{fn} \circ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{init-symbols}^{\Delta_{fn}}(\overline{\langle p, \langle \ell_{init}, T'_{init} \rangle \rangle}) \\ \boxed{\Sigma[\mathcal{L}_{fn(\texttt{id})} \xmapsto{\texttt{cmap}} \texttt{labelMappings}(\Sigma, \mathcal{H}, \mathcal{L}, \phi)]} \end{pmatrix}}$$

INITIALIZE

$$\frac{\begin{array}{ll} & \Gamma\{C\} = \langle \overline{f : T}, c \rangle \\ \boxed{\Sigma' = \Sigma[\ell \xmapsto{\texttt{cmap}} \texttt{tupleMappings}(\Sigma, \mathcal{H}, \mathcal{L}, \phi, \ell_{new})]} & \ell_{new} = \texttt{uniqueID}() \\ \mathcal{L}_{tuple} = \texttt{tupleLocalConfig}(\phi, \ell_{new}) & \theta = \texttt{container-of-tuple} \\ \mathcal{H}' = \mathcal{H}[\ell_{new} \mapsto \varnothing] & T_{new} = \langle \texttt{ref}, C, \theta, \texttt{mutable}, \texttt{movable} \rangle \\ & \overline{T'_{init} = \texttt{importType}(T_{init}, T)} \end{array}}{\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{init}^{\Delta}(C, \phi, \overline{\langle \ell_{init}, T_{init} \rangle}) \\ \boxed{\Sigma} \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H}' \quad \boxed{\mathcal{U} = \{\ell_{new}\}} \\ \mathcal{L}_{tuple} \circ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{init-fields}^{\Delta}(\langle \ell_{new}, T_{new} \rangle, \overline{\langle f, \langle \ell_{init}, T'_{init} \rangle \rangle}) \\ \boxed{\Sigma'[\ell_{new} \mapsto \texttt{ptype}(\Sigma', \mathcal{H}, \mathcal{L}_{tuple}, T_{new})]} \end{pmatrix}}$$

Figure 4.15: Transition Rules — Expressions

`labelMappings`() function, and it translates container labels into physical containers for the new scope.

With both of these maps established, the function call operation begins with the frame `init-symbols` which takes each passed parameter from $\overline{\langle \ell_{init}, T_{init} \rangle}$ and defines the respective symbols within the new scope. The function `funcLocalConfig`() places the function body into the new frame-stack.

The rule for `init`() initializes a new tuple using the passed parameters as the initial values. It works like a function call even though there is no function body to run. Here, the frame `init-fields` handles the parameters. The function `tupleLocalConfig` performs the same role as `funcLocalConfig`() and `tupleMappings` performs the same role as `labelMappings`().

The unique portion of `init`() is the allocation of a new heap location ℓ_{new} and its insertion into $\mathcal{U}$ recognising it as an uninitialized tuple. When the `init-fields` operation completes, it will remove ℓ_{new}, and $\mathcal{U}$ will be empty again.

4.8.4 Parameter Passing and Symbol Initialization

Figure 4.16 details rules dealing with symbol initialization and passing parameters to functions. The frame `init-symbols` takes a list of variables to be initialized and processes the first variable. Either Init-Symbols-Value or Init-Symbols-Ref can apply depending on the kind of the first variable. When initializing a value, a copy must be made using `copy-tuple`, and in this case, `init-symbol` is pushed onto the stack to consume the copy and do the variable assignment. After `init-symbol` is complete, the remainder of the variables are processed by another `init-symbols` frame, which is also pushed to the stack. When the variable list becomes empty, rule Init-Symbols-Value returns `void` which completes the operation.

Rule Init-Symbols-Ref is simpler, as it bypasses the `copy-tuple` step. Rules Init-Symbol-Value and Init-Symbol-Ref add the symbol v to the local configuration and map the variable to a newly allocated and initialized *Box* or *Ref*. The mapping from v to *Box*/*Ref* is fixed and access to the actual value of a variable requires a heap lookup.

4.8.5 Field Initialization

Field initialization behaves similarly to parameter initialization. Instead of defining symbols in the local configuration, the tuple construct in the heap has fields added to it

INIT-SYMBOLS-VALUE

$$\frac{\begin{array}{ll} & T_{\texttt{(kind)}} = \texttt{value} \\ \ell'_{init1} = \texttt{toTuple}(\mathcal{H}, \ell_{init1}) & \mathcal{H}(\ell_{init1}) \neq \texttt{null} \\ \overline{\mathcal{F}_{next}} = \texttt{init-symbol}^{\Delta}(v_1, \square) & \\ \qquad \circ\ \texttt{init-symbols}^{\Delta}(\texttt{tail}(\overline{\langle v, \langle \ell_{init}, T_{init} \rangle \rangle})) & T_1 = \Delta(v_1) \end{array}}{\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{init-symbols}^{\Delta}(\overline{\langle v, \langle \ell_{init}, T_{init} \rangle \rangle}) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L}[\texttt{push}\ \overline{\mathcal{F}_{next}}] \circ \overline{\mathcal{L}}] \\ \texttt{copy-tuple}^{\Delta}(T_1, \langle \ell'_{init1}, T_{init1} \rangle) \end{pmatrix}}$$

INIT-SYMBOLS-REF

$$\frac{\begin{array}{ll} \ell'_{init1} = \texttt{toTuple}(\mathcal{H}, \ell_{init1}) & T_1 = \Delta(v_1) \\ \mathcal{F}_{next} = \texttt{init-symbols}^{\Delta}(\texttt{tail}(\overline{\langle v, \langle \ell_{init}, T_{init} \rangle \rangle})) & T_{1\texttt{(kind)}} = \texttt{ref} \end{array}}{\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{init-symbols}^{\Delta}(\overline{\langle v, \langle \ell_{init}, T_{init} \rangle \rangle}) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L}[\texttt{push}\ \mathcal{F}_{next}] \circ \overline{\mathcal{L}} \\ \texttt{init-symbol}^{\Delta}(v_1, \langle \ell'_{init1}, T_{init1} \rangle) \end{pmatrix}}$$

INIT-SYMBOLS-COMPLETE

$$\frac{}{\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{init-symbols}^{\Delta}(\varnothing) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{void} \end{pmatrix}}$$

INIT-SYMBOL-VALUE

$$\frac{\begin{array}{ll} & T = \Delta(v) \\ & T_{\texttt{(kind)}} = \texttt{value} \\ \mathcal{H}' = \mathcal{H}[\ell_{\textit{dst-box}} \mapsto \ell_{init}] & \ell_{\textit{dst-box}} = \texttt{uniqueID}() \end{array}}{\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{init-symbol}^{\Delta}(v, \langle \ell_{init}, T_{init} \rangle) \\ \boxed{\Sigma} \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H}' \\ \mathcal{L}[\texttt{decl}\ \ v \mapsto \ell_{\textit{dst-box}}] \circ \overline{\mathcal{L}} \\ \texttt{void} \\ \boxed{\Sigma[\ell_{\textit{dst-box}} \mapsto \texttt{ptype}(\Sigma, \mathcal{H}', \mathcal{L}, T)]} \end{pmatrix}}$$

INIT-SYMBOL-REF

$$\frac{\begin{array}{ll} & T = \Delta(v) \\ & T_{\texttt{(kind)}} = \texttt{ref} \\ \mathcal{H}' = \mathcal{H}[\ell_{\textit{dst-ref}} \mapsto \ell_{init}] & \ell_{\textit{dst-ref}} = \texttt{uniqueID}() \end{array}}{\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{init-symbol}^{\Delta}(v, \ell_{init}) \\ \boxed{\Sigma} \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H}' \\ \mathcal{L}[\texttt{decl}\ \ v \mapsto \ell_{\textit{dst-ref}}] \circ \overline{\mathcal{L}} \\ \texttt{void} \\ \boxed{\Sigma[\ell_{\textit{dst-ref}} \mapsto \texttt{ptype}(\Sigma, \mathcal{H}', \mathcal{L}, T)]} \end{pmatrix}}$$

Figure 4.16: Transition Rules — Symbol Initialization

incrementally. When all fields have been initialized, the tuple is removed from the set of uninitialized tuples $\mathcal{U}$. Rules for field initialization are defined in two figures: 4.17 and 4.18. Each of the five rules is similar to its counterpart for parameter initialization. Also similar is the immutability of a tuple as each field name maps to a location in the heap for that field.

$$\textsc{Init-Fields-Value}\quad \frac{\begin{array}{l} T_{field} = \Gamma(T_{tuple})(f) \\ T_{field(\texttt{kind})} = \texttt{value} \\ \ell'_{init1} = \texttt{toTuple}(\mathcal{H}, \ell_{init1}) \\ \overline{\mathcal{F}_{next}} = \texttt{init-field}^{\Delta}(\mathcal{R}_{tuple}, f_1, \square) \\ \qquad\qquad \circ\ \texttt{init-fields}^{\Delta}(\mathcal{R}_{tuple}, \texttt{tail}(\overline{\langle f, \langle \ell_{init}, T_{init} \rangle \rangle})) \end{array}}{\left(\begin{array}{c} \mathcal{H} \quad \boxed{\mathcal{U} \cup \mathcal{R}_{tuple(\texttt{loc})}} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{init-fields}^{\Delta}(\mathcal{R}_{tuple}, \overline{\langle f, \langle \ell_{init}, T_{init} \rangle \rangle}) \end{array}\right) \rightarrow \left(\begin{array}{c} \mathcal{H} \quad \boxed{\mathcal{U} \cup \mathcal{R}_{tuple(\texttt{loc})}} \\ \mathcal{L}[\texttt{push}\ \ \overline{\mathcal{F}_{next}}] \circ \overline{\mathcal{L}}] \\ \texttt{copy-tuple}^{\Delta}(T_{field}, \langle \ell'_{init1}, T_{init1} \rangle) \end{array}\right)}$$

$$\textsc{Init-Fields-Ref}\quad \frac{\begin{array}{l} T_{field} = \Gamma(T_{tuple})(f) \\ T_{field(\texttt{kind})} = \texttt{ref} \\ \ell'_{init1} = \texttt{toTuple}(\mathcal{H}, \ell_{init_n}) \\ \mathcal{F}_{next} = \texttt{init-fields}^{\Delta}(\mathcal{R}_{tuple}, \texttt{tail}(\overline{\langle f, \langle \ell_{init}, T_{init} \rangle \rangle})) \end{array}}{\left(\begin{array}{c} \mathcal{H} \quad \boxed{\mathcal{U} \cup \mathcal{R}_{tuple(\texttt{loc})}} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{init-fields}^{\Delta}(\mathcal{R}_{tuple}, \overline{\langle f, \langle \ell_{init}, T_{init} \rangle \rangle}) \end{array}\right) \rightarrow \left(\begin{array}{c} \mathcal{H} \quad \boxed{\mathcal{U} \cup \mathcal{R}_{tuple(\texttt{loc})}} \\ \mathcal{L}[\texttt{push}\ \ \mathcal{F}_{next}] \circ \overline{\mathcal{L}} \\ \texttt{init-field}^{\Delta}(\mathcal{R}_{tuple}, f_1, \langle \ell'_{init1}, T_{init1} \rangle) \end{array}\right)}$$

$$\textsc{Init-Fields-Complete}\quad \frac{}{\left(\begin{array}{c} \mathcal{H} \quad \boxed{\mathcal{U} \cup \ell_{tuple}} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{init-fields}^{\Delta}(\langle \ell_{tuple}, T_{tuple} \rangle, \varnothing) \end{array}\right) \rightarrow \left(\begin{array}{c} \mathcal{H} \quad \boxed{\mathcal{U}} \\ \overline{\mathcal{L}} \\ \langle \ell_{tuple}, \texttt{exportType}(\mathcal{L}_{(\texttt{lmap})}, T_{tuple}) \rangle \end{array}\right)}$$

Figure 4.17: Transition Rules — Field Initialization (1)

Init-Field-Value

$$\frac{\begin{array}{ll} & \boxed{T_{\mathit{field}} = \Gamma(T_{\mathit{tuple}})(f)} \\ \mathit{tuple} = \mathcal{H}(\ell_{\mathit{tuple}})[f \mapsto \ell_{\mathit{dst\text{-}box}}] & T_{\texttt{(kind)}} = \texttt{value} \\ \mathcal{H}' = \mathcal{H}[\ell_{\mathit{dst\text{-}box}} \mapsto \ell_{\mathit{init}},\ \ell_{\mathit{tuple}} \mapsto \mathit{tuple}] & \ell_{\mathit{dst\text{-}box}} = \texttt{uniqueID}() \end{array}}{\left(\begin{array}{c} \mathcal{H} \quad \boxed{\mathcal{U} \cup \ell_{\mathit{tuple}}} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{init-field}^{\Delta}(\langle \ell_{\mathit{tuple}}, T_{\mathit{tuple}} \rangle, f, \langle \ell_{\mathit{init}}, T_{\mathit{init}} \rangle) \\ \boxed{\Sigma} \end{array}\right) \rightarrow \left(\begin{array}{c} \mathcal{H}' \quad \boxed{\mathcal{U} \cup \ell_{\mathit{tuple}}} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{void} \\ \boxed{\Sigma[\ell_{\mathit{dst\text{-}box}} \mapsto \texttt{ptype}(\Sigma, \mathcal{H}', \mathcal{L}, T_{\mathit{field}})]} \end{array}\right)}$$

Init-Field-Ref

$$\frac{\begin{array}{ll} & \boxed{T_{\mathit{field}} = \Gamma(T_{\mathit{tuple}})(f)} \\ \mathit{tuple} = \mathcal{H}(\ell_{\mathit{tuple}})[f \mapsto \ell_{\mathit{dst\text{-}ref}}] & T_{\texttt{(kind)}} = \texttt{ref} \\ \mathcal{H}' = \mathcal{H}[\ell_{\mathit{dst\text{-}ref}} \mapsto \ell_{\mathit{init}},\ \ell_{\mathit{tuple}} \mapsto \mathit{tuple}] & \ell_{\mathit{dst\text{-}ref}} = \texttt{uniqueID}() \end{array}}{\left(\begin{array}{c} \mathcal{H} \quad \boxed{\mathcal{U} \cup \ell_{\mathit{tuple}}} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{init-field}^{\Delta}(\langle \ell_{\mathit{tuple}}, T_{\mathit{tuple}} \rangle, f, \langle \ell_{\mathit{init}}, T_{\mathit{init}} \rangle) \\ \boxed{\Sigma} \end{array}\right) \rightarrow \left(\begin{array}{c} \mathcal{H}' \quad \boxed{\mathcal{U} \cup \ell_{\mathit{tuple}}} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{void} \\ \boxed{\Sigma[\ell_{\mathit{dst\text{-}ref}} \mapsto \texttt{ptype}(\Sigma, \mathcal{H}', \mathcal{L}, T_{\mathit{field}})]} \end{array}\right)}$$

Figure 4.18: Transition Rules — Field Initialization (2)

4.8.6 Object Copy Rules

The rules to copy objects are a third instance of the initialization strategy we've discussed with parameter and field initialization. We are initializing fields, so the greatest similarity is to tuple initialization. However, there is also another layer of complexity with copying, because we carry an accumulated record of all tuples that have been copied as we recursively copy a self-contained object.

The copy-map $\mathcal{A}$ has tuple heap locations as its domain and range. The domain represents the source tuple being copied and the range are the new tuples. At the completion of the copy process, the domain of $\mathcal{A}$ will contain the original tuple to copy plus all tuples reachable from that tuple. The copy-map is used to make sure that source tuples are only copied once. When a nested tuple is encountered a second time through a second alias, the copy-map is used to determine the appropriate tuple location that should be used for the respective alias in the copy. As an example, if a self-contained tuple contains a cycle of references within it, the use of the map will prevent an infinite loop.

