Safe Initialization of Objects

Fengyun Liu, Ondřej Lhoták, Aggelos Biboudis, Paolo G. Giarrusso, Martin Odersky

Technical Report

École Polytechnique Fédérale de Lausanne

September 10, 2020

Contents

Chapter 1

Introduction

Life is too short to spend time chasing down irreproducible bugs,
and money is too valuable to waste on the purchase of flaky software.

— Andrew W. Appel, *Modern Compiler Implementation in ML*

Object-oriented programming is unsafe if objects cannot be initialized safely. The following code shows a simple initialization problem [1]:

```
1 class Hello {
2   val message = "hello, " + name
3   val name = "Alice"
4 }
5 println(new Hello().message))
```

The code above at runtime will print "`hello, null`" instead of "`hello, Alice`", as the field `name` is not initialized, thus holds the value `null`, when it is used in the second line.

Software that are vulnerable to such errors crash or misbehave at runtime and may incur financial loss or cause safety accidents. The errors, sometimes simple sometimes subtle, require programmer efforts to debug and fix, which drains valuable engineering resources and harms programmer productivity.

Such errors have come into existence since the inception of object-oriented programming 60 years ago. Despite the fact that object-oriented programming has become the dominant programming paradigm in the software industry for more than 30 years, programmers are still frequently bothered and frustrated by initialization errors in software development.

Joe Duffy, in his popular blog post *on partially constructed objects* [13], wrote:

> Not only are partially-constructed objects a source of consternation for everyday programmers, they are also a challenge for language designers wanting to provide guarantees around invariants, immutability and concurrency-safety, and non-nullability.

This thesis addresses the problem. We study principles, abstractions and algorithms for checking safe initialization of objects. The solutions presented in the thesis are validated theoret-

[1] In the absence of special notes, the code examples are in Scala.

ically and supported empirically, which can be readily used to improve current languages and design future languages.

1.1 The Problem

Safe initialization of objects is becoming a challenge as the code in constructors is getting more complex. From past research [14, 16, 12, 9, 24, 19, 11], two initialization requirements are identified and commonly recognized.

Requirement 1: usage of "this" inside the constructor. The usage of already initialized fields in the constructor is safe and supported by almost all industrial languages. Based on an extensive study of over sixty thousand classes, Gil *et al.* [14] report that over 8% constructors include method calls on `this`. Method calls on `this` can be used to compute initial values for field initialization or serve as a private channel between the superclass and subclass.

Requirement 2: creation of cyclic data structures. Cyclic data structures are common in programming. For example, the following code shows the initialization of two mutually dependent objects:

```
1 class Parent { val child: Child = new Child(this) }
2 class Child(parent: Parent)
```

The objective is to allow cyclic data structures while preventing accidental premature usage of aliased objects. Accessing fields or calling methods on those aliased objects under initialization is an orthogonal concern, the importance of which is open to debate.

Tony Hoare introduced `null` to programming languages, which is commonly known as *the billion dollar mistake* [15]. Therefore, a good solution should not resort to `null` to create cyclic data structures.

A common misunderstanding is that initialization problems only occur in strict languages. For example, the following Haskell code loops forever:

```
1 a = if b then 10 else 20
2 b = a >= 10
3 main = putStrLn (show a)
```

Indeed, there are no initialization errors that cause null-pointer exceptions or segmentation fault. However, the situation is arguably worse because in the language we may not distinguish initialization errors from non-termination.

1.2 Theoretical Challenges

There are three theoretical challenges to attack the problems above.

Challenge 1: virtual method calls. While direct usage of already initialized fields via `this` is relatively easy to handle, indirect usage via virtual method calls poses a challenge. Such methods could be potentially overridden in a subclass, which makes it difficult to statically check whether it is safe to call such a method. This can be demonstrated by the following example:

```
abstract class AbstractFile {
  def name: String
  val extension: String = name.substring(4)
}
class RemoteFile(url:String) extends AbstractFile {
  val localFile: String = url.hashCode // error
  def name: String = localFile
}
```

According to the semantics of Scala (Java is the same), fields of a superclass are initialized before fields of a subclass, so initialization of the field `extension` proceeds before `localFile`. The field `extension` in the class `AbstractFile` is initialized by calling the abstract method `name`. The latter, implemented in the child class `RemoteFile`, accesses the uninitialized field `localFile`.

Challenge 2: aliasing. It is well-known that aliasing complicates program reasoning and it is challenging to develop practical type systems to support reasoning about aliasing [8, 33]. It is also the case for safe initialization: if a field aliases `this`, we may not assume the object pointed to by the field is fully initialized. This can be seen from the following example:

```
class Knot {
  val self = this
  val n: Int = self.n // error
}
```

In the code above, the field `self` is an alias of `this`, thus we may not use it as a fully initialized value. Aliasing may also happen indirectly through method calls, as the following code shows:

```
class Foo {
  def f() = this
  val n: Int = f().n // error
}
```

Challenge 3: typestate polymorphism. Every newly created object goes through several typestates [37]: starting from a state where all fields are uninitialized until all of them are assigned. If a method does not access any fields on $this$, then it should be able to be called on any typestate of $this$. For example, in the following class `C`, we should be able to call the method `g` regardless of the initialization state of `this`:

```
class C {
  val a = ...
  // ...
  def g(): Int = 100
}
```

The challenge is how to support this feature succinctly without syntactic overhead.

1.3 Practical Challenges

Practical object-oriented programming languages usually have extra language features, such as *first-class functions, inner classes, traits* and *properties*. Each such feature poses additional challenges for safe initialization.

First-Class Functions. Functions may capture `this` and the usage of functions may reach uninitialized fields, as the following example demonstrates:

```
class Rec {
  val even = (n: Int) => n == 0 || odd(n - 1)
  even(6)                                          // error
  val odd = (n: Int) => n == 1 || even(n - 1)
  even(6)                                          // ok
}
```

In the code above, the first call `even(6)` is problematic, as it indirectly uses the yet uninitialized field `odd`. In contrast, the last line is fine, as `odd` is already initialized.

Traits. Traits enable flexible code reuse [21, 26], and they are a key language feature of Scala. Unlike interfaces in Java, it is possible to define fields and methods in traits. Traits are initialized following a scheme called *linearization* [2], which complicates the initialization check. The following example illustrates the subtlety related to linearization:

```
trait TA { val x = "EPFL" }
trait TB { def x: String; val n = x.length }
class Foo extends TA with TB
class Bar extends TB with TA
new Foo // ok
new Bar // error
```

In the code above, the class `Foo` and class `Bar` only differ in the order in which the traits are mixed in. For the class `Foo`, the body of the trait `TA` is evaluated before the body of `TB`, thus the expression `new Foo` works as expected. In contrast, `new Bar` throws an exception, because the body of the trait `TB` is evaluated first, so the field `x` in `TA` is not yet initialized when it is used in `TB`.

Inner Classes. Inner classes [30] are supported in most object-oriented languages. They create more complexity for initialization, as the instances of inner classes have privileged access to the outer `this`. The following example illustrates an interaction between inner classes and outer classes during initialization:

```
class Trees {
  class ValDef { counter += 1 }
  class EmptyValDef extends ValDef
  val theEmptyValDef = new EmptyValDef
  private var counter = 0
}
```

The code above is problematic, as the field `counter` is indirectly used before being initialized during evaluation of the expression `new EmptyValDef`.

Properties. In languages such as Scala and Kotlin, fields are actually properties, accesses of public fields are dynamic method calls, as the following code shows:

```
class A { val a = "Bonjour"; val b: Int = a.size }
class B extends A { override val a = "Hi" }
new B
```

The code above will throw a null-pointer exception at runtime when initializing the field `A.b`, as the code `a.size` will access the field `B.a`, which is not yet initialized.

1.4 Engineering Challenges

Designing a safe initialization system for a practical programming language is an art that strikes a balance between *safety*, *usability*, *expressiveness*, *performance* and *simplicity*.

Safety. A safe initialization checker soundly over-approximates program semantics. A practical checker should be sound for common and reasonable usage, and the checks should always terminate. While safety is a noble goal in theoretical work, in practice it may be weakened with unsafe switches, such as `@unchecked`, to support rare code patterns. Otherwise, the system may become unnecessarily complex for common usage, which harms usability or even makes the system impractical.

Usability. A user-friendly initialization system should not incur much syntactic overhead. The overhead usually manifests itself in the form of annotations. The rules imposed by the system should be easy to learn and reason about. The error messages should be informative and facilitate bug fixes.

Expressiveness. An expressive initialization system should support common and reasonable initialization patterns. In particular, usage of already initialized fields, calling methods on *this*, and creation of cyclic data structures should be supported.

Performance. It is recommended that checks for programs should be enabled as compiler errors whenever possible [4]. Consequently, the checker has to be fast to be integrated in a compiler.

Simplicity. Compilers are subtle pieces of software with high complexity. An initialization system should not compound the complexity. In particular, for statically typed languages, it should not complicate the core type system of the compiler. Ideally, the initialization system should be able to be explained by a simple theory.

1.5 Existing Work

There have been attempts to address the challenges both in industrial languages and academic research.

1.5.1 Industrial Languages

Existing programming languages sit at two extremes. On one extreme, we find languages such as Java, C++, Scala, where programmers can use `this` even if it is not fully initialized, devoid of any safety guarantee. On the other extreme, we find languages such as Swift, which ensures safe initialization, but is overly restrictive. In Swift, the initialization of cyclic data structures is not supported, calling methods on `this` is forbidden, even the usage of already initialized fields is limited. For example, in the following Swift code, while the usage of `x` to initialize `y` is allowed, the usage of `y` to initialize `f` is illegal, which is a surprise:

```
class Position {
    var x, y: Int
    var f: () -> Int
    init() {
        x = 4
        y = x * x                    // OK
        f = { () -> Int in self.y } // error
    }
}
```

The current Swift compiler is incapable of handling the usage of already initialized fields inside a function. It reports the following error for the code above:

```
code/position.swf:7:14: error: 'self' captured by a closure before all members were
initialized
        f = { () -> Int in self.y } // error
            ^
code/position.swf:3:10: note: 'self.f' not initialized
    var f: () -> Int
        ^
```

1.5.2 Masked Types

Masked types [16] is an expressive, flow-sensitive type-and-effect system that addresses the theoretical challenges.

A masked type $T\backslash f$ denotes objects of the type T, where the masked field f cannot be accessed. Each method has an effect signature of the form $\overline{M_1} \rightsquigarrow \overline{M_2}$, which means that the method can only be called if `this` conforms to the masks $\overline{M_1}$, and the resulting masks for `this` after the call is $\overline{M_2}$. A fully initialized object of class C has all its fields initialized, it thus takes an unmasked type C. However, there are several obstacles to make the system practical.

First, the system incurs cognitive load and syntactic overhead. Many concepts are introduced in the system, such as *subclass masks*, *conditional masks*, *abstract masks*, each with non-trivial syntax. The paper mentions that inference can help to remove the syntactic burden. However, it leaves open the formal development of such an inference system.

Second, the system, while expressive, is insufficient for simple and common use cases due to the missing support for *typestate polymorphism*. This can be seen from the following example,

where we want the method `g` to be called for any initialization state of `this`:

```
1 class C {
2   def g(): Int = 100 // effect of g: ∀M.M ⇝ M
3 }
```

As the method `g` can be called for `this` with any masks, we would like to give it the (imaginary) polymorphic effect signature $\forall M.M \rightsquigarrow M$, which is not supported. Even if an extension of the system supports polymorphic effect signatures, it will only incur more syntactic overhead.

1.5.3 The Freedom Model

Summers and Müller [12] propose a light-weight, flow-insensitive type system for safe initialization, which we call *the Freedom Model*.

The freedom model classifies objects into two groups: *free*, that is under initialization, and *committed*, that is transitively initialized. Field accesses on free objects may get `null`, while committed objects can be used safely. To support typestate polymorphism, it introduces the typestate *unclassified*, which means either *free* or *committed*. With subtyping, typestate polymorphism becomes just *subtyping polymorphism*.

The freedom model supports the creation of cyclic data structures with light-weight syntax. However, the formal system does not address the usage of already initialized fields in the constructor. When an object is free, accessing its field will return a value of the type `unclassified C?`, which means the value could be `null`, free or committed. In the implementation, they introduce *committed-only fields* which can be assumed to be committed with the help of a dataflow analysis. However, the paper leaves open the formal treatment of the dataflow analysis. Our work will address the problem.

Moreover, the abstraction *free* is too coarse for some use cases. This is demonstrated by the following example:

```
1 class Parent {
2   var child = new Child(this)
3   var tag: Int = child.tag     // error in freedom model
4 }
5 class Child(parent: Parent @free) {
6   var tag: Int = 10
7 }
```

According to the freedom model, the expression `child` in line 3 will be typed as *free*, thus the type system cannot tell whether the field `child.tag` is initialized or not. But conceptually we know that all fields of `child` are initialized by the constructor of the class `Child`. In this work we propose a new abstraction to improve expressiveness in such cases.

To our best knowledge, no existing proposals can be easily extended to handle complex language features, such as *inner classes*, *functions*, *properties* and *traits*.

1.6 Contribution

Based on the study of this thesis, we implement an initialization system for Scala 3, which supports inner classes, traits, properties and functions. No annotations are required for the system to work. Given the following Scala program:

```
1  abstract class AbstractFile {
2    def name: String
3    val extension: String = name.substring(4)
4  }
5
6  class RemoteDoc(url:String) extends AbstractFile {
7    val localFile: String = url.hashCode // error
8    def name: String = localFile
9  }
```

Our system will report that the field `localFile` is used before initialization:

```
1  -- Error: code/AbstractFile.scala:7:4 ------------------------------
2  7 |   val localFile: String = url.hashCode + ".tmp" // error
3    |       ^
4    |    Access non-initialized field value localFile. Calling trace:
5    |     -> val extension: String = name.substring(4) [ AbstractFile.scala:3 ]
6    |      -> def name: String = localFile              [ AbstractFile.scala:8 ]
```

Our system also supports interactions between an inner class and an outer class without the need for any annotation, as the following code shows

```
1      class Trees {
2        class ValDef { counter += 1 }     // error
3        class EmptyValDef extends ValDef
4
5        val theEmptyValDef = new EmptyValDef
6        private var counter = 0
7      }
```

In the code above, the field `counter` is used before it is initialized. If we move `counter` at line 6 before line 5, our system will not issue any warnings.

Our work makes contributions in the following areas:

1. Identification of initialization principles. We identify and advocate four design principles which originate from formal reasoning about initialization: *monotonicity*, *authority*, *stackability* and *scopability*. The principles enable *local reasoning* as well sound *strong updates* in a flow-insensitive system.

2. Principled design of class constructors. Based on the principles, we propose that class-based programming languages should adopt class parameters and mandatory field initializers. As far as we know, this is the first work that bases the syntactic design of constructors on initialization principles.

3. Better understanding of local reasoning. *Local reasoning* is a key requirement for simple

and fast initialization systems. However, while prior work [12] takes advantage of local reasoning to design simple initialization systems, the concept of local reasoning is neither mentioned nor defined precisely. Identifying local reasoning as a concept with a better understanding enables it to be applied in the design of future initialization systems.

4. A more expressive type-based model. We propose a more expressive type-based model for initialization based on the abstractions *cold*, *warm* and *hot*. The introduction of the abstraction *warm* improves the expressiveness of the freedom model [12] which classifies objects as either initialized (i.e. cold) or uninitialized (i.e. hot).

5. A novel type-and-effect inference system. We propose a type-and-effect inference system for a practical fragment of the type-based model. Existing work usually depends on some unspecified inference or analysis to cut down syntactic overhead [16, 12, 11]. We are the first to present a formal inference system on the problem of safe initialization. Meanwhile, to our knowledge, we are the first to demonstrate the technique of aliasing control in a type-and-effect system with the concept of *potentials*.

6. Implementation in Scala 3. We implement an initialization system in the Scala 3 compiler and evaluate it on several real-world projects. The system is capable of handling complex language features, such as inner classes, traits and functions.

Chapter 2

Principles: Monotonicity, Authority, Stackability and Scopability

Civilization advances by extending the number of important operations which we can perform without thinking about them.

— Alfred North Whitehead, *An Introduction to Mathematics*

In this chapter, we identify four design principles from the formal reasoning about initialization. The principles shed light on the syntactic design of class constructors.

2.1 Principles

We uphold four design principles for initialization systems, namely *monotonicity*, *authority*, *stackability* and *scopability*.

2.1.1 Monotonicity

Roughly, monotonicity means that the initialization states cannot be reversed. Monotonicity is essential for the safe usage of initialized fields.

One obvious violation of monotonicity is to assign `null` to a field which is already initialized with a non-null value. To fix the billion-dollar mistake [15], `null` can be removed from the language in favor of the type `Option[T]`.

If a system guarantees that *initialized fields continue to be initialized*, we say it observes **weak monotonicity**.

However, stronger concepts of monotonicity are required for initialization safety. The following example shows that assignment of non-null values may also cause problems:

```
1 trait Reporter { def report(msg: String): Unit }
2 class FileReporter(ctx: Context) extends Reporter {
3     ctx.typer.reporter = this            // problem: ctx reaches uninitialized object
4     val file: File = new File("report.txt")
```

```
5     def report(msg: String) = file.write(msg)
6 }
```

In the code above, suppose `ctx` is a transitively initialized value. Now the assignment at line 3 makes `this`, which is not initialized, reachable from `ctx`. This makes any operation on `ctx` dangerous, as it may indirectly reach uninitialized fields of the current object.

If a system additionally guarantees that *transitively initialized objects continue to be transitively initialized*, we say it observes **strong monotonicity**.

However, to enable safe usage of already initialized fields of an object under initialization, we need an even stronger concept, as the following example demonstrates:

```
1 trait Reporter { def report(msg: String): Unit }
2 class ProxyReporter(underlying: Reporter) extends Reporter {
3   this.underlying = this     // problematic, should be rejected
4   // ...
5   val n = 10
6   // ...
7 }
```

In the code above, we assume that initially the field `underlying` of the class `ProxyReporter` is transitively initialized — thus the object may be used freely. However, at line 3 we reassign the field with `this`, which is not initialized. The assignment is valid in the context of strong monotonicity, because the object that the field `underlying` used to refer to is not changed, and strong monotonicity is only about the initialization state of objects rather than fields. The assignment makes the initialization state of the field `underlying` go backward. Now usage of the field `underlying` may potentially reach uninitialized fields.

The initialization state of an object not only includes the fields that are assigned, but also the initialization states of the objects that the fields point to. **Perfect monotonicity** enforces that the initialization state of a field, i.e. the initialization state of the object that it points to, is monotone across mutations.

2.1.2 Authority

The principle of authority says that the fields of an object may only be initialized at specific locations in the class constructor. That is, each field should have a unique location in the constructor where it is officially initialized. Unrestricted initialization of a field breaks monotonicity thus compromises initialization safety. This can be demonstrated by the code example below:

```
1 class Knot {
2   var self: Knot = { e; this }
3   var n: Int = 10
4   // ...
5 }
```

In the code above, suppose that a side effect of evaluating the expression `e` is to initialize the field `this.self` to a transitively initialized value. Now initializing the field `self` with the value `this` breaks monotonicity!

To ensure that the initialization at line 2 maintains monotonicity, we have to make sure that after executing `e`, the field `self` is not initialized with a more advanced state. A simple approach to guarantee that is to enforce that fields may only be initialized at specific locations in the constructor.

Note that assigning to a field before its formal initialization is semantically harmless as long as we do not regard the field as initialized at the point of assignment. However, as such assignments will be overwritten by field initialization later, we think such code patterns should be rejected.

A natural consequence of the principle of authority is that it is only safe to formally promote the initialization state of an object at the end of the constructor, after all its fields are officially initialized.

The principle of authority is a direct consequence of enforcing monotonicity in a *flow-insensitive* system. Conceptually, in a flow-sensitive system, we may advance the initialization state of an object at any location, without worrying about whether the advancement at one location is backwards relative to another location. However, in a flow-insensitive system, if the initialization status of an object may be advanced at arbitrary locations, there is no guarantee that an advancement of object status at one location is indeed an advancement, as the object might be already advanced further at another location.

The principle of authority is the key to ensure sound strong updates in a flow-insensitive system. We will see that flow-insensitivity affords a simple solution to the challenge of *typestate polymorphism* and it enables simpler initialization systems (chapter 4).

2.1.3 Stackability

The principle of *stackability* stipulates that all fields of an object should be initialized at the end of the class constructor. Consequently, we know that all fields of a freshly created object can be safely accessed.

Note that having all fields of an object initialized does not mean the object is *transitively* initialized, as fields of the object may point to an uninitialized object directly or indirectly.

If we push an object on a stack when it comes into existence, and remove it from the stack when all its fields are assigned, we will find that the object to be removed is always at the top of the stack. That is why we call the principle *stackability*. This is illustrated in Figure 2.1.

Stackability is broken in languages Java and C++, as they do not require that all fields are assigned at the end of the constructor. Control effects such as exceptions may also compromise this property, as the following example shows:

```
1 class MyException(val b: B) extends Exception("")
2
3 class A {
4   val b = try { new B } catch { case myEx: MyException => myEx.b }
5   println(b.a)            // error: a is not initialized
6 }
7
8 class B {
9   throw new MyException(this) // problem: should be rejected
```

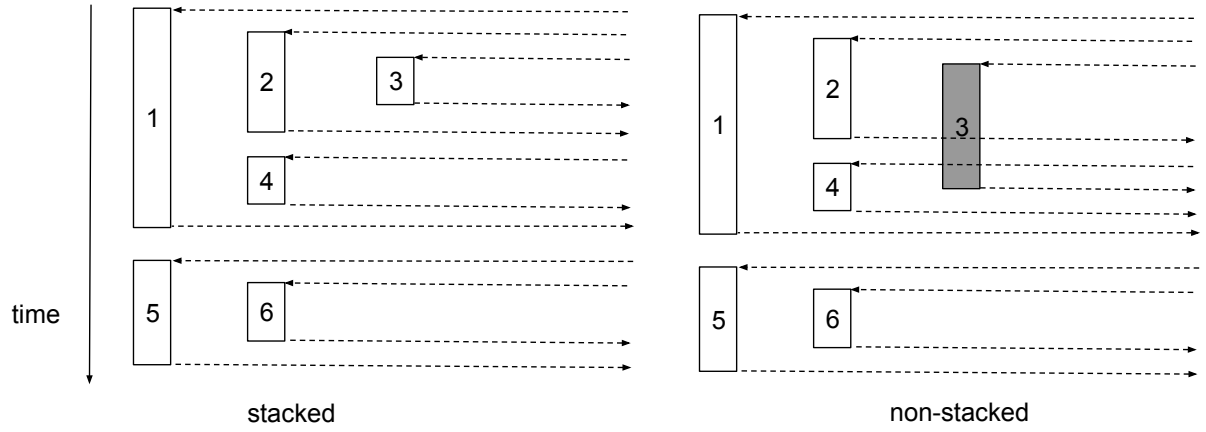


Figure 2.1 – Each block represents the initialization duration of an object, i.e., from the creation of the object to the point where all fields are assigned.

```
10    val a: Int = 1
11  }
```

In the code above, the exception teleports the uninitialized value wrapped in an exception. To enforce stackability, we need to ensure that values to control effects are fully initialized.

Without this principle, it will be difficult, for an initialization checker and programmers, to reason about when an object becomes fully initialized. An initialization checker would have to explicitly track the set of initialized fields of an object in the program, which gets complex in the presence of aliasing.

With stackability, the system guarantees that object initialization form stacked time regions, which greatly simplifies the reasoning about initialization. For example, in the following code, the object pointed to by the field $this.child$ must be fully initialized if the object pointed to by $this$ in the class `Parent` is fully initialized.

```
1  class Parent { var child: Child = new Child(this) }
2  class Child(parent: Parent)
```

2.1.4 Scopability

The principle of scopability says that the access to uninitialized objects should be controlled by static scoping. Intuitively, it means that a method may only access pre-existing uninitialized objects through its environment, i.e. method parameters and `this`.

Objects under initialization are dangerous when used without care, therefore the access to them should be controlled. Scopability imposes discipline on accessing uninitialized objects. If we regard uninitialized objects as capabilities, then scopability restricts that there should be no side channels for accessing those capabilities. All accesses have to go through the explicit channel, i.e. method parameters and `this`. This makes it much easier to control the capabilities: we only need to impose rules on the explicit channel.

Control effects like coroutines, delimited control, resumable exceptions may break the property, as they can transport a value outwards in the stack (not in scope) to be reachable from

the current scope. Global fields can also serve as a teleport thus breaking this property, as the following example shows:

```
object O { var a: A = null }
class A {
  O.a = this          // problem: should be rejected
  val b = new B()
  val name = "a"
}

class B {
  val size = O.a.name.length
}
```

In the code above, the global object `O` teleports an uninitialized instance of class `A`, which results in an initialization error in class `B` when the object is used. To restore scopability, we need to ensure that the teleported values are fully initialized.

2.2 Design of Constructors

We reflect on the design of constructors from the perspective of safe initialization.

2.2.1 A Critique of Traditional Constructors

In the light of the design principles, we find that traditional class constructors, like those in Java, C++, C♯, Swift, etc., are flawed in several aspects. The following example shows a typical class with traditional class constructors in Java:

```
class Car {
  int modelYear;
  String modelName;

  public Car(int year, String name) {
    modelYear = year;
    modelName = name;
  }
}
```

In traditional class constructors, there is no distinction between *field initialization* and *field reassignment* in the syntax. The fields of a class are usually declared in the class body, and then initialized by assignment. From the syntax alone, it is not easy to distinguish field initialization from field reassignment. This causes several problems.

First, it makes enforcement of monotonicity more difficult. As we learned above, monotonicity can only be broken with reassignment, thus special rules should be enforced for field reassignment. For example, while a field may be initialized with an object under initialization, it is generally unsafe to do so for reassignment. The indistinguishability in syntax makes it harder

to enforce different rules to field reassignment and field initialization.

Second, the syntax obscures when a field is *officially* initialized, as required by the principle of authority. It is syntactically possible to initialize a field in a method or inside if-expressions. This unnecessarily complicates the reasoning about the safe usage of already initialized fields.

Third, it complicates the check whether all fields of a class are initialized at the end of the constructor. This is because assignment of a field may happen in an *if* branch or in a method, thus non-trivial analysis has to be employed to enforce stackability. In the freedom model [12], *definite assignment analysis* [7] is used to ensure that all fields of a class are assigned at the end of the constructor.

Fourth, it complicates the check that an immutable field is not reassigned. The system of masked types [16] introduces the concept of *must-mask* to support initialization of immutable fields, e.g. `C\d!` means that the field `d` is definitely not assigned, while a normal mask `C\d` means the field `d` may not be assigned.

In object-oriented programming languages, programmers sometimes create convenience constructors to call the main constructor (constructor chaining). For example, Scala has the concept of *secondary constructors* which are supposed to eventually call the implicit *primary constructor* [22]. In Swift, there are *designated initializers* and *convenience initializers*, the latter are supposed to call the former [1]. Our critique of constructors does not extend to convenience constructors or secondary constructors, as they have no impact on the reasoning principles of initialization.

2.2.2 Class Parameters and Mandatory Field Initializers

The separation of field declaration and initialization is caused by the fact that the arguments for object initialization are only available as constructor parameters. However, that is not the only possibility for the design of object initialization. For example, with Scala-like class parameters, there is no need for explicit constructors:

```
1 class RemoteDoc(url:String) {
2   val localFile: String = url.hashCode
3   def name: String = localFile
4 }
```

We advocate Scala-like class parameters and mandatory field initialization on declaration. This design has many benefits:

1. We may distinguish field initialization and reassignment in syntax, which makes it possible to give different rules to enforce monotonicity.

2. The principle of authority is followed with no syntactic overhead, as a field is always officially initialized when it is declared.

3. It enforces stackability in syntax, as fields are initialized when they are declared.

4. Enforcing no reassignment of immutable fields becomes as simple as a syntactic check.

language	`class C(f:Int)`	`var f:Int = e`	typed	year
Java	○	○	●	1995
C#	○	○	●	2000
D	○	○	●	2001
Scala	●	◐	●	2003
Ceylon	●	○	●	2011
Dart	○	○	●	2011
Kotlin	●	◐	●	2011
TypeScript	○	○	●	2012
Crystal	○	○	●	2014
Swift	○	○	●	2014
JavaScript	○	○	○	2015

Table 2.1 – Object initialization of industrial object-oriented languages. **typed** means whether the language is statically typed. **year** is the year the language or language feature is first introduced. The remaining two columns are about whether the language supports class parameters and whether field initializers are mandatory.

In Table 2.1, we list the design of object initialization in industrial object-oriented programming languages. As can be seen, Scala and Kotlin are the only two languages that favor class parameters and mandatory field initialization. However, in Scala, it is possible to work around mandatory field initializers via the syntax `var x: T = _`. In Kotlin, *late-initialized properties* serve as an escape from mandatory field initializers [1].

2.3 Related Work

Monotonicity is a well-known technique called *heap monotonic typestate* to ensure soundness in the presence of aliasing [23]. Monotonicity is enforced in *raw types* [24], *masked types* [16] and *the freedom model* [12].

The freedom model enforces *strong monotonicity* instead of *perfect monotonicity*. As a result, it has to strengthen monotonicity via an extension called *committed-only fields* to support the usage of already initialized fields in the implementation.

The principle of stackability dates back to *delayed types* [19], and is followed in the freedom model [12]. However, it is not enforced in *masked types* [16]. Consequently, the system has to track the initialized fields of an object explicitly in the system. To deal with aliasing, the authors introduced *conditionally masked types* to type check the following program:

```
1 class Knot {
2     var self: Knot\self[this.self] = this
3 }
```

The conditional mask `Knot\self[this.self]` describes that the field `self` references a partially initialized object, which will become fully initialized when the field `this.self` is initialized.

[1] https://kotlinlang.org/docs/reference/properties.html#late-initialized-properties-and-variables

The principle of scopability is related to *lexical scoping* or *static scoping* [32]. A language that follows the discipline of static scoping naturally enjoys the property unless the language supports control effects, such as delimited control [20, 35] and algebraic effects [6]. In the presence of control effects, we need to ensure that the teleported values are fully initialized to maintain scopability. In contrast, *dynamic scoping* [17] essentially violates the property.

Masked types [16] is a *flow-sensitive* system, thus it does not need to resort to the principle of authority. However, the choice of flow-sensitivity makes it difficult for the system to address the challenge of *typestate polymorphism*. In general, they have to resort to parametric polymorphism to solve the problem, which further complicates their system.

The freedom model [12] cleverly uses a *flow-insensitive* type system to support the creation of cyclic data structures, and it addresses the challenge of typestate polymorphism via subtyping. However, the principle of authority is not made clear in their work.

2.4 Conclusion

In this chapter, we identify four design principles from formal reasoning about initialization, namely *monotonicity*, *authority*, *stackability* and *scopability*. Based on the principles, we propose that traditional class constructors should be replaced by class parameters and mandatory field initializers.

One topic we do not touch is what do the principles mean formally? What roles do they play in formal reasoning about initialization? That will be the topic of the next two chapters, where the principles are formalized and their roles in the proofs uncovered.

Chapter 3

Local Reasoning

Assumptions are the things you don't know you're making.

— Douglas Adams, Mark Cawardine, "Last Chance to See"

An important insight in the freedom model [12] is that *if a constructor is called with only transitively initialized arguments, the resulting object is transitively initialized.* We give this insight a name, *local reasoning* of initialization; it enables reasoning about initialization without the global analysis of a program, which is the key for simple and fast initialization systems. The insight also holds for method calls: if the receiver and arguments of a method call are transitively initialized, so is the result.

But how can we justify the insight? While a justification can be found in the soundness proof of the freedom model, it is obscured in a monolithic proof structure (see Lemma 1 in the freedom model paper [12]). We provide a modular understanding of local reasoning by identifying three semantic properties, which we call *weak monotonicity*, *stackability* and *scopability*. Identifying local reasoning as a concept with a better understanding enables it to be applied in the design of future initialization systems. The properties can be explained roughly as follows:

- Weak monotonicity: initialized fields continue to be initialized.
- Stackability: all fields of a class should be initialized at the end of the class constructor.
- Scopability: access to objects under initialization should be controlled by static scoping.

To study the properties more formally, we first introduce a small language.

3.1 A Small Language

Our language resembles a subset of Scala having only top-level classes, mutable fields and methods.

$$
\begin{array}{rcl}
\mathcal{P} \in Program & ::= & (\overline{\mathcal{C}}, D) \\
\mathcal{C} \in Class & ::= & class\ C(\overline{\hat{f}{:}T})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \} \\
\mathcal{F} \in Field & ::= & var\ f{:}T = e \\
e \in Exp & ::= & x \mid this \mid e.f \mid e.m(\overline{e}) \mid new\ C(\overline{e}) \mid e.f = e; e \\
\mathcal{M} \in Method & ::= & def\ m(\overline{x{:}T}) : T = e \\
S, T, U \in Type & ::= & C
\end{array}
$$

A program $\mathcal{P}$ is composed of a list of class definitions and an entry class. The entry class must have the form $class\ D\ \{\ def\ main : T = e\ \}$. The program runs by executing e.

A class definition contains class parameters ($\hat{f}{:}T$), field definitions ($var\ f{:}T = e$) and method definitions. Class parameters are also fields of the class. All class fields are mutable. As a convention, we use f to range over all fields, and $\hat{f}$ to only range over class parameters.

An expression (e) can be a variable (x), self reference ($this$), field access ($e.f$), method call ($e.m(\overline{e})$), class instantiation ($new\ D(\overline{e})$), or block expression ($e.f = e; e$). The block expression is introduced to avoid introducing the syntactic category of statements in the presence of assignments, which simplifies the presentation and meta-theory.

A method definition is standard. We restrict the method body to just expressions. This choice simplifies the meta-theory without loss of expressiveness thanks to block expressions.

The following constructs are used in defining the semantics:

$$
\begin{array}{rcl}
\Xi \in ClassTable & = & ClassName \rightharpoonup Class \\
\sigma \in Store & = & Loc \rightharpoonup Obj \\
\rho \in Env & = & Name \rightharpoonup Value \\
o \in Obj & = & ClassName \times (Name \rightharpoonup Value) \\
l, \psi \in Value & = & Loc
\end{array}
$$

We use ψ to denote the value of $this$, σ corresponds to the heap, ρ corresponds to the local variable environment of the current stack frame.

The big-step semantics, presented in 3.1 is standard, thus we omit detailed explanation. The only note is that non-initialized fields are represented by missing keys in the object, instead of a *null* value. Newly initialized objects have no fields, and new fields are gradually inserted during initialization until all fields defined by the class have been assigned.

Note that this language does not enjoy initialization safety, and it is the task of later sections to make it safe. However, the language enjoys local reasoning of initialization.

3.2 Abstractions: Cold, Warm, Hot

3.2.1 Intuition

In object-oriented programming, objects are connected with each other to form a graph. The objects are connected via fields: the field value of an object may point to another object. Naturally, the initialization status of an object not only includes the initialization status of its own fields, but also the initialization status of all reachable objects. Consequently, we may classify objects in the graph into three categories according to their initialization status:

Program evaluation $\boxed{[\![(\overline{\mathcal{C}}, D)]\!] = (l, \sigma)}$

$$
\begin{array}{lcl}
[\![(\overline{\mathcal{C}}, D)]\!] & = & [\![e]\!]\,(\{\ l \mapsto (D, \emptyset)\ \}, \emptyset, l) \\
 & & \text{where } \Xi = \overline{C \to \mathcal{C}} \text{ and } l \text{ is a fresh location} \\
 & & \text{and } \Xi(D) = class\ D\ \{\ def\ main : T = e\ \}
\end{array}
$$

Expression evaluation $\boxed{[\![e]\!]\,(\sigma, \rho, \psi) = (l, \sigma')}$

$$
\begin{array}{lcl}
[\![x]\!]\,(\sigma, \rho, \psi) & = & (\rho(x), \sigma) \\
[\![this]\!]\,(\sigma, \rho, \psi) & = & (\psi, \sigma) \\
[\![e.f]\!]\,(\sigma, \rho, \psi) & = & (\omega(f), \sigma_1) \text{ where } (l_0, \sigma_1) = [\![e]\!]\,(\sigma, \rho, \psi) \\
 & & \text{and } (\_, \omega) = \sigma_1(l_0) \\
[\![e_0.m(\overline{e})]\!]\,(\sigma, \rho, \psi) & = & [\![e_1]\!]\,(\sigma_2, \rho_1, l_0) \\
 & & \text{where } (l_0, \sigma_1) = [\![e_0]\!]\,(\sigma, \rho, \psi) \\
 & & \text{and } (C, \_) = \sigma_1(l_0) \\
 & & \text{and } lookup(C, m) = def\ m(\overline{x{:}T}) : T = e_1 \\
 & & \text{and } (\overline{l}, \sigma_2) = [\![\overline{e}]\!]\,(\sigma_1, \rho, \psi) \\
 & & \text{and } \rho_1 = \overline{x \mapsto l} \\
[\![new\ C(\overline{e})]\!]\,(\sigma, \rho, \psi) & = & (l, \sigma_3) \\
 & & \text{where } (\overline{l}, \sigma_1) = [\![\overline{e}]\!]\,(\sigma, \rho, \psi) \\
 & & \text{and } \sigma_2 = [l \mapsto (C, \emptyset)]\sigma_1 \text{ where } l \text{ is fresh} \\
 & & \text{and } \sigma_3 = init(l, \overline{l}, C, \sigma_2) \\
[\![e_1.f = e_2; e]\!]\,(\sigma, \rho, \psi) & = & [\![e]\!]\,(\sigma_3, \rho, \psi) \\
 & & \text{where } (l_1, \sigma_1) = [\![e_1]\!]\,(\sigma, \rho, \psi) \\
 & & \text{and } (l_2, \sigma_2) = [\![e_2]\!]\,(\sigma_1, \rho, \psi) \\
 & & \text{and } \sigma_3 = assign(l_1, f, l_2, \sigma_2)
\end{array}
$$

Initialization

$$
\begin{array}{lcl}
init(\psi, \overline{l}, C, \sigma) & = & [\![\overline{\mathcal{F}}]\!]\,(\sigma_1, \psi) \\
 & & \text{where } lookup(C) = class\ C(\overline{\hat{f}{:}T})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \} \\
 & & \text{and } \sigma_1 = assign(\psi, \overline{\hat{f}}, \overline{l}, \sigma) \\
[\![var\ f : D = e]\!]\,(\sigma, \psi) & = & assign(\psi, f, l_1, \sigma_1) \text{ where } (l_1, \sigma_1) = [\![e]\!]\,(\sigma, \emptyset, \psi)
\end{array}
$$

Helpers

$$
\begin{array}{lcl}
[\![\overline{e}]\!]\,(\sigma, \rho, \psi) & = & fold\ \overline{e}\ (Nil, \sigma)\ f \text{ where} \\
 & & f\ (ls, \sigma_1)\ e = let\ (l, \sigma_2) = [\![e]\!]\,(\sigma_1, \rho, \psi)\ in\ (l :: ls, \sigma_2) \\
[\![\overline{\mathcal{F}}]\!]\,(\sigma, \psi) & = & fold\ \overline{\mathcal{F}}\ \sigma\ f \text{ where } f\ \sigma_1\ \mathcal{F} = [\![\mathcal{F}]\!]\,(\sigma_1, \psi) \\
assign(\psi, f, l, \sigma) & = & [\psi \mapsto (C, [f \mapsto l]\omega)]\sigma \text{ where } (C, \omega) = \sigma(\psi) \\
assign(\psi, \overline{f}, \overline{l}, \sigma) & = & [\psi \mapsto (C, [\overline{f \mapsto l}]\omega)]\sigma \text{ where } (C, \omega) = \sigma(\psi)
\end{array}
$$

Figure 3.1 – Big-step semantics, defined as a definitional interpreter.

- Objects with uninitialized fields.
- Objects with their own fields initialized, but reaching objects with uninitialized fields.
- Objects which do not reach any object with uninitialized fields (including themselves).

We name the categories above *cold, warm* and *hot* respectively. If the initialization of objects

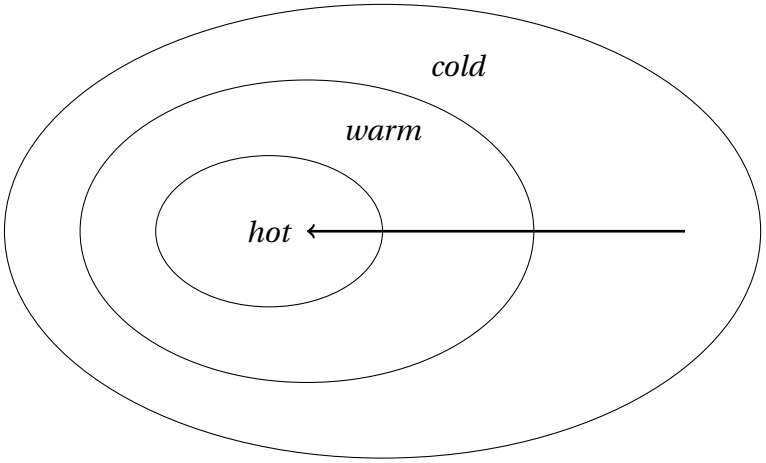


Figure 3.2 – The arrow indicates the states that an object passes through during its initialization.

is monotone, then object initialization is the process of going from cold to warm and eventually to hot.

The initialization status of objects matters, because objects of different status may be used differently in the program. If an object is cold, then it is illegal to access fields on the object, as they might not be initialized yet. For a warm object, field access is generally safe. However, the result of the access must be used with care, as it may be cold or warm. For hot objects, they can be used freely without the worry of reaching an uninitialized field, as all reachable objects from a hot object are initialized.

From the perspective of usage, we may safely treat a hot object as a warm or cold object, or a warm object as a cold object in the program, without worrying about creating initialization errors. This suggests the following slightly adapted definition:

cold	A cold object *may* have uninitialized fields.
warm	A warm object has all its fields initialized.
hot	A hot object has all its fields initialized and only reaches hot objects.

The relationship of the three states are illustrated in Figure 3.2. If we think of the states in terms of maturity levels, then the states form a lattice: $hot \sqsubseteq warm \sqsubseteq cold$. We will employ the ordering to check whether a value conforms to the expected initialization state of a parameter in a method call in chapter 4. Passing a cold value as a hot value to a method is an error, while it is safe to use a hot value as a cold value.

3.2.2 Formal Definitions

DEFINITION 3.1 (reachability)**.** *An object l' is reachable from l in the heap σ, written $\sigma \vDash l \rightsquigarrow l'$, is defined below:*

$$\frac{l \in dom(\sigma)}{\sigma \vDash l \rightsquigarrow l} \qquad \frac{\sigma \vDash l_0 \rightsquigarrow l_1 \qquad (\_, \omega) = \sigma(l_1) \qquad \exists\, f.\ \omega(f) = l_2 \qquad l_2 \in dom(\sigma)}{\sigma \vDash l_0 \rightsquigarrow l_2}$$

LEMMA 3.1 (transitivity of reachability). *If* $\sigma \vDash l_1 \rightsquigarrow l_2$ *and* $\sigma \vDash l_2 \rightsquigarrow l_3$, *then* $\sigma \vDash l_1 \rightsquigarrow l_3$.