Copy-Tuple

$$\frac{\overline{F}' = \texttt{copy-discard-env}^{\Delta}(\square) \circ \mathcal{L}_{(\texttt{fstack})}}{\begin{pmatrix}\mathcal{H}\\ \mathcal{L}\circ\overline{\mathcal{L}}\\ \texttt{copy-tuple}^{\Delta}(T_{dst},\langle \ell_{src},T_{src}\rangle)\end{pmatrix} \rightarrow \begin{pmatrix}\mathcal{H}\\ \mathcal{L}[\texttt{fstack}\mapsto \overline{F}']\circ\overline{\mathcal{L}}\\ \texttt{copy-tuple2}^{\Delta}(T_{dst},\langle\ell_{src},T_{src}\rangle,\varnothing)\end{pmatrix}}$$

Copy-Tuple-2

$$\frac{\begin{array}{ll}\boxed{\Sigma' = \Sigma[\ell_{new} \overset{\texttt{cmap}}{\longmapsto} \texttt{tupleMappings}(\Sigma,\mathcal{H},\mathcal{L},\phi,\ell_{new})]} & \\ \phi = \{\texttt{container-of-tuple}\mapsto T_{(\texttt{cont})}\} & \\ T_{new} = \langle T_{dst(\texttt{kind})}, T_{dst(\texttt{class})}, \theta, T_{dst(\texttt{mut})}, T_{dst(\texttt{mob})}\rangle & \langle\overline{f:T},\_\rangle = \Gamma(T_{src(\texttt{class})}) \\ \mathcal{L}_{new} = \texttt{tupleLocalConfig}(\phi,\ell_{new}) & \overline{f\mapsto\ell} = \mathcal{H}(\ell_{src}) \\ \mathcal{A}' = \mathcal{A}[\ell_{src}\to\ell_{new}] & \ell_{new} = \texttt{uniqueID}() \\ \mathcal{H}' = \mathcal{H}[\ell_{new}\mapsto\varnothing] & \theta = \texttt{container-of-tuple}\end{array}}{\begin{pmatrix}\mathcal{H}\quad\boxed{\mathcal{U}}\\ \mathcal{L}\circ\overline{\mathcal{L}}\\ \texttt{copy-tuple2}^{\Delta}(T_{dst},\langle\ell_{src},T_{src}\rangle,\mathcal{A})\\ \boxed{\Sigma}\end{pmatrix} \rightarrow \begin{pmatrix}\mathcal{H}'\quad\boxed{\mathcal{U}\cup\ell_{new}}\\ \mathcal{L}_{new}\circ\mathcal{L}\circ\overline{\mathcal{L}}\\ \texttt{copy-fields}^{\Delta}(\langle\ell_{new},T_{new}\rangle,\overline{\langle f,\langle\ell,T\rangle\rangle},\mathcal{A}')\\ \boxed{\Sigma'[\ell_{new}\mapsto\texttt{ptype}(\Sigma,\mathcal{H}',\mathcal{L},T_{dst})]}\end{pmatrix}}$$

Copy-Value

$$\frac{\begin{array}{l}T_{1(\texttt{kind})} = \texttt{value} \lor\ \ell'_1 \notin \texttt{dom}(\mathcal{A})\\ T_{field1} = \Gamma(T_{dst(\texttt{class})})(f_1)\\ \ell'_1 = \texttt{toTuple}(\mathcal{H},\ell_1)\\ \overline{\mathcal{F}_{next}} = \texttt{copy-init}^{\Delta}(\mathcal{R}_{tuple},f_1,\square)\\ \qquad\quad \circ\ \texttt{copy-fields}^{\Delta}(\mathcal{R}_{tuple},\texttt{tail}(\overline{\langle f,\langle\ell,T\rangle\rangle}),\square)\end{array}}{\begin{pmatrix}\mathcal{H}\quad\boxed{\mathcal{U}\cup\mathcal{R}_{tuple(\texttt{loc})}}\\ \mathcal{L}\circ\overline{\mathcal{L}}\\ \texttt{copy-fields}^{\Delta}(\mathcal{R}_{tuple},\overline{\langle f,\langle\ell,T\rangle\rangle},\mathcal{A})\end{pmatrix} \rightarrow \begin{pmatrix}\mathcal{H}\quad\boxed{\mathcal{U}\cup\mathcal{R}_{tuple(\texttt{loc})}}\\ \mathcal{L}[\texttt{push}\ \overline{\mathcal{F}_{next}}]\circ\overline{\mathcal{L}}\\ \texttt{copy-tuple2}^{\Delta}(T_{field1},\langle\ell'_1,T_1\rangle,\mathcal{A})\end{pmatrix}}$$

Copy-Ref

$$\frac{\begin{array}{l}T_{1(\texttt{kind})} = \texttt{ref} \land\ (\ell'_1 \in \texttt{dom}(\mathcal{A}) \lor\ \ell'_1 = \texttt{null-ref})\\ \ell'_1 = \texttt{if}\ \ \ell_1 = \texttt{null-ref then null-ref else}\ \mathcal{A}(\texttt{toTuple}(\mathcal{H},\ell_1))\\ \mathcal{F}_{next} = \texttt{copy-fields}^{\Delta}(\mathcal{R}_{tuple},\texttt{tail}(\overline{\langle f,\langle\ell,T\rangle\rangle}),\square)\end{array}}{\begin{pmatrix}\mathcal{H}\quad\boxed{\mathcal{U}\cup\mathcal{R}_{tuple(\texttt{loc})}}\\ \mathcal{L}\circ\overline{\mathcal{L}}\\ \texttt{copy-fields}^{\Delta}(\mathcal{R}_{tuple},\overline{\langle f,\langle\ell,T\rangle\rangle},\mathcal{A})\end{pmatrix} \rightarrow \begin{pmatrix}\mathcal{H}\quad\boxed{\mathcal{U}\cup\mathcal{R}_{tuple(\texttt{loc})}}\\ \mathcal{L}[\texttt{push}\ \mathcal{F}_{next}]\circ\overline{\mathcal{L}}\\ \texttt{copy-init}^{\Delta}(\mathcal{R}_{tuple},f_1,\langle\langle\ell'_1,T_1\rangle,\mathcal{A}\rangle)\end{pmatrix}}$$

Figure 4.19: Transition Rules — Copy Tuple (1)

Copy-Complete

$$\frac{}{\begin{pmatrix} \mathcal{H} \quad \boxed{\mathcal{U} \cup \ell_{tuple}} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{copy-fields}^{\Delta}(\langle \ell_{tuple}, T_{tuple} \rangle, \varnothing, \mathcal{A}) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \quad \boxed{\mathcal{U}} \\ \overline{\mathcal{L}} \\ \langle \langle \ell_{tuple}, \texttt{exportType}(\mathcal{L}_{\texttt{(lmap)}}, T_{tuple}) \rangle, \mathcal{A} \rangle \end{pmatrix}}$$

Copy-Init-Value

$$\frac{\begin{array}{c} tuple = \mathcal{H}(\ell_{tuple})[f \mapsto \ell_{dst\text{-}box}] \\ \mathcal{H}' = \mathcal{H}[\ell_{dst\text{-}box} \mapsto \ell, \ell_{tuple} \mapsto tuple] \end{array} \qquad \begin{array}{l} \boxed{T_{field} = \Gamma(T_{tuple})(f)} \\ T_{\texttt{(kind)}} = \texttt{value} \\ \ell_{dst\text{-}box} = \texttt{uniqueID}() \end{array}}{\begin{pmatrix} \mathcal{H} \quad \boxed{\mathcal{U} \cup \ell_{tuple}} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{copy-init}^{\Delta}(\langle \ell_{tuple}, T_{tuple} \rangle, f, \langle \langle \ell, T \rangle, \mathcal{A} \rangle) \\ \boxed{\Sigma} \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H}' \quad \boxed{\mathcal{U} \cup \ell_{tuple}} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \mathcal{A} \\ \boxed{\Sigma[\ell_{dst\text{-}box} \mapsto \texttt{ptype}(\Sigma, \mathcal{H}', \mathcal{L}, T_{field})]} \end{pmatrix}}$$

Copy-Init-Ref

$$\frac{\begin{array}{c} tuple = \mathcal{H}(\ell_{tuple})[f \mapsto \ell_{dst\text{-}ref}] \\ \mathcal{H}' = \mathcal{H}[\ell_{dst\text{-}ref} \mapsto \ell, \ell_{tuple} \mapsto tuple] \end{array} \qquad \begin{array}{l} \boxed{T_{field} = \Gamma(T_{tuple})(f)} \\ T_{\texttt{(kind)}} = \texttt{ref} \\ \ell_{dst\text{-}ref} = \texttt{uniqueID}() \end{array}}{\begin{pmatrix} \mathcal{H} \quad \boxed{\mathcal{U} \cup \ell_{tuple}} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{copy-init}^{\Delta}(\langle \ell_{tuple}, T_{tuple} \rangle, f, \langle \langle \ell, T \rangle, \mathcal{A} \rangle) \\ \boxed{\Sigma} \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H}' \quad \boxed{\mathcal{U} \cup \ell_{tuple}} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \mathcal{A} \\ \boxed{\Sigma[\ell_{dst\text{-}ref} \mapsto \texttt{ptype}(\Sigma, \mathcal{H}', \mathcal{L}_{new}, T_{field})\}} \end{pmatrix}}$$

Copy-Discard-Env

$$\frac{}{\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{copy-discard-env}^{\Delta}(\mathcal{R}, \mathcal{A} \rangle) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \mathcal{R} \end{pmatrix}}$$

Figure 4.20: Transition Rules — Copy Tuple (2)

4.8.7 Null De-Reference Guards

There are a number of places where the transition rules require access to a tuple. In these cases, if a `null-ref` were used as the tuple, the machine would break. To prevent this, there are a number of rules defined in figure 4.21 to trap null violations and safely halt the machine. There is one rule for each frame that cannot tolerate a `null-ref`.

Assign-Value-NPE

$$\frac{\mathcal{H}(\ell_{rhs}) = \texttt{null-ref}}{\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{assign-value}^{\Delta}(\langle \ell_{lhs}, T_{lhs} \rangle, \langle \ell_{rhs}, T_{rhs} \rangle) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{NPE} \end{pmatrix}}$$

Return-Value-NPE

$$\frac{\mathcal{H}(\ell_{ret}) = \texttt{null-ref}}{\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{return}^{\Delta}(T, \langle \ell_{ret}, T_{ret} \rangle) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{NPE} \end{pmatrix}}$$

Field-NPE

$$\frac{\mathcal{H}(\ell_{tuple}) = \texttt{null-ref}}{\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{field}^{\Delta}(\langle \ell_{tuple}, T_{tuple} \rangle, f) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{NPE} \end{pmatrix}}$$

Init-Symbols-NPE

$$\frac{T = \Delta(v_1) \qquad T_{\texttt{(kind)}} = \texttt{value} \qquad \mathcal{H}(\ell_{init1}) = \texttt{null-ref}}{\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{init-symbols}^{\Delta}(\overline{\langle v, \langle \ell_{init}, T_{init} \rangle \rangle}) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{NPE} \end{pmatrix}}$$

Init-Fields-NPE

$$\frac{T = \Delta(v_1) \qquad T_{\texttt{(kind)}} = \texttt{value} \qquad \mathcal{H}(\ell_{init1}) = \texttt{null-ref}}{\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{init-fields}^{\Delta}(\langle \ell_{tuple}, T_{tuple} \rangle, \overline{\langle f, \langle \ell_{init}, T_{init} \rangle \rangle}) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{NPE} \end{pmatrix}}$$

Copy-Fields-NPE

$$\frac{T = \Delta(v_1) \qquad T_{\texttt{(kind)}} = \texttt{value} \qquad \mathcal{H}(\ell_{init1}) = \texttt{null-ref}}{\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{copy-fields}^{\Delta}(\langle \ell_{tuple}, T_{tuple} \rangle, \overline{\langle f, \langle \ell, T \rangle \rangle}, \mathcal{A}) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{NPE} \end{pmatrix}}$$

Figure 4.21: Transition Rules — Null Pointer Violations

4.8.8 Decomposition Rules

Our final set of transition rules in figure 4.22 and 4.23 take care of decomposing complex frames into discrete operations. Sub-expressions are removed from their parent frame, leaving a hole (□) in place of the expression. The expression is placed on the frame-stack and becomes a result after it is executed. Some frames contain a list of expressions (e.g., `call()`) and each one is extracted in turn until all sub-expressions have been replaced with results. At this stage, the completed frame is ready to be applied to one of previous rules we've discussed.

D-Let

$$\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{let}^{\Delta}(T, l, e, s) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L}[\texttt{fstack} \mapsto \texttt{let}(T, \ell, \square, s) \circ \mathcal{L}_{(\texttt{fstack})}] \circ \overline{\mathcal{L}} \\ e^{\Delta} \end{pmatrix}$$

D-Assign-Value-1

$$\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{assign-value}^{\Delta}(T, e_{lhs}, e_{rhs}) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L}[\texttt{push assign-value}(T, \square, e_{rhs})] \circ \overline{\mathcal{L}} \\ e^{\Delta}_{lhs} \end{pmatrix}$$

D-Assign-Value-2

$$\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{assign-value}^{\Delta}(T, \ell_{lhs}, e_{rhs}) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L}[\texttt{push assign-value}(T, \ell_{lhs}, \square)] \circ \overline{\mathcal{L}} \\ e^{\Delta}_{rhs} \end{pmatrix}$$

D-Assign-Reference-1

$$\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{assign-ref}^{\Delta}(e_{lhs}, e_{rhs}) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L}[\texttt{push assign-ref}(\square, e_{rhs})] \circ \overline{\mathcal{L}} \\ e^{\Delta}_{lhs} \end{pmatrix}$$

D-Assign-Reference-2

$$\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{assign-ref}^{\Delta}(\ell_{lhs}, e_{rhs}) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L}[\texttt{push assign-ref}(\ell_{lhs}, \square)] \circ \overline{\mathcal{L}} \\ e^{\Delta}_{rhs} \end{pmatrix}$$

Figure 4.22: Transition Rules — Decompositions (1)

D-Return

$$\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{return}^{\Delta}(e) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L}[\texttt{push return}(\Box)] \circ \overline{\mathcal{L}} \\ e^{\Delta} \end{pmatrix}$$

D-field

$$\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{field}^{\Delta}(e, f) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L}[\texttt{push field}(\Box, f)] \circ \overline{\mathcal{L}} \\ e^{\Delta} \end{pmatrix}$$

D-Call-Function

$$\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{call}^{\Delta}(F, \phi, \ell_1 \ldots \ell_{i-1}, e_i \ldots e_n) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L}[\texttt{push call}(F, \phi, \ell_1 \ldots \ell_{i-1}, \Box, e_{i+1} \ldots e_n)] \circ \overline{\mathcal{L}} \\ e_i^{\Delta} \end{pmatrix}$$

D-Initialize-Tuple

$$\begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \texttt{init}^{\Delta}(C, \ell_1 \ldots \ell_{i-1}, e_i \ldots e_n) \end{pmatrix} \rightarrow \begin{pmatrix} \mathcal{H} \\ \mathcal{L}[\texttt{push init}(C, \ell_1 \ldots \ell_{i-1}, \Box, e_{i+1} \ldots e_n)] \circ \overline{\mathcal{L}} \\ e_i^{\Delta} \end{pmatrix}$$

Figure 4.23: Transition Rules — Decompositions (2)

4.9 Configuration Typing

With the transitions rules defined, we turn our attention back to typing. We typed our frames in section 4.6 and now we address the remainder of the configuration. Figure 4.24 contains the `config-ok` judgment, which validates the entire configuration. This includes fully typing the heap, all frame-stacks and all local variables. Establishing `config-ok` relies on many other rules to type each of the various components. First, we'll look at the `heap-ok` judgment.

The heap is comprised of a set of abstract locations, and one of the conditions of `heap-ok` is that every location have a corresponding typing in Σ. For each of these locations we apply the `location-ok` judgment to ensure that the heap and its typing are in agreement.

There are four rules, also in figure 4.24, that establish `location-ok`. The rule Object-OK handles all tuple locations that are fully initialized (not in $\mathcal{U}$). It looks up the set of fields from Γ and confirms that each field is present and the heap location associated with the field has a compatible type using the `f-match` judgment which we'll detail soon.

Configuration-OK

$$\frac{\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{H}\ \texttt{heap-ok} \qquad \Gamma; \Sigma; \mathcal{H} \vdash \mathcal{L}[\texttt{push}\ \ \mathcal{CF}^{\Delta}] \circ \overline{\mathcal{L}} : (\texttt{Void} \to \mathcal{T}, \mathcal{U} \to \varnothing)}{\Gamma; \Sigma; \mathcal{U} \vdash \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \mathcal{CF}^{\Delta} \end{pmatrix} \texttt{config-ok}}$$

Heap-OK

$$\frac{\forall\ \ell_i \in \mathcal{H}.\ \Gamma; \Sigma; \mathcal{H} \vdash \ell_i\ \texttt{location-ok} \qquad \texttt{dom}(\mathcal{H}) = \texttt{dom}(\Sigma)}{\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{H}\ \texttt{heap-ok}}$$

Object-OK

$$\frac{\begin{array}{c} \ell \in \mathit{TupleLocation} \qquad \ell \notin \mathcal{U} \qquad \mathcal{T}_{\mathit{obj}} = \Sigma(\ell) \\ \forall f_i \in \Gamma(\mathcal{T}_{\mathit{obj}(\texttt{class})}).\ \Sigma; \mathcal{H}; \ell \vdash \Gamma(\mathcal{T}_{\mathit{obj}(\texttt{class})})(f_i)\ \ \texttt{f-match}\ \ \Sigma(\mathcal{H}(\ell)(f_i)) \end{array}}{\Gamma; \Sigma; \mathcal{U}; \mathcal{H} \vdash \ell\ \texttt{location-ok}}$$

Object-OK-Uninit

$$\frac{\begin{array}{c} \ell \in \mathit{TupleLocation} \qquad \ell \in \mathcal{U} \\ \mathcal{T}_{\mathit{obj}} = \Sigma(\ell) \qquad \forall f_i \in \mathcal{H}(\ell).\ \Sigma; \mathcal{H}; \ell \vdash \Gamma(\mathcal{T}_{\mathit{obj}(\texttt{class})})(f_i)\ \ \texttt{f-match}\ \ \Sigma(\mathcal{H}(\ell)(f_i)) \end{array}}{\Gamma; \Sigma; \mathcal{U}; \mathcal{H} \vdash \ell\ \texttt{location-ok}}$$

Null-Ref-OK

$$\frac{\ell \in \mathit{RefLocation} \qquad \mathcal{H}(\ell) = \texttt{null-ref}}{\Gamma; \Sigma; \mathcal{U}; \mathcal{H} \vdash \ell\ \texttt{location-ok}}$$

Ref/Box-OK

$$\frac{\begin{array}{c} \ell \in \mathit{SymbolLocation} \\ \mathcal{H}(\ell) \neq \texttt{null-ref} \qquad \mathcal{T}_{\mathit{ref}} = \Sigma(\ell) \qquad \mathcal{T}_{\mathit{tuple}} = \Sigma(\mathcal{H}(\ell)) \qquad \mathcal{T}_{\mathit{ref}}\ \ \texttt{p-match}\ \ \mathcal{T}_{\mathit{tuple}} \end{array}}{\Gamma; \Sigma; \mathcal{U}; \mathcal{H} \vdash \ell\ \texttt{location-ok}}$$

Figure 4.24: Configuration Typing Rules (1)

For uninitialized objects, the rule Object-OK-Uninit is used and the validation is similar, but the set of fields that are type checked are only the fields that are currently present in the heap. Next we have a special rule, for reference heap locations that are set to `null-ref`. This case requires no further validation, as all reference types are permitted to be `null-ref`.

For the normal reference case as well as boxes, rule Ref/Box-OK is used. This validates that the physical type of the box or reference matches the physical type of the tuple that is referenced by the box or reference.

In figure 4.25, we have the next set of rules. The `f-match` judgment is defined in the rule Field-Match, and contains the condition `within`, which is perhaps the most important condition for establishing proper containment properties. The `within` judgment is defined in rule Containment and requires that for a container Θ_1 to be considered *within* another container Θ_2, it either must be the same container ($\Theta_1 = \Theta_2$) or the container of Θ_1 must be *within* Θ_2. Nested containers unwrap as many layers of containers as needed to find the container of interest. Note that we are considering physical containers, which are identified by a heap location scope identifier within the configuration. Establishing this properly means that containers behave the way our common sense dictates they should. This means that no reference that is inside the container may refer to an object that is outside container.

Returning to the judgment `f-match`, it relies on three additional judgments: `t-match`, to compare logical to physical types, `p-match`, which checks that two physical types match, and finally `c-match`, which checks that containers match. We'll now look at this sequence of judgments in more detail.

The second condition of `f-match` is `t-match` which verifies that the logical type T is compatible with its physical type $\mathcal{T}$. The `t-match` judgment uses the `ptype`() function to convert it into a physical type. Now, with two physical types, we validate them using the `p-match` judgment.

`p-match` validates the sub-components of the physical type. For a class there must be an exact match. For mutability, there must either be a match or the left-hand side container must be read-only. This mutability logic mirrors the type rules of the language. The third condition validates the containers and delegates this to our last matching rule `c-match`.

The rule Container-Match takes two physical containers Θ_{ref} and Θ_{value}, and determines if they are compatible. Either $\Theta_{ref} = \Theta_{value}$ or Θ_{ref} must be set to `unknown-cont`, which indicates that it is compatible with any container.

Field-Match

$$\frac{\begin{array}{c}\mathcal{L} = \texttt{tupleLocalConfig}(\varnothing, \ell) \\ \Sigma \vdash \mathcal{T}_{\texttt{(cont)}}\ \texttt{within}\ \Sigma(\ell_{tuple})_{\texttt{(cont)}} \qquad \Sigma; \mathcal{H}; \mathcal{L} \vdash T\ \texttt{t-match}\ \mathcal{T}\end{array}}{\Sigma; \mathcal{H}; \ell_{tuple} \vdash T\ \texttt{f-match}\ \mathcal{T}}$$

Type-Match

$$\frac{\texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}, T)\ \texttt{p-match}\ \mathcal{T}}{\Sigma; \mathcal{H}; \mathcal{L} \vdash T\ \texttt{t-match}\ \mathcal{T}}$$

Physical-Match

$$\frac{\begin{array}{c}\mathcal{T}_{ref\texttt{(cont)}}\ \texttt{c-match}\ \mathcal{T}_{tuple\texttt{(cont)}} \\ \mathcal{T}_{ref\texttt{(class)}} = \mathcal{T}_{tuple\texttt{(class)}} \qquad \mathcal{T}_{ref\texttt{(mut)}} = \mathcal{T}_{tuple\texttt{(mut)}} \lor \mathcal{T}_{ref\texttt{(mut)}} = \texttt{read-only}\end{array}}{\mathcal{T}_{ref}\ \texttt{p-match}\ \mathcal{T}_{tuple}}$$

Container-Match

$$\frac{\Theta_{ref} = \Theta_{value} \lor \Theta_{ref} = \texttt{unknown-cont}}{\Theta_{ref}\ \texttt{c-match}\ \Theta_{value}}$$

Vars-OK

$$\frac{\forall v_i \in \texttt{dom}(\mathcal{V}).\ \Gamma; \Delta \vdash \texttt{var}(v_i) : T_i \qquad \Sigma; \mathcal{H}; \mathcal{L} \vdash T_i\ \ \texttt{t-match}\ \ \Sigma(\mathcal{V}(v_i))}{\Gamma; \Delta; \Sigma; \mathcal{L} \vdash \mathcal{V}\ \texttt{vars-ok}}$$

Containment

$$\frac{\Theta_1 = \Theta_2 \lor \Theta_2 = \texttt{null-cont} \lor \Sigma(\Theta_1)_{\texttt{(cont)}}\ \texttt{within}\ \Theta_2}{\Sigma \vdash \Theta_1\ \texttt{within}\ \Theta_2}$$

Copy-Map-OK

$$\frac{\forall \ell_i \in \texttt{dom}(\mathcal{A}).\ \ell_i \in \Sigma \land \ell_i \notin \texttt{range}(\mathcal{A}) \qquad \forall \ell_j \in \texttt{range}(\mathcal{A}).\ \ell_j \in \Sigma}{\Sigma \vdash \mathcal{A}\ \texttt{copy-map-ok}}$$

CMap-OK

$$\frac{\begin{array}{c}\mathcal{F}^{\Delta_{caller}} = \texttt{head}(\mathcal{L}_{caller\texttt{(fstack)}}) \qquad \Gamma; \Delta_{caller} \vdash \mathcal{L}_{\texttt{(ctx,lmap)}}\ \texttt{lmap-ok} \\ \phi = \mathcal{L}_{\texttt{(lmap)}} \qquad \mathcal{M} = \Sigma_{\texttt{(cmap)}}(\mathcal{L}_{\texttt{(id)}}) \qquad \texttt{dom}(\phi) = \texttt{dom}(\mathcal{M}) \\ \forall \theta_i \in \texttt{dom}(\mathcal{M}).\ \mathcal{M}(\theta_i) = \texttt{labelToCont}(\Sigma, \mathcal{H}, \mathcal{L}, \theta_i) = \texttt{labelToCont}(\Sigma, \mathcal{H}, \mathcal{L}_{caller}, \texttt{mapLabel}(\phi, \theta_i))\end{array}}{\Sigma; \mathcal{H} \vdash \mathcal{L}_{caller}\ \texttt{cmap-ok}\ \mathcal{L}}$$

Figure 4.25: Configuration Typing Rules (2)

We've completed the fields discussion, and move on to the `vars-ok` judgment in the rule VARS-OK. Like fields, each variable must satisfy the `t-match` judgments. Unlike fields, there is no check needed for `within`, because local symbols aren't fields of contained objects.

The `copy-map-ok` is a simple rule that ensures that when copying an object, none of the tuples reachable from the source tuple appear in the range of $\mathcal{A}$. The converse case must be true as well; none of the new tuples in the range of $\mathcal{A}$ can appear in the domain.

The last rule of figure 4.25 is CMAP-OK. Recall that a container map $\mathcal{M}$ converts container labels in a scope $\mathcal{L}$ to physical containers. Also, container maps are built using information from mapping ϕ as well as physical container labels from the parent scope $\mathcal{L}_{caller}$. The judgment, written $\Sigma; \mathcal{H} \vdash \mathcal{L}_{caller}$ `cmap-ok` $\mathcal{L}$, is judging $\mathcal{M}$ relative to a caller and callee. All labels in $\mathcal{M}$ must be mapped to physical containers which are in agreement with the physical containers of the corresponding container labels in the caller's scope.

This completes all the judgments needed to validate the heap in our configuration. What remains is the stack of local configurations and the frame-stacks within them.

Local Configuration Typing

The second component of our master `config-ok` judgment is the typing of the local configuration stack, which is typed by the rules in figure 4.26. This judgment works by induction, but first we'll look at how it validates the current local configuration $\mathcal{L}$. The local variables in $\mathcal{L}$ are checked using `vars-ok`. Scope $\mathcal{L}$ is created by scope `head`($\overline{\mathcal{L}}$) and these two scopes are used to validate $\mathcal{L}$'s container map $\mathcal{M}$ using `cmap-ok`. The frame-stack of $\mathcal{L}$ is validated using the frame-stack typing rules in the next section.

The last condition of the local configuration typing is the inductive step to verify the tail of the local configuration stack. The final typing of the stack is of the form $(\mathcal{T} \to \mathcal{T}'', \mathcal{U} \to \varnothing)$. The first relation $\mathcal{T} \to \mathcal{T}''$ means that this local configuration stack

T-LOCAL-CONFIG-STACK

$$\frac{\begin{array}{c}\Gamma; \Delta; \Sigma \vdash \mathcal{L}\ \texttt{vars-ok} \qquad \Sigma; \mathcal{H} \vdash \texttt{head}(\overline{\mathcal{L}})\ \texttt{cmap-ok}\ \mathcal{L} \\ \Gamma; \Sigma; \mathcal{H}; \mathcal{L} \vdash \mathcal{L}_{(\texttt{fstack})} : (\mathcal{T} \to \mathcal{T}', \mathcal{U} \to \mathcal{U}') \qquad \Gamma; \Sigma; \mathcal{H} \vdash \overline{\mathcal{L}} : (\mathcal{T}' \to \mathcal{T}'', \mathcal{U}' \to \varnothing)\end{array}}{\Gamma; \Sigma; \mathcal{H} \vdash \mathcal{L} \circ \overline{\mathcal{L}} : (\mathcal{T} \to \mathcal{T}'', \mathcal{U} \to \varnothing)}$$

T-LOCAL-CONFIG-STACK-EMPTY

$$\frac{}{\Gamma; \Sigma; \mathcal{H} \vdash \varnothing : (\mathcal{T} \to \mathcal{T}, \mathcal{U} \to \mathcal{U})}$$

Figure 4.26: Local Configuration Typing

is a computation taking $\mathcal{T}$ as input and producing $\mathcal{T}''$ as output. The first $\mathcal{T}$ matches the input type of $\mathcal{L}_{\texttt{(fstack)}}$ and the $\mathcal{T}''$ matches with the inductive typing of the stack tail.

The second component of the typing is $\mathcal{U} \to \varnothing$, which indicates that this stack begins with a configuration where the heap locations in $\mathcal{U}$ are uninitialized and the computation ends with no uninitialized objects.

Returning to the definition of `config-ok`, notice how it melds the current frame into the current local configuration's frame-stack before typing it. By placing the current frame $\mathcal{CF}$ into $\mathcal{L}_{\texttt{(fstack)}}$ before typing the local configuration stack, we don't need to special case the typing of $\mathcal{CF}$ as it happens automatically as part of the local configuration typing.

Frame-Stack Typing

To type the frame-stacks, we first separate the frames into groups. Open frames are typed differently than closed frames, and frames that work with uninitialized tuples need special treatment. Except for stacks with the frame `return` on the top, all frame-stacks can be typed generically without specific rules for each frame. Return statements have special treatment to support the case of returning early from a function, and as mentioned earlier, this capability is not taken advantage of in the current system.

$\mathcal{UF}_{same} = \{\mathcal{R}^*\texttt{, void, copy-tuple-2, init-field, copy-init}\}$
$\mathcal{UF}_{done} = \{\texttt{init-fields, copy-fields, copy-complete}\}$
$\mathcal{UF} = \mathcal{UF}_{done} \cup \mathcal{UF}_{same}$

We define sets of frames to split them according to their usage of uninitialized tuples. Frames that type with $\mathcal{U} \to \mathcal{U}$ are in $\mathcal{UF}_{same}$ and frames that type as $\mathcal{U} \cup \ell \to \mathcal{U}$ are in $\mathcal{UF}_{done}$. No frame-stack type ever needs to add elements to $\mathcal{U}$, because frames that do expand $\mathcal{U}$ always push an additional frame to the stack to remove the new uninitialized tuple. This preserves the $\mathcal{U} \to \mathcal{U}$ typing.

The frame-stack typing rules appear in figure 4.27. Similar to local configuration typing, frame-stacks are typed as $(\mathcal{T} \to \mathcal{T}', \mathcal{U} \to \mathcal{U}')$. The different rules vary based on which frame is on the top of the stack. In general, the top frame is typed directly and the remainder of the stack is typed inductively, with the rule T-Empty-Stack terminating the induction. The output of the top frame must be consistent with the input of the inductively typed frames. Frames are logically typed as T, and the frame-stack uses physical types $\mathcal{T}$. The function `ptype`() is found in each rule to do the conversion.

Stacks with an open frame $\mathcal{OF}$ on top only differ from stacks with closed frames $\mathcal{CF}$ in that the input type is always `void` in the rule T-Closed-Stack, and it's the type of the

T-Empty-Stack

$$\frac{}{\Gamma;\Sigma;\mathcal{H};\mathcal{L}\vdash [] : (\mathcal{T}\to\mathcal{T},\mathcal{U}\to\mathcal{U})}$$

T-Open-Stack

$$\frac{\begin{array}{c} \mathcal{OF}\notin\mathcal{UF} \\ \mathcal{CF}\neq\texttt{return(\_)} \qquad \Gamma;\Delta\cup\{\Box:T\};\Sigma\vdash\mathcal{OF}:T' \qquad \Gamma;\Sigma;\mathcal{H};\mathcal{L}\vdash\overline{\mathcal{F}}:(\mathcal{T}'\to\mathcal{T}'',\varnothing\to\varnothing) \\ \mathcal{T}=\texttt{ptype}(\Sigma,\mathcal{H},\mathcal{L},T) \qquad \mathcal{T}'=\texttt{ptype}(\Sigma,\mathcal{H},\mathcal{L},T') \end{array}}{\Gamma;\Sigma;\mathcal{H};\mathcal{L}\vdash\mathcal{OF}^{\Delta}\circ\overline{\mathcal{F}}:(\mathcal{T}\to\mathcal{T}'',\varnothing\to\varnothing)}$$

T-Closed-Stack

$$\frac{\begin{array}{c} \mathcal{CF}\notin\mathcal{UF} \qquad \mathcal{CF}\neq\texttt{return(\_)} \qquad \Gamma;\Delta;\Sigma;\mathcal{H};\mathcal{L}\vdash\mathcal{CF}:T \\ \mathcal{T}=\texttt{ptype}(\Sigma,\mathcal{H},\mathcal{L},T) \qquad \Gamma;\Sigma;\mathcal{H};\mathcal{L}\vdash\overline{\mathcal{F}}:(\mathcal{T}\to\mathcal{T}',\varnothing\to\varnothing) \end{array}}{\Gamma;\Sigma;\mathcal{H};\mathcal{L}\vdash\mathcal{CF}^{\Delta}\circ\overline{\mathcal{F}}:(\texttt{Void}\to\mathcal{T}',\varnothing\to\varnothing)}$$

T-Immediate-Return

$$\frac{\begin{array}{c} \Gamma;\Delta;\Sigma;\mathcal{H};\mathcal{L}\vdash\texttt{return}(\mathcal{CF}):T_{callee} \\ T=\texttt{exportType}(\mathcal{L}_{\texttt{(lmap)}},T_{callee}) \qquad \mathcal{T}=\texttt{ptype}(\Sigma,\mathcal{H},\mathcal{L},T) \end{array}}{\Gamma;\Sigma;\mathcal{H};\mathcal{L}\vdash\texttt{return}^{\Delta}(\mathcal{CF})\circ\_:(\texttt{Void}\to\mathcal{T},\varnothing\to\varnothing)}$$

T-Uninit-Closed-Stack

$$\frac{\begin{array}{c} \mathcal{CF}\in\mathcal{UF}_{same} \qquad \Gamma;\Delta;\Sigma;\mathcal{H};\mathcal{L}\vdash\mathcal{CF}:T \\ \mathcal{T}=\texttt{ptype}(\Sigma,\mathcal{H},\mathcal{L},T) \qquad \Gamma;\Sigma;\mathcal{H};\mathcal{L}\vdash\overline{\mathcal{F}}:(\mathcal{T}\to\mathcal{T}',\mathcal{U}\to\mathcal{U}') \end{array}}{\Gamma;\Sigma;\mathcal{H};\mathcal{L}\vdash\mathcal{CF}^{\Delta}\circ\overline{\mathcal{F}}:(\texttt{Void}\to\mathcal{T}',\mathcal{U}\to\mathcal{U}')}$$

T-Uninit-Open-Stack

$$\frac{\begin{array}{c} \mathcal{OF}\in\mathcal{UF}_{same} \qquad \Gamma;\Delta\cup\{\Box:T\};\Sigma\vdash\mathcal{OF}:T' \qquad \Gamma;\Sigma;\mathcal{H};\mathcal{L}\vdash\overline{\mathcal{F}}:(\mathcal{T}'\to\mathcal{T}'',\mathcal{U}\to\mathcal{U}') \\ \mathcal{T}=\texttt{ptype}(\Sigma,\mathcal{H},\mathcal{L},T) \qquad \mathcal{T}'=\texttt{ptype}(\Sigma,\mathcal{H},\mathcal{L},T') \end{array}}{\Gamma;\Sigma;\mathcal{H};\mathcal{L}\vdash\mathcal{OF}^{\Delta}\circ\overline{\mathcal{F}}:(\mathcal{T}\to\mathcal{T}'',\mathcal{U}\to\mathcal{U}')}$$

T-Uninit-Closed-Stack-Done

$$\frac{\begin{array}{c} \mathcal{CF}\in\mathcal{UF}_{done} \qquad \Gamma;\Delta;\Sigma;\mathcal{H};\mathcal{L}\vdash\mathcal{CF}:T \qquad \Gamma;\Sigma;\mathcal{H};\mathcal{L}\vdash\overline{\mathcal{F}}:(\mathcal{T}\to\mathcal{T}',\mathcal{U}\to\mathcal{U}') \\ \mathcal{T}=\texttt{ptype}(\Sigma,\mathcal{H},\mathcal{L},T) \qquad \mathcal{T}'=\texttt{ptype}(\Sigma,\mathcal{H},\mathcal{L},T') \end{array}}{\Gamma;\Sigma;\mathcal{H};\mathcal{L}\vdash\mathcal{CF}^{\Delta}\circ\overline{\mathcal{F}}:(\texttt{Void}\to\mathcal{T}',\mathcal{U}\cup\{\ell\}\to\mathcal{U}')}$$

T-Uninit-Open-Stack-Done

$$\frac{\begin{array}{c} \mathcal{OF}\in\mathcal{UF}_{done} \\ \Gamma;\Delta\cup\{\Box:T\};\Sigma;\mathcal{H};\mathcal{L}\vdash\mathcal{CF}:T' \qquad \Gamma;\Sigma;\mathcal{H};\mathcal{L}\vdash\overline{\mathcal{F}}:(\mathcal{T}'\to\mathcal{T}'',\mathcal{U}\to\mathcal{U}') \\ \mathcal{T}=\texttt{ptype}(\Sigma,\mathcal{H},\mathcal{L},T) \qquad \mathcal{T}'=\texttt{ptype}(\Sigma,\mathcal{H},\mathcal{L},T') \end{array}}{\Gamma;\Sigma;\mathcal{H};\mathcal{L}\vdash\mathcal{OF}^{\Delta}\circ\overline{\mathcal{F}}:(\mathcal{T}\to\mathcal{T}'',\mathcal{U}\cup\{\ell\}\to\mathcal{U}')}$$

Figure 4.27: Frame-Stack Typing

open frame in the rule T-Open-Stack.