Proof. By induction on the definition of reachability. □

DEFINITION 3.2 (reachability for set of locations).

$$\begin{aligned} \sigma \vDash L \rightsquigarrow l &\triangleq \exists l' \in L.\sigma \vDash l' \rightsquigarrow l \\ \sigma \vDash l \rightsquigarrow L &\triangleq \exists l' \in L.\sigma \vDash l \rightsquigarrow l' \end{aligned}$$

DEFINITION 3.3 (cold). *An object is cold if it exists in the heap, formally*

$$\sigma \vDash l : cold \triangleq l \in dom(\sigma)$$

DEFINITION 3.4 (warm). *An object is warm if all its fields are assigned, formally*

$$\sigma \vDash l : warm \triangleq \exists (C, \omega) = \sigma(l) \bigwedge fields(C) \subseteq dom(\omega)$$

DEFINITION 3.5 (hot). *An object is hot if all reachable objects are warm, formally*

$$\sigma \vDash l : hot \triangleq l \in dom(\sigma) \bigwedge \forall l'.\sigma \vDash l \rightsquigarrow l' \Longrightarrow \sigma \vDash l' : warm$$

From the definition, it is easy to see that hot implies $warm$ and $warm$ implies $cold$.
An immediate result of the definition of *hot* is transitivity:

LEMMA 3.2 (hot transitivity). *If* $\sigma \vDash l : hot$ *and* $\sigma \vDash l \rightsquigarrow l'$, *then* $\sigma \vDash l' : hot$.

Proof. For any l'', if $\sigma \vDash l' \rightsquigarrow l''$, then $\sigma \vDash l \rightsquigarrow l''$. As l is hot, thus $\sigma \vDash l'' : warm$. □

DEFINITION 3.6 (initialization state of sets). *A set of objects has an initialization state* μ *if all objects in the set have that initialization state, formally*

$$\sigma \vDash L : \mu \triangleq \forall l \in L.\sigma \vDash l : \mu$$

3.3 Formal Local Reasoning

3.3.1 Three Concepts of Monotonicity

The idea of *monotonicity* dates back to *heap monotonic typestates* [25]. There are, however, at least three different concepts of monotonicity.

Weak monotonicity means initialized fields continue to be initialized. More formally, we may prove the following theorem:

THEOREM 3.1 (Weak Monotonicity).

$$[\![e]\!](\sigma, \rho, \psi) = (l, \sigma') \Longrightarrow \sigma \preceq \sigma'$$

In the above, the predicate *weak monotonicity* ($\sigma \preceq \sigma'$) is defined below:

DEFINITION 3.7 (Weak Monotonicity)**.**

$$\sigma \preceq \sigma' \quad \triangleq \quad \forall l \in dom(\sigma).(C,\omega) = \sigma(l) \quad \Longrightarrow \quad (C,\omega') = \sigma'(l)$$

While weak monotonicity is sufficient to justify local reasoning, stronger monotonicity is required for initialization safety. For example, the freedom model [12] enforces *strong monotonicity*:

$$\sigma \preceq \sigma' \quad \triangleq \quad \forall l \in dom(\sigma).\sigma \vDash l : \mu \Longrightarrow \sigma' \vDash l : \mu$$

In the above, we abuse the notation by using μ to denote either *cold*, *warm* or *hot*. Strong monotonicity additionally ensures that hot objects continue to be hot. Therefore, it is always safe to use hot objects freely. However, to enforce safer usage of already initialized fields of non-hot objects, we need an even stronger concept, *perfect monotonicity*:

$$\begin{aligned}\sigma \preceq \sigma' \quad \triangleq \quad & \forall l \in dom(\sigma). \quad (C,\omega) = \sigma(l) \Longrightarrow \\ & (C,\omega') = \sigma'(l) \quad \bigwedge \quad \forall f \in dom(\omega).\sigma \vDash \omega(f) : \mu \Longrightarrow \sigma' \vDash \omega'(f) : \mu\end{aligned}$$

In the above, we abuse the notation by writing directly $\omega'(f)$ to require that $dom(\omega) \subseteq dom(\omega')$. Perfect monotonicity in addition ensures that initialization states of object fields are monotone. It will be problematic if a field is initially assigned a hot value and later reassigned to a non-hot value. The formal system of the freedom model [12] actually allows this. Consequently, in the implementation, they have to introduce *committed-only fields* to restore perfect monotonicity to support the safe usage of already initialized fields with the help of a dataflow analysis.

3.3.2 Stackability

Conceptually, stackability ensures that all newly created objects during the evaluation of an expression e are *warm*, i.e. all fields of the objects are assigned. Formally, the insight can be proved as a theorem:

THEOREM 3.2 (Stackability)**.**

$$[\![e]\!]\,(\sigma,\rho,\psi) = (l,\sigma') \Longrightarrow \sigma \ll \sigma'$$

The predicate $\sigma \ll \sigma'$ is defined below, which says that for any object in the heap σ', either the object is *warm*, or the object pre-exists in the heap σ.

DEFINITION 3.8 (Stacking)**.**

$$\sigma \ll \sigma' \quad \triangleq \quad \forall l \in dom(\sigma').\sigma' \vDash l : warm \bigvee l \in dom(\sigma)$$

Definite assignment [7] can be used to enforce stackability in programming languages. Java, however, only requires that final fields are initialized.

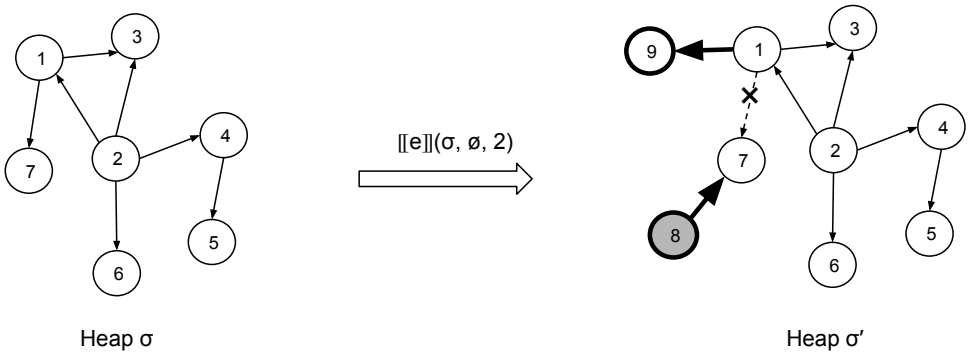


Figure 3.3 – Each circle represents an object and numbers are locations. An arrow means that an object holds a reference to another object. The thick circles and links on the right heap are new objects and links created during the execution. The gray circle indicates the result of the evaluation.

3.3.3 Scopability

Scopability says that the access to uninitialized objects should be controlled by static scoping. Intuitively, it means that a method may only access pre-existing uninitialized objects through its environment, i.e. method parameters and `this`.

Objects under initialization are dangerous when used without care, therefore the access to them should be controlled. Scopability imposes discipline on accessing uninitialized objects. If we regard uninitialized objects as capabilities, then scopability restricts that there should be no side channels for accessing those capabilities. All accesses have to go through the explicit channel, i.e. method parameters and `this`. In contrast, global variables or control-flow effects such as algebraic effects may serve as side channels for teleporting values under initialization. To maintain local reasoning, an initialization system needs to make sure that only initialized values may travel by side channels.

More formally, we can prove the following theorem:

THEOREM 3.3 (Scopability)**.**

$$[\![e]\!]\,(\sigma, \rho, \psi) = (l, \sigma') \Longrightarrow (\sigma, codom(\rho) \cup \{\ \psi\ \}) \lessdot (\sigma', \{\ l\ \})$$

In the above, the predicate $(\sigma, L) \lessdot (\sigma', L')$ is defined below:

DEFINITION 3.9 (Scoping)**.** *A set of addresses $L' \subseteq dom(\sigma')$ is* scoped *by a set of addresses $L \subseteq dom(\sigma)$, written $(\sigma, L) \lessdot (\sigma', L')$, is defined as follows*

$$(\sigma, L) \lessdot (\sigma', L') \quad \triangleq \quad \forall l \in dom(\sigma).\quad \sigma' \vDash L' \rightsquigarrow l \Longrightarrow \sigma \vDash L \rightsquigarrow l$$

The theorem means that if e evaluates to l, then l and every location l' reachable from l in the new heap is either fresh, in that it did not exist in the old heap, or it was reachable from $codom(\rho) \cup \{\ \psi\ \}$ in the old heap.

Note that in the definition of *scoping*, we use $\sigma_1 \vDash L_1 \rightsquigarrow l$ instead of $\sigma_2 \vDash L_1 \rightsquigarrow l$. This is because in a language with mutation, l may no longer be reachable from L_1 in σ_2 due to reassignment. This can be seen in Figure 3.3. Due to scopability, we have $(\sigma, \{2\}) \lessdot (\sigma', \{8\})$. It means if the result object 8 reaches any object which pre-exists in the heap σ, then the object must be reachable from object 2 in the heap σ. The object 7, which is reachable from the object 2 in the heap σ, is no longer reachable from object 2 in the heap σ' due to the removal of the link from object 1 to object 7.

However, the definition of *scoping* is only part of the story. In an evaluation $[\![e]\!](\sigma, \rho, \psi) = (l, \sigma')$, not only the result l is scoped by $codom(\rho) \cup \psi$, but also existing scoped relations should continue to hold. In other words, existing scoping relations are *preserving*. More formally, the following property is also a result of scopability:

$$[\![e]\!](\sigma, \rho, \psi) = (l, \sigma') \Longrightarrow \sigma \rightsquigarrow \sigma' \lessdot codom(\rho) \cup \{\ \psi\ \})$$

The predicate $\sigma_1 \rightsquigarrow \sigma_2 \lessdot L$ is defined as follows:

DEFINITION 3.10 (Scoping Preservation)**.**

$$\begin{aligned} \sigma_1 \rightsquigarrow \sigma_2 \lessdot L \quad \triangleq \quad & \forall \sigma_0, L_0, L_1. \\ & (\sigma_0, L_0) \lessdot (\sigma_1, L) \bigwedge (\sigma_0, L_0) \lessdot (\sigma_1, L_1) \\ & \Longrightarrow (\sigma_0, L_0) \lessdot (\sigma_2, L_1) \end{aligned}$$

Scoping preservation is a triple relation among σ_2, σ_1 and L. Intuitively, the set L can be thought of as the current stack frame. It is needed because not all lexical scopings are preserving due to reassignment; only those scopings by a stack frame that is still on the stack. In the definition, we may think of L_0 as the previous stack frame, $(\sigma_0, L_0) \lessdot (\sigma_1, L)$ ensures L_0 is still on the stack. The definition says that any set L_1 which is scoped by L_0 continues to be scoped by L_0 in the heap migration from σ_1 to σ_2. In particular, L_1 can be L.

Scoping preservation is illustrated in Figure 3.4. In the heap migration from σ to σ', we have $(\sigma, 1) \lessdot (\sigma', 6)$, as the object 6 only reaches the object 7 in σ', the latter is also reachable from the object 1 in σ. However, we do not have $(\sigma, 1) \lessdot (\sigma'', 6)$, because the object 6 reaches the object 2 in σ'', which is not reachable from the object 1 in σ. Thus the scoping relation $(\sigma, 1) \lessdot (\sigma', 6)$ is not preserved in the heap migration from σ' to σ''.

In contrast, in the heap migration from σ to σ', we have $(\sigma, 4) \lessdot (\sigma', 6)$, as the object 6 only reaches the object 7 in σ', the latter is also reachable from the object 4 in σ. Meanwhile, we also have $(\sigma, 4) \lessdot (\sigma'', 6)$, because all objects in σ that the object 6 may reach in σ'', are reachable from the object 4 in σ. Thus the scoping relation $(\sigma, 4) \lessdot (\sigma', 6)$ is preserved in the heap migration from σ' to σ''.

Without scoping preservation, in an evaluation $[\![e]\!](\sigma, \rho, \psi) = (l, \sigma')$, we cannot even conclude that $(\sigma, L) \lessdot (\sigma', L)$, where $L = codom(\rho) \cup \{\ \psi\ \}$. With the property, we may prove $(\sigma, L) \lessdot (\sigma', L)$ by choosing $\sigma_0 = \sigma$ and $L' = L$, as $(\sigma, L) \lessdot (\sigma, L)$ holds trivially. The two properties are interdependent, thus they are proved together.

This property is related to separation logic [28], where the part of the heap that a command

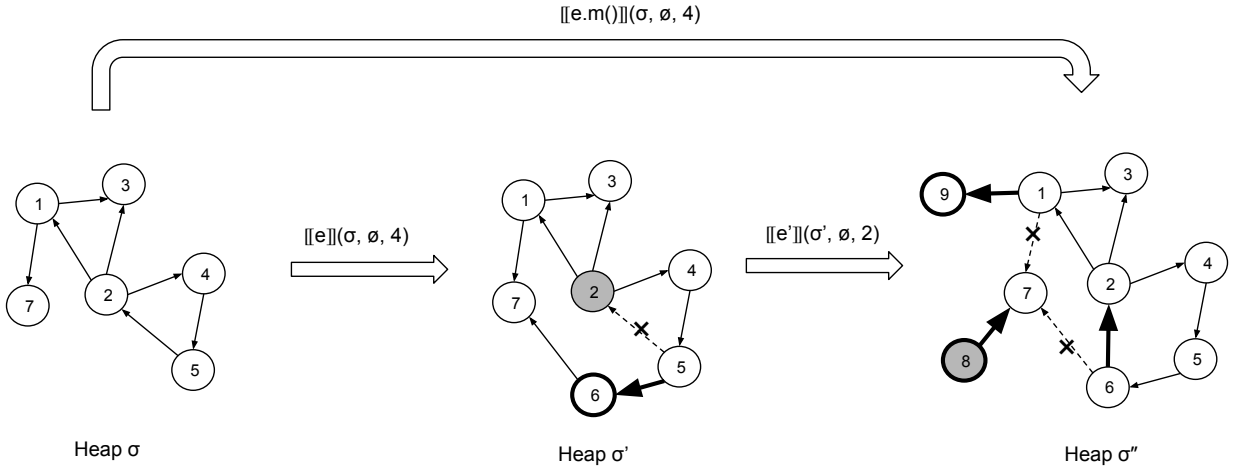


Figure 3.4 – Illustration of scoping preservation. Gray circles indicate the result of the evaluation. Thick circles and thick arrows represent newly created objects and links. We assume the body of the method m is e'.

actually uses is called its *footprint*. Here, we over-approximate the footprint of an expression by the set of objects reachable from $codom(\rho)$ and ψ, and we may think unreachable heap regions from $codom(\rho)$ and ψ are valid frames for an expression.

3.3.4 Local Reasoning

With weak monotonicity, stackability and scopability, we may prove the theorem of local reasoning.

LEMMA 3.3 (Local Reasoning)**.** *The following proposition holds*

$$\frac{(\sigma, L) \lessdot (\sigma', L') \qquad \sigma \ll \sigma' \qquad \sigma \preceq \sigma' \qquad \sigma \vDash L : hot}{\sigma' \vDash L' : hot}$$

Proof. Let's consider a reachable object l from L', i.e. $\sigma' \vDash L' \rightsquigarrow l$. Depending on whether $l \in dom(\sigma)$, there are two cases.

- **Case** $l \notin dom(\sigma)$.
  Use the fact that $\sigma \ll \sigma'$, we know $\sigma' \vDash l : warm$.

- **Case** $l \in dom(\sigma)$.
  Use the fact that $(\sigma, L) \lessdot (\sigma', L')$, we have $\sigma \vDash L \rightsquigarrow l$. From the premise $\sigma \vDash L : hot$, we have $\sigma \vDash l : warm$. From $\sigma \preceq \sigma'$, we have $\sigma' \vDash l : warm$.

In both cases, we have $\sigma' \vDash l : warm$, by definition we have $\sigma' \vDash L' : hot$. □

THEOREM 3.4 (Local Reasoning)**.** *The following proposition holds:*

$$\frac{[\![e]\!](\sigma, \rho, \psi) = (l, \sigma') \qquad \sigma \vDash \{\ \psi\ \} \cup codom(\rho) : hot}{\sigma' \vDash l : hot}$$

Proof. Immediate from Lemma 3.3, the preconditions are satisfied by Theorem 3.3, Theorem 3.1 and Theorem 3.2. □

In particular, if e is a method body, we can conclude that if the receiver and all the method parameters are hot, then the return value is also hot.

This theorem echoes the insight in the freedom model [12]: if a constructor is called with all committed arguments, then the constructed object is also committed.

3.4 Proof of Scopability

3.4.1 Lemmas

LEMMA 3.4 (Scoping Reflexivity). *For all* $\sigma, L_2 \subseteq L_1$, *we have* $(\sigma, L_1) \lessdot (\sigma, L_2)$.

Proof. Immediate from the definition of scoping. □

LEMMA 3.5 (Scoping Subset). *If* $(\sigma_1, L_1) \lessdot (\sigma_2, L_2 \cup L)$, *then* $(\sigma_1, L_1) \lessdot (\sigma_2, L)$.

Proof. Immediate from the definition of scoping. □

LEMMA 3.6 (Scoping Union). *If* $(\sigma_1, L) \lessdot (\sigma_2, L_1)$ *and* $(\sigma_1, L) \lessdot (\sigma_2, L_2)$, *then* $(\sigma_1, L) \lessdot (\sigma_2, L_1 \cup L_2)$.

Proof. Immediate from the definition of scoping. □

LEMMA 3.7 (Scoping Reachability). *For all* σ, l_1, l_2, *if* $\sigma \vDash l_1 \rightsquigarrow l_2$, *then* $(\sigma, l_1) \lessdot (\sigma, l_2)$.

Proof. Immediate from the definition of scoping. □

LEMMA 3.8 (Scoping Transitivity). *Given*

- $dom(\sigma_1) \subseteq dom(\sigma_2)$
- $(\sigma_1, L_1) \lessdot (\sigma_2, L_2)$
- $(\sigma_2, L_2) \lessdot (\sigma_3, L_3)$

then

- $(\sigma_1, L_1) \lessdot (\sigma_3, L_3)$

Proof. From the definition of $scoping$, we need to prove that for any $l_3 \in L_3$ and $l \in dom(\sigma_1)$, if (A1) holds, then (A2) holds too:

- (A1) $\sigma_3 \vDash l_3 \rightsquigarrow l$
- (A2) $\exists l_1 \in L_1.\ \sigma_1 \vDash l_1 \rightsquigarrow l$

From $l \in dom(\sigma_1)$ and $dom(\sigma_1) \subseteq dom(\sigma_2)$, we have

- (B1) $l \in dom(\sigma_2)$

Use (A1), (B1) and $(\sigma_2, L_2) \lessdot (\sigma_3, L_3)$, we have

- (C1) $\exists l_2 \in L_2 . \sigma_2 \vDash l_2 \rightsquigarrow l$

Pick l_2 in (C1), now use (C1) and $(\sigma_1, L_1) \lessdot (\sigma_2, L_2)$, we arrive at (A2) immediately. □

LEMMA 3.9 (Preserving Transitivity). *Given*

- $\sigma_1 \rightsquigarrow \sigma_2 \lessdot L_1$

- $\sigma_2 \rightsquigarrow \sigma_3 \lessdot L_2$
- $(\sigma_1, L_1) \lessdot (\sigma_2, L_2)$

then

- $\sigma_1 \rightsquigarrow \sigma_3 \lessdot L_1$

Proof. From the definition of scoping preservation, we need to prove that for any σ_0, $L \subseteq dom(\sigma_1)$ and $L_0 \subseteq dom(\sigma_0)$, if (A1) and (A2) hold, then (A3) holds too:

- (A1) $(\sigma_0, L_0) \lessdot (\sigma_1, L_1)$
- (A2) $(\sigma_0, L_0) \lessdot (\sigma_1, L)$
- (A3) $(\sigma_0, L_0) \lessdot (\sigma_3, L)$

From (A1), (A2) and the premise $\sigma_1 \rightsquigarrow \sigma_2 \lessdot L_1$, we have

- (B1) $(\sigma_0, L_0) \lessdot (\sigma_2, L)$

From the premise $(\sigma_1, L_1) \lessdot (\sigma_2, L_2)$ and (A1), we use Lemma 3.8:

- (C1) $(\sigma_0, L_0) \lessdot (\sigma_2, L_2)$

From (B1), (C1) and the premise $\sigma_2 \rightsquigarrow \sigma_3 \lessdot L_2$, we arrive at (A3) immediately. □

LEMMA 3.10 (Preserving Regularity - Degenerate Case)**.**
If $\sigma_1 \rightsquigarrow \sigma_2 \lessdot L$*, then* $(\sigma_1, L) \lessdot (\sigma_2, L)$.

Proof. By the definition of scoping preservation with $\sigma_0 = \sigma_1$, $L_1 = L$ and $L_0 = L$, the precondition $(\sigma_1, L) \lessdot (\sigma_1, L)$ holds trivially. □

LEMMA 3.11 (Preserving Regularity)**.** *If* $\sigma_1 \rightsquigarrow \sigma_2 \lessdot L$*,* $(\sigma_0, L) \lessdot (\sigma_1, L)$ *and* $(\sigma_0, L) \lessdot (\sigma_1, L_1)$*, then* $(\sigma_0, L) \lessdot (\sigma_2, L_1)$.

Proof. By the definition of scoping preservation. □

LEMMA 3.12 (Preserving Transitivity - Degenerate Case)**.** *Given*

- $\sigma_1 \rightsquigarrow \sigma_2 \lessdot L_1$
- $\sigma_2 \rightsquigarrow \sigma_3 \lessdot L_1$

then

- $\sigma_1 \rightsquigarrow \sigma_3 \lessdot L_1$

Proof. Use Lemma 3.9 and Lemma 3.10. □

LEMMA 3.13 (Assignment)**.** *Given*

(1) $\sigma_1 \rightsquigarrow \sigma_2 \lessdot L_1$
(2) $(\sigma_1, L_1) \lessdot (\sigma_2, l)$
(3) $(\sigma_1, L_1) \lessdot (\sigma_2, l')$
(4) $(C, \omega) = \sigma_2(l)$
(5) $\omega' = [f \mapsto l']\omega$
(6) $\sigma_2' = [l \mapsto (C, \omega')]\sigma_2$

then we have

(a) $\sigma_1 \rightsquigarrow \sigma_2' \lessdot L_1$
(b) $(\sigma_1, L_1) \lessdot (\sigma_2', l)$

Proof. We prove the two separately.

Proof of $\sigma_1 \rightsquigarrow \sigma_2' \lessdot L_1$.

From the definition of scoping preservation, we need to prove that for any σ_0, $L \subseteq dom(\sigma_1)$, $L_0 \subseteq dom(\sigma_0)$, if (A1) and (A2) hold, then (A3) holds:

- (A1) $(\sigma_0, L_0) \lessdot (\sigma_1, L_1)$
- (A2) $(\sigma_0, L_0) \lessdot (\sigma_1, L)$
- (A3) $(\sigma_0, L_0) \lessdot (\sigma_2', L)$

From (A1), (A2) and $\sigma_1 \rightsquigarrow \sigma_2 \lessdot L_1$, we have

- (B1) $(\sigma_0, L_0) \lessdot (\sigma_2, L)$

From $(\sigma_1, L_1) \lessdot (\sigma_2, l')$, (A1) and Lemma 3.8, we have

- (C1) $(\sigma_0, L_0) \lessdot (\sigma_2, l')$

To prove (A3), we need to show that for any $l_2 \in L$ and $l_0 \in dom(\sigma_0)$, if (D1) holds, then (D2) holds too:

- (D1) $\sigma_2' \vDash l_2 \rightsquigarrow l_0$
- (D2) $\exists l \in L_0.\sigma_0 \vDash l \rightsquigarrow l_0$

For (D1), we consider the path that begins from l_2 to l_0. Note that the two heap graphs σ_2 and σ_2' only differ by the edge $l \rightsquigarrow l'$.

If the path does not contain the edge $l \rightsquigarrow l'$, then the path must exist in σ_2, i.e. $\sigma_2 \vDash l_2 \rightsquigarrow l_0$. Now use (B1) we arrive at (D2) immediately.

If the path contains the edge $l \rightsquigarrow l'$, then we have $\sigma_2 \vDash l' \rightsquigarrow l_0$. Now use (C1), we arrive at (D2) too.

Proof of $(\sigma_1, L_1) \lessdot (\sigma_2', l)$.

Suppose that there exists $l_1 \in dom(\sigma_1)$, such that $\sigma_2' \vDash l \rightsquigarrow l_1$. We consider the path that begins from l to l_1 without loops. Note that the two heap graphs σ_2 and σ_2' only differ by the edge $l \rightsquigarrow l'$.

If the path from l to l_1 does not contain the edge $l \rightsquigarrow l'$, then the path must exist in σ_2, i.e. $\sigma_2 \vDash l \rightsquigarrow l_1$. Now from precondition (2) we have $\exists l_a \in L_1.\sigma_1 \vDash l_a \rightsquigarrow l_1$, which completes the proof goal.

Otherwise, if the path from l to l_1 contains the edge $l' \rightsquigarrow l$, then l_1 is reachable from l', i.e. $\sigma_2 \vDash l' \rightsquigarrow l_2$. Now use precondition (3), we arrive at the result similarly as above. □

3.4.2 Theorem

THEOREM 3.5 (Scopability)**.** *If* $[\![e]\!](\sigma, \rho, \psi) = (l, \sigma')$*, then we have*

(1) $(\sigma, codom(\rho) \cup \{\ \psi\ \}) \lessdot (\sigma', l)$

(2) $\sigma \rightsquigarrow \sigma' \lessdot codom(\rho) \cup \{\ \psi\ \}$

Proof. By induction on the structure of e.

- **case** $e = x$

  (1) holds by choosing the existential to be $\rho(x)$.

  (2) holds trivially by Lemma 3.4 due to $\sigma' = \sigma$.

- **case** $e = this$
  (1) holds by choosing the existential to be ψ.
  (2) holds trivially by Lemma 3.4 due to $\sigma' = \sigma$.

- **case** $e = e_0.f$
  From the induction hypothesis on e_0, we know there exists l_0 such that:
  - (A1) $[\![e_0]\!](\sigma, \rho, \psi) = (l_0, \sigma')$
  - (A2) $(\sigma, \psi \cup codom(\rho)) \lessdot (\sigma', l_0)$
  - (A3) $\sigma \rightsquigarrow \sigma' \lessdot \psi \cup codom(\rho)$
  - (A4) $(C, \omega) = \sigma'(l_0)$
  - (A5) $l = \omega(f)$

  From $\sigma' \vDash l_0 \rightsquigarrow l$ and Lemma 3.7 we have
  - (B1) $(\sigma', l) \lessdot (\sigma', l_0)$

  From (A2), (B1) and Lemma 3.8 we have
  - (C1) $(\sigma, \psi \cup codom(\rho)) \lessdot (\sigma', l)$

  (1) holds from (C1).
  (2) holds from (A3).

- **case** $e = e_0.m(\overline{e})$
  From the induction hypothesis on e_0, we know there exists l_0 and σ_0 such that:
  - (A1) $[\![e_0]\!](\sigma, \rho, \psi) = (l_0, \sigma_0)$
  - (A2) $(\sigma, \psi \cup codom(\rho)) \lessdot (\sigma_0, l_0)$
  - (A3) $\sigma \rightsquigarrow \sigma_0 \lessdot \psi \cup codom(\rho)$
  - (A4) $(\sigma, \psi \cup codom(\rho)) \lessdot (\sigma_0, \psi \cup codom(\rho))$ ▷ By (A3) and Lemma 3.10
  - (A5) $(C, \omega) = \sigma_0(l_0)$

  By using induction hypothesis repeatedly on all arguments, we get σ_i such that
  - (B1) $[\![e_i]\!](\sigma_{i-1}, \rho, \psi) = (l_i, \sigma_i)$
  - (B2) $(\sigma_{i-1}, \psi \cup codom(\rho)) \lessdot (\sigma_i, l_i)$
  - (B3) $\sigma_{i-1} \rightsquigarrow \sigma_i \lessdot \psi \cup codom(\rho)$
  - (B4) $(\sigma_{i-1}, \psi \cup codom(\rho)) \lessdot (\sigma_i, \psi \cup codom(\rho))$ ▷ By (B3) and Lemma 3.10

  As the method call succeeds, we must have
  - (D1) $lookup(C, m) = @\mu\ def\ m(\overline{x_i{:}T_i}) : T = e_m$
  - (D2) $\rho' = \overline{x_i{:}l_i}$
  - (D3) $[\![e_m]\!](\sigma_n, \rho', l_0) = (l_m, \sigma_m)$

  Now we can use the induction hypothesis for e_m:
  - (E1) $(\sigma_n, l_0 \cup codom(\rho')) \lessdot (\sigma_m, l_m)$

– (E2) $\sigma_n \rightsquigarrow \sigma_m \lessdot l_0 \cup codom(\rho')$

From (A2), (A4), (B2), (B4) and Lemma 3.11, we have
– (F1) $(\sigma, \psi \cup codom(\rho)) \lessdot (\sigma_n, l_0 \cup codom(\rho'))$
– (F2) $\sigma \rightsquigarrow \sigma_n \lessdot \psi \cup codom(\rho)$ ▷ By Lemma 3.12

(1) holds from (E1), (F1) and Lemma 3.8.
(2) holds from (E2), (F1), (F2) and Lemma 3.9.

- **case** $e = new\ C(\overline{e})$
Let $\sigma_0 = \sigma$, use induction hypothesis on arguments repeatedly:
– (A1) $(\sigma_{i-1}, \psi \cup codom(\rho)) \lessdot (\sigma_i, l_i)$
– (A2) $\sigma_{i-1} \rightsquigarrow \sigma_i \lessdot \psi \cup codom(\rho))$

From (A1), (A2) and Lemma 3.11, we have
– (C1) $(\sigma, \psi \cup codom(\rho)) \lessdot (\sigma_n, \{l_1, \cdots, l_n\})$
– (C2) $\sigma \rightsquigarrow \sigma_n \lessdot \psi \cup codom(\rho)$ ▷ By Lemma 3.12

We define σ'_0 with a fresh location l:
– (D1) $\sigma'_0 = \sigma_n \cup \{l \mapsto (C, \overline{\hat{f}_i = l_i})\}$

We have the following:
– (E1) $(\sigma, \psi \cup codom(\rho)) \lessdot (\sigma'_0, l)$ ▷ from (C1) and (D1)
– (E2) $\sigma_n \rightsquigarrow \sigma'_0 \lessdot \psi \cup codom(\rho)$ ▷ from (D1) and definition
– (E3) $\sigma \rightsquigarrow \sigma'_0 \lessdot \psi \cup codom(\rho)$ ▷ by (C2), (E2) and Lemma 3.12

We perform induction on m — the number of fields in the class body.

The basic case $m = 0$ trivially holds from (E1) and (E3).

Let us consider $m = i + 1$. From induction hypothesis, we have
– (F1) $(\sigma, \psi \cup codom(\rho)) \lessdot (\sigma'_i, l)$
– (F2) $\sigma \rightsquigarrow \sigma'_i \lessdot \psi \cup codom(\rho)$
– (F3) $(\sigma, \psi \cup codom(\rho)) \lessdot (\sigma'_i, \psi \cup codom(\rho))$ ▷ By (F2) and Lemma 3.10

Consider the (i+1)-th field in the class body $var\ f_{i+1} : T_{i+1} = e_{i+1}$. Use induction hypothesis on e_{i+1}, we know there exists $l_{i+1}, \hat{\sigma}_{i+1}$ such that:
– (G1) $(\sigma'_i, l) \lessdot (\hat{\sigma}_{i+1}, l_{i+1})$
– (G2) $\sigma'_i \rightsquigarrow \hat{\sigma}_{i+1} \lessdot l$

Now we have
– (I1) $(\sigma, \psi \cup codom(\rho)) \lessdot (\hat{\sigma}_{i+1}, l_{i+1})$ ▷ from (F1), (G1) and Lemma 3.8

– (I2) $(\sigma, \psi \cup codom(\rho)) \lessdot (\hat{\sigma}_{i+1}, l)$ ▷ from (F1), (F3), (G2) and Lemma 3.11
– (I3) $\sigma \rightsquigarrow \hat{\sigma}_{i+1} \lessdot \psi \cup codom(\rho)$ ▷ from (F1), (F2), (G2) and Lemma 3.9

We define σ'_{i+1} as follows:
– (H1) $(C, \omega) = \hat{\sigma}_{i+1}(l)$
– (H2) $\omega' = [f_{i+1} \mapsto l_{i+1}]\omega$
– (H3) $\sigma'_{i+1} = [l \mapsto (C, \omega')]\hat{\sigma}_{i+1}$

The conclusion follows from Lemma 3.13.

- **case** $e = (e_0.f = e_1; e_2)$
From the induction hypothesis on e_0, we know there exists l_0 and σ_0 such that:
– (A1) $(\sigma, \psi \cup codom(\rho)) \lessdot (\sigma_0, l_0)$
– (A2) $\sigma \rightsquigarrow \sigma_0 \lessdot \psi \cup codom(\rho)$
– (A3) $(\sigma, \psi \cup codom(\rho)) \lessdot (\sigma_0, \psi \cup codom(\rho))$ ▷ By (A2) and Lemma 3.10

From the induction hypothesis on e_1, we know there exists l_1 and σ_1 such that:
– (B1) $(\sigma_0, \psi \cup codom(\rho)) \lessdot (\sigma_1, l_1)$
– (B2) $\sigma_0 \rightsquigarrow \sigma_1 \lessdot \psi \cup codom(\rho)$
– (B3) $(\sigma, \psi \cup codom(\rho)) \lessdot (\sigma_1, l_1)$ ▷ from (B1), (A3) and Lemma 3.8
– (B4) $(\sigma, \psi \cup codom(\rho)) \lessdot (\sigma_1, l_0)$ ▷ from (A1), (A3) and (B2)

We define σ'_1 as follows
– (C1) $(C, \omega) = \sigma_1(l_0)$
– (C2) $\omega' = [f \mapsto l_1]\omega$
– (C3) $\sigma'_1 = [l_0 \mapsto (C, \omega')]\sigma_1$

Now from Lemma 3.13 we have
– (D1) $(\sigma, \psi \cup codom(\rho)) \lessdot (\sigma'_1, l)$
– (D2) $\sigma \rightsquigarrow \sigma'_1 \lessdot \psi \cup codom(\rho)$
– (D3) $(\sigma, \psi \cup codom(\rho)) \lessdot (\sigma'_1, \psi \cup codom(\rho))$ ▷ By (D2) and Lemma 3.10

Now use induction hypothesis on e_2, we have
– (E1) $(\sigma'_1, \psi \cup codom(\rho)) \lessdot (\sigma_2, l_2)$
– (E2) $\sigma'_1 \rightsquigarrow \sigma_2 \lessdot \psi \cup codom(\rho)$

(1) holds from by (D3), (E1) and Lemma 3.8.
(2) holds from by (D2), (E2) and Lemma 3.12.

□

3.5 Proof of Weak Monotonicity

3.5.1 Lemmas

LEMMA 3.14 (Reflexivity). *For all* σ, $\sigma \preceq \sigma$.

Proof. By the definition of the predicate weak monotonicity. □

LEMMA 3.15 (Transitivity). *If* $\sigma_1 \preceq \sigma_2$ *and* $\sigma_2 \preceq \sigma_3$, *then* $\sigma_1 \preceq \sigma_3$.

Proof. By the definition of the predicate weak monotonicity. □

LEMMA 3.16 (Assignment). *For all* $\sigma, \sigma', l, l', f$, *if*
(1) $(C, \omega) = \sigma(l)$
(2) $\omega' = [f \mapsto l']\omega$
(3) $\sigma' = [l \mapsto (C, \omega')]\sigma$
then $\sigma \preceq \sigma'$

Proof. Immediate by definition, as assignment does not cause fields to be uninitialized. □

LEMMA 3.17 (Warm Monotone). *If* $\sigma \preceq \sigma'$ *and* $\sigma \vDash l : warm$, *then* $\sigma' \vDash l : warm$.

Proof. Immediate by definition. □

3.5.2 Theorem

THEOREM 3.6 (Weak Monotonicity). *If* $[\![e]\!](\sigma, \rho, \psi) = (l, \sigma')$, *then* $\sigma \preceq \sigma'$.

Proof. By induction on the structure of e.

- **case** $e = x$
  Trivial due to $\sigma' = \sigma$.

- **case** $e = this$
  Trivial due to $\sigma' = \sigma$.

- **case** $e = e_0.f$
  From the induction hypothesis on e_0, we know there exists l_0 such that:
  - (A1) $\sigma \preceq \sigma'$
  - (A2) $(T_0, \omega) = \sigma'(l_0)$
  - (A3) $l = \omega(f)$

- **case** $e = e_0.m(\overline{e})$
  From the induction hypothesis on e_0, we know there exists l_0 and σ_0 such that:
  - (A1) $\sigma \preceq \sigma_0$
  - (A2) $(C, \omega) = \sigma_0(l_0)$

  By using induction hypothesis repeatedly on all arguments, we get σ_i such that

– (B1) $\sigma_{i-1} \preceq \sigma_i$

As the method call succeeds, we must have
– (D1) $lookup(C, m) = @\mu\ def\ m(\overline{x_i{:}T_i}) : T = e_m$
– (D2) $\rho' = \overline{x_i{:}l_i}$
– (D3) $[\![e_m]\!]\,(\sigma_n, \rho', l_0) = (l_m, \sigma_m)$

Now we can use the induction hypothesis for e_m:
– (E1) $\sigma_n \preceq \sigma_m$

Now by transitivity, we have
– (F1) $\sigma \preceq \sigma_m$

- **case** $e = new\ C(\overline{e})$
Let $\sigma_0 = \sigma$, use induction hypothesis on arguments repeatedly:
– (A1) $\sigma_{i-1} \preceq \sigma_i$

By transitivity, we have
– (B1) $\sigma \preceq \sigma_n$

We define σ'_0 with a fresh location l:
– (C1) $\sigma'_0 = \sigma_n \cup \{l \mapsto (C, \overline{\hat{f}_i = l_i})\}$

We have the following:
– (D1) $\sigma_n \preceq \sigma'_0$ ▷ from (C1) and definition
– (D2) $\sigma \preceq \sigma'_0$ ▷ by (B1), (D1) and transitivity

Suppose the class has m fields in class body, we prove that the following invariant holds:
– (E1) $\sigma'_0 \preceq \sigma'_m$

By induction on m. The basic case $m = 0$ trivially holds as $\sigma'_m = \sigma'_0$.

Let us consider $m = i + 1$. From induction hypothesis, we have
– (F1) $\sigma'_0 \ll \sigma'_i$
– (F2) $\sigma'_0 \preceq \sigma'_i$

Consider the (i+1)-th field in the class body $var\ f_{i+1} : T_{i+1} = e_{i+1}$. Use induction hypothesis on e_{i+1}, we know there exists $l_{i+1}, \hat{\sigma}_{i+1}$ such that:
– (G1) $[\![e_{i+1}]\!]\,(\sigma'_i, \emptyset, l) = (l_{i+1}, \hat{\sigma}_{i+1})$
– (G2) $\sigma'_i \preceq \hat{\sigma}_{i+1}$

We define σ'_m as follows:

– (H1) $(C, \omega) = \hat{\sigma}_{i+1}(l)$
– (H2) $\omega' = [f_{i+1} \mapsto l_{i+1}]\omega$
– (H3) $\sigma'_m = [l \mapsto (C, \omega')]\hat{\sigma}_{i+1}$

Now use the Lemma Assignment, we have
– (I1) $\sigma'_0 \preceq \sigma'_m$

The conclusion $\sigma \preceq \sigma'_m$ holds from (I1) and (D2) by transitivity.

- **case** $e = (e_0.f = e_1; e_2)$
From the induction hypothesis on e_0, we know there exists σ_0 such that:
– (A1) $\sigma \preceq \sigma_0$

From the induction hypothesis on e_1, we know there exists σ_1 such that:
– (B1) $\sigma_0 \preceq \sigma_1$

We define σ'_1 as follows
– (C1) $(C, \omega) = \sigma_1(l_0)$
– (C2) $\omega' = [f \mapsto l_1]\omega$
– (C3) $\sigma'_1 = [l_0 \mapsto (C, \omega')]\sigma_1$

By Lemma Assignment, we have the following holds
– (D1) $\sigma_1 \preceq \sigma'_1$

Now use induction hypothesis on e_2, we have
– (E1) $\sigma'_1 \preceq \sigma_2$

The conclusion follows by transitivity.

□

3.6 Proof of Stackability

3.6.1 Lemmas

LEMMA 3.18 (Reflexivity)**.** *For all* σ, $\sigma \ll \sigma$.

Proof. By the definition of the predicate stacking. □

LEMMA 3.19 (Transitivity)**.** *If* $\sigma_1 \ll \sigma_2, \sigma_2 \ll \sigma_3$ *and* $\sigma_2 \preceq \sigma_3$, *then* $\sigma_1 \ll \sigma_3$.

Proof. From premises, we have
- (A1) $\forall l_3 \in dom(\sigma_3).\sigma_3 \vDash l_3 : warm \vee l_3 \in dom(\sigma_2)$
- (A2) $\forall l_2 \in dom(\sigma_2).\sigma_2 \vDash l_2 : warm \vee l_2 \in dom(\sigma_1)$

Consider $l_3 \in dom(\sigma_3)$, if we have $\sigma_3 \vDash l_3 : warm$, we are done. If $\sigma_3 \vDash l_3 : warm$ does not hold, then from (A1) we have $l_3 \in dom(\sigma_2)$. Now use (A2), it cannot be the case that $\sigma_2 \vDash l_2 : warm$ due to $\sigma_2 \preceq \sigma_3$. Therefore, we have $l_3 \in dom(\sigma_1)$, which completes the proof. □

LEMMA 3.20 (Assignment). *For all* $\sigma, \sigma', l, l', f$, *if*
(1) $(C, \omega) = \sigma(l)$
(2) $\omega' = [f \mapsto l']\omega$
(3) $\sigma' = [l \mapsto (C, \omega')]\sigma$
then $\sigma \ll \sigma'$

Proof. The assignment does not create new objects, thus the result holds trivially. □

3.6.2 Theorem

THEOREM 3.7 (Stackability). *If* $[\![e]\!](\sigma, \rho, \psi) = (l, \sigma')$, *then* $\sigma \ll \sigma'$.

Proof. By induction on the structure of e.

- **case** $e = x$
  Trivial due to $\sigma' = \sigma$.

- **case** $e = this$
  Trivial due to $\sigma' = \sigma$.

- **case** $e = e_0.f$
  From the induction hypothesis on e_0, we know there exists l_0 such that:
  - (A1) $\sigma \ll \sigma'$
  - (A2) $(C, \omega) = \sigma'(l_0)$
  - (A3) $l = \omega(f)$

- **case** $e = e_0.m(\overline{e})$
  From the induction hypothesis on e_0, we know there exists l_0 and σ_0 such that:
  - (A1) $\sigma \ll \sigma_0$
  - (A2) $\sigma \preceq \sigma_0$ ▷ By Theorem 3.6
  - (A3) $(C, \omega) = \sigma_0(l_0)$

  By using induction hypothesis repeatedly on all arguments, we get σ_i such that
  - (B1) $\sigma_{i-1} \ll \sigma_i$
  - (B2) $\sigma_{i-1} \preceq \sigma_i$ ▷ By Theorem 3.6

  As the method call succeeds, we must have
  - (D1) $lookup(C, m) = @\mu\ def\ m(\overline{x_i{:}T_i}) : T = e_m$
  - (D2) $\rho' = \overline{x_i{:}l_i}$
  - (D3) $[\![e_m]\!](\sigma_n, \rho', l_0) = (l_m, \sigma_m)$

Now we can use the induction hypothesis for e_m:

- (E1) $\sigma_n \ll \sigma_m$
- (E2) $\sigma_n \preceq \sigma_m$ ▷ By Theorem 3.6

Now by transitivity, we have

- (F1) $\sigma \ll \sigma_m$

- **case** $e = new\ C(\overline{e})$

Let $\sigma_0 = \sigma$, use induction hypothesis on arguments repeatedly:

- (A1) $\sigma_{i-1} \ll \sigma_i$
- (A2) $\sigma_{i-1} \preceq \sigma_i$ ▷ By Theorem 3.6

By transitivity we have

- (B1) $\sigma \ll \sigma_n$
- (B2) $\sigma \preceq \sigma_n$ ▷ By Theorem 3.6

We define σ'_0 with a fresh location l:

- (C1) $\sigma'_0 = \sigma_n \cup \{l \mapsto (C, \overline{\hat{f}_i = l_i})\}$

We have the following:

- (D1) $\sigma_n \preceq \sigma'_0$ ▷ from (C1) and definition
- (D2) $\sigma \preceq \sigma'_0$ ▷ by (B2), (D1) and transitivity

Suppose the class has m fields in class body, we prove that the following invariant holds:

- (E1) $\sigma'_0 \ll \sigma'_m$
- (E2) $\sigma'_0 \preceq \sigma'_m$

By induction on m. The basic case $m = 0$ trivially holds as $\sigma'_m = \sigma'_0$.