The rules T-Uninit-Closed-Stack and T-Uninit-Open-Stack mirror the rules for working with frames that always run with $\mathcal{U} = \varnothing$. The only difference is that they type with a free variable $\mathcal{U}$ rather than explictly specifying $\varnothing$.

The last two rules handle frames that remove an element from $\mathcal{U}$. These are rules that complete an initialization operation. Here, there is an extra free variable ℓ which is removed from $\mathcal{U}$. Otherwise, these rules operate like the others.

This completes the frame-stack typing, which is also the final set of rules needed to establish `config-ok`. With our configuration fully typed, the next chapter will prove the soundness of the operational semantics.

Chapter 5

Type Safety

Here, we establish type safety for our container operational semantics and show that container restrictions expressed by the type rules are indeed obeyed during execution. Given a properly typed program and a configuration initialized to Initial-Config as defined in section 4.8.1, the machine will either reach a terminal state or run forever.

Recall that the specification of the operational semantics in chapter 4 simultaneously defined the semantics as well as configuration typing rules which were segregated into boxed regions. We will write $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$ to refer to the combined relation defined by the rules of chapter 4. In general, arguments will begin with a valid configuration, a configuration typing, as well as the set of uninitialized heap locations, and will show that the subsequent state is also valid. This relation is also a function, however, we do not require determinism for this proof and we will ignore this fact.

Following the approach developed by Wright and Felleisen [20], progress and preservation theorems are presented. Note that the key novel property of this system that we are proving is the enforcement of container restrictions. This property is encoded into the configuration typing; the `within` judgment defined in the rule Containment gives us the result we want. Showing that the `config-ok` property holds establishes the containment result as well. In other words, any program that satisfies the container type rules is guaranteed to execute with correct containment.

5.1 Progress and Preservation

Theorem 5.1.1 *Progress*

If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok` *then the current frame of* $\mathcal{C}$ *will either be in a terminal state or there exists a rule to advance the configuration such that* $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \to \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$*.*

Proof: *First we make the observation that for each frame defined in figure 4.3 there is at least one rule defined in section 4.8. However, many of the rules have conditions that must be satisfied, and it must be shown that for each frame, the conditions will indeed be satisfied.*

First we'll address sub-expressions. For every frame that contains sub-expressions there exists a decomposition rule in section 4.8.8. Further, none of these rules impose any conditions. For the remainder of the frames we will discuss, it is assumed that all sub-expressions have been evaluated and will appear as results in their respective frames.

There are 6 frames that are handled by rules with no conditions: `void`, `null`, *result,* `statement-seq`, `copy-tuple` *and* `copy-discard-environment`*. The remainder of the frames will be addressed individually.*

`pop-local`*: By rule* T-POP-LOCAL *variable* $v \in \Delta$ *and by* VARS-OK v *is present in the local configuration.*

`assign-value`*: The rule* ASSIGN-VALUE-NPE *catches any null de-reference attempt. The type rule* T-ASSIGN-VALUE *asserts that the left-hand-side is a* `movable` *and a* `value` *which ensures that* ℓ_{lhs} *is a BoxLocation.* T-LOCATION *ensures* ℓ_{lhs} *and* ℓ_{rhs} *are valid heap addresses.*

`assign-ref`*: The type rule* T-ASSIGN-REF *asserts that the left-hand-side is a* `movable` *and a* `ref` *which ensures that* ℓ_{lhs} *is a RefLocation.* T-LOCATION *ensures* ℓ_{lhs} *and* ℓ_{rhs} *are valid heap addresses.*

`return`*: Either the rule* RETURN-VALUE *or* RETURN-REF *is used depending on the kind of return value and* RETURN-VALUE-NPE *will protect against null de-reference. The type rule* T-LOCATION *ensures* ℓ_{ret} *is a valid heap address.*

`field`*: The rule* FIELD-NPE *catches any null de-reference attempt. The rule* T-FIELD *requires that the field* f *exists in the class of* ℓ_{tuple}*. The rules* T-LOCATION *and* `heap-ok` *ensure that* ℓ_{tuple} *is a valid heap address referring to an object with a field* f*.*

`variable`*: The rule* T-VARIABLE *requires that the variable* v *exists in* Δ *and* VARS-OK *assures that it is also present in the local configuration.*

`call-function`*: The rule* T-FUNCTION *requires that the function* f *exists in* Γ*.*

`init`*: The rule* T-INIT *requires that the class* C *exists in* Γ*.*

`init-symbols`*: When* $T_{\texttt{(kind)}} =$ `value`*,* Init-Symbols-NPE *will protect against null dereference. Heap locations* ℓ_{init} *are valid by* T-Location*.*

`init-symbol`*: The rule* T-Init-Symbol *ensures* $v \in \Delta$ *and that there is no null dereference.*

`init-fields`*: When* $T_{\texttt{(kind)}} =$ `value`*,* Init-Fields-NPE *will protect against null dereference. Heap locations in* $\overline{\ell_{init}}$ *are valid by* T-Location*.*

`init-field`*: The rule* T-Init-Field *ensures that* f *is a field of* ℓ_{tuple} *and that there is no null de-reference.*

`copy-tuple2`*: The rule* T-Copy-Tuple-2 *ensures that* T_{dst} *is a valid type and* T-Locations *ensures that* ℓ_{src} *is valid.*

`copy-fields`*: When* $T_{\texttt{(kind)}} =$ `value`*,* Copy-Fields-NPE *will protect against null dereference. Heap locations in* $\overline{\ell_{init}}$ *are valid by* T-Location*.*

`copy-init`*: The rule* T-copy-init *ensures that* f *is a field of* ℓ_{tuple} *and that there is no null de-reference.*

Therefore, all well typed frames have corresponding rules to process them and progress is assured.

Theorem 5.1.2 *Preservation*
If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`*,* $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$
then $\Gamma; \Sigma'; \mathcal{U}' \vdash \mathcal{C}'_{\texttt{(heap)}}$ `config-ok`

Proof: *By lemma 5.2.1, the* `heap-ok` *condition is satisfied in* $\mathcal{C}'$*, and the local configuration stack is satisfied by lemma 5.3.1. Therefore,* `config-ok` *is preserved in* $\mathcal{C}'$*.*

5.2 Heap Validation Lemmas

Lemma 5.2.1 *Heap OK*
If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`*,* $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$
then $\Gamma; \Sigma'; \mathcal{U}' \vdash \mathcal{C}'$ `heap-ok`

Proof: *By lemma 5.2.3, we know that all non-modified heap locations are properly typed. Lemma 5.2.4 establishes the typing for newly allocated heap locations. And finally, lemma 5.2.6 shows that modified heap locations preserve their typing. Therefore, we know all heap locations in* $\mathcal{C}'$ *are* `location-ok`*.*

To establish that $\texttt{dom}(\mathcal{H}) = \texttt{dom}(\Sigma)$*, we observe that for the eight rules that expand the heap (* Init-Symbol-Value*,* Init-Symbol-Ref*,* Init-Field-Value*,* Init-Field-Ref*,* Copy-Tuple-2*,* copy-init-Value*,* copy-init-Ref*and* Initialize*), each makes equivalent additions to both* $\mathcal{H}$ *and* Σ*. No operation removes elements from either set.*

Lemma 5.2.2 *Immutable Heap Locations*
If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$
then $\forall\ \ell \in \mathcal{H}$ *where* $\ell \notin \mathcal{U}$ *and* $\Sigma(\ell)_{(\texttt{mobility})} = \texttt{fixed}$. $\mathcal{H}'(\ell) = \mathcal{H}(\ell)$

Proof: *We consider the two rules in the operational semantics that modify existing heap locations:* Assign-Copied *and* Assign-Ref*. Both of these rules are typed with lhs* : $T_{(\texttt{mobility})} = \texttt{movable}$*, which implies that the physical type* $\mathcal{T} = \Sigma(\ell)$ *will match with* $\mathcal{T}_{(\texttt{mobility})} = \texttt{movable}$ *since the* `ptype()` *function preserves the mobility property. Therefore, these assignment rules cannot change the values stored in* `fixed` *locations.*

Lemma 5.2.3 *Non-Interference*
If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$, $\mathcal{H} = \mathcal{C}_{(\texttt{heap})}$, $\mathcal{H}' = \mathcal{C}'_{(\texttt{heap})}$
then $\forall\ \ell \in \mathcal{H}$ *where* $\mathcal{H}'(\ell) = \mathcal{H}(\ell)$. $\Gamma; \Sigma; \mathcal{U}; \mathcal{H}' \vdash \ell$ `location-ok`

Proof: `config-ok` *implies that* ℓ *is* `location-ok` *in* $\mathcal{C}$*. Of all the conditions in* `location-ok` *across its four rules, the only one that depends on heap contents at locations other than* ℓ *is* `f-match`*. For initialized tuples, the* `f-match` *judgment was satisfied in* $\mathcal{C}$ *and continues to hold in* $\mathcal{C}'$ *by lemma 5.5.3 for all fields of* ℓ*.*

Otherwise, if $\ell \in \mathcal{U}$*, we know we are currently initializing the fields of tuple* ℓ *in an order that respects their dependencies. This means any previous field cannot depend on the contents of subsequent fields and no other heap locations are modified during initialization of a tuple. Thus,* `f-match` *must continue to hold for the initialized fields by lemma* `f-match`*.*

Therefore, for all unmodified heap locations, if the `location-ok` *judgment held in* $\mathcal{C}$*, it will continue to hold in* $\mathcal{C}'$*.*

Lemma 5.2.4 *New Locations OK*
If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$, $\mathcal{H} = \mathcal{C}_{(\texttt{heap})}$, $\mathcal{H}' = \mathcal{C}'_{(\texttt{heap})}$, $\ell \in \mathcal{H}'$ *and* $\ell \notin \mathcal{H}$
then $\Gamma; \Sigma; \mathcal{U}; \mathcal{H}' \vdash \ell$ `location-ok`

Proof: *There are eight rules in the operational semantics that introduce new heap locations. For each, we show that* `location-ok` *holds.*

Case Init-Symbol-Value

This rule allocates a new box $\ell_{\textit{dst-box}} : T$ and initializes it. By the frame typing, we know that $\langle \ell_{\textit{init}}, T_{\textit{init}} \rangle$ satisfies T-Location *and $T_{\textit{dst-box}} = T_{\textit{init}}$. Therefore, $\mathcal{T}_{\textit{init}} = \Sigma(\ell_{\textit{init}}) =$ `ptype`$(\Sigma, \mathcal{H}, \mathcal{L}, T_{\textit{init}})$ and we define $\ell_{\textit{dst-box}} \mapsto$ `ptype`$(\Sigma, \mathcal{H}, \mathcal{L}, T_{\textit{dst-box}})$ in Σ. After updating the heap with $\ell_{\textit{dst-box}} \mapsto \ell_{\textit{init}}$, we have satisfied the* `p-match` *condition of* Ref/Box-OK, *because the $T_{\textit{dst-box}} = T_{\textit{init}}$ equality makes the two invocations of* `ptype`*() identical.*

Case Init-Symbol-Ref

This rule allocates a new reference $\ell_{\textit{dst-box}} : T$ and initializes it. By the frame typing, we know that $\langle \ell_{\textit{init}}, T_{\textit{init}} \rangle$ satisfies T-Location *and $T_{\textit{init}} \widetilde{\succ} T_{\textit{init}}$. Therefore, $\mathcal{T}_{\textit{init}} = \Sigma(\ell_{\textit{init}}) =$ `ptype`$(\Sigma, \mathcal{H}, \mathcal{L}, T_{\textit{init}})$ and we define $\ell_{\textit{dst-box}} \mapsto$ `ptype`$(\Sigma, \mathcal{H}, \mathcal{L}, T_{\textit{dst-box}})$ in Σ. After updating the heap with $\ell_{\textit{dst-box}} \mapsto \ell_{\textit{init}}$, we satisfy the* `p-match` *condition of* Ref/Box-OK *by lemma 5.2.5.*

Case Init-Field-Value

This rule allocates a new box $\ell_{\textit{dst-box}} : T_{\textit{field}}$ and initializes it. By the frame typing, we know that $\langle \ell_{\textit{init}}, T_{\textit{init}} \rangle$ satisfies T-Location *and $T_{\textit{field}} = T_{\textit{init}}$. Therefore, $\mathcal{T}_{\textit{init}} = \Sigma(\ell_{\textit{init}}) =$ `ptype`$(\Sigma, \mathcal{H}, \mathcal{L}, T_{\textit{init}})$ and we define $\ell_{\textit{dst-box}} \mapsto$ `ptype`$(\Sigma, \mathcal{H}, \mathcal{L}, T_{\textit{field}})$ in Σ. After updating the heap with $\ell_{\textit{dst-box}} \mapsto \ell_{\textit{init}}$, we have satisfied the* `p-match` *condition of* Ref/Box-OK *because $T_{\textit{field}} = T_{\textit{init}}$ makes the two invocations of* `ptype`*() identical.*

Case Init-Field-Ref

This rule allocates a new reference $\ell_{\textit{dst-ref}}$ and initializes it. By the frame typing, we know that $\langle \ell_{\textit{init}}, T_{\textit{init}} \rangle$ satisfies T-Location *and $T_{\textit{field}} \widetilde{\succ} T_{\textit{init}}$. Therefore, $\mathcal{T}_{\textit{init}} = \Sigma(\ell_{\textit{init}}) =$ `ptype`$(\Sigma, \mathcal{H}, \mathcal{L}, T_{\textit{init}})$ and we define $\ell_{\textit{dst-box}} \mapsto$ `ptype`$(\Sigma, \mathcal{H}, \mathcal{L}, T_{\textit{field}})$ in Σ. After updating the heap with $\ell_{\textit{dst-box}} \mapsto \ell_{\textit{init}}$, we satisfy the* `p-match` *condition of* Ref/Box-OK *by lemma 5.2.5.*

Case Copy-Tuple-2

The Copy-Tuple-2 *rule creates a new uninitialized tuple at location $\ell_{\textit{new}}$, which trivially satisfies* Object-OK-Uninit *because currently no fields of this tuple have been initialized yet.*

Case Copy-Init-Value

This rule allocates a new box $\ell_{\textit{dst-box}} : T_{\textit{field}}$ and initializes it. By the frame typing, we know that $\langle \ell_{\textit{init}}, T_{\textit{init}} \rangle$ satisfies T-Location *and $T_{\textit{field}} = T_{\textit{init}}$. Therefore $\mathcal{T} = \Sigma(\ell_{\textit{init}}) =$ `ptype`$(\Sigma, \mathcal{H}, \mathcal{L}, T_{\textit{init}})$ and we define $\ell_{\textit{dst-box}} \mapsto$ `ptype`$(\Sigma, \mathcal{H}, \mathcal{L}, T_{\textit{field}})$ in Σ. After updating* *the heap with* $\ell_{\textit{dst-box}} \mapsto \ell$ *we have satisfied the* `p-match` *condition of* Ref/Box-OK *because* $T_{\textit{field}} = T_{\textit{init}}$ *makes the 2 invocations of* `ptype`*() identical.*

Case Copy-Init-Ref

This rule allocates a new reference field $\ell_{\textit{dst-ref}} : T_{\textit{field}}$ *and initializes it. By the frame typing, we know that* $\langle \ell_{\textit{init}}, T_{\textit{init}} \rangle$ *satisfies* T-Location *and that* $T_{\textit{field}} \tilde{\succ} T_{\textit{init}}$*. Therefore,* $\mathcal{T} = \Sigma(\ell) = \texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}, T_{\textit{init}})$ *and we define* $\ell_{\textit{dst-box}} \mapsto \texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}, T_{\textit{field}})$ *in* Σ*. After updating the heap with* $\ell_{\textit{dst-ref}} \mapsto \ell$*, we have satisfied the* `p-match` *condition of* Ref/Box-OK *by lemma 5.2.5.*

Case Initialize

This rule adds a new uninitialized tuple to the heap. Because the tuple has all fields uninitialized, the Object-OK-Uninit *rule will be trivially satisfied.*

Lemma 5.2.5 Ref-Initializable *Implies* `p-match`
If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok` *and two types:* $T_{\textit{src}}$ *and* $T_{\textit{dst}}$*, defined within scope* $\mathcal{L} \in \mathcal{C}$ *such that* $T_{\textit{dst}} \tilde{\succ} T_{\textit{src}}$ *and* $T_{\textit{dst}(\texttt{kind})} = \texttt{ref}$ *then the associated physical types will match;* $\mathcal{T}_{\textit{dst}}$ `p-match` $\mathcal{T}_{\textit{src}}$*.*

Proof: *With* $\mathcal{T}_{\textit{dst}} = \texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}, T_{\textit{dst}})$ *and* $\mathcal{T}_{\textit{src}} = \texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}, T_{\textit{src}})$*, we know that the* `ptype`*() function preserves the class and mutability sub-components which have the same requirements in* Ref-Initializable *and* Physical-Match*. Finally, the container-labels must either match resulting in matching* `ptype`*() output or* $T_{\textit{dst}(\texttt{label})} = \texttt{unknown}$ *which maps to* `unknown-cont` *and satisfies* `p-match`*. Therefore,* $T_{\textit{dst}} \tilde{\succ} T_{\textit{src}}$ *implies* $\mathcal{T}_{\textit{dst}}$ `p-match` $\mathcal{T}_{\textit{src}}$*.*

Lemma 5.2.6 *Modified Locations OK*
If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`*,* $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$*,* $\mathcal{H} = \mathcal{C}_{(\texttt{heap})}$*,* $\mathcal{H}' = \mathcal{C}'_{(\texttt{heap})}$*,* $\ell \in \mathcal{H}$*,* $\mathcal{H}'(\ell) \neq \mathcal{H}(\ell)$
then $\Gamma; \Sigma'; \mathcal{U}'; \mathcal{H}' \vdash \ell$ `location-ok`

Proof: *By cases across the eight rules that make heap modifications; for each case, we show that* `location-ok` *holds for* ℓ*.*

Cases Init-Field-Value, Init-Field-Ref, copy-init-Value and copy-init-Ref

These four cases are all proved by the same argument; the proof for Init-Field-Value *is presented.*