Let us consider $m = i + 1$. From induction hypothesis, we have

- (F1) $\sigma'_0 \ll \sigma'_i$
- (F2) $\sigma'_0 \preceq \sigma'_i$ ▷ By Theorem 3.6

Consider the (i+1)-th field in the class body $var\ f_{i+1} : T_{i+1} = e_{i+1}$. Use induction hypothesis on e_{i+1}, we know there exists $l_{i+1}, \hat{\sigma}_{i+1}$ such that:

- (G1) $[\![e_{i+1}]\!]\,(\sigma'_i, \emptyset, l) = (l_{i+1}, \hat{\sigma}_{i+1})$
- (G2) $\sigma'_i \ll \hat{\sigma}_{i+1}$
- (G3) $\sigma'_i \preceq \hat{\sigma}_{i+1}$ ▷ By Theorem 3.6

We define σ'_m as follows:

- (H1) $(C, \omega) = \hat{\sigma}_{i+1}(l)$
- (H2) $\omega' = [f_{i+1} \mapsto l_{i+1}]\omega$

– (H3) $\sigma'_m = [l \mapsto (C, \omega')]\hat{\sigma}_{i+1}$

Now use the Lemma Assignment, we have
– (I1) $\sigma'_0 \ll \sigma'_m$
– (I2) $\sigma'_0 \preceq \sigma'_m$

As all fields of the object l has been assigned, we have
– (J1) $\sigma'_m \vDash l : warm$

Suppose there exists $l_c \in dom(\sigma'_m)$ where l_c is not warm in σ'_m. We have
– (K1) $l_c \in dom(\sigma'_0)$ ▷ from (I1)
– (K2) $l_c \neq l$ ▷ because l is warm

From the definition of σ'_0 in (C1), we have $l_c \in dom(\sigma_n)$. Note that l_c cannot be warm in σ_n. Otherwise, l_c will be warm in σ'_m due to the following fact:
– (L1) $\sigma_n \preceq \sigma'_m$ ▷ from (D1) and (I2)

Now from (B1) and $\neg\sigma_n \vDash l_c : warm$, we have $l_c \in \sigma$, which completes the proof that $\sigma \ll \sigma'_m$.

- **case** $e = (e_0.f = e_1; e_2)$
From the induction hypothesis on e_0, we know there exists σ_0 such that:
– (A1) $\sigma \ll \sigma_0$
– (A2) $\sigma \preceq \sigma_0$ ▷ By Theorem 3.6

From the induction hypothesis on e_1, we know there exists σ_1 such that:
– (B1) $\sigma_0 \ll \sigma_1$
– (B2) $\sigma_0 \preceq \sigma_1$ ▷ By Theorem 3.6

We define σ'_1 as follows
– (C1) $(C, \omega) = \sigma_1(l_0)$
– (C2) $\omega' = [f \mapsto l_1]\omega$
– (C3) $\sigma'_1 = [l_0 \mapsto (C, \omega')]\sigma_1$

By Lemma Assignment, we have the following holds
– (D1) $\sigma_1 \ll \sigma'_1$
– (D2) $\sigma_1 \preceq \sigma'_1$ ▷ By Theorem 3.6

Now use induction hypothesis on e_2, we have
– (E1) $\sigma'_1 \ll \sigma_2$
– (E2) $\sigma'_1 \preceq \sigma_2$ ▷ By Theorem 3.6

The conclusion follows by transitivity.

□

3.7 Mechanization

Clément Blaudeau worked on Coq mechanization of local reasoning. The code is located in the following address:

`https://github.com/clementblaudeau/celsius`

During the mechanization, he found that several well-formedness conditions are missing. First, in the definition of *scoping*, we need to add the requirement that $dom(\sigma_1) \subseteq dom(\sigma_2)$ (Definition 3.9). Otherwise, in proving the Lemma 3.9, we will be unable to supply the evidence $dom(\sigma_1) \subseteq dom(\sigma_2)$ required by the Lemma 3.8. With the addition in the definition, we may remove the premise $dom(\sigma_1) \subseteq dom(\sigma_2)$ from the Lemma 3.8.

Second, we need to assume that field values are valid locations of the heap, i.e.

$$wf(\sigma) \quad \triangleq \quad \forall l.(C,\omega) = \sigma(l) \wedge l' = \omega(f) \Longrightarrow l' \in dom(\sigma)$$

We will be able to prove the following theorem:

$$\frac{wf(\sigma) \qquad codom(\rho) \cup \{\ \psi\ \} \subseteq dom(\sigma) \qquad [\![e]\!](\sigma,\rho,\psi) = (l,\sigma')}{wf(\sigma') \wedge l \in dom(\sigma')}$$

We need to add $wf(\sigma)$ and $codom(\rho) \cup \{\ \psi\ \} \subseteq dom(\sigma)$ as premises to the theorem of scopability (Theorem 3.5) and the theorem of local reasoning (Theorem 3.4).

Third, the initial definition of weak monotonicity does not enforce that the class of an object remains the same, which poses a difficulty in proving that a warm object continues to be warm. We already fixed the problem in the definition.

We choose to not address the first two mechanization findings due to the following considerations. First, the fixes will clutter the proofs without addition of insights. Second, we feel it is valuable to keep some gaps in the pen and paper proofs such that interested readers can learn more from the mistakes. Third, there might be different fixes.

3.8 Discussion

One might attempt to define scoping preservation as follows:

DEFINITION 3.11 (Scoping Preservation - Incorrect Attempt)**.**

$$\sigma_1 \rightsquigarrow \sigma_2 \lessdot L \quad \triangleq \quad \forall \sigma_0, L'. \quad (\sigma_0, L) \lessdot (\sigma_1, L') \Longrightarrow (\sigma_0, L) \lessdot (\sigma_2, L')$$

This definition says that if a set of locations L' is scoped by L, it continues to be scoped by L during the heap migration from σ_1 to σ_2. While the definition is reasonable, the induction hypothesis it generates is too weak for the case of method calls.

Suppose we have a method call evaluation $[\![this.m(e)]\!](\sigma_1,\rho,\psi) = (l_m,\sigma_3)$, we need to prove

1. $(\sigma_1, codom(\rho) \cup \{\ \psi\ \}) \lessdot (\sigma_3, l_m)$
2. $\sigma_1 \rightsquigarrow \sigma_3 \lessdot codom(\rho) \cup \{\ \psi\ \}$

The first proof goal says that the final result is scoped by the initial evaluation environment. The second proof goal says that all locations previously scoped by the initial evaluation environment continue to be scoped.

The evaluation recurs on the argument e and method body e_m:

(1) $[\![e]\!]\,(\sigma_1, \rho, \psi) = (l_e, \sigma_2)$
(2) $[\![e_m]\!]\,(\sigma_2, \{\ x \mapsto l_e\ \}, \psi) = (l_m, \sigma_3)$

where we get the following as induction hypothesis:

(A1) $\sigma_1 \rightsquigarrow \sigma_2 \lessdot codom(\rho) \cup \{\ \psi\ \}$
(A2) $(\sigma_1, codom(\rho) \cup \{\ \psi\ \}) \lessdot (\sigma_2, l_e)$
(A3) $\sigma_2 \rightsquigarrow \sigma_3 \lessdot \{\ \psi_1, l_e\ \}$
(A4) $(\sigma_1, \{\ \psi_2, l_e\ \}) \lessdot (\sigma_3, l_m)$

We may prove the first goal by the following steps:

(B1) $(\sigma_1, codom(\rho) \cup \{\ \psi\ \}) \lessdot (\sigma_2, \psi)$. ▷ From (A1)
(B2) $(\sigma_1, codom(\rho) \cup \{\ \psi\ \}) \lessdot (\sigma_2, \{\ l_e, \psi\ \})$ ▷ From (B1) and (A2)
(B3) $(\sigma_1, codom(\rho) \cup \{\ \psi\ \}) \lessdot (\sigma_3, l_m)$ ▷ From (B2), (A4) and transitivity of scoping

However, we get stuck on the second goal. The reason is that the induction hypothesis (A3) is too weak: it only says that all locations scoped by $\{\ \psi_1, l_e\ \}$ continue to be scoped, while we want to prove that all locations scoped by $codom(\rho) \cup \{\ \psi\ \}$ continue to be scoped. In particular, we are unable to prove $(\sigma_1, codom(\rho) \cup \{\ \psi\ \}) \lessdot (\sigma_3, codom(\rho))$, which is an immediate result if the second goal holds.

3.9 Conclusion

We formally prove local reasoning (Theorem 3.4), which provides the semantic justification for the importance of the principles for initialization. The property of scopability obviously holds, but its proof is not obvious at all — we need to prove at the same time that scoping is preserving relative to some set of addresses (Theorem 3.5).

Chapter 4

The Basic Model

Simplicity is the ultimate sophistication.

— Leonardo da Vinci

In this chapter, we present a type system for safe initialization based on the abstractions *hot*, *warm* and *cold*.

4.1 The Formal Language

We formalize the basic model in a class-based language with mutations. Our language resembles a subset of Scala having only top-level classes, mutable fields and methods.

$$
\begin{array}{rcl}
\mathcal{P} \in Program & ::= & (\overline{\mathcal{C}}, D) \\
\mathcal{C} \in Class & ::= & class\ C(\overline{\hat{f}{:}T})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \} \\
\mathcal{F} \in Field & ::= & var\ f{:}T = e \\
e \in Exp & ::= & x \mid this \mid e.f \mid e.m(\overline{e}) \mid new\ C(\overline{e}) \mid e.f = e; e \\
\mathcal{M} \in Method & ::= & @\mu\ def\ m(\overline{x{:}T}) : T = e \\
S, T, U \in Type & ::= & C^{\mu} \\
\mu \in mode & ::= & hot \mid warm \mid cold
\end{array}
$$

The only change compared to the language in the previous chapter is the definition of types. Types are composed of a mode μ and a class name, written as C^{μ}. The mode consists of the three abstractions: *hot*, *warm* and *cold*.

Methods are now annotated with modes, i.e. $@\mu\ def\ m(\overline{x{:}T}) : T = e$, the mode μ means $this$ has the type C^{μ} inside the method m of the class C. The semantics of the language remain the same as the language introduced in the last chapter.

Mode Lattice

$$hot \sqsubseteq warm \sqsubseteq cold$$

$$\sqcup(\mu_1, \mu_2) = max(\mu_1, \mu_2)$$

$$\sqcap(\mu_1, \mu_2) = min(\mu_1, \mu_2)$$

Subtyping

$$T <: T \text{ (S-REFL)} \qquad \frac{T_1 <: T_2 \qquad T_2 <: T_3}{T_1 <: T_3} \text{ (S-TRANS)} \qquad \frac{\mu_1 \sqsubseteq \mu_2}{C^{\mu_1} <: C^{\mu_2}} \text{ (S-MODE)}$$

Figure 4.1 – Lattice and Subtyping

4.2 Type System

4.2.1 Subtyping

The subtyping rules are presented in Figure 4.1. The ordering, meet and join of the lattice are defined naturally. The rule S-REFL and S-TRANS are standard for subtyping. The rule S-MODE extends the mode lattice to types.

4.2.2 Definition Typing

The typing rules for definitions are presented in Figure 4.2. When type checking a program $(\overline{C}, e)$ with the rule T-PROG, the system checks that every class is well-typed, and the entry expression is well typed.

When type checking a class, the rule T-CLASS first checks the field definitions. Then it checks that each method is well-typed.

When type checking a field definition $var\ \ f{:}T\ =\ e$, the rule T-FIELD ensures that the expression e can be typed as T in an empty environment. The type of $this$ is assumed to be C^{cold}.

When type checking a method, the rule T-METHOD checks that the method body e conforms to the method return type S, in the environment of method parameters $\overline{x{:}T}$, assuming $this$ to take the mode of the method.

4.2.3 Expression Typing

The typing rules for expressions are presented in Figure 4.3. The typing judgements for expressions have the form $\Gamma; T_1 \vdash e : T_2$, which means that the expression e can take the type T_2, given the environment Γ, and the type T_1 as the type of $this$.

These and later definitions assume helper methods $fieldType(C, f)$, $methodType(C, m)$ and $constrType(C)$ to look up in class table Ξ the type, respectively, of field $C.f$, of method $C.m$ and of the constructor of C.

Program Typing $\boxed{\vdash \mathcal{P}}$

$$\frac{\Xi = \overline{C \to \mathcal{C}} \qquad \Xi(D) = class\ D\ \{\ def\ main : T = e\ \} \qquad \emptyset; D^{hot} \vdash e : T \qquad \overline{\Xi \vdash \mathcal{C}}}{\vdash (\overline{\mathcal{C}}, D)} \text{ (T-PROG)}$$

Class Typing $\boxed{\Xi \vdash \mathcal{C}}$

$$\frac{\Omega_0 = \overline{\hat{f}} \qquad \overline{\Xi; C^{\Omega_i} \vdash \mathcal{F}_i \quad \Omega_{i+1} = \Omega_i \cup \{\ f_i\ \}} \qquad \overline{\Xi; C \vdash \mathcal{M}}}{\Xi \vdash class\ C(\overline{\hat{f}{:}T})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \}} \text{ (T-CLASS)}$$

Field Typing $\boxed{\Xi; C^{\Omega} \vdash \mathcal{F}}$

$$\frac{\emptyset; C^{\Omega} \vdash e : T}{\Xi; C^{\Omega} \vdash var\ f : T = e} \text{ (T-FIELD)}$$

Method Typing $\boxed{\Xi; C \vdash \mathcal{M}}$

$$\frac{\overline{x{:}T}; C^{\mu} \vdash e : S}{\Xi; C \vdash @\mu\ def\ m(\overline{x{:}T}) : S = e} \text{ (T-METHOD)}$$

Figure 4.2 – Definition Typing

The rule T-SUB is standard in type systems with subtyping. The rule T-VAR looks up the type of the variable from the environment, and the rule T-THIS assumes the type T for $this$.

The rule T-SELHOT ensures that field selection on a *hot* object always returns a *hot* value. The rule T-SELWARM enforces that field selection on a *warm* object takes the type of the field.

When checking a new-expression $new\ C(\overline{e})$, the rule T-NEW first checks that the arguments $\overline{e}$ conform to the types of the class parameters. If any of the arguments is *cold* or *warm*, then the result is *warm*; otherwise, the result is *hot*. This rule is expressed as $\mu = (\sqcup\ \mu_i) \sqcap warm$. Note that this rule is more expressive than the following rule:

$$\frac{\overline{C_i^{\mu_i}} = constrType(C) \qquad \Gamma; T \vdash e_i : C_i^{\mu_i} \qquad \mu' = (\sqcup\ \mu_i) \sqcap warm}{\Gamma; T \vdash new\ C(\overline{e}) : C^{\mu'}} \text{ (T-NEW')}$$

The difference is that the rule T-NEW uses the actual mode of the actual arguments, while the rule T-NEW' uses the mode of the formal parameters. The former achieves a kind of *mode-polymorphism* — it gives more precise types when the arguments are hot. The justification of the rule is based on *local reasoning* of initialization (Chapter 3).

The same insight applies to the rule T-INVOKE: if the actual receiver and arguments of a method call are hot, then the result should be hot regardless of the declared method result type. Otherwise, the result takes the declared method result type. The rule T-INVOKE also checks

Expression Typing $\boxed{\Gamma; T \vdash e : T}$

$$\frac{\Gamma; T_1 \vdash e : T_2 \qquad T_2 <: T_3}{\Gamma; T_1 \vdash e : T_3} \quad \text{(T-SUB)}$$

$$\frac{x : U \in \Gamma}{\Gamma; T \vdash x : U} \quad \text{(T-VAR)}$$

$$\Gamma; T \vdash this : T \quad \text{(T-THIS)}$$

$$\frac{\Gamma; T \vdash e : D^{hot} \qquad C^{\mu} = fieldType(D, f)}{\Gamma; T \vdash e.f : C^{hot}} \quad \text{(T-SELHOT)}$$

$$\frac{\Gamma; T \vdash e : D^{warm} \qquad U = fieldType(D, f)}{\Gamma; T \vdash e.f : U} \quad \text{(T-SELWARM)}$$

$$\frac{\overline{T_i} = constrType(C) \qquad \Gamma; T \vdash e_i : C_i^{\mu_i} \qquad C_i^{\mu_i} <: T_i \qquad \mu = (\sqcup\, \mu_i) \sqcap warm}{\Gamma; T \vdash new\ C(\overline{e}) : C^{\mu}} \quad \text{(T-NEW)}$$

$$\frac{\begin{array}{c}\Gamma; T \vdash e : C^{\mu_0} \qquad (\mu_m, \overline{T_i}, D^{\mu_r}) = methodType(C, m) \\ \mu_0 \sqsubseteq \mu_m \qquad \Gamma; T \vdash e_i : D_i^{\mu_i} \qquad D_i^{\mu_i} <: T_i \qquad \mu = (\sqcup\, \mu_i = hot)?hot : \mu_r\end{array}}{\Gamma; T \vdash e.m(\overline{e}) : D^{\mu}} \quad \text{(T-INVOKE)}$$

$$\frac{\Gamma; T \vdash e_1.f : C^{\mu} \qquad \Gamma; T \vdash e_2 : C^{hot} \qquad \Gamma; T \vdash e : T_1}{\Gamma; T \vdash e_1.f = e_2; e : T_1} \quad \text{(T-BLOCK)}$$

Figure 4.3 – Expression Typing

that the receiver e is well-typed and its mode conforms to the mode of the method, and the arguments $\overline{e}$ confirm to the types of method parameters.

When checking a block expression $e_1.f = e_2; e$, the rule T-BLOCK ensures that only assignment of hot values is allowed. Note that this typing rule would disallow assignment to a cold value because $e_1.f$ will not type check. While assigning hot values to fields of cold values will not cause soundness problems, this rule is motivated for two reasons: (1) it disallows code like `class A { a=5; var a=10 }`, which assigned to a variable in vain before it is initialized; (2) it enforces modularity of initialization, as all fields of a class have to be assigned inside the class. Note that allowing assignment of hot values to a field of a cold object does not necessarily violate the principle of authority, as long as the field is not assumed to hold a hot value after the assignment in the type system.

4.2.4 Typing Example

We demonstrate how the following circular data structure can be type checked in our formal model.

```
1 class Parent {
2   var child: Child @warm = new Child(this)
3 }
4 class Child(parent: Parent @cold) {
5   var tag: Int = 10
6 }
```

The main steps of type derivation for type checking the class `Parent` are given below:

$$\dfrac{\dfrac{\dfrac{\emptyset; Parent^{cold} \vdash this : Parent^{cold} \qquad \mu = (\sqcup cold) \sqcap warm = warm}{\emptyset; Parent^{cold} \vdash new\ Child(this) : Child^{warm}}\ \text{(T-NEW)}}{Parent \vdash var\ c : Child@warm = new\ Child(this)}\ \text{(T-FIELD)}}{\Xi \vdash\ class\ Parent\ \{\ var\ c : Child@warm = new\ Child(this)\ \}}\ \text{(T-CLASS)}$$

4.3 Extension

The type system does not allow the usage of already initialized fields, as `this` is given the type C^{cold} in checking the fields of a class C. The design is intentional to make the model simple and reusable. We will show systems that support the usage of fields in later chapters.

As a simple exercise, we may also extend the type system with the type C^{Ω} to support the usage of already initialized fields:

$$\begin{array}{lcl} \Omega & ::= & \{\ f_1, f_2, \ldots\ \} \\ \mu & ::= & cold \mid warm \mid hot \mid \Omega \\ T & ::= & C^{\mu} \end{array}$$

In the above, we introduce the type C^{Ω} to support the usage of already initialized fields — Ω denotes the set of initialized fields. The type is well-formed only if Ω contains only fields of the class C. The lattice for modes μ is defined below:

$$hot \sqsubseteq \mu \qquad warm \sqsubseteq \Omega \qquad \Omega_1 \cup \Omega_2 \sqsubseteq \Omega_1 \qquad \mu \sqsubseteq cold$$

The modes hot and $cold$ are respectively bottom and top of the lattice, and Ω is in the middle.

Programmers do not need to use the type C^{Ω} explicitly in the program. We adapt the typing rule for classes and fields to handle the type automatically:

$$\dfrac{\Omega_0 = \overline{\hat{f}} \qquad \overline{\Xi; C^{\Omega_i} \vdash \mathcal{F}_i \qquad \Omega_{i+1} = \Omega_i \cup \{\ f_i\ \}} \qquad \overline{\Xi; C \vdash \mathcal{M}}}{\Xi \vdash class\ C(\overline{\hat{f}:T})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \}}\ \text{(T-CLASS)}$$

$$\dfrac{\emptyset; C^{\Omega} \vdash e : T}{\Xi; C^{\Omega} \vdash var\ f : T = e}\ \text{(T-FIELD)}$$

As can be seen from above, when checking a field $\mathcal{F}_i$, we set the type of $this$ to be C^{Ω_i}, where Ω_i is the set of already initialized fields. We just need one more typing rule to support field access:

$$\frac{\Gamma; T \vdash e : D^{\Omega} \qquad f \in \Omega \qquad U = fieldType(D, f)}{\Gamma; T \vdash e.f : U} \qquad \text{(T-SELOBJ)}$$

4.4 Discussion

4.4.1 Promotion before Commitment

The paper *freedom before commitment* [12] popularized the concept *commitment point*, where a single object or a group of cyclic objects are formally taken as initialized. The insight of commitment point is an instance of local reasoning (chapter 3).