This rule updates the uninitialized heap location $\ell_{\textit{tuple}}$ *to add a new field* f *located at* $\ell_{\textit{dst-box}}$*, which we know is* `location-ok` *by lemma 5.2.4. We have* $\mathcal{T} = \Sigma(\ell_{\textit{dst-box}}) =$

`ptype`$(\Sigma, \mathcal{H}', \mathcal{L}, T_{field})$ *as the right-hand side of the* `f-match` *condition in rule* Object-OK-Uninit *for this new field f. Now, the sub-condition* `t-match` *requires that* `ptype`$(\Sigma, \mathcal{H}', \mathcal{L}, T_{field})$ `p-match` $\mathcal{T}$*, which is satisfied because the LHS is identical to the RHS.*

All other fields in ℓ_{tuple} are unchanged, satisfied `f-match` *in $\mathcal{H}$ and will continue to satisfy* `f-match` *in $\mathcal{H}'$ by lemma 5.5.3. Therefore, all conditions of* Object-OK-Uninit *are satisfied and* `location-ok` *holds.*

Case Assign-Ref

For Assign-Ref*, we know that if the RHS is null then* `location-ok` *is trivially satisfied by rule* Null-Ref-OK*. If non-null then* Ref/Box-OK *is the relevant rule to establish* `location-ok` *for our modified heap location ℓ_{lhs}. We must show $\mathcal{T}_{lhs}$* `p-match` *$\mathcal{T}_{rhs}$. From the type rule* T-Assign-Ref*, we know that $T_{lhs} \widehat{\succ} T_{rhs}$. Further, from the typing of the* Assign-Ref *frame, we know that $\langle \ell_{lhs}, T_{lhs} \rangle$ and $\langle \ell_{rhs}, T_{rhs} \rangle$ satisfy the* T-Location *rule. Therefore, $\mathcal{T}_{lhs} = \Sigma(\ell_{lhs}) =$* `ptype`$(\Sigma, \mathcal{H}, \mathcal{L}, T_{lhs})$ *and $\mathcal{T}_{rhs} = \Sigma(\ell_{rhs}) =$* `ptype`$(\Sigma, \mathcal{H}, \mathcal{L}, T_{rhs})$*. After updating the heap with $\ell_{lhs} \mapsto \ell_{rhs}$, we can apply lemma 5.4.1, and we have satisfied the* `p-match` *condition of* Ref/Box-OK*.*

Case Assign-Copied

In this case, we must satisfy Ref/Box-OK*. From the type rule* T-Assign-Copied*, we know that $T_{lhs} = T_{rhs}$ and $\langle \ell_{lhs}, T_{lhs} \rangle$ and $\langle \ell_{rhs}, T_{rhs} \rangle$ satisfy the* T-Location *rule. Therefore, $\mathcal{T}_{lhs} = \Sigma(\ell_{lhs}) =$* `ptype`$(\Sigma, \mathcal{H}, \mathcal{L}, T_{lhs})$*, and for the left-hand side $\mathcal{T}_{rhs} = \Sigma(\ell_{rhs}) =$* `ptype`$(\Sigma, \mathcal{H}, \mathcal{L}, T_{rhs})$*. After updating the heap with $\ell_{lhs} \mapsto \ell_{rhs}$, we have satisfied the* `p-match` *condition of* Ref/Box-OK*, because the $T_{lhs} = T_{rhs}$ equality makes the two invocations of* `ptype`*() identical.*

Cases Init-Fields-Complete and Copy-Complete

These two cases simply remove ℓ_{tuple} from $\mathcal{U}$. With all fields populated, the check for Object-OK-Uninit *covers every field and is identical to* Object-OK*.*

5.3 Frame-Stack Lemmas

Lemma 5.3.1 *Local Configuration Stack OK*

If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$, $\mathcal{C} = \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \mathcal{CF} \end{pmatrix}$, $\mathcal{C}' = \begin{pmatrix} \mathcal{H}' \\ \mathcal{L}' \circ \overline{\mathcal{L}}' \\ \mathcal{CF}' \end{pmatrix}$

then $\Gamma; \Sigma; \mathcal{H} \vdash \mathcal{L}'[\texttt{fstack} \mapsto \mathcal{CF}' \circ \mathcal{L}_{(\texttt{fstack})}] \circ \overline{\mathcal{L}}' : (\mathcal{T} \rightarrow \mathcal{T}'', \mathcal{U} \rightarrow \varnothing)$

Proof: *The conditions of* T-Local-Config-Stack *are established in two lemmas. The* `cmap-ok` *condition is satisfied by lemma 5.3.2. For the frame-stack typing and* `vars-ok`*, the combined consideration of the frame-stack typing of $\mathcal{L}'$ as well as the tail of the local configuration stack are satisfied by lemma 5.3.3.*

Lemma 5.3.2 *Container Map OK*

If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U}\rangle \to \langle \mathcal{C}', \Sigma', \mathcal{U}'\rangle$, $\mathcal{C} = \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \mathcal{CF} \end{pmatrix}$, $\mathcal{C}' = \begin{pmatrix} \mathcal{H}' \\ \mathcal{L}' \circ \overline{\mathcal{L}}' \\ \mathcal{CF}' \end{pmatrix}$

then $\Sigma; \mathcal{H}' \vdash \mathtt{head}(\overline{\mathcal{L}'})$ `cmap-ok` $\mathcal{L}'$

Proof: *By lemma 5.5.5, the* `cmap-ok` *condition will continue to be satisfied for existing pairs of local configurations in the stack. To cover the remaining cases, we now consider the three operations that add new scopes to the stack.*

Call-Function
With $\mathcal{L}_{callee} = \mathtt{funcLocalConfig()}$, *we have* $\mathcal{L}' \circ \overline{\mathcal{L}}' = \mathcal{L}_{callee} \circ \mathcal{L} \circ \overline{\mathcal{L}}$

We can see that $\Sigma; \mathcal{H} \vdash \mathcal{L}$ `cmap-ok` $\mathcal{L}_{callee}$ *by first observing that* `lmap-ok` *is satisfied by the typing of* `call`() *and that* $\mathtt{dom}(\phi) = \mathtt{dom}(\mathcal{M})$ *by construction.*

To establish the container equality condition, we can see that the mapping $\mathcal{M} = \Sigma_{(\mathtt{cmap})}(\mathcal{L}_{fn(\mathtt{id})})$ *is explicitly set by the* `labelMappings`() *function to satsify this condition of* `cmap-ok`*. By definition,* $\forall \theta \in \mathtt{dom}(\mathcal{M}).\ \mathcal{M}(\theta) = \mathtt{labelToCont}(\Sigma, \mathcal{H}, \mathcal{L}, \mathtt{mapLabel}(\phi, \theta))$ *which is the right-hand side of the container equality condition in rule* CMap-OK*. On the left-hand side, we have* $\mathtt{labelToCont}(\Sigma, \mathcal{H}, \mathcal{L}_{callee}, \theta)$*. Examining the implementation of* `labelToCont`()*, we can see that it returns* $\mathcal{M}(\theta)$ *for mapped labels, which as we've seen was defined as* $\mathtt{labelToCont}(\Sigma, \mathcal{H}, \mathcal{L}, \mathtt{mapLabel}(\phi, \theta))$*, thus showing that the two sides of the container equality condition are equivalent. Therefore,* Call-Function *satisfies all of the conditions of* `cmap-ok`*.*

Initialize *and* Copy-Tuple-2
The reasoning for these two cases is identical; Initialize *is presented.*
With $\mathcal{L}_{tuple} = \mathtt{tupleLocalConfig()}$, *we have* $\mathcal{L}' \circ \overline{\mathcal{L}}' = \mathcal{L}_{tuple} \circ \mathcal{L} \circ \overline{\mathcal{L}}$

We can see that $\Sigma; \mathcal{H} \vdash \mathcal{L}$ `cmap-ok` $\mathcal{L}_{callee}$ *by observing that* `lmap-ok` *is satisfied by the typing of* `init`() *and that* $\mathtt{dom}(\phi) = \mathtt{dom}(\mathcal{M})$ *by construction.*

The logic for satisfying the container equality condition is the same as it was for the Call-Function *case. Briefly, the mapping* $\mathcal{M} = \Sigma_{(\mathtt{cmap})}(\mathcal{L}_{tuple(\mathtt{id})})$ *is explicitly set by the* `tupleMappings`() *function such that label-matching condition of* `cmap-ok` *is satisfied. Therefore, rules* Initialize *and* Copy-Tuple-2 *satisfy all the conditions of* `cmap-ok`*.*

Lemma 5.3.3 *Local Configuration Frame-Stack OK*

If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}\ \texttt{config-ok}$, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \to \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$, $\mathcal{C} = \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \mathcal{CF} \end{pmatrix}$, $\mathcal{C}' = \begin{pmatrix} \mathcal{H}' \\ \mathcal{L}' \circ \overline{\mathcal{L}}' \\ \mathcal{CF}' \end{pmatrix}$

then $\Gamma; \Sigma'; \mathcal{H}' \vdash \mathcal{L}'[\texttt{fstack} \mapsto \mathcal{CF}' \circ \mathcal{L}'_{(\texttt{fstack})}] : (\texttt{Void} \to \mathcal{T}, \mathcal{U} \to \mathcal{U}')$ *and* $\Gamma; \Sigma'; \mathcal{H}' \vdash \overline{\mathcal{L}}' : (\mathcal{T} \to \mathcal{T}', \mathcal{U}' \to \varnothing)$

Proof: *We prove this lemma by parts and we'll group rules of the operational semantics by how they modify the frame-stack.*

First, lemma 5.3.4 proves our claim for operations that add a new local configuration to the stack. This includes function calls, which is the most complex case and directly demonstrates the management of container labels and containers across scopes.

Similarly, lemma 5.3.5 proves our claim for operations that remove local configurations from the stack. It shows that containers are managed properly for returned values.

We address a simpler group of rules in lemma 5.3.6, where operations make changes to to the current frame-stack, but do not create or remove scopes from the local configuration stack.

Finally, the simplest cases are proven by lemma 5.3.7. Here, no changes are made to the local configuration stack.

The combination of the four lemmas above proves the entire claim.

Lemma 5.3.4 *Local Configuration Frame-Stack OK (open scope)*

If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}\ \texttt{config-ok}$, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \to \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$, $\mathcal{C} = \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \mathcal{CF} \end{pmatrix}$, $\mathcal{C}' = \begin{pmatrix} \mathcal{H}' \\ \mathcal{L}' \circ \mathcal{L} \circ \overline{\mathcal{L}} \\ \mathcal{CF}' \end{pmatrix}$,
$\Gamma; \Sigma; \mathcal{H} \vdash \mathcal{L} \circ \overline{\mathcal{L}} : (\mathcal{T} \to \mathcal{T}', \mathcal{U}' \to \varnothing)$

then $\Gamma; \Sigma'; \mathcal{H}' \vdash \mathcal{L}'[\texttt{fstack} \mapsto \mathcal{CF}' \circ \mathcal{L}'_{(\texttt{fstack})}] : (\texttt{Void} \to \mathcal{T}, \mathcal{U} \to \mathcal{U}')$ *and* $\Gamma; \Sigma'; \mathcal{H}' \vdash \mathcal{L} \circ \overline{\mathcal{L}} : (\mathcal{T} \to \mathcal{T}', \mathcal{U}' \to \varnothing)$

Proof: *The type of* $\mathcal{L} \circ \overline{\mathcal{L}}$ *is preserved in* $\mathcal{C}'$ *by lemma 5.5.7.*

To pass values across scopes, our reasoning must also take into account that the determination of physical types from logical types depends on scope. Each local configuration has a mapping ϕ *to map container labels to equivalent labels in a parent scope as well as an associated mapping* $\mathcal{M}$ *which is used by* $\texttt{ptype}()$ *to determine the physical type. When passing a parameter of type* T_1 *in* $\mathcal{L}_{caller}$ *to a child scope as type* T_2 *in* $\mathcal{L}_{callee}$*, we must show that* $\texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}_{caller}, T_1) = \texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}_{callee}, T_2)$.

New scopes do not contain any variables, so they trivially satisfy `vars-ok`.

We prove the type of $\mathcal{L}'[\texttt{fstack} \mapsto \mathcal{CF}' \circ \mathcal{L}'_{(\texttt{fstack})}]$ *by cases.*

Call-Function
We must determine the type of $\mathcal{L}'[\texttt{fstack} \mapsto \mathcal{CF}' \circ \mathcal{L}'_{(\texttt{fstack})}]$ *where*
$\mathcal{CF}' \circ \mathcal{L}'_{(\texttt{fstack})} = \{\texttt{init-symbols}(), s\}$

For the new scope, we require $\mathcal{L}$ `cmap-ok` $\mathcal{L}'$*, and lemma 5.3.2 establishes this judgment. By the typing of* T-Call-Function *and* V-Function*, we know that frame* `init-symbols()` *types as* `Void`*, since the input parameters will match by lemma 5.4.4. Further, from the call typing, we know the function body frame* s *in* $\mathcal{L}_{callee}$ *will evaluate to* T_{ret}*. We have*

$\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \texttt{call-function}(\dots) : \texttt{exportType}(\phi, T_{ret})$
$\phi = \mathcal{L}'_{(\texttt{lmap})}$
$\mathcal{T} = \texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}, \texttt{exportType}(\phi, T_{ret}))$
$\mathcal{L}' = \texttt{funcLocalConfig}(\dots)$
$\mathcal{T}_{ret} = \texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}', T_{ret})$

To establish $\mathcal{T} = \mathcal{T}_{ret}$*, we apply lemma 5.4.2. Therefore, the new frame-stack has the required type of* $\Gamma; \Sigma'; \mathcal{H}' \vdash \mathcal{L}'[\texttt{fstack} \mapsto \mathcal{CF}' \circ \mathcal{L}'_{(\texttt{fstack})}] : (\texttt{Void} \to \mathcal{T}, \varnothing \to \varnothing)$ *with* $\mathcal{U}$ *and* $\mathcal{U}'$ *empty.*

Initialize *and* Copy-Tuple-2
The proofs of these two cases follow the same reasoning; Initialize *is presented.*

We must determine the type of $\mathcal{L}'[\texttt{fstack} \mapsto \mathcal{CF}' \circ \mathcal{L}'_{(\texttt{fstack})}]$ *where*
$\mathcal{CF}' \circ \mathcal{L}'_{(\texttt{fstack})} = \{\texttt{init-symbols}()\}$

For the new scope, we require $\mathcal{L}$ `cmap-ok` $\mathcal{L}'$*, and lemma 5.3.2 establishes this judgment. By the typing of* T-init*, we know that frame* `init-fields()` *types as* `Void`*, since the input parameters will match by lemma 5.4.4. We have*

$\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \texttt{init}(\dots) :$
$T_{new} = \langle \texttt{ref}, \texttt{container-of-tuple}, \theta, \texttt{mutable}, \texttt{movable} \rangle$
$\phi = \mathcal{L}'_{(\texttt{lmap})}$
$\mathcal{T} = \texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}, \texttt{exportType}(\phi, T_{new}))$
$\mathcal{L}' = \texttt{tupleLocalConfig}(\dots)$
$\mathcal{T}_{ret} = \texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}', T_{new})$

To establish $\mathcal{T} = \mathcal{T}_{new}$*, we apply lemma 5.4.2. Therefore, the new frame-stack has the required type of* $\Gamma; \Sigma'; \mathcal{H}' \vdash \mathcal{L}'[\texttt{fstack} \mapsto \mathcal{CF}' \circ \mathcal{L}'_{(\texttt{fstack})}] : (\texttt{Void} \to \mathcal{T}, \mathcal{U} \to \mathcal{U} \cup \{\ell_{new}\})$.

Lemma 5.3.5 *Local Configuration Frame-Stack OK (close scope)*
If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ $\texttt{config-ok}$, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \to \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$, $\mathcal{C} = \begin{pmatrix} \mathcal{H} \\ \mathcal{L}_{pop} \circ \mathcal{L} \circ \overline{\mathcal{L}} \\ \mathcal{CF} \end{pmatrix}$, $\mathcal{C}' = \begin{pmatrix} \mathcal{H}' \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \mathcal{CF}' \end{pmatrix}$
then $\Gamma; \Sigma'; \mathcal{H}' \vdash \mathcal{L}[\texttt{fstack} \mapsto \mathcal{CF}' \circ \mathcal{L}_{\texttt{(fstack)}}] : (\texttt{Void} \to \mathcal{T}, \mathcal{U} \to \mathcal{U}')$ *and*
$\Gamma; \Sigma'; \mathcal{H}' \vdash \overline{\mathcal{L}} : (\mathcal{T} \to \mathcal{T}', \mathcal{U}' \to \varnothing)$

Proof: *The type of* $\overline{\mathcal{L}}$ *is preserved in* $\mathcal{C}'$ *by lemma 5.5.7.*

To return a value to the calling scope our reasoning must also take into account that the determination of physical types from logical types depends on scope. Each local configuration has a mapping ϕ *from container labels to equivalent labels in a parent scope as well as an associated mapping* $\mathcal{M}$ *which is used by* $\texttt{ptype}()$ *to determine the physical type. When returning a construct of type* T_1 *in* $\mathcal{L}_{callee}$ *to the parent scope as type* T_2 *in* $\mathcal{L}_{caller}$*, we must show that* $\texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}_{callee}, T_1) = \texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}_{caller}, T_2)$.

Return-Value *and* Return-Reference
We must determine the type of $\mathcal{L}[\texttt{fstack} \mapsto \mathcal{CF}' \circ \mathcal{L}_{\texttt{(fstack)}}]$ *where*
$\mathcal{CF}' \circ \mathcal{L}_{\texttt{(fstack)}} = \langle \ell_{ret}, \texttt{exportType}(\phi, T_{ret}) \rangle \rangle \circ \mathcal{L}$ *and* $\phi = \mathcal{L}_{pop\texttt{(lmap)}}$.

From T-Local-Config-Stack *we know that* $\mathcal{L}_{pop}$ $\texttt{cmap-ok}$ $\mathcal{L}$ *and the returned value will match the expected input to* $\mathcal{L}$*, preserving the type of the new frame-stack. We have*

$\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L}_{pop} \vdash \texttt{return}(\dots) : T_{ret}$
$\mathcal{T}_{ret} = \texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}_{pop}, T_{ret})$
$\mathcal{T} = \texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}, \texttt{exportType}(\phi, T_{ret}))$

To establish $\mathcal{T} = \mathcal{T}_{ret}$*, we apply lemma 5.4.2. Therefore, the new frame-stack has the required type of* $\Gamma; \Sigma'; \mathcal{H}' \vdash \mathcal{L}[\texttt{fstack} \mapsto \mathcal{CF}' \circ \mathcal{L}_{\texttt{(fstack)}}] : (\texttt{Void} \to \mathcal{T}, \varnothing \to \varnothing)$.

Copy-Complete *and* Init-Fields-Complete
We must determine the type of $\mathcal{L}[\texttt{fstack} \mapsto \mathcal{CF}' \circ \mathcal{L}_{\texttt{(fstack)}}]$ *where*
$\mathcal{CF}' \circ \mathcal{L}_{\texttt{(fstack)}} = \langle \ell_{tuple}, \texttt{exportType}(\phi, T_{tuple}) \rangle \rangle \circ \mathcal{L}$ *and* $\phi = \mathcal{L}_{pop\texttt{(lmap)}}$.

From T-Local-Config-Stack*, we know that* $\mathcal{L}_{pop}$ $\texttt{cmap-ok}$ $\mathcal{L}$ *and the returned value will match the expected input to* $\mathcal{L}$*, preserving the type of the new frame-stack. We have*

$\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L}_{pop} \vdash \texttt{copy-fields}(\dots) : T_{tuple}$
$\mathcal{T}_{tuple} = \texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}_{pop}, T_{tuple})$
$\mathcal{T} = \texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}, \texttt{exportType}(\phi, T_{tuple}))$

To establish $\mathcal{T} = \mathcal{T}_{tuple}$*, we apply lemma 5.4.2. Therefore, the new frame-stack has the required type of* $\Gamma; \Sigma'; \mathcal{H}' \vdash \mathcal{L}[\texttt{fstack} \mapsto \mathcal{CF}' \circ \mathcal{L}_{\texttt{(fstack)}}] : (\texttt{Void} \to \mathcal{T}, \mathcal{U} \cup T_{tuple} \to \mathcal{U})$.

Lemma 5.3.6 *Local Configuration Frame-Stack OK (same scope)*

If $\Gamma;\Sigma;\mathcal{U}\vdash\mathcal{C}$ $\texttt{config-ok}$, $\langle\mathcal{C},\Sigma,\mathcal{U}\rangle\rightarrow\langle\mathcal{C}',\Sigma',\mathcal{U}'\rangle$, $\mathcal{C}=\begin{pmatrix}\mathcal{H}\\ \mathcal{L}\circ\overline{\mathcal{L}}\\ \mathcal{CF}\end{pmatrix}$, $\mathcal{C}'=\begin{pmatrix}\mathcal{H}'\\ \mathcal{L}'\circ\overline{\mathcal{L}}\\ \mathcal{CF}'\end{pmatrix}$,
$\mathcal{L}_{(\texttt{id})}=\mathcal{L}'_{(\texttt{id})}$
then $\Gamma;\Sigma';\mathcal{H}'\vdash\mathcal{L}'[\texttt{fstack}\mapsto\mathcal{CF}'\circ\mathcal{L}'_{(\texttt{fstack})}]:(\texttt{Void}\rightarrow\mathcal{T},\mathcal{U}\rightarrow\mathcal{U}')$ *and*
$\Gamma;\Sigma';\mathcal{H}'\vdash\overline{\mathcal{L}}:(\mathcal{T}\rightarrow\mathcal{T}',\mathcal{U}'\rightarrow\varnothing)$

Proof: *The set of rules we are considering here modify the current local frame-stack, but do not create new scopes or modify parent scopes. By lemma 5.5.7, we know that the type of $\overline{\mathcal{L}}$ is preserved, as well as the type of each individual unmodified frame by lemma 5.5.9. We must show the type of $\mathcal{CF}\circ\mathcal{L}_{(\texttt{fstack})}$ is the same as the type of $\mathcal{CF}'\circ\mathcal{L}'_{(\texttt{fstack})}$ so that* T-Local-Config-Stack *retains the same typing.*

When reasoning about the type of the frame-stack, we break the stack into a modified portion and a stable tail portion as follows; $\mathcal{CF}\circ\mathcal{L}=\overline{\mathcal{F}_{old}}\circ\overline{\mathcal{F}_{tail}}$ *and* $\mathcal{CF}'\circ\mathcal{L}'=\overline{\mathcal{F}_{new}}\circ\overline{\mathcal{F}_{tail}}$*. The argument in each of the cases below is that the type of* $\overline{\mathcal{F}_{old}}$ *is the same as* $\overline{\mathcal{F}_{new}}$*. Then, by lemma 5.5.8,* $\overline{\mathcal{F}_{tail}}$ *preserves its type. With these two results combined, we have completed the typing of* $\mathcal{CF}'\circ\mathcal{L}'_{(\texttt{fstack})}$ *in* $\mathcal{C}'$ *and show it to be the same as in* $\mathcal{C}$*.*

Where the typing of the new frames is non-obvious, there will be further justification based on the conditions of the relevant type rule.

Next-Frame
$\overline{\mathcal{F}_{old}}=\{\texttt{void}\}:(\texttt{Void}\rightarrow\texttt{Void},\mathcal{U}\rightarrow\mathcal{U})$
$\overline{\mathcal{F}_{new}}=\{\}:(\texttt{Void}\rightarrow\texttt{Void},\mathcal{U}\rightarrow\mathcal{U})$

Consume-Result
$\overline{\mathcal{F}_{old}}=\{result,\mathcal{OF}\}:(\texttt{Void}\rightarrow\mathcal{T},\mathcal{U}\rightarrow\mathcal{U})$
$\overline{\mathcal{F}_{new}}=\{\mathcal{OF}[\Box\mapsto result]\}:(\texttt{Void}\rightarrow\mathcal{T},\mathcal{U}\rightarrow\mathcal{U})$
The resulting closed frame will be well typed because its components inherited from the open frame were well typed and the substituted result has the same typing as $\Box$*.*

Pop-Local
$\overline{\mathcal{F}_{old}}=\{\texttt{pop-local}^{\Delta\cup v}(v)\}:(\texttt{Void}\rightarrow\texttt{Void},\varnothing\rightarrow\varnothing)$
$\overline{\mathcal{F}_{new}}=\{\texttt{void}^{\Delta}(v)\}:(\texttt{Void}\rightarrow\texttt{Void},\varnothing\rightarrow\varnothing)$
`pop-local()` *will remove* v *from* $\mathcal{L}_{(\texttt{vars})}$ *and we know the corresponding change is also made to* Δ *from the previous frame-stack typing. The remainder of the conditions in* `vars-ok` *are as they were in* $\mathcal{C}$*, thus* `vars-ok` *is preserved in* $\mathcal{C}'$*.*

Let
$\overline{\mathcal{F}_{old}}=\{\texttt{let}^{\Delta}(v)\}:(\texttt{Void}\rightarrow\texttt{Void},\varnothing\rightarrow\varnothing)$

$\overline{\mathcal{F}_{new}} = \{\texttt{init-symbol}^{\Delta}(v), s^{\Delta\cup v}, \texttt{pop-local}^{\Delta\cup v}(v)\} : (\texttt{Void} \rightarrow \texttt{Void}, \varnothing \rightarrow \varnothing)$
The new frame's components are all extracted from well typed components from the original `let` *frame. Frame* `init-symbol()` *will add* v *to* $\mathcal{L}_{\texttt{(vars)}}$ *and* `pop-local(`v`)` *will remove it, leaving the* `vars-ok` *conditions as they were in* $\mathcal{C}$*. Thus* `vars-ok` *is preserved in* $\mathcal{C}'$*.*

Assign-Value
$\overline{\mathcal{F}_{old}} = \{\texttt{assign-value}()\} : (\texttt{Void} \rightarrow \texttt{Void}, \mathcal{U} \rightarrow \mathcal{U})$
$\overline{\mathcal{F}_{new}} = \{\texttt{copy-tuple}(), \texttt{assign-copied}(\square)\} : (\texttt{Void} \rightarrow \texttt{Void}, \mathcal{U} \rightarrow \mathcal{U})$
The new frame's components are all extracted from well typed components from the original `assign-value` *frame.* `assign-copied()` *consumes the result from* `copy-tuple()` *resulting in a frame of type* `Void`*.*

Init-Symbols-Value
$\overline{\mathcal{F}_{old}} = \{\texttt{init-symbols}()\} : (\texttt{Void} \rightarrow \texttt{Void}, \varnothing \rightarrow \varnothing)$
$\overline{\mathcal{F}_{new}} = \{\texttt{copy-tuple}(), \texttt{init-symbol}(\square), \texttt{init-symbols}()\} : (\texttt{Void} \rightarrow \texttt{Void}, \varnothing \rightarrow \varnothing)$
The new frame's components are all extracted from well typed components from the original `init-symbols` *frame.* `init-symbol()` *consumes the result from* `copy-tuple()`*, resulting in a frame of type* `Void`*. The next* `init-symbols()` *is also typed as* `Void`*, so the new stack retains its type.*

Init-Symbols-Ref
$\overline{\mathcal{F}_{old}} = \{\texttt{init-symbols}()\} : (\texttt{Void} \rightarrow \texttt{Void}, \varnothing \rightarrow \varnothing)$
$\overline{\mathcal{F}_{new}} = \{\texttt{init-symbol}(), \texttt{init-symbols}()\} : (\texttt{Void} \rightarrow \texttt{Void}, \varnothing \rightarrow \varnothing)$
The new frame's components are all extracted from well typed components from the original `init-symbols` *frame.*

Init-Symbol-Value *and* Init-Symbol-Ref
$\overline{\mathcal{F}_{old}} = \{\texttt{init-symbol}(v)\} : (\texttt{Void} \rightarrow \texttt{Void}, \varnothing \rightarrow \varnothing)$
$\overline{\mathcal{F}_{new}} = \{\texttt{void}^{\Delta\cup v}\} : (\texttt{Void} \rightarrow \texttt{Void}, \varnothing \rightarrow \varnothing)$
`init-symbol()` *will add* v *to* $\mathcal{L}_{\texttt{(vars)}}$ *in agreement with the frame typing of* `init-symbol`*. The remainder of the* `vars-ok` *conditions are as they were in* $\mathcal{C}$*, thus* `vars-ok` *is preserved in* $\mathcal{C}'$*.*

Init-Fields-Value
$\overline{\mathcal{F}_{old}} = \{\texttt{init-fields()}\} : (\texttt{Void} \rightarrow \texttt{Void}, \mathcal{U} \rightarrow \mathcal{U})$
$\overline{\mathcal{F}_{new}} = \{\texttt{copy-tuple}(), \texttt{init-field}(\square), \texttt{init-fields}()\} : (\texttt{Void} \rightarrow \texttt{Void}, \mathcal{U} \rightarrow \mathcal{U})$
The new frame's components are all extracted from well typed components from the original `init-fields` *frame.* `init-field()` *consumes the result from* `copy-tuple()` *resulting in a frame of type* `Void`*, and the next* `init-fields()` *is also typed as* `Void`*, so the new stack retains its type.*

Init-Fields-Ref

$\overline{\mathcal{F}_{old}} = \{\texttt{init-fields}()\} : (\texttt{Void} \rightarrow \texttt{Void}, \mathcal{U} \rightarrow \mathcal{U})$
$\overline{\mathcal{F}_{new}} = \{\texttt{init-field}(), \texttt{init-fields}()\} : (\texttt{Void} \rightarrow \texttt{Void}, \mathcal{U} \rightarrow \mathcal{U})$
The new frame's components are all extracted from well typed components from the original `init-fields` *frame.*

Copy-Tuple
$\overline{\mathcal{F}_{old}} = \{\texttt{copy-tuple}()\} : (\texttt{Void} \rightarrow \mathcal{T}, \varnothing \rightarrow \varnothing)$
$\overline{\mathcal{F}_{new}} = \{\texttt{copy-tuple2}(), \texttt{discard-copy-env}()\} : (\texttt{Void} \rightarrow \mathcal{T}, \varnothing \rightarrow \varnothing)$
The frame typing conditions of `copy-tuple2` *are satisfied by the typing of* `copy-tuple`*. After simplifying the result of* `copy-tuple2()` *using* `discard-copy-env()`*, the typing matches the original and the new stack retains its type.*

Copy-Value
$\overline{\mathcal{F}_{old}} = \{\texttt{copy-fields()}\} : (\texttt{Void} \rightarrow \texttt{Void}, \mathcal{U} \rightarrow \mathcal{U})$
$\overline{\mathcal{F}_{new}} = \{\texttt{copy-tuple2}(), \texttt{copy-init}(\square), \texttt{copy-fields}()\} : (\texttt{Void} \rightarrow \texttt{Void}, \mathcal{U} \rightarrow \mathcal{U})$
The new frame's components are all extracted from well typed components from the original `copy-fields` *frame.* `copy-init()` *consumes the result from* `copy-tuple()`*, resulting in a frame of type* `Void`*, and the next* `copy-fields()` *is also typed as* `Void`*.* `copy-tuple2()` *adds location ℓ_{new} to $\mathcal{U}$ and* `copy-fields()` *removes it, therefore the new stack retains its type.*

Copy-Ref
$\overline{\mathcal{F}_{old}} = \{\texttt{copy-fields}()\} : (\texttt{Void} \rightarrow \texttt{Void}, \mathcal{U} \rightarrow \mathcal{U})$
$\overline{\mathcal{F}_{new}} = \{\texttt{copy-init}(), \texttt{copy-fields}()\} : (\texttt{Void} \rightarrow \texttt{Void}, \mathcal{U} \rightarrow \mathcal{U})$
The new frame's components are all extracted from well typed components from the original `init-fields` *frame.*

Lemma 5.3.7 *Local Configuration Frame-Stack OK (simple replacement)*
If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}\ \texttt{config-ok}$, $\langle \mathcal{C}, \Sigma, \mathcal{U}\rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}'\rangle$, $\mathcal{C} = \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \mathcal{CF} \end{pmatrix}$, $\mathcal{C}' = \begin{pmatrix} \mathcal{H}' \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \mathcal{CF}' \end{pmatrix}$
then $\Gamma; \Sigma'; \mathcal{H}' \vdash \mathcal{L}'[\texttt{fstack} \mapsto \mathcal{CF}' \circ \mathcal{L}'_{(\texttt{fstack})}] : (\texttt{Void} \rightarrow \mathcal{T}, \mathcal{U} \rightarrow \mathcal{U}')$ *and*
$\Gamma; \Sigma'; \mathcal{H}' \vdash \overline{\mathcal{L}}' : (\mathcal{T} \rightarrow \mathcal{T}', \mathcal{U}' \rightarrow \varnothing)$

Proof: *Here we consider rules that only modify the current frame, leaving the the frame-stack unmodified. By lemma 5.5.7, we know the typing of the $\overline{\mathcal{L}}$ is unaltered in $\mathcal{C}'$. We will show that the new frame has the same type as the old frame; $\mathcal{CF} : T$ and $\mathcal{CF}' : T$. This equality ensures that* T-Closed-Stack *will return the same stack typing after by applying lemma 5.5.8. Therefore $\Gamma; \Sigma'; \mathcal{H}' \vdash \mathcal{L}[\texttt{fstack} \mapsto \mathcal{CF}' \circ \mathcal{L}_{(\texttt{fstack})}] : (\texttt{Void} \rightarrow \mathcal{T}, \mathcal{U} \rightarrow \mathcal{U}')$ will have the same type as before.*

Rule	$\mathcal{CF}$ **Type**	$\mathcal{CF}'$ **Type**
Copy-Discard-Env	T	$\langle \ell, T \rangle : T$ *by lemma* 5.2.3
Assign-Copied	`Void`	`void` : `Void`
Assign-Ref	`Void`	`void` : `Void`
Field-Access	T_{field}	$\langle \ell'_{field}, T_{field} \rangle : T_{field}$ *by lemma* 5.2.3
Variable-Access	T	$\langle \ell, T \rangle : T$ *by lemma* 5.2.3
Init-Field-Value	`Void`	`void` : `Void`
Init-Field-Ref	`Void`	`void` : `Void`
copy-init-Value	*CopyEnv*	$\mathcal{A}$: *CopyEnv by old frame typing*
copy-init-Ref	*CopyEnv*	$\mathcal{A}$: *CopyEnv by old frame typing*

5.4 Logical to Physical Consistency

Lemma 5.4.1 *Logical to Physical Type Match*
If $\Gamma; \Sigma; \mathcal{U} \vdash \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \mathcal{CF} \end{pmatrix}$ `config-ok`
and two frames $\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \mathcal{F}_{lhs} : T_{lhs}$ *and* $\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \mathcal{F}_{rhs} : T_{rhs}$ *such that* $T_{lhs} \overset{\frown}{\succ} T_{rhs}$,
then there will also be a match between physical types with
$\texttt{ptype}(\Sigma; \mathcal{H}, \mathcal{L}, T_{lhs})$ `p-match` $\texttt{ptype}(\Sigma; \mathcal{H}, \mathcal{L}, T_{rhs})$

Proof: *The* Ref-Initializable *rule which defines* $T_{lhs} \overset{\frown}{\succ} T_{rhs}$ *provides enough information to meet the three conditions of* Physical-Match. *First we consider both the class and mutability sub-components of the type, both of which* `ptype` *directly maps into the physical type. The requirements of* Ref-Initializable *are the same as* Physical-Match, *so these two conditions will be satisfied. For the container condition, we look at two sub-cases. If the left-hand-side container label is* `unknown`, *then* `ptype`*() will map it to* `unknown-cont`, *which will satisfy* Physical-Match. *Otherwise, the container labels must be equal and map to the same physical container to satify the condition.*

Lemma 5.4.2 *Returned Values Match*

If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$, $\mathcal{C} = \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \mathcal{L}_{caller} \circ \overline{\mathcal{L}} \\ \mathcal{CF} \end{pmatrix}$
and a type T *in scope* $\mathcal{L}$ *then*
$\texttt{ptype}(\Sigma; \mathcal{H}, \mathcal{L}, T)$ `p-match` $\texttt{ptype}(\Sigma; \mathcal{H}, \mathcal{L}_{caller}, \texttt{exportType}(\mathcal{L}_{(\texttt{lmap})}, T))$

Proof: `ptype()` *and* `exportType()` *both directly map the class, mutability and mobility components directly into the physical type, guaranteeing that they will match. For the container label,* `exportType()` *maps the label using* $\texttt{mapLabel}(\phi, T_{\texttt{(label)}})$*, which will map correctly by lemma 5.4.3, and all components of the physical types will match.*

Lemma 5.4.3 *Returned Containers Match*

If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$, $\mathcal{C} = \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \mathcal{L}_{caller} \circ \overline{\mathcal{L}} \\ \mathcal{CF} \end{pmatrix}$, $\phi = \mathcal{L}_{\texttt{(lmap)}}$ *and a heap location* $\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \ell : T$ *with container label* $\theta = T_{\texttt{(label)}}$ *then* $\texttt{labelToCont}(\Sigma; \mathcal{H}, \mathcal{L}_{caller}, \texttt{mapLabel}(\phi, \theta))$ `c-match` $\texttt{labelToCont}(\Sigma; \mathcal{H}, \mathcal{L}, \theta)$

Proof: *If* $\theta \in \texttt{dom}(\phi)$*, then we can immediately see that the* CMap-OK *condition of* T-Local-Config-Stack *gives us the result we need. If, on the other hand,* θ *is a path, then the root container* θ_{root} *of* θ *must exist in* ϕ *by type rule* `lmap-ok`. CMap-OK *then guarantees that*
$\texttt{labelToCont}(\Sigma; \mathcal{H}, \mathcal{L}, \theta_{root})$ `c-match` $\texttt{labelToCont}(\Sigma; \mathcal{H}, \mathcal{L}_{caller}, \texttt{mapLabel}(\phi, \theta_{root}))$

With the base of the path correctly mapped, following the path is a deterministic sequence of immutable field accesses which will lead to the same final container and satisfy `c-match`.