Our system has a promotion step before commitment: an object is first promoted to *warm* before it is eventually committed as hot. This promotion step is explicit in the meta-theory of the basic model (chapter 5).

The meta-theory of the freedom model [12] introduces the concept *locally initialized* which means all fields of an object are assigned. The concept is almost the same as *warm* in our formal model. However, the concept is not introduced as an abstraction for programming.

Given the importance of *warm* and promotion in the formal reasoning of initialization, we believe it is worth making it available as an abstraction for programming.

4.4.2 Authority, Flow-Insensitivity and Typestate Polymorphism

As expected, local reasoning plays an important role in the soundness of the system. An important semantic property that is not covered by the insight of local reasoning is what we call *authority*, which is defined in terms of store typings below (see chapter 5):

$$\frac{\forall l \in dom(\Sigma).\Sigma(l) = C^{\Omega} \Longrightarrow \Sigma'(l) = C^{\Omega}}{\Sigma \rhd \Sigma'}$$

In the above, Σ and Σ' are the store typings before and after evaluating an expression. Intuitively, it says that we may not advance the initialization state of existing objects during evaluation of an expression. It leaves the only possibility to advance object state at special locations in the constructor.

Without this property, we may not prove *perfect monotonicity*. The reason can be demonstrated by the following program:

```
1 class C { var x: D @warm = e; var y: Int = 10 }
```

In the code above, suppose the type of ψ (the value for $this$) starts as $C^{\emptyset}$, and a side-effect of evaluating e updates the type of ψ to C^{μ}. After assigning the value of e, denoted as l_e, to the field x, we update the type of this to $C^{\mu'}$. We would like $\mu' \sqsubseteq \{\ x\ \}$ to record that field x is initialized, and monotonicity requires that $\mu' \sqsubseteq \mu$. The property of authority ensures that $\mu = \emptyset$, which enables

one simple solution to these constraints, namely $\mu' = \{ x \}$. This is a sound choice because we do know that the field x has been assigned the value l_e, which is of the type D^{warm} (known by induction hypothesis) as required by the semantic typing of the object ψ as C^x.

Without the property of authority, it would be allowed to update the type of ψ as a side-effect of evaluating e, for example to C^{hot}. Then the constraint $\mu' \sqsubseteq \mu$ would force μ to be *hot*. However, there is no guarantee that l_e is transitively initialized. From the induction hypothesis, we only know that it has the type D^{warm}. So setting μ' to *hot* would be unsound, since this might no longer be transitively initialized after $this.x$ is assigned the value l_e.

Note that the definition only talks about types of the form C^{Ω}. The store typing never contains types like C^{cold}, a value takes such a type by subtyping. Authority for values of the type C^{warm} is not necessary for soundness. The reason is that the next monotone state is always C^{hot}, it is impossible for monotonicity to fail. For the type C^{hot}, monotonicity guarantees that the type stays the same.

The semantic property *authority* reflects a subtle design of the system. A key design idea of the freedom model, which is also followed in the current system, is the usage of subtyping to support typestate polymorphism. However, one point that is not made clear in the freedom model [12] is that a *flow-insensitive* system is needed to support typestate polymorphism via subtyping. In a flow-sensitive system, we cannot resort to subtyping because a method has to represent the typestates of $this$ both before and after the method call, which generally requires parametric polymorphism. In a flow-insensitive system, the typestate of $this$ does not change inside a method. Therefore, there is only the need to represent the current typestate of $this$, thus subtyping can be used.

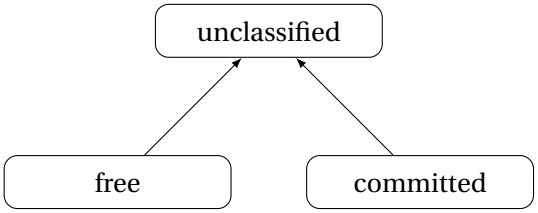
In a flow-insensitive system, how can we safely advance object typestates, i.e. *strong updates*? It is unsafe to do so at arbitrary points in the program due to flow-insensitivity, as the strong update may break monotonicity if the typestate of the object has been advanced further via aliases elsewhere. The property of authority suggests that it is only safe to perform strong updates by an outstanding alias at definite locations in the program. The outstanding alias is not unique due to the presence of aliasing to create cyclic data structures. In the experimental language, the outstanding alias is `this`, and the locations are the points of field initializations.

4.5 Related Work

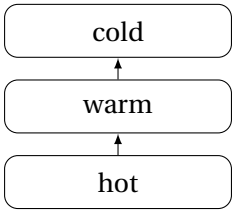
The basic model is inspired by *the freedom model* [12]. The models are illustrated in Figure 4.4. Both models introduce three initialization states. In the freedom model the states have the following meaning:

- **committed**: the object is initialized.
- **free**: the object is under initialization.
- **unclassified**: the object may be initialized or under initialization.

The lifetime of an object passes from *free* to *committed* in the freedom model. In the basic model, however, three states are used to describe the lifetime of an object, which goes from *cold* to *warm* and eventually to *hot*.



(a) The Freedom Model



(b) The Basic Model

Figure 4.4 – The freedom model and the basic model. The arrow means whether a value in the source state is a value in the target state.

Conceptually, *free* roughly corresponds to *cold*, and *committed* corresponds to *hot*. However, there are two crucial differences between the two models:

- In the basic model, hot is a subtype of $cold$, whereas in the freedom model, $committed$ is not a subtype of $free$.
- The basic model introduces $warm$, which has no correspondent in the freedom model.

The first difference is justified both semantically and practically. Semantically, a fully initialized object should be able to be used as an object under initialization. Practically, if a method may be called on an object under initialization, it should be able to be called when the object is fully initialized.

The second difference aims for improved expressiveness. For example, our model supports the following code example, while it is impossible to type check the code in the freedom model:

```
1 class Parent {
2   var child = new Child(this)
3   var tag: Int = child.tag     // OK in the basic model, error in freedom model
4 }
5
6 class Child(parent: Parent @cold) {
7   var tag: Int = 10
8 }
```

The difference lies in the way the expression `new Child(this)` is handled. In our model, the expression returns a *warm* value, while in the freedom model it returns a *free* value. As a result, the selection of the field `tag` is safe in the basic model, as all fields of warm values are initialized. In contrast, the freedom model cannot tell whether the field `tag` is initialized or not.

To be fair, there are code examples that type check in the freedom model, but fail to type check in the basic model. The following is one such example:

```
1 class A(b: B) {
2   var c: C = new C(this)
3   b.m(this)
4 }
5 class B {
6   def m(a: A @free): Unit = a.c = new C(a) // !!
```

```
}
class C(a: A @free)
```

The assignment `a.c = new C(a)` in class `B` will be rejected in the basic model, as `new C(a)` is a value under initialization (it holds a reference to a free value `a`). In the basic model, it is only possible to assign hot values to fields of cold objects, while in the freedom model it is possible to assign non-committed values to fields of non-committed values.

However, we are unaware of such code patterns in our case studies. This is also observed in the empirical studies of the freedom model [12]:

> "*So far in the code we have examined, we have not seen any cases where an object escapes from its constructor and then has its fields written via other methods.*"

Another reason to reject the code is that it breaks monotonicity: the field before the reassignment might hold a hot value, and after the assignment it holds a warm value, monotonicity is thus compromised. As a result, it is unsafe to use already initialized fields in the freedom model. In the implementation, the freedom model has to introduce another modifier *committed-only* to reject the assignment above for commited-only fields. This tweak restores monotonicity and enables the usage of already initialized fields. Therefore, we think this loss of expressiveness is a justified design choice.

The meta-theory of the freedom model [12] introduces the concept *locally initialized* which means all fields of an object are assigned, which is almost the same as *warm* in our formal model. However, the concept is not introduced as an abstraction for programming.

Qi and Myers [16] introduce an expressive type system for initialization based on masked types. In the system, methods and constructors have effects, which are essentially the mappings of initialization status of `this` before and after the call. To support cyclic data structures, they introduce *conditional masked types*, written $T \backslash f[x_1.g_1, \ldots, x_n.g_n]$. It means that the field f points to a partially initialized object, which will become fully initialized when all fields $x_i.g_i$ are initialized. Using conditional masked types, we may type check the example in the beginning of the chapter as follows:

```
class Parent {
  val child: Child\parent[this.child, this.name] = new Child(this)
  val name = "parent"
}

class Child(parent: Parent\child\name) {
  println(parent.name)                // error: name is not initialized
}
```

Despite the expressiveness, the system is verbose and it lacks simple abstractions that may reduce cognitive burdens for programmers.

Fähndrich et al. [24] introduce raw types like $T^{\text{raw}(S)}$ — a value of such a type is possibly under initialization, and all fields up to the superclass S are initialized. Class fields may not hold raw values, thus it does not support creating cyclic data structures. To overcome the limitation,

they introduce *delayed types* [19]. The system ensures that the initialization of objects forms stacked time regions.

The Billion-Dollar Fix [9] introduces a new linguistic construct *placeholders* and *placeholder types* to support initialization of circular data structures. The work is orthogonal to the current work, in that we are constrained from introducing new language constructs and semantics.

4.6 Conclusion

We presented a type-based system based on the three abstractions , namely *hot*, *warm* and *cold*. The introduction of the abstraction *warm* improves the expressiveness of the freedom model [12], which classifies objects either as committed (i.e. hot) or free (i.e. cold).

The type system, however, requires type annotations on methods and fields, which impacts usability. Meanwhile, it is not obvious how to extend the model to support language features like inheritance, traits, inner classes, and properties. In later chapters, we will show how to develop an inference system for a practical fragment of the basic model to cut down the syntactic overhead.

Chapter 5

Meta-Theory: The Basic Model

Mathematics, rightly viewed, possesses not only truth, but supreme beauty.

— Bertrand Russell

In this chapter, we prove soundness of the extended system (which includes the type C^{Ω}) presented in the last chapter.

5.1 Approach

To show that the type system is sound, we follow the approach of definitional interpreters [10, 5]. In order to reason about the definitional interpreter formally as a function, it has to be total. It means we need to deal with non-termination and errors explicitly. The standard approach is to introduce a fuel k to deal with non-termination, and handle errors with an option monad.

The step-indexed interpreter defined in Coq can be found in Appendix A. In the definition, we use `None` to represent timeouts, `Some None` to represent errors, and `Some(Some(l, s))` to represent the successful result.

We always assume a well-typed class table Ξ, which is part of the implicit context. In the Coq definition, the class table is parameterized, thus there is no need to pass it around everywhere, which makes the presentation clean (Appendix A).

The soundness statement says that well-typed programs do not go wrong:

PROPOSITION 5.1 (Soundness)**.** *If* $\vdash \mathcal{P}$*, then* $\forall k.\ evalProg(\mathcal{P})(k) \neq Some\ None$

5.2 Definitions

We let Σ range over store typings, that is finite maps from locations to types:

$$\Sigma \in StoreTyping = Loc \rightharpoonup Type$$

Store typing can be seen as an abstraction of the concrete heap. It also establishes the relationship between static semantics, i.e. typing and concrete semantics.

Store typing $\boxed{\Sigma \vDash \sigma}$

$$\frac{\forall\, l \in dom(\Sigma).\ \Sigma \vDash \sigma(l) : \Sigma(l)}{\Sigma \vDash \sigma}$$

Monotonicity $\boxed{\Sigma_1 \preccurlyeq \Sigma_2}$

$$\frac{\forall\, l \in dom(\Sigma_1).\ \Sigma_2(l) <: \Sigma_1(l)}{\Sigma_1 \preccurlyeq \Sigma_2}$$

Authority $\boxed{\Sigma_1 \rhd \Sigma_2}$

$$\frac{\forall l \in dom(\Sigma_1).\Sigma_1(l) = C^{\Omega} \Longrightarrow \Sigma_2(l) = C^{\Omega}}{\Sigma_1 \rhd \Sigma_2}$$

Stackability $\boxed{\Sigma_1 \ll \Sigma_2}$

$$\frac{\forall l \in dom(\Sigma_2).\Sigma_2 \vDash l : warm \bigvee l \in dom(\Sigma_1)}{\Sigma_1 \ll \Sigma_2}$$

Value typing $\boxed{\Sigma \vDash l : T}$

$$\frac{\Sigma(l) = T_1 \qquad T_1 <: T_2}{\Sigma \vDash l : T_2}$$

Environment typing $\boxed{\Gamma; \Sigma \vDash \rho}$

$$\emptyset; \Sigma \vDash \emptyset \qquad\qquad \frac{\Gamma; \Sigma \vDash \rho \qquad \Sigma \vDash l : T}{\Gamma, x{:}T; \Sigma \vDash \rho, x{:}l}$$

Convenience Definitions

$$\frac{\Sigma \vDash l : D^{\mu}}{\Sigma \vDash l : D} \qquad \frac{\Sigma \vDash l : D^{\mu}}{\Sigma \vDash l : \mu} \qquad \frac{\forall l \in L.\Sigma \vDash l : \mu}{\Sigma \vDash L : \mu}$$

Figure 5.1 – Store typing, environment typing and value typing

We next define typing judgements on the runtime state. The definitions are presented in Figure 5.1.

The store typing judgement $\Sigma \vDash \sigma$ guarantees that the object stored at each store location is well-typed according to Σ. Other typing judgments then refer to Σ to verify the type of a location. Since objects can form cycles, verifying that an object matches a certain type without using store

Object typing $\boxed{\Sigma \vDash o : T}$

$$\frac{(C,\omega) = o \qquad \forall\, f \in dom(\omega).\ \Sigma \vDash \omega(f) : fieldType(C,f)}{\Sigma \vDash o : C^{dom(\omega)}}$$

$$\frac{(C,\omega) = o \qquad \forall\, f \in dom(\omega).\ \Sigma \vDash \omega(f) : fieldType(C,f)}{\Sigma \vDash o : C^{cold}}$$

$$\frac{(C,\omega) = o \qquad \forall\, f \in fields(C).\ \Sigma \vDash \omega(f) : fieldType(C,f)}{\Sigma \vDash o : C^{warm}}$$

$$\frac{(C,\omega) = o \qquad \forall\, f \in fields(C).\ (\_, D) = fieldType(C,f) \bigwedge \Sigma \vDash \omega(f) : D^{hot}}{\Sigma \vDash o : C^{hot}}$$

Figure 5.2 – Object typing

typings would run into cycles, and introducing store typings is a standard solution [27, Chapter 13].

Store typings evolve during evaluation, so we define an ordering across store typing $\preccurlyeq$, such that if Σ_1 updates to Σ_2 then $\Sigma_1 \preccurlyeq \Sigma_2$. Moreover, we show that if $\Sigma_1 \preccurlyeq \Sigma_2$ then any typing judgment established under store typing Σ_1 still holds under store typing Σ_2: that is, typing judgments referring to store typings are *monotonic* (Lemmas 5.3, 5.4 and 7.2). As usual, if $\Sigma_1 \preccurlyeq \Sigma_2$, then the updated store typing Σ_2 can have more entries than Σ_1.

During the initialization of an object, we need to reason that a field is not already taken as initialized in the store typing (whether it is initialized in the heap does not matter), so that after the field initialization, we may update the store typing to include the field without breaking monotonicity. Updating the store typing with C^{Ω} will violate monotonicity if the object under initialization is already taken as $warm$ or hot in the store typing. This is what the property *authority* guarantees: if $\Sigma_1 \rhd \Sigma_2$, then it ensures that if $\Sigma_1(l) = C^{\Omega}$, then $\Sigma_2(l) = C^{\Omega}$. It means that the initialization status of a field in the type system (not in the heap) may only advance during field initialization. Consequently, the initialization status of an object can be safely promoted at the end of class constructors (Lemma 5.20). This property is a direct result of the principle of authority.

For expression soundness, we need to show that the store typing follows the principle of stackability, written as $\Sigma_1 \ll \Sigma_2$. The property $\Sigma_1 \ll \Sigma_2$ is an over-approximation of $\sigma_1 \ll \sigma_2$ (Lemma 5.2).

The value typing judgement $\Sigma \vDash l : T$ is standard: it just retrieves the type of the location from the store typing. As our language supports subtyping, we allow subtyping on values as well.

The environment typing judgement $\Gamma; \Sigma \vDash \rho$ is also standard, and lifts value typing to environments. In other words, it ensures that the value of a variable in the runtime environment ρ

corresponds to the type of the variable in the typing environment Γ.

To simplify presentation, we also assume some convenience definitions, $\Sigma \vDash l : D$, $\Sigma \vDash l : \mu$ and $\Sigma \vDash L : \mu$.

Finally, we get to object typing, which is presented in Figure 5.2. The object typing judgement $\Sigma \vDash o : T$ checks that the object o is well-typed according to the semantics of initialization states. For example, to check that an object is *hot*, it verifies that every field of the object is hot. The typing of objects in the store typing is an over-approximation of the actual states of objects in the heap (Lemma 5.1).

5.3 Over-Approximation Lemmas

The lemmas here are not used in the soundness proof. However, we find it insightful to show them here.

LEMMA 5.1 (Object Over-Approximation)**.** *The store typing is an over-approximation of initialization states of the heap, formally*

$$\frac{\Sigma \vDash \sigma \qquad \Sigma \vDash l : \mu}{\sigma \vDash l : \mu}$$

Proof. The case where $\mu = cold$ and $\mu = warm$ are easy from the definitions. For the case $\mu = hot$, we know that any object l' reachable from l can be typed *hot* from the Lemma 5.9. By definition of object typing, we know all fields of the object are assigned, therefore $\sigma \vDash l' : warm$. Now $\sigma \vDash l : hot$ holds by definition. □

LEMMA 5.2 (Over-Approximation of Stackability)**.** $\Sigma_1 \ll \Sigma_2$ *is an over-approximation of* $\sigma_1 \ll \sigma_2$*, formally*

$$\frac{\Sigma_1 \vDash \sigma_1 \qquad \Sigma_2 \vDash \sigma_2 \qquad dom(\Sigma_2) = dom(\sigma_2) \qquad \Sigma_1 \ll \Sigma_2}{\sigma_1 \ll \sigma_2}$$

Proof. Consider $l \in dom(\sigma_2)$, from $\Sigma_1 \ll \Sigma_2$, we know $\Sigma_2 \vDash l : warm \bigvee l \in dom(\Sigma_1)$. If we have $\Sigma_2 \vDash l : warm$, from Lemma 5.1, we have $\sigma_2 \vDash l : warm$. If we have $l \in dom(\Sigma_1)$, by definition we have $l \in dom(\sigma_1)$. Thus, for all $l \in dom(\sigma_2)$, either $\sigma_2 \vDash l : warm$ or $l \in dom(\sigma_2)$, which completes the proof. □

However, the property of monotonicity is *not* an over-approximation of weak monotonicity. Logically, it should not be due to the fact that the store tying is an over-approximation of objects (Lemma 5.1). The reason is that the actual heap is too noisy, it may capture properties that are not assumed in the store typing, i.e. from $\sigma \vDash l : \mu$ we may not conclude $\Sigma \vDash l : \mu$. Therefore, the following proposition does not hold:

$$\frac{\Sigma_1 \vDash \sigma_1 \qquad \Sigma_2 \vDash \sigma_2 \qquad \Sigma_1 \preceq \Sigma_2}{\sigma_1 \preceq \sigma_2}$$

5.4 Monotonicity Lemmas

LEMMA 5.3 (Value Typing Monotonicity)**.** *If* $\Sigma_1 \preccurlyeq \Sigma_2$ *and* $\Sigma_1 \vDash l : T$*, then* $\Sigma_2 \vDash l : T$*.*

Proof. We use inversion on value typing and the definition of store typing ordering to show $\Sigma_2(l) <: \Sigma_1(l) <: T$. We can conclude by transitivity of subtyping. □

LEMMA 5.4 (Environment Typing Monotonicity)**.** *If* $\Sigma_1 \preccurlyeq \Sigma_2$ *and* $\Gamma; \Sigma_1 \vDash \rho$*, then* $\Gamma; \Sigma_2 \vDash \rho$*.*

Proof. We use induction on $\Gamma; \Sigma_1 \vDash \rho$ to apply value typing monotonicity (Lemma 5.3) to each environment entry. □

LEMMA 5.5 (Object Typing Monotonicity)**.** *If* $\Sigma \preccurlyeq \Sigma'$ *and* $\Sigma \vDash o : T$*, then* $\Sigma' \vDash o : T$*.*

Proof. By inversion on the given derivation of object typing, the definition of store typing ordering, and value typing monotonicity (Lemma 5.3). □

LEMMA 5.6 (Monotonicity Reflexivity)**.** *For all* Σ*,* $\Sigma \preceq \Sigma$*.*

Proof. Immediate by definition of monotonicity. □

LEMMA 5.7 (Monotonicity Transitivity)**.** *If* $\Sigma_1 \preceq \Sigma_2$ *and* $\Sigma_2 \preceq \Sigma_3$*, then* $\Sigma_1 \preceq \Sigma_3$*.*

Proof. By definition of monotonicity and transitivity of subtyping. □

LEMMA 5.8 (Environment Regularity)**.** *If* $\Gamma \mid \Sigma \vDash \rho$ *and* $\Gamma; U \vdash x{:}T$*, then* $\Sigma \vDash \rho(x) : T$*.*

Proof. By induction on the definition of environment typing. □

LEMMA 5.9 (Hot Transitivity)**.** *If an object is typed as hot, then any object reachable from it is typed as hot. Formally,*

$$\frac{\Sigma \vDash l : hot \qquad \Sigma \vDash \sigma \qquad \sigma \vDash l \rightsquigarrow l'}{\Sigma \vDash l' : hot}$$

Proof. By structural induction on the derivation of reachability. □

5.5 Authority Lemmas

LEMMA 5.10 (Authority Reflexivity)**.** *For all* Σ*,* $\Sigma \rhd \Sigma$*.*

Proof. Immediate by definition of monotonicity. □

LEMMA 5.11 (Authority Transitivity)**.** *If* $\Sigma_1 \rhd \Sigma_2$*,* $\Sigma_2 \rhd \Sigma_3$ *and* $\Sigma_1 \preceq \Sigma_2$*, then* $\Sigma_1 \rhd \Sigma_3$*.*

Proof. By definition of authority. The premise $\Sigma_1 \preceq \Sigma_2$ is used to ensure that for any location l, if $l \in dom(\Sigma_1)$, then $l \in dom(\Sigma_2)$. □

5.6 Stackability Lemmas

LEMMA 5.12 (Stackability - Reflexivity). *For all* Σ, $\Sigma \ll \Sigma$.