Lemma 5.4.4 *Passed Parameters Match*

If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$, $\mathcal{C} = \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \mathcal{L}_{caller} \circ \overline{\mathcal{L}} \\ \mathcal{CF} \end{pmatrix}$, $\phi = \mathcal{L}_{\texttt{(lmap)}}$*, and type* T_{passed} *in* $\mathcal{L}_{caller}$ *and* T_{parm} *in* $\mathcal{L}$ *such that* $\texttt{exportType}(\phi, T_{parm}) \overset{\sim}{\succ} T_{passed}$ *then* $\texttt{ptype}(\Sigma; \mathcal{H}, \mathcal{L}, \texttt{importType}(T_{passed}, T_{parm}))$ `p-match` $\texttt{ptype}(\Sigma; \mathcal{H}, \mathcal{L}_{caller}, T_{passed})$

Proof: *First, focusing on the container component of the physical type, note that the* `importType()` *function simply swaps the container label of* T_{passed} *and sets it to the label from* T_{parm}*. The* `ptype()` *function maps labels to containers using the* `labelToCont()` *function, and for* `p-match` *to hold,* `c-match` *must be satisfied. With* $\theta_{passed} = T_{passed\texttt{(label)}}$ *and* $\theta_{parm} = T_{parm\texttt{(label)}}$*, we must show that*
$\texttt{labelToCont}(\Sigma; \mathcal{H}, \mathcal{L}, \theta_{parm})$ `c-match` $\texttt{labelToCont}(\Sigma; \mathcal{H}, \mathcal{L}_{caller}, \theta_{passed})$.

From the definition of $\overset{\sim}{\succ}$ *we know that* $\texttt{mapLabel}(\phi, \theta_{parm})$ `l-match` θ_{passed}*, then with* `l-match` *established, lemma 5.4.5 provides the* `c-match` *judgment we need.*

For the remaining components of the physical types, the `ptype` *function simply maps them unchanged, and by the* $\overset{\sim}{\succ}$ *relation, we know those components will be satisfied in* `p-match`.

Lemma 5.4.5 *Passed Containers Match*

If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U}\rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}'\rangle$, $\mathcal{C} = \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \mathcal{L}_{caller} \circ \overline{\mathcal{L}} \\ \mathcal{CF} \end{pmatrix}$,

$\phi = \mathcal{L}_{(\texttt{lmap})}$*, and container label* θ_{passed} *in* $\mathcal{L}_{caller}$ *and* θ_{parm} *in* $\mathcal{L}$ *such that*
`mapLabel`(ϕ, θ_{parm}) `l-match` θ_{passed}
then `labelToCont`$(\Sigma; \mathcal{H}, \mathcal{L}, \theta_{parm})$ `c-match` `labelToCont`$(\Sigma; \mathcal{H}, \mathcal{L}_{caller}, \theta_{passed})$

Proof: *From the definition of* `l-match`*, we know that either* $\theta_{passed} =$ `mapLabel`(ϕ, θ_{parm}) *or that* $\theta_{parm} =$ `unknown`*. We'll first consider the equality case. By lemma 5.4.3, we know that the inverse mapping holds*
`labelToCont`$(\Sigma; \mathcal{H}, \mathcal{L}_{caller},$ `mapLabel`$(\phi, \theta_{parm}))$ `c-match` `labelToCont`$(\Sigma; \mathcal{H}, \mathcal{L}, \theta_{parm})$
By our equality, we can substitute in θ_{passed}*, yielding*
`labelToCont`$(\Sigma; \mathcal{H}, \mathcal{L}_{caller}, \theta_{passed})$ `c-match` `labelToCont`$(\Sigma; \mathcal{H}, \mathcal{L}, \theta_{parm})$ *which is backwards from what we need. However, we have already assumed that* $\theta_{parm} \neq$ `unknown`*, which means that* `c-match` *degrades to a simple equality check, which is symmetric. Therefore the terms can be swapped, giving the desired result.*

In the case where $\theta_{parm} =$ `unknown`, `labelToCont`$(\Sigma; \mathcal{H}, \mathcal{L}, \theta_{parm})$ *will evaluate to* `unknown-cont` *which satisfies* `c-match`*.*

5.5 Preservation Lemmas

This section contains a series of lemmas showing that the rules of the dependent type system are such that the typing only depends on immutable objects. This leads to the useful property that a typing that is valid in configuration $\mathcal{C}$ will continue to be valid in a subsequent configuration $\mathcal{C}'$.

Lemma 5.5.1 *Stable* `ptype()` *Evaluation*

If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U}\rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}'\rangle$, $\mathcal{C} = \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \mathcal{CF} \end{pmatrix}$, $\mathcal{C}' = \begin{pmatrix} \mathcal{H}' \\ \mathcal{L}' \circ \overline{\mathcal{L}}' \\ \mathcal{CF}' \end{pmatrix}$,

$\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \mathcal{F} : T$, $\mathcal{T} =$ `ptype`$(\Sigma, \mathcal{H}, \mathcal{L}, T)$, $\mathcal{L}_{(\texttt{id})} = \mathcal{L}'_{(\texttt{id})}$
then $\mathcal{T} =$ `ptype`$(\Sigma', \mathcal{H}', \mathcal{L}', T)$.

Proof: *We examine the implementation of* `ptype()` *and note that the class, mutability and mobility components pass directly from* T *to* $\mathcal{T}$ *unaffected by the heap or configuration*

typing. The final component of the physical type is the container, which `ptype()` *computes using the function* `labelToCont()`*. By lemma 5.5.2, we know that* $\texttt{labelToCont}(\Sigma, \mathcal{H}, \mathcal{L}, T_{\texttt{(label)}}) = \texttt{labelToCont}(\Sigma', \mathcal{H}', \mathcal{L}', T_{\texttt{(label)}})$.
Therefore, all components of the physical types are identical and `ptype()` *is stable.*

Lemma 5.5.2 *Stable* `labelToCont()` *Evaluation*
If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$, $\mathcal{C} = \begin{pmatrix} \mathcal{H} \\ \mathcal{L} \circ \overline{\mathcal{L}} \\ \mathcal{CF} \end{pmatrix}$, $\mathcal{C}' = \begin{pmatrix} \mathcal{H}' \\ \mathcal{L}' \circ \overline{\mathcal{L}}' \\ \mathcal{CF}' \end{pmatrix}$,
$\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \mathcal{F} : T$, $\Theta = \texttt{labelToCont}(\Sigma, \mathcal{H}, \mathcal{L}, T_{\texttt{(label)}})$, $\mathcal{L}_{\texttt{(id)}} = \mathcal{L}'_{\texttt{(id)}}$
then $\Theta = \texttt{labelToCont}(\Sigma', \mathcal{H}', \mathcal{L}', T_{\texttt{(label)}})$.

Proof: *For* $\theta = T_{\texttt{(label)}}$*, we consider the cases in the implementation of the* `labelToCont()` *function, first addressing the simple cases. The cases* `unknown-label` *and* `null-label` *are simple constants with no dependencies on any environment. The default case returns* $\Sigma_{\texttt{(cmap)}}(\mathcal{L}_{\texttt{(id)}})(\theta)$*, which will be consistent because the container map is immutable.*

Finally, when θ *is a path, the heap will be accessed. Since type* T *is a valid type, the path of* θ *has satisfied* `path-ok`*. By rules* V-SP-Tuple *or* V-SP-Var*, the base of the path must be* `fixed`*. Further, by the rule* V-SP-Step*, any fields accessed along the path will also be* `fixed`*, and by lemma 5.2.2, we know that all of these locations will have the same value in* $\mathcal{H}$ *and* $\mathcal{H}'$*. Therefore, all heap information that the path depends on is the same in* $\mathcal{H}$ *and* $\mathcal{H}'$*, which forces the results to match.*

Therefore, for all possible values for θ*,* `labelToCont()` *is stable.*

Lemma 5.5.3 `f-match` *Preserved*
If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$, $\mathcal{H} = \mathcal{C}_{\texttt{(heap)}}$, $\mathcal{H}' = \mathcal{C}'_{\texttt{(heap)}}$, $\ell \in \mathcal{H}$,
$\Sigma; \mathcal{H}; \ell \vdash T$ `f-match` $\mathcal{T}$
then $\Sigma'; \mathcal{H}'; \ell \vdash T$ `f-match` $\mathcal{T}$

Proof: *We know from* `t-match` *in* $\mathcal{C}$ *that* $\mathcal{T} = \texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}, T)$ *and by lemma 5.5.1 we know that* $\mathcal{T} = \texttt{ptype}(\Sigma', \mathcal{H}', \mathcal{L}, T)$*. Therefore the* `t-match` *condition of* `f-match` *continues to hold.*

The second condition of interest is the `within` *judgment, which is preserved by lemma 5.5.4. If all dependencies of* `fmatch` *are preserved, we conclude that* `fmatch` *is also preserved.*

Lemma 5.5.4 *Containment —* `within` *Preserved*
If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$, $\mathcal{H} = \mathcal{C}_{\texttt{(heap)}}$, $\mathcal{H}' = \mathcal{C}'_{\texttt{(heap)}}$, $\Sigma \vdash$

Θ_1 `within` Θ_2
then $\Sigma' \vdash \Theta_1$ `within` Θ_2

Proof: *The* `within` *judgment uses* Σ *to recursively look up parent containers. The type environment only grows and is never modified. Therefore* $\Sigma'(\Theta_1) = \Sigma(\Theta_1)$, *making the entire judgment preserved in* $\mathcal{C}'$.

Lemma 5.5.5 CMap-OK *Preserved*
If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \to \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$, $\Sigma; \mathcal{H} \vdash \mathcal{L}_{caller}$ `cmap-ok` $\mathcal{L}_{callee}$, $\mathcal{L}'_{caller} \in \mathcal{C}'$, $\mathcal{L}'_{caller(\texttt{id})} = \mathcal{L}_{caller(\texttt{id})}$, $\mathcal{L}'_{callee} \in \mathcal{C}'$, $\mathcal{L}'_{callee(\texttt{id})} = \mathcal{L}_{callee(\texttt{id})}$
then $\Sigma'; \mathcal{H}' \vdash \mathcal{L}'_{caller}$ `cmap-ok` $\mathcal{L}'_{callee}$

Proof: *Looking at each of the conditions in rule* CMap-OK, *we'll start with* `lmap-ok`, *which is a pure typing rule and independent from the configuration. The lookup of* Δ *from the caller scope is equivalent in* $\mathcal{C}'$, *because the frame-stack of the caller is fixed while the callee executes. Therefore, the* `lmap-ok` *condition continues to hold in* $\mathcal{C}'$.

Both ϕ *and* $\mathcal{M}$ *are immutable, which leaves the final container equality to consider. Here, we appeal to lemma 5.5.2 to establish that the left and right sides of this equality will evaluate to the same values in* $\mathcal{C}$ *and* $\mathcal{C}'$ *for each* $\theta_i \in \texttt{dom}(\mathcal{M})$. *Since the equality held in* $\mathcal{C}$, *it will continue to hold in* $\mathcal{C}'$

Lemma 5.5.6 `vars-ok` *Preserved*
If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \to \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle, \mathcal{L} \in \mathcal{C}, \mathcal{L}' \in \mathcal{C}', \mathcal{L}_{(\texttt{vars})} = \mathcal{L}'_{(\texttt{vars})}$
then $\Gamma; \Delta; \Sigma' \vdash \mathcal{L}'_{(\texttt{vars})}$ `vars-ok`

Proof: *We know that we have* `heap-ok` *in* $\mathcal{C}$, *which means that every local symbol is mapped to a box or reference which is* `location-ok`. $T_{variable} = T_{box/ref}$ *implies that* $\Sigma(\ell_{box/ref}) = \texttt{ptype}(\Sigma, \mathcal{H}, \mathcal{L}, T_{box/ref}) = \texttt{ptype}(\Sigma; \mathcal{H}, \mathcal{L}, T_{variable})$. *Therefore, we conclude that the* `t-match` *judgment is satisfied for all local variables.*

Lemma 5.5.7 *Local Configuration Stack Typing Preserved*
If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \to \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$, $\mathcal{H} = \mathcal{C}_{(\texttt{heap})}$, $\mathcal{H}' = \mathcal{C}'_{(\texttt{heap})}$, $\Gamma; \Sigma; \mathcal{H} \vdash \mathcal{L} \circ \overline{\mathcal{L}} : (\mathcal{T} \to \mathcal{T}'', \mathcal{U}' \to \varnothing)$
then $\Gamma; \Sigma'; \mathcal{H}' \vdash \mathcal{L} \circ \overline{\mathcal{L}} : (\mathcal{T} \to \mathcal{T}'', \mathcal{U} \to \varnothing)$

Proof: *The* `cmap-ok` *judgment continues to hold in* $\mathcal{C}'$ *by lemma 5.5.5. Since the set of variables on the stack has not changed, the* `vars-ok` *judgment remains true, by lemma 5.5.6. Next, by lemma 5.5.8, we have* $\Gamma; \Sigma'; \mathcal{H}' \vdash \mathcal{L}_{(\texttt{fstack})} : (\mathcal{T} \to \mathcal{T}', \mathcal{U} \to \mathcal{U}')$

Finally, by induction on this lemma, we have $\Gamma; \Sigma'; \mathcal{H}' \vdash \overline{\mathcal{L}} : (\mathcal{T}' \rightarrow \mathcal{T}'', \mathcal{U}' \rightarrow \varnothing)$ *. Rule* T-Local-Config-Stack-Empty *provides the base case and guarantees that the induction terminates. Therefore, all conditions of the rule* T-Local-Config-Stack *are met, and we can conclude that an unmodified stack of local configurations retains its typing in* $\mathcal{C}$*.*

Lemma 5.5.8 *Frame-Stack Typing Preserved*
If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$, $\mathcal{H} = \mathcal{C}_{\texttt{(heap)}}$, $\mathcal{H}' = \mathcal{C}'_{\texttt{(heap)}}$,
$\Gamma; \Sigma; \mathcal{H}; \mathcal{L} \vdash \overline{\mathcal{F}} : (\mathcal{T} \rightarrow \mathcal{T}'', \mathcal{U} \rightarrow \mathcal{U}')$
then $\Gamma; \Sigma'; \mathcal{H}'; \mathcal{L} \vdash \overline{\mathcal{F}} : (\mathcal{T} \rightarrow \mathcal{T}'', \mathcal{U} \rightarrow \mathcal{U}')$

Proof: *We consider the conditions of the frame-stack typing rules as defined in section 4.9 in their entirety. There is significant similarity among the rules, and we'll reason by cases over the kinds of conditions that need to be satisfied rather than over each rule individually.*

First, all conditions of the form $\mathcal{T} = \texttt{ptype}()$ *will continue to hold by the stability of* `ptype()` *established in lemma 5.5.1. All induction on the tail of the frame-stack is satisfied by induction on this lemma. The base case is typed by rule* T-Empty-Stack*, which guarantees that the induction terminates. Finally, typing of individual frames is preserved by lemma 5.5.9. This completes the set of conditions common to most of the frame-stack typing rules.*

The rule T-Immediate-Return *is special, and we consider its unique condition* $T = \texttt{exportType}(\mathcal{L}_{\texttt{(lmap)}}, T_{callee})$*. We know* T_{callee} *is preserved by lemma 5.5.1 and* $\mathcal{L}_{\texttt{(lmap)}}$ *is immutable. Therefore, this condition is also preserved, and we conclude that all conditions appearing in the frame-stack typing rules continue to hold in* $\mathcal{C}'$*.*

Lemma 5.5.9 *Frame Typing Preserved*
If $\Gamma; \Sigma; \mathcal{U} \vdash \mathcal{C}$ `config-ok`, $\langle \mathcal{C}, \Sigma, \mathcal{U} \rangle \rightarrow \langle \mathcal{C}', \Sigma', \mathcal{U}' \rangle$, $\mathcal{H} = \mathcal{C}_{\texttt{(heap)}}$, $\mathcal{H}' = \mathcal{C}'_{\texttt{(heap)}}$,
$\Gamma; \Delta; \Sigma; \mathcal{H}; \mathcal{L} \vdash \mathcal{F} : T$
then $\Gamma; \Delta; \Sigma'; \mathcal{H}'; \mathcal{L}' \vdash \mathcal{F} : T$

Proof: *Here, we distinguish frames that are the original statements and expressions of the language from the extended frames that are typed in section 4.6. For the proper language components, we don't need to concern ourselves with the extended typing environment as the rules only depend on* Γ *and* Δ*. Therefore we conclude that all statements and expressions have the same typing in* $\mathcal{C}'$*.*

For the extended typing rules of section 4.6, we use the same strategy of lemma 5.5.8 and show preservation for the various kinds of conditions in these rules rather than address each rule directly. All type lookups in Γ, Δ *or* Σ *will be the same because* Γ *and* Δ *are the* *same and existing entries in Σ are immutable. For frames that hold other frames and inductively type those sub-frames, we inductively apply this lemma to show the sub-frames retain their same typing.*

Simple logical predicates on types will remain the same, because all the type information consumed is preserved. For example, the condition $(T_{i\,\texttt{(kind)}} = \texttt{ref} \wedge T_i \widetilde{\succ} T_{init_i}) \vee (T_i = T_{init_i})$ *from rule* T-Init-Symbols *is composed of typings that are preserved.*

For rules that test `copy-map-ok`*, we know that this condition will hold, because $\mathcal{A}$ is the same as before. Note that in each instance of $\mathcal{A}$ in the rules, it is part of the frame itself, and this lemma is only concerned with typing identical frames.*

Finally, the rule T-Location *has a* `ptype()` *condition, which is preserved by lemma 5.5.1, and a* `t-match` *condition which does not depend on the heap.*

Therefore, all of the conditions present in the rules of section 4.6 will be preserved in $\mathcal{C}'$.

Chapter 6

Related Work

6.1 Ownership Types

Aliasing causes many problems for imperative programs. Reasoning about a system with aliases is problematic. Generally, when considering a method's behavior, there will be limited or no knowledge of what external aliases can exist. This interferes with proving code correctness and forces an optimizing compiler to make pessimistic choices.

Although aliasing is not the focus of our container language, there can be reductions in aliasing with this system. We encourage pass-by-value semantics by providing well-defined copying. Once you've made a fresh copy of an object, external aliasing is eliminated. Although our semantics define a potentially expensive copy algorithm, an alternative implementation could use persistent data structures like a functional language. The cost of a copy would become zero at the expense of increased update time. If such a system were implemented, pass by value wouldn't incur a performance penalty and aliasing could be avoided in many cases.

Ownership types were first proposed by David G. Clarke et al. in [9] as a way to enforce encapsulation, which results in all aliasing concerns being local and easier to reason about. Significant work in this area has been continued by many researchers and two surveys can be found in [8] and [14].

Although our containment system doesn't involve owners, the fact that a container has no outgoing references means it is self-contained and implicitly everything in the container is *owned* by the container. We make no attempt to enforce encapsulation, but an extension of container-typing could add restrictions on the use of containers. If a container is declared as encapsulated, then only privileged code would be able to hold references into that container.

In this thesis, we are focused on containment, which is converse of encapsulation in that it disallows outgoing references rather than incoming. However, despite having converse goals, the fundamental type-system machinery needed to manage containment has similarities to ownership types. With ownership types, an object can contain references to owned sub-objects. Types are augmented to indicate that certain fields represent *owned* objects. When an object is *owned*, the type system must prevent the creation of external aliases. Both systems must control the assignment of references.

The paper *Sheep Cloning with Ownership Types* by Paley et al. [12] recognizes that ownership types can be leveraged for the purpose of object copying. This thesis fundamentally builds on their insight. They recognize the difficulty in manually implementing cloning code, and they present a hybrid between deep cloning and shallow cloning. Their work uses the declared owner of nested objects to determine which objects should be copied. Our approach mirrors theirs, and although we use containers instead of owners, there is fundamental similarity. This thesis, differs in that we remove the goals of ownership types so that we can exclusively focus on automated functionality related to self-contained objects. Their hybrid approach contrasts with our system as we have chosen to explicitly prevent a hybrid copy. Their work is further refined in [16]. Here, they refactor their work to facilitate proving its soundness.

Although still firmly encapsulation-focused, the work of Bettini et al. [3] introduces the concept of boxes, which is similar to our notion of a container. They recognize that making every object encapsulated is to too restrictive. Their duality between boxed and un-boxed classes is similar to the data and entity classes of the container language. In contrast, the ownership type system of Boyapati et al. [5] also tries to make a less restrictive system, but takes a different approach. It uses module boundaries as encapsulation barriers. Inner classes would have full access to objects owned by the outer class. It's appealing that they could avoid introducing a new concept like a box or container, but using a module as a container would be too broad, and we would lose pass-by-value semantics.

Ownership types are a dependent type system, as is our container type system. In *Ownership Type Systems and Dependent Classes*, Dietl et al. [10] implement ownership types on top of a dependent type system. They conclude that specialized syntax for ownership types is still desirable, even though ownership types can be fully expressed in a more general way. However, by establishing an implementation of ownership types in a general dependent-type system, they have shown a new way to reason about ownership types. A similar treatment of our container work could yield additional insights.

The work of Cameron et al. [7] also evaluates ownership types from a more theoretical viewpoint and compares them with more fundamental dependent type systems such as Dependent ML. Again, applying this type of analysis to our container type system would be an worthwhile exercise.

Huang et al. [11] develop type inferencing of ownership types. They argue that the overhead of ownership specification hinders the adoption of ownership types. We make a similar argument with respect to containers. Any adoption is unlikely unless systems like these are simple and easy to use.

6.2 Serialization

Serialization in object-oriented systems is another broad field of research and is also widely used in practice. In our system, we claim that objects that are candidates for serialization should always be self-contained, and our container system enforces that property. This greatly simplifies the problem of serialization. Here, we'll compare our container system to a number of other systems.

In *Instant Pickles* by Miller et al. [13], they develop a pickle combinator with the flexibility to customize to many data formats. To enhance our system so the output format could be customized would require a reflection system. If this was in place, containment could continue to provide the nice property that no per-class code would be needed for serialization. A generic piece of code could reflect on the meta-data and using an algorithm similar to the one presented in this thesis, produce a custom serialized form of a self-contained object. This paper also considers inheritance and versioning which is absent from our work.

In the Fibonacci system [1], Albano et al. have developed a programming language for object databases. They build a system with similar ideas to an entity-relationship model. They state that "any value, irrespective of its type, has the same rights to persistence". Here, we take a different stance, where non-self-contained objects can only be persisted as part of a larger self-contained object. Although this language is object oriented, they have added concepts like associations, which effectively create a data model like the database schema in a relational database.

Nestmann et al. [15] develop a system not to copy objects, but to migrate them. They go to great lengths to achieve transparency in their distributed system with surrogate objects working locally to forward calls to remote objects. Effectively, they extend their application's heap across physical machines. This work is pursuing a goal opposite to ours in that we want to make message passing easier to reduce the need for a shared heap.

6.3 Operational Semantics

The operational semantics of chapter 4 were initially modeled using the MJ technical report [4] as inspiration, although the design drifted apart as our semantics were developed. In the frame stack typing of the MJ system, they duplicated logic from their type rules in order to re-build an equivalent typing environment needed for the stack typing. In our system, with many more frames to consider, this would have become unmanageable. Like the MJ system, we need the typing environment for our frame-stack typing as well. Our typing environment is only built by the type rules, and we bind it to our frames to avoid rebuilding it a second time. This blunt-force strategy worked well and saved significant duplication of work which would have cluttered the frame stack typing.

Chapter 7

Future Work

This chapter looks at three areas where the ideas of this thesis can be extended and presents ideas to address them.

7.1 Sub-Containers

One significant issue with the container type system is that it requires precise specification of a container. For example, if a reference has container `c`, then that reference is not allowed to point to an object contained with `c.nested`. This could be fixed by introducing a sub-container constraint for references. For example, figure 7.1 contains a proposed `::>` specifier which constrains a reference to be *within* a container and not necessarily have that precise container. This allows for nesting containers without exposing internal structure.

With an imprecise container constraint, it would then also be natural to allow narrowing a reference to a specific container. This could be done by a matching predicate which refines the type to a precise container label. E.g. `if r within nested then ...`

One concern with this narrowing behavior is that up until now, the soundness of the container type checking implied that there is no extra run-time overhead associated with container typing. In order to support the narrowing behavior, there would need to be an extra field in each object to indicate which container it was in. Perhaps this could be minimized with global optimization as only objects that participate in narrowing would require this extra overhead.

```
class Inner( value : Int )
constructor Inner( value : Int ) {
    self.value = value;
}

class Outer( fixed nested : Inner )
constructor Outer( value : Int ) {
    self.nested = Inner( value );
}

class Main ()
constructor Main() {
    fixed var multiLevel = Outer(1);

    iref r1 : Int :: multiVar =^ null;
    r1 =^ multiLevel.inner.value;       // COMPILER ERROR! container mismatch

    iref r2 : Int :: multiVar.nested =^ null;
    r2 =^ multiLevel.inner.value;       // OK, containers match

    iref r3 : Int ::> multiVar =^ null;
    r3 =^ multiLevel.inner.value;       // OK, with new ::> sub-container!
}
```

Figure 7.1: Sub-Container Notation

7.2 Roles

In chapter 2 we presented an algorithm to do a deep comparison between two self-contained tuples (figure 2.11). Conceptually, this algorithm must assign an identity to each of the nested objects. A mapping is created indicating a correspondence between nested tuples, saying tuple `x` on the left hand side matches with tuple `y` on the right hand side. A natural extension of this is to say that an entity has an identity relative to its container. This is in contrast to the typical instance-identity notion, where the event of an object's creation defines its identity. With a container-relative identity, it becomes possible to have two container objects and relate nested entities that fulfill the same role with respect to their container.

The identities from the deep comparison algorithm are simply the first path to reach a nested object by a depth-first search. If a concept of a role were added to the language, then this mechanically-derived identity could be replaced with something more meaningful. The idea would be that all entities would be assigned a role. The indirect reference `iref` shown in the demonstration language was added with the idea that it would be a reference to a role. In the most direct case, an entity could be described as a having the role granted by a special *role* field of a container object. This idea can be extended by longer paths to reach nested entities. Every entity's identity could be described in the form *(role-of-granting-object).role-name.* This would be a meaningful description rather than a memory address. Identities would be preserved over a serialization and de-serialization round trip.

Descriptive identities can also be represented directly in the language as data. A non-self-contained entity class could be converted into a pure data class by replacing all references with materialized identity paths. Conversion functions could be automatically generated. The inverse operation (data to entity) is simply a function that takes a container object and an identity path of a nested entity and returns a reference.

With descriptive identities materialized as data, entities can be translated into pure data objects and can then be serialized. Additional functionality can be built on this as well. We've looked at boolean equality comparison, but you could also compare two objects and return the difference between them. With descriptive identity paths, you could return a list of new, deleted and modified entities. It's commonplace to *diff* two text files to see what has changed. Version control systems store a sequence of modifications to text and can re-create any version of a file by applying the recorded changes. This capability would also be useful in a distributed system. Replicas of data could be synchronized by sending only changes to the data. Many such systems like this exist, but not directly supported by a programming language.

If features like those in a version control system could be brought into an object-oriented language, then implementing an undo button would be a trivial task. You would be able to roll back the state of the program after an exception is caught.

Many details have been glossed over in this description, however with additional effort, the foundation of containers can be extended to become a richer data model with enhanced capabilities.

Although this work was not completed in this thesis, research related to roles was reviewed and we'll compare two of those works now.

In the work of Steimann [18], the connection is made between an object assigned to a variable and a role. Variables have purposes and when a variable refers to an object, it gives that object a role. The paper then argues that variables should be typed with interfaces specific to the role needed by the variable. Further, the design of interfaces should be role-centric, essentially making interfaces and roles the same thing. Although this paper is more focused on object oriented modeling, parallels can be drawn with our proposal. The recognition that the roles of an object can change throughout its lifetime is an important one. If we develop a system that links object identity to roles, then we

```
class Demo( v1 : Int, v2 : Int )
constructor Demo() {
    v1 = 1;
    v2 = 2;
}

method Main.test( iref r1 : Int :: ‘c, iref r2 : Int :: ‘c } {
    ...
}

class Main ()
constructor Main() {
    var demo = Demo();
    self.test( demo.v1, demo.v2 );   // COMPILER ERROR! demo is not fixed!
    // The paths demo.v1 and demo.v2 clearly have the same container, but because
    // var demo is not declared as fixed, the container labels are ’unknown’
}
```

Figure 7.2: Data Flow Example

need to fully consider how to handle object identity when roles change. This will be a challenging problem.

An alternative approach to identity is proposed in [19] by Vaziri et al.. Here they introduce an explicit concept called a relation-type, where a programmer designates a set of immutable fields to act as that object's identity. The class of an object is also considered part of its identity, so the fields only need to be unique with respect to their class. This model is similar to a relational database and makes identity a concrete part of every object. This system will inherit the difficulties databases have with unique key generation, such as the efficient allocation of unique keys across a distributed system. However, explicit identity nicely bypasses the problem of changing roles. Their relation-type model would work well to satisfy our goal of easy serialization.

7.3 Data Flow Analysis

We have the restriction that all container labels can only depend on `fixed` symbols. This restriction can be an annoyance in some cases, one of which is outlined in figure 7.2. Here, we have a method that requires both of its parameters to be within the same container. The code in `Main` clearly passed two references that are within the same container, but because the variable `demo` is not declared as `fixed`, the type system asserts that the container labels are unknown and the method call does not type check.

This issue could be fixed with data flow analysis. If a symbol is determined to be fixed within a region of code, then an automatic rewrite could be applied to inject a temporary reference that is fixed.

Chapter 8

Conclusion

In this thesis, we've taken the abstract idea of a container isolating its contents from the outside world and built two systems based on this idea. First, a broader implementation in Haskell that attempted to solve pragmatic issues such as reducing the amount of extra syntax needed using a container inferencing algorithm. By developing example code and test cases, we have a sense of what programming in a container language could be like. This system is too immature to evaluate its suitability as an implementation language for real object-oriented programs. However, in the code that has been written, no major problems have been found and simple data structures such as a linked list with containment were easy to develop.

The second system of operational semantics showed that our dependent type system is sound. Code using containers can be statically typed such that there will be no run-time violation of containers. This is an important result since it means that all container type information can be erased at run-time and a container system can run without any additional overhead compared to an equivalent system without containers.

Adding the ability to enforce self-containment creates a *kind* of object which is simple to reason about. There are a number of algorithms that are easy to implement when an object is self contained, but extremely tricky otherwise. Serialization, deep object copying and deep object equality comparisons can be built into the language. The opportunity exists to eliminate a significant amount of glue code by leveraging containers and self-contained objects.

A great deal of work would be required to turn this demonstration system into a practical programming language. This thesis has shown that core idea of containment is sound, and demonstrated that containers can have practical benefits. We hope this work will motivate future research into containers.

References

[1] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. Fibonacci: A programming language for object databases. *VLDB J.*, 4(3):403–444, 1995.

[2] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In Mamdouh Ibrahim and Satoshi Matsuoka, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002*, pages 311–330. ACM, 2002.

[3] Lorenzo Bettini, Ferruccio Damiani, Kathrin Geilmann, and Jan Schäfer. Combining traits with boxes and ownership types in a java-like setting. *Sci. Comput. Program.*, 78(2):218–247, 2013.

[4] Gavin M Bierman, MJ Parkinson, and AM Pitts. Mj: An imperative core calculus for java and java with effects. Technical report, University of Cambridge, Computer Laboratory, 2003.

[5] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In Alex Aiken and Greg Morrisett, editors, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 213–223. ACM, 2003.

[6] John Boyland. Alias burying: Unique variables without destructive reads. *Softw. Pract. Exp.*, 31(6):533–553, 2001.

[7] Nicholas Robert Cameron, Sophia Drossopoulou, and James Noble. Understanding ownership types with dependent types. In Dave Clarke, James Noble, and Tobias

Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 84–108. Springer, 2013.

[8] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 15–58. Springer, 2013.

[9] David G. Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In Bjørn N. Freeman-Benson and Craig Chambers, editors, *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada, October 18-22, 1998*, pages 48–64. ACM, 1998.

[10] W. Dietl and P. Müller. Ownership Type Systems and Dependent Classes. In *Foundations of Object-Oriented Languages (FOOL)*, January 2008.

[11] Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. Inference and checking of object ownership. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 181–206. Springer, 2012.

[12] Paley Li, Nicholas Cameron, and James Noble. Sheep cloning with ownership types. In *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*, page 59, 2012.

[13] Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. Instant pickles: generating object-oriented pickler combinators for fast and extensible serialization. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 183–202. ACM, 2013.

[14] Alan Mycroft and Janina Voigt. Notions of aliasing and ownership. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 59–83. Springer, 2013.

[15] Uwe Nestmann, Hans Hüttel, Josva Kleist, and Massimo Merro. Aliasing models for object migration. In Patrick Amestoy, Philippe Berger, Michel J. Daydé, Iain S. Duff, Valérie Frayssé, Luc Giraud, and Daniel Ruiz, editors, *Euro-Par '99 Parallel Processing, 5th International Euro-Par Conference, Toulouse, France, August 31 - September 3, 1999, Proceedings*, volume 1685 of *Lecture Notes in Computer Science*, pages 1353–1368. Springer, 1999.

[16] Nicholas Cameron Paley Li and James Noble. Dynamic semantics of sheep cloning: Proving cloning. *International Workshop on Aliasing, Capabilities, and Ownership (IWACO)*, 2014.

[17] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.*, 60-61:17–139, 2004.

[18] Friedrich Steimann. Role= interface: a merger of concepts. *Journal of Object Oriented Programming*, 14(4):23–32, 2001.

[19] Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. Declarative object identity using relation types. In Erik Ernst, editor, *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 54–78. Springer, 2007.

[20] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.