Proof. By the definition of the predicate stacking. □

LEMMA 5.13 (Stackability - Transitivity). *If* $\Sigma_1 \ll \Sigma_2, \Sigma_2 \ll \Sigma_3$ *and* $\Sigma_2 \preceq \Sigma_3$, *then* $\Sigma_1 \ll \Sigma_3$.

Proof. From premises, we have

- (A1) $\forall l_3 \in dom(\Sigma_3).\Sigma_3 \vDash l_3 : warm \vee l_3 \in dom(\Sigma_2)$
- (A2) $\forall l_2 \in dom(\Sigma_2).\Sigma_2 \vDash l_2 : warm \vee l_2 \in dom(\Sigma_1)$

Consider $l_3 \in dom(\Sigma_3)$, if we have $\Sigma_3 \vDash l_3 : warm$, we are done. If $\Sigma_3 \vDash l_3 : warm$ does not hold, then from (A1) we have $l_3 \in dom(\Sigma_2)$ Now use (A2), it cannot be the case that $\Sigma_{22} : warm$ due to $\Sigma_2 \preceq \Sigma_3$. Therefore, we have $l_3 \in dom(\Sigma_1)$, which completes the proof. □

5.7 Local Reasoning

LEMMA 5.14 (Synchronization). *If we have*

- $\Sigma \vDash \sigma$
- $\forall l'.\sigma \vDash l \rightsquigarrow l' \Longrightarrow \Sigma \vDash l' : warm$

then there exists Σ' *such that*

1. $\Sigma' \vDash \sigma, \Sigma \preceq \Sigma', \Sigma \rhd \Sigma'$ *and* $\Sigma \ll \Sigma'$
2. $\forall l'.\sigma \vDash l \rightsquigarrow l' \Longrightarrow \Sigma' \vDash l' : hot$

Proof. Define Σ' as follows:

$$\Sigma'(l') = \begin{cases} D^{hot} & \text{if } \sigma \vDash l \rightsquigarrow l' \text{ and } \Sigma \vDash l' : D^{\mu}, \\ \Sigma(l') & otherwise \end{cases}$$

It is straightforward to show that $\Sigma \rhd \Sigma'$, because we never change typing for an object l' where $\Sigma(l') = C^{\overline{f}}$. The proof goals $\Sigma \preccurlyeq \Sigma'$ and $\Sigma \ll \Sigma'$ follow immediately from definition of Σ'. The proof goal that all values reachable from l can be typed as hot also follows from the definition of Σ'. We must then prove $\Sigma' \vDash \sigma$, that is:

$$\forall l' \in dom(\Sigma').\Sigma' \vDash \sigma(l') : \Sigma'(l')$$

For any location l' that is not reachable from l, we have $\Sigma'(l') = \Sigma(l') = C^{\mu}$, $\Sigma' \vDash \sigma(l') : C^{\mu}$ follows from object typing monotonicity (Lemma 7.2).

If instead $\sigma \vDash l \rightsquigarrow l'$, $\Sigma' \vDash \sigma(l') : C^{hot}$ follows from the definition of object typing. □

LEMMA 5.15 (Local Reasoning). *The following proposition holds*

$$\frac{(\sigma_1, L_1) \lessdot (\sigma_2, L_2) \qquad \Sigma_1 \ll \Sigma_2 \qquad \Sigma_1 \rhd \Sigma_2 \qquad \Sigma_1 \preceq \Sigma_2 \qquad \Sigma_1 \vDash \sigma_1 \qquad \Sigma_2 \vDash \sigma_2 \qquad \Sigma_1 \vDash L_1 : hot}{\exists \Sigma'.\Sigma_1 \ll \Sigma' \qquad \Sigma_1 \preceq \Sigma' \qquad \Sigma_1 \rhd \Sigma' \qquad \Sigma' \vDash \sigma_2 \qquad \Sigma' \vDash L_2 : hot}$$

Proof. Let's consider a reachable object l from L_2, i.e. $\sigma_2 \vDash L_2 \rightsquigarrow l$. Depending on whether $l \in dom(\sigma_1)$, there are two cases.

- **Case** $l \notin dom(\Sigma_1)$.
  Use the fact that $\Sigma_1 \ll \Sigma_2$, we know $\Sigma_2 \vDash l : warm$.

- **Case** $l \in dom(\Sigma_1)$.
  Use the fact that $(\sigma_1, L_1) \lessdot (\sigma_2, L_2)$, we have $\sigma_1 \vDash L_1 \rightsquigarrow l$. From the premise that L_1 is hot and Lemma 5.9, we have $\Sigma_1 \vDash l : warm$. From $\Sigma_1 \preceq \Sigma_2$, we have $\Sigma_2 \vDash l : warm$.

In both cases, we have $\Sigma_2 \vDash l : warm$. Now we may use Lemma 5.14 for all locations in L_2, the conclusion follows immediately. □

5.8 Selection Lemmas

LEMMA 5.16 (Hot Selection). *If* $\Sigma \vDash \sigma, \Sigma \vDash l : C^{hot}, (C, \omega) = \sigma(l)$ *and* $fieldType(C, f) = D^{\mu}$, *then* $\Sigma \vDash \omega(f) : D^{hot}$.

Proof. It is easy to know $\sigma \vDash l : hot$, thus all its fields are assigned. By object typing for $\sigma(l)$ we immediately get the result. □

LEMMA 5.17 (Warm Selection). *If* $\Sigma \vDash \sigma, \Sigma \vDash l : C^{warm}, (C, \omega) = \sigma(l)$ *and* $fieldType(C, f) = T$, *then* $\Sigma \vDash \omega(f) : T$.

Proof. By the definition of object typing for $\sigma(l)$. □

LEMMA 5.18 (Object Selection). *If* $\Sigma \vDash \sigma, \Sigma \vDash l : C^{\overline{f}}, (C, \omega) = \sigma(l), f \in \overline{f}$ *and* $fieldType(C, f) = T$, *then* $\Sigma \vDash \omega(f) : T$.

Proof. By the definition of object typing for $\sigma(l)$. □

5.9 Initialization Lemmas

LEMMA 5.19 (Field Initialization). *If we have*

- $\Sigma \vDash \sigma$
- $\Sigma(l) = C^{\overline{f}}$
- $(C, \omega) = \sigma(l)$
- $fieldType(C, f') = T$
- $\Sigma \vDash l' : T$
- $\omega' = [f' \mapsto l']\omega$
- $\sigma' = [l \mapsto (C, \omega')]\sigma$
- $\Sigma' = \Sigma \cup \{\ l \mapsto C^{\overline{f} \cup f'}\ \}$

then

- $\Sigma' \vDash \sigma'$
- $\Sigma \preceq \Sigma'$
- $\Sigma \ll \Sigma'$

Proof. The conclusion $\Sigma \preceq \Sigma'$ and $\Sigma \ll \Sigma'$ holds by definition. $\Sigma' \vDash \sigma'$ as the only changed object l continues to type check, other objects type check due to monotonicity. □

LEMMA 5.20 (Promotion)**.** *If we have*

- $\Sigma_1 \rhd \Sigma_2$
- $\Sigma_1 \preceq \Sigma_2$
- $l \notin dom(\Sigma_1)$
- $\Sigma_2(l) = C^{\overline{f}}$
- $\Sigma_2 \vDash \sigma_2$
- $\overline{f} = fields(C)$
- $\Sigma_3 = \Sigma_2 \cup \{\ l \mapsto C^{warm}\ \}$

then

- $\Sigma_1 \rhd \Sigma_3$
- $\Sigma_1 \preceq \Sigma_3$
- $\Sigma_3 \vDash \sigma_2$

Proof. $\Sigma_1 \rhd \Sigma_3$ holds because $l \notin dom(\Sigma_1)$ and $\Sigma_1 \rhd \Sigma_2$. $\Sigma_1 \preceq \Sigma_3$ holds by transitivity of monotonicity. $\Sigma_3 \vDash \sigma_2$ holds as the typing for l holds by definition, other values type check due to monotonicity. □

5.10 Theorem

THEOREM 5.1 (Expression Soundness)**.** *Given*

(1) $\Gamma; U \vdash e : T$
(2) $\Gamma; \Sigma \vDash \rho$
(3) $\Sigma \vDash \sigma$
(4) $\Sigma; \sigma \vDash \psi : U$
(5) $[\![e]\!](\rho, \sigma, \psi)(k) = Some\ result$

then there exists l, σ', Σ' *such that:*

(a) $result = Some(l, \sigma')$
(b) $\Sigma \preceq \Sigma', \Sigma \ll \Sigma', \Sigma \rhd \Sigma'$ *and* $\Sigma' \vDash \sigma'$
(c) $\Sigma'; \sigma' \vDash l : T$

Proof. By induction on k. The case k = 0 trivially holds. For the case k = n + 1, perform induction on the typing judgement (1).

In each case, to deal with fuel, we show that any recursive calls evaluating subexpressions of e do not "time out" — that is, they do not return `None`. This follows because $[\![e]\!](\sigma, \rho, \psi)(k)$ itself does not time out.

- **case** $e = x$
  Choose $l = \rho(x)$, $\sigma' = \sigma$ and $\Sigma' = \Sigma$.

- **case** $e = this$
  Choose $l = \psi$, $\sigma' = \sigma$ and $\Sigma' = \Sigma$.

- **case** $e = e_0.f$
  From the induction hypothesis on e_0, we know there exists l_0, σ_0 and Σ_0 such that:
  - (A1) $\Sigma \preceq \Sigma_0$, $\Sigma \ll \Sigma_0$, $\Sigma \rhd \Sigma_0$ and $\Sigma_0 \vDash \sigma_0$
  - (A2) $\Sigma_0 \vDash l_0 : T_0$

  There are only two possible rules for type checking e.
  - **case** T-SELHOT. We know $T_0 = D^{hot}$, use Lemma 5.16.
  - **case** T-SELWARM. We know $T_0 = D^{warm}$, use Lemma 7.26.
  - **case** T-SELOBJ. We know $T_0 = D^{\overline{f}}$, $f \in \overline{f}$ use Lemma 5.18.

- **case** $e = e_0.m(\overline{e})$
  From the induction hypothesis on e_0, we know there exists l_0 and σ_0 such that:
  - (A1) $\Sigma \preceq \Sigma_0$, $\Sigma \ll \Sigma_0$, $\Sigma \rhd \Sigma_0$ and $\Sigma_0 \vDash \sigma_0$
  - (A2) $\Sigma_0 \vDash l_0 : T_0$
  - (A3) $(C, \omega) = \sigma_0(l_0)$

  Due to the typing for e, we have
  - (B1) $lookup(C, m) = @\mu\ def\ m(\overline{x_i{:}T_i}) : T_r = e_m$
  - (B2) $\Gamma; U \vDash e_i : T_i$

  By using induction hypothesis repeatedly on all arguments, we get l_i, σ_i and Σ_i such that
  - (C1) $\Sigma_{i-1} \preceq \Sigma_i$, $\Sigma_{i-1} \ll \Sigma_i$, $\Sigma_{i-1} \rhd \Sigma_i$ and $\Sigma_i \vDash \sigma_i$
  - (C2) $\Sigma_i \vDash l_i : T_i$

  We prepare the environment for evaluating the method body e_m as follows:
  - (D2) $\rho' = \overline{x_i{:}l_i}$
  - (D3) $[\![e_m]\!]\,(\sigma_n, \rho', l_0) = Some(result)$

  Now we can use the induction hypothesis for e_m:
  - (E1) $\Sigma_n \preceq \Sigma_m$, $\Sigma_n \ll \Sigma_m$, $\Sigma_n \rhd \Sigma_m$ and $\Sigma_m \vDash \sigma_m$
  - (E2) $\Sigma_n \vDash l_m : T_r$

  The case $T = T_r$ follows immediately by transitivity lemmas. The difficult case is to reason that l_m is hot if l_0 and l_i are hot. In the latter case, we use the Lemma 5.15.

- **case** $e = new\ C(\overline{e})$
  From the typing rule T-NEW, we have
  - (A1) $\overline{T_i} = constructorType(C)$
  - (A2) $\Gamma; U \vDash e_i : C_i^{\mu_i}$
  - (A3) $C_i^{\mu_i} <: T_i$
  - (A4) $\mu = (\sqcup \mu_i) \sqcap warm$
  - (A5) $T = C^{\mu}$

Let $\Sigma_0 = \Sigma$, use induction hypothesis on arguments repeatedly:

- (B1) $\Sigma_{i-1} \preceq \Sigma_i$, $\Sigma_{i-1} \ll \Sigma_i$, $\Sigma_{i-1} \rhd \Sigma_i$ and $\Sigma_i \vDash \sigma_i$
- (B2) $\Sigma_i \vDash l_i : T_i$

By transitivity, we have

- (C1) $\Sigma \preceq \Sigma_n$
- (C1) $\Sigma \ll \Sigma_n$
- (C2) $\Sigma \rhd \Sigma_n$
- (C3) $\Sigma_n \vDash l_i : T_i$

We define σ'_0 and Σ'_0 with a fresh location l:

- (D1) $\sigma'_0 = \sigma_n \cup \{l \mapsto (C, \overline{\hat{f}_i = l_i})\}$
- (D2) $\Sigma'_0 = \Sigma_n \cup \{l \mapsto C^{\overline{\hat{f}_i}}\}$
- (D3) $\Sigma_n \preceq \Sigma'_0$, $\Sigma_n \rhd \Sigma'_0$ and $\Sigma'_0 \vDash \sigma'_0$ $\rhd$ By definition

By transitivity, we have:

- (E1) $\Sigma \preceq \Sigma'_0$
- (E2) $\Sigma \rhd \Sigma'_0$

Suppose the class has m fields in class body, we prove that the following invariant holds:

- (F1) $\Sigma \preceq \Sigma'_m$
- (F2) $\Sigma \rhd \Sigma'_m$
- (F3) $\Sigma'_0 \ll \Sigma'_m$
- (F4) $\Sigma'_m(l) = C^{\{\ \hat{f}_1,\ldots,\hat{f}_n,f_1,\ldots,f_m\ \}}$

By induction on m. The basic case $m = 0$ trivially holds as $\sigma'_m = \sigma'_0$ and $\Sigma'_m = \Sigma'_0$.

Let us consider $m = i + 1$. From induction hypothesis, we have

- (G1) $\Sigma \preceq \Sigma'_i$
- (G2) $\Sigma \rhd \Sigma'_i$
- (G3) $\Sigma'_0 \ll \Sigma'_i$
- (G4) $\Sigma'_i(l) = C^{\{\ \hat{f}_1,\ldots,\hat{f}_n,f_1,\ldots,f_i\ \}}$

Consider the (i+1)-th field in the class body $var\ f_{i+1} : T_{i+1} = e_{i+1}$. Use induction hypothesis on e_{i+1}, we know there exists $l_{i+1}, \hat{\sigma}_{i+1}$ such that:

- (H1) $[\![e_{i+1}]\!](\sigma'_i, \emptyset, l) = (l_{i+1}, \hat{\sigma}_{i+1})$
- (H2) $\Sigma'_i \preceq \hat{\Sigma}_{i+1}$, $\Sigma'_i \ll \hat{\Sigma}_{i+1}$, $\Sigma'_i \rhd \hat{\Sigma}_{i+1}$ and $\hat{\Sigma}_{i+1} \vDash \hat{\sigma}_{i+1}$
- (H3) $\hat{\Sigma}_{i+1} \vDash l_{i+1} : T_{i+1}$

Now we have:

- (I1) $\Sigma \preceq \hat{\Sigma}_{i+1}$

– (I2) $\Sigma \rhd \hat{\Sigma}_{i+1}$ ▷ From (H2), (G1), (G2) and Lemma 5.11
– (I3) $\Sigma'_0 \ll \hat{\Sigma}_{i+1}$
– (I4) $\hat{\Sigma}_{i+1}(l) = C^{\{\ \hat{f}_1,\ldots,\hat{f}_n,f_1,\ldots,f_i\ \}}$ ▷ From (H2) and (G4)

We define σ'_m and Σ'_m as follows:
– (H1) $(C,\omega) = \hat{\sigma}_{i+1}(l)$
– (H2) $\omega' = [f_{i+1} \mapsto l_{i+1}]\omega$
– (H3) $\sigma'_m = [l \mapsto (C,\omega')]\hat{\sigma}_{i+1}$
– (H4) $\Sigma'_m = \hat{\Sigma}_{i+1} \cup \{\ l \mapsto C^{\{\ \hat{f}_1,\ldots,\hat{f}_n,f_1,\ldots,f_i,f_{i+1}\ \}}\ \}$

Now use Lemma 5.19, we have
– (K1) $\hat{\Sigma}_{i+1} \preceq \Sigma'_m$
– (K2) $\hat{\Sigma}_{i+1} \ll \Sigma'_m$
– (K3) $\Sigma'_m \vDash \sigma'_m$

Using transitivity again, we
– (L1) $\Sigma \preceq \Sigma'_m$ ▷ From (I1) and (K1)
– (L2) $\Sigma \rhd \Sigma'_m$ ▷ From (I2), (H4) and $l \notin dom(\Sigma)$
– (L3) $\Sigma'_0 \ll \Sigma'_m$ ▷ From (I3) and (K2)

As $l \notin dom(\Sigma)$, we can use Lemma 5.20:
– (M1) $\Sigma' = \Sigma'_m \cup \{\ l \mapsto C^{warm}\ \}$
– (M2) $\Sigma' \vDash \sigma'_m$
– (M3) $\Sigma \preceq \Sigma'$
– (M4) $\Sigma \rhd \Sigma'$

The proof goal $\Sigma \ll \Sigma'$ holds from (C1) and $\Sigma_n \ll \Sigma'$. The latter holds from (L3), (M1), (D2) and the fact that l now is warm.

If $\mu = warm$, the conclusion is immediate by choosing l, $\sigma' = \sigma'_m$ and Σ'.

When $\mu = hot$, then by the typing rule T-NEW all arguments $\overline{l_i}$ are hot. In this case, we need to repeat the induction steps above with an additional hypothesis:
– (N1) $(\sigma_n, \overline{l_i}) \lessdot (\sigma_m, l)$

It holds trivially in the case $m = 0$, as its fields are exactly $\overline{l_i}$. For the case $m = i+1$, it continues to hold by induction hypothesis. Now use Lemma 5.15, we arrive at the conclusion.

- **case** $e = (e_0.f = e_1; e_2)$
From the typing derivation, we have
– (A1) $\Gamma; U \vDash e_0 : C^{\mu_0}$

– (A2) $D^{\mu} = fieldTpye(C, f)$
– (A3) $\Gamma; U \vDash e_1 : D^{hot}$
– (A4) $\Gamma; U \vDash e_2 : T_2$

From the induction hypothesis on e_0, we know there exists l_0, σ_0 and Σ_0 such that:
– (B1) $\Sigma \preceq \Sigma_0$, $\Sigma \ll \Sigma_0$, $\Sigma \rhd \Sigma_0$ and $\Sigma_0 \vDash \sigma_0$
– (B2) $\Sigma_0 \vDash l_0 : C^{\mu_0}$

From the induction hypothesis on e_1, we know there exists l_1, σ_1 and Σ_1 such that:
– (C1) $\Sigma_0 \preceq \Sigma_1$, $\Sigma_0 \ll \Sigma_1$, $\Sigma_0 \rhd \Sigma_1$ and $\Sigma_1 \vDash \sigma_1$
– (C2) $\Sigma_1 \vDash l_1 : D^{hot}$

We define σ_1' and Σ_1' as follows
– (D1) $(C, \omega) = \sigma_1(l_0)$
– (D2) $\omega' = [f \mapsto l_1]\omega$
– (D3) $\sigma_1' = [l_0 \mapsto (C, \omega')]\sigma_1$
– (D4) $\Sigma_1' = \Sigma_1$

We have the following by definition
– (E1) $\Sigma_1 \preceq \Sigma_1'$, $\Sigma_1 \ll \Sigma_1'$, $\Sigma_1 \rhd \Sigma_1'$ and $\Sigma_1' \vDash \sigma_1'$

Now use induction hypothesis on e_2, there exists l_2, σ_2 and Σ_2 such that
– (F1) $\Sigma_1' \preceq \Sigma_2$, $\Sigma_1' \ll \Sigma_2$, $\Sigma_1' \rhd \Sigma_2$ and $\Sigma_2 \vDash \sigma_2$
– (F2) $\Sigma_2 \vDash l_2 : T_2$

The conclusion follows by transitivity.

□

COROLLARY 5.1 (Soundness). *If* $\vdash \mathcal{P}$, *then* $\forall k.\ evalProg(\mathcal{P})(k) \neq Some\ None$.

5.11 Discussion

5.11.1 Monotonicity

One might attempt to define monotonicity without resorting to store typing:

DEFINITION 5.1 (Monotonicity - Wrong Attempt).

$$\sigma \preceq \sigma' \quad \triangleq \quad \forall l \in dom(\sigma).\quad (C, \omega) = \sigma(l) \Longrightarrow \\ (C, \omega') = \sigma'(l) \quad \bigwedge \quad \forall f \in dom(\omega). \sigma \vDash \omega(f) : \mu \Longrightarrow \sigma' \vDash \omega'(f) : \mu$$

This definition, while reasonable, is too rigid to be used, as it enforces more constraints than what the type system assumes. One particular difficulty with this definition is that in reasoning

about monotonicity during object initialization, we need to reason that either a field is vacant or it holds a value that is less initialized than the value produced by the field initializer. This creates unnecessary complexity in the proofs.

Semantically, it is fine to assign to a field before its formal initialization, as long as the type system only assumes that the field is assigned after its formal initialization in the constructor. Store typing helps here to abstract over the concrete heap and thus avoids the nasty details. The property of authority ensures that the initialization status of an object is only officially advanced in the constructor, even if the actual state of the object in the heap already advances.

A related question is: can we prove monotonicity separately? The concept of monotonicity we adopt in our system is *perfect monotonicity*. Unlike *weak monotonicity*, the proof of perfect monotonicity depends on expression soundness in the case of assignment. Consequently, it is impossible to prove monotonicity separately from expression soundness.

5.11.2 Stackability

Another question is, why do we need to define *stackability* as $\Sigma \ll \Sigma'$, instead of using the definition $\sigma \ll \sigma'$ from chapter 3? The reason is that $\sigma \ll \sigma'$ is too weak, as it only says what is happening in the actual heap, but it says nothing about the abstract heap, i.e. the store typing. More technically, in the Lemma 5.15, we need to reason that all newly created objects in σ_2 are formally typed as *warm* in Σ_2. This is required when we advance the store typing in Lemma 5.14, as it is only safe to advance the state of objects which are already taken as warm in the store typing. Otherwise, we run the risk of breaking authority ($\Sigma_1 \rhd \Sigma_2$), which is necessary in reasoning monotonicity ($\Sigma_1 \preceq \Sigma_2$) during field initialization and in the promotion of an object to warm when all fields of the object are formally initialized.

5.11.3 Local Reasoning

Local reasoning (Lemma 5.15) plays an important role in the soundness proof. In contrast to local reasoning presented in chapter 3, which is defined on the concrete heap, the local reasoning in this chapter is defined on the abstract heap, i.e. the store typing. The *abstract* local reasoning is more complex as we also need to show that authority ($\Sigma_1 \rhd \Sigma_2$), stackability ($\Sigma_1 \ll \Sigma_2$), and monotonicity ($\Sigma_1 \preceq \Sigma_2$) are preserved in the update of the store typing.

5.12 Conclusion

In this chapter, we proved soundness of the basic model. Local reasoning (Lemma 5.15) plays an important role in the proof. The proof also shows that the semantic property *authority* is important for soundness, which justifies the principle of authority.

The meta-theory developed here is used as a foundation for developing the meta-theory of the type-and-effect system in the next chapter.

Chapter 6

An Inference System

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.

— Tony Hoare

Joe Duffy, in his popular blog post [13], wrote the following about the type-based approach to safe initialization:

> To be honest, the reason this approach has likely not yet seen widespread use is that the cost is not commensurate with the benefit. ... For systems programmers, this makes sense. For many other programmers, it would be useless ceremony with no perceived value.

In this chapter, we present an inference system for a practical fragment of the basic model to cut down its syntactic overhead.

6.1 Motivation

If we use the type system presented in the last chapter, we would have to add the annotations `@cold` in the following program to make it type check:

```
1 abstract class AbstractFile {
2   @cold def name: String
3   val extension: String = name.substring(4)
4 }
5
6 class RemoteFile(url:String) extends AbstractFile {
7   val localFile: String = url.hashCode
8   @cold def name: String = localFile // error
9 }
```

The annotations incur overhead in programming thus harm usability. Meanwhile, we find the type-based approach not expressive enough as it suffers from the *fragile base class* problem, as the following code demonstrates:

```
class Base { def g(): String = "hello" }
class Foo extends Base { val a = this.g() }
class Bar extends Base {
  val b: String = "b"
  override def g(): String = this.b
}
```

This program is correct. However, if we follow a type-based approach, in order to call `g()` in the class `Foo`, the method `Base.g` has to be annotated `@cold`, so that it may not access any fields on `this`. For soundness, the overriding method `Bar.g` has to be annotated `@cold` too: but now it may not access the field `this.b` in the body of the method `Bar.g`. This unnecessarily restricts expressiveness of the system.

Meanwhile, it is unclear to us how to extend the type system to support complex language features, such as traits. Reasoning about initialization gets subtle in the presence of traits [21, 26], as the following example shows:

```
trait TA { val x = "EPFL" }
trait TB { def x: String; val n = x.length }
class Foo extends TA with TB
class Bar extends TB with TA
new Foo // ok
new Bar // error
```

In the code above, the class `Foo` and class `Bar` only differ in the order in which the traits are mixed in. For the class `Foo`, the body of the trait `TA` is evaluated before the body of `TB`, thus the expression `new Foo` works as expected. In contrast, `new Bar` throws an exception, because the body of the trait `TB` is evaluated first, and at the time the field `x` in `TA` is not yet initialized when it is used in `TB`.

In the following, we identify a practical fragment of the basic model, and then develop an effective inference system for the fragment that scales better to complex language features and significantly reduces syntactic overhead of the basic model.

6.2 A Practical Fragment

The fragment of the basic model that we identify demands that (1) *method arguments must be hot*, and (2) *non-hot class parameters must be annotated*. The fragment supports calling methods on $this$ in the constructor, as well as creation of cyclic data structures. There are several considerations for the restrictions.

First, from practical experience, there is little need to use non-hot values as method arguments. Meanwhile, virtual method calls on `this` are allowed, which covers most use cases in practice [14].

Second, it agrees with good programming practices that values under initialization should not escape [18]. Therefore, when there is the need to pass non-hot arguments to a constructor, it is a good practice to mark it explicitly.

Third, demanding method arguments to be hot saves us from changing the core type system of a language to check safe overriding of virtual methods. Integrating a type-based system in the compiler poses an engineering challenge, as the following example demonstrates:

```
class Knot {
  val self: Knot @cold = this
}
```

In the code above, the type of the field `self` depends on *when* we ask for its type. If it is queried during the initialization of the object, then it has the type `Knot @cold`. Otherwise, it has the type `Knot`. We do not see a principled way to implement the type-based solution in the Scala 3 compiler.

Finally, for such a fragment, there exists an effective inference algorithm in the style of type-and-effect systems [36, 31], which can significantly cut down the syntactic overhead of type-based approaches.

6.3 The Design

In this section, we discuss the design ideas behind the type-and-effect inference system.

6.3.1 Potentials and Effects

Consider the following erroneous program, which accesses the field y before it is initialized:

```
class Point {
    var x: Int = this.y      // Point.this.y!
    var y: Int = 10
}
```

A natural idea to ensure safe initialization is to analyze the fields that are accessed at each step of initialization, and check that only initialized fields are accessed. This leads to the fundamental effect in initialization: **field access effect**, e.g. $C.this.f!$.

Fields may also be accessed *indirectly* through method calls, as the following code shows:

```
class Point {
    var x: Int = this.m()  // Point.this.m◊
    var y: Int = 10
    def m(): Int = this.y  // Point.this.y!
}
```

For this case, we may introduce method calls as effects, which act as placeholders for the actual effects that happen in the method: **method call effects**, e.g. $C.this.m◊$.

If we first analyze effects of the method m and map the effect $Point.this.m\Diamond$ to the set of effects $\{Point.this.y!\}$, then we may effectively check the initialization error in the code above.

On subtlety in the design is how to handle aliasing. We illustrate with the following example:

```
class Knot {
    var self  = this      // potentials of "self": {  Knot.this  }
    var x: Int = self.x   // effects of "self.x":  {  Knot.this.self.x !  }
}
```

In the code above, the field `x` is used via the alias `self` before it is initialized. To check such errors, we need a way to represent the aliasing information in the system. That leads us to the concept of **potentials**. A set of potentials represents aliasing of objects possibly under initialization. If an expression has an empty set of potentials, it means at runtime the value of the expression is always hot.

A potential encodes aliasing information in the form of paths, such as `C.this`, `C.this.f` or `C.this.m`. The paths are of finite length, and the maximum length can be parameterized. In the code example above, the field `self` takes the potential of its initializer, i.e. the set `{ Knot.this }`. Now an initialization checker may take advantage of the aliasing information and report an error for the code `self.x`.

To enforce that all arguments to method calls are hot, we introduce **promotion effects** that promote potentials to be hot, e.g. $C.this\uparrow$. The checking system will check that only hot objects are promoted. The following example illustrates the usage of the effect:

```
class Point {
    var x: Int = this.m()        // Point.this.m◊
    def m(): Int = call(this)    // Point.this↑
}
```

In the code above, the method call effect $Point.this.m\Diamond$ incurs the promotion effect $Point.this\uparrow$. The system finds that at the point of the call `this.m()`, the value of `this` is not hot, such promotion is thus illegal.

The promotion effect has a semantic interpretation. Semantically, potentials keep track of objects possibly under initialization in order to maintain a *directed segregation* of initialized objects and objects under initialization: the latter may point to the former, but not vice versa. A promotion effect means that the object pointed to by the potential ascends to the initialized world, and the system gives up on tracking it. The system will have to ensure that by the time this happens, the object is hot.

Note that field access $C.this.a!$ and field promotion $C.this.a\uparrow$ are different effects, because field access does not necessarily need to promote the field, as demonstrated by the following example:

```
class Knot {
  var a = this
  var b = this.a // Knot.this.a! , but no promotion
}
```

Aliasing and promotion may also happen through methods, as the following example shows:

```
1 class Fact {
2     var value = escape(this.m()) // Fact.this.m↑
3     def m() = this               // potentials of m: {   Fact.this   }
4 }
```

The type-and-effect system knows that the return value of the method `m` aliases `this`, thus the promotion of `this.m()` at line 2 indirectly promotes `this`.

A similar distinction is drawn on methods: (1) the method invocation effect $C.this.m\Diamond$ means that the method `m` is called with the receiver `this`; (2) the method promotion effect $C.this.m\uparrow$ means that the return value of the call `this.m` is promoted.

6.3.2 Two-Phase Checking

A common issue in program analysis is how to deal with recursive methods. We tackle the problem with *two phase checking*. In the first phase, the system computes effect summaries for methods and fields. In the second phase, the system checks that no fields are used before they are initialized. During the checking, it uses the effect summaries from the first phase. For example, assume the following program:

```
1 class Foo {
2   var a: Int = h()
3   def h(): Int = g()
4   def g(): Int = h()
5 }
```

In the first phase, the computed summary for the methods `h` and `g` is as follows:

method	effects	potentials
h	$\{\ Foo.this.g\Diamond\ \}$	$\{\ Foo.this.g\ \}$
g	$\{\ Foo.this.h\Diamond\ \}$	$\{\ Foo.this.h\ \}$

In the second phase, while checking the method call `h()`, the analysis propagates the effects associated with the method `h` until it reaches the fixed point $\{\ Foo.this.g\Diamond, Foo.this.h\Diamond\ \}$. As the set does not contain accesses to any uninitialized fields of `this` nor invalid promotion, the program passes the check. Note that the domain of effects has to be finite for the existence of the fixed point.

6.3.3 Full-Construction Analysis

Another common issue in initialization is how to handle virtual method calls. The approach we take is *full-construction analysis*: we treat the constructors of concrete classes as entry points, and check all super constructors as if they were inlined. The analysis spans the full duration of object construction. This way, all virtual method calls on `this` can be resolved statically. From our experience, full-construction analysis greatly improves user experience, as no annotations are required for the interaction between subclasses and superclasses.

Full-construction analysis easily solves the *fragile base class problem* mentioned at the beginning of the chapter. Moreover, we believe it is the only practical way to handle complex language features such as properties and traits. In languages such as Scala and Kotlin, fields are actually properties, public field accesses are dynamic method calls, as the following code shows:

```
1 class A { val a = "Bonjour"; val b: Int = a.size }
2 class B extends A { override val a = "Hi" }
3 new B
```

In the code above, when the constructor of class `B` calls the constructor of class `A`, the expression `a.size` will dynamically dispatch to read the field `a` declared in class `B`, not the field `a` declared in class `A`. This results in a null-pointer exception at runtime because at the time the field `a` in class `B` is not yet initialized. Without full-construction analysis, it is difficult to make the analysis sound for the code above.

6.3.4 Cyclic Data Structures

Cyclic data structures are supported with an annotation $@cold$ on class parameters, as the following example demonstrates:

```
1 class Parent { val child: Child = new Child(this) }
2 class Child(parent: Parent @cold) {
3   val friend: Friend = new Friend(this.parent)
4 }
5 class Friend(parent: Parent @cold) { val tag = 10 }
```

The annotation `@cold` indicates that the actual argument to `parent` during object construction might not be initialized. The type-and-effect system will ensure that the field `parent` is not used directly or indirectly when instantiating `Child`. However, aliasing the field to another cold class parameter is fine, thus the code `new Friend(this.parent)` at line 3 is accepted by the system. This allows programmers to create complex aliasing structures during initialization.

Our system tracks the return value of `new Child(this)` as the set of potentials $\{\ warm[Child]\ \}$. All fields of a warm value are assigned, but they may hold values that are not fully initialized.

6.4 Formalization

We start by introducing the syntax and semantics of our experimental language.

6.4.1 Syntax and Semantics

Our language is almost the same as the language introduced in chapter 3, except for the definition of class parameters. In a class definition like $class\ C(\overline{\hat{f}{:}T})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \}$, we introduce *cold class parameters*, which have the syntax $\tilde{f}$. Cold class parameters may take a value that is not transitively initialized. A class parameter $\hat{f}$ is also a field of its defining class. By default, we

use f to range over all fields, $\hat{f}$ to range over class parameters, and $\tilde{f}$ to range over cold class parameters.

The tilde annotation on $\tilde{f}$ is only used in the type-and-effect system; it does not have runtime semantics. That is the only annotation that is required in source code.

The semantics is the same as the language in chapter 3, we thus omit the details.

6.4.2 Effects and Potentials

As seen from Figure 6.1, the definition of potentials (π) and effects (ϕ) depends on *roots* (β). Roots are the shortest path that represents an alias of a value that may not be transitively initialized. There are three kinds of roots in the current system:

- $C.this$ represents aliasing of the underlying object referenced by *this* inside class C.
- $warm[C]$ represents aliasing of an instance of the class C, all fields of which are assigned, but it may not be transitively initialized.
- $cold$ represents a value whose initialization status is unknown. It is used to represent the potential of cold class parameters. Field access or method calls on such an object is forbidden.

Potentials (π) represent aliasing information. They extend roots with field aliasing $\beta.f$ and method aliasing $\beta.m$. Field aliasing $\beta.f$ represents aliasing of the field f of β, while method aliasing $\beta.m$ represents aliasing of the return value of the method m with the receiver β.

Effects (ϕ) include field accesses, method calls and promotions of possibly uninitialized values. A promotion effect is represented with $\pi\uparrow$, which enforces that the potential π is transitively initialized. The field access effect $\beta.f!$ means that the field f is accessed on β. The method call effect $\beta.m\Diamond$ means that the method m is called on β.

There are three helpers for the creation of potentials and effects:

- Field selection: $select(\Pi, f)$
- Method call: $call(\Pi, m)$
- Class instantiation: $init(C, \overline{\hat{f}_i = \Pi_i})$

The helper method $select(\Pi, f)$ has five cases:

- selection of cold class parameter $\tilde{f}$ on β
- selection of non-cold class parameter $\hat{f}$ on β
- selection of body field f on $C.this$ or $warm[C]$
- selection of body field f on $cold$
- selection of a field f on $\beta.f$ or $\beta.m$

In the first case, the field $\tilde{f}$ may hold a value that is not transitively initialized, thus the potential is represented as *cold*. The effect is empty, as class parameters are always initialized before the class body is executed.

For the same reason, in the second case, the effects are empty. The potentials are empty because a non-cold class parameter may only hold a value that is transitively initialized, thus we do not need to track it in the system.

In the third case, selecting a body field on $C.this$ produces the effect $C.this.f!$ due to field access, and the potential $C.this.f$ due to the fact that the field f may hold a value which is not transitively initialized. The case for $select(warm[C], m)$ is similar.

Potentials and Effects

$$
\begin{array}{rcll}
T & ::= & C \mid D \mid E \mid \cdots & \text{type} \\
\beta & ::= & C.this \mid warm[C] \mid cold & \text{root} \\
\pi & ::= & \beta \mid \beta.f \mid \beta.m & \text{potential} \\
\Pi & ::= & \{\, \pi_1, \pi_2, \cdots \} & \text{potentials} \\
\phi & ::= & \pi\uparrow \mid \beta.f! \mid \beta.m\Diamond & \text{effect} \\
\Phi & ::= & \{\, \phi_1, \phi_2, \cdots \} & \text{effects} \\
\Omega & ::= & \{\, f_1, f_2, \cdots \} & \text{fields} \\
\Delta & ::= & \overline{f_i \mapsto (\Phi_i, \Pi_i)} & \text{field summary} \\
\mathcal{S} & ::= & \overline{m_i \mapsto (\Phi_i, \Pi_i)} & \text{method summary} \\
\mathcal{E} & ::= & \overline{C \mapsto (\Delta, \mathcal{S})} & \text{effect table}
\end{array}
$$

Select

$$
\begin{array}{rcl}
select(\Pi, f) & = & \Pi.map(\pi \Rightarrow select(\pi, f)).reduce(\oplus) \\
select(\beta, \tilde{f}) & = & (\emptyset, \{cold\}) \\
select(\beta, \hat{f}) & = & (\emptyset, \emptyset) \\
select(\beta, f) & = & (\{\beta.f!\}, \{\beta.f\}) \text{ where } \beta \neq cold \\
select(cold, f) & = & (\{cold\uparrow\}, \emptyset) \\
select(\pi, f) & = & (\{\pi\uparrow\}, \emptyset) \text{ where } \pi = \beta.f \text{ or } \pi = \beta.m
\end{array}
$$

Call

$$
\begin{array}{rcl}
call(\Pi, m) & = & \Pi.map(\pi \Rightarrow call(m, \pi)).reduce(\oplus) \\
call(\beta, m) & = & (\{\beta.m\Diamond\}, \{\beta.m\}) \text{ where } \beta \neq cold \\
call(cold, m) & = & (\{cold\uparrow\}, \emptyset) \\
call(\pi, m) & = & (\{\pi\uparrow\}, \emptyset) \text{ where } \pi = \beta.f \text{ or } \pi = \beta.m
\end{array}
$$

Init

$$
\begin{array}{rcl}
init(C, \overline{\hat{f}_i = \Pi_i}) & = & (\cup\overline{\Pi_{k\neq j}\uparrow}, \{warm[C]\}) \text{ if } \exists \tilde{f}_j, \Pi_j \neq \emptyset \\
init(C, \overline{\hat{f}_i = \Pi_i}) & = & (\cup\overline{\Pi_i\uparrow}, \emptyset)
\end{array}
$$

Helpers

$$
\begin{array}{rcl}
\Pi\uparrow & = & \{\, \pi\uparrow \mid \pi \in \Pi \,\} \\
(A_1, A_2) \oplus (B_1, B_2) & = & (A_1 \cup B_1, A_2 \cup B_2)
\end{array}
$$

Figure 6.1 – Definition of Potentials and Effects

The case $select(cold, f)$ simply promotes the potential $cold$, which will cause the program to be rejected if the effect is triggered during object initialization.

In the last case, the system just promotes the potential π, which is equivalent to say that π is a transitively initialized value, thus the potential of the selection is empty. The system restricts the length of effects and potentials to make the domain finite. In the formalization, we set the length to 2. When the maximum length is reached, we promote the potential and give up tracking of it in the system.

The helper method $call(\Pi, m)$ distinguishes three cases:

- the receiver is $C.this$ or $warm[C]$
- the receiver is $cold$
- the receiver is $\beta.f$ or $\beta.m$

In the first case, it produces the effect $\beta.m\Diamond$ and potential $\beta.m$ — remember $\beta.m\Diamond$ is a placeholder to say all effects associated with the method m, and $\beta.m$ to all potentials associated with the return value of the method m.

The case $call(cold, m)$ simply promotes the potential $cold$, which will cause the program to be rejected if the effect is triggered during object initialization.

In the last case, the system just promotes the potential π, just as the case of selection. The resulting potential is empty, which means the result is also transitively initialized. This is guaranteed by local reasoning of initialization, given that both the receiver and arguments are transitively initialized.

The helper method $init(C, \overline{\hat{f}_i = \Pi_i})$ distinguishes two cases:

- the class parameters of C accept values under initialization and there exists at least one corresponding argument whose potential is non-empty.

- either class C does not accept values under initialization or all potentials for cold class parameters $\tilde{f}$ are empty.

In the first case, it promotes all potentials that do not correspond to cold class parameters, which is equivalent to saying that these arguments are transitively initialized. The potentials corresponding to cold class parameters $\tilde{f}$ are absorbed by the fields $\tilde{f}$ in the resulting potential $warm[C]$.

In the second case, it promotes all potentials of the arguments to ensure that they are transitively initialized. The result potential is empty, i.e., it must be a transitively initialized value, which is guaranteed by local reasoning (chapter 3).

To simplify our presentation, we use the syntax $\Pi\uparrow$ to denote the set $\{\ \pi\uparrow \mid\ \pi \in \Pi\ \}$.

6.4.3 Expression Typing

Expression typing (Figure 6.2) has the form $\Gamma; C \vdash e : D\ !\ (\Phi, \Pi)$, which means that the expression e in class C under the environment Γ, can be typed as D, and it produces effects Φ and has the potential Π. Generally, when typing an expression, the effects of sub-expressions will **accumulate**, while potentials may be **refined** (via selection), **promoted** (used as arguments to methods).

The definitions assume several helper methods, such as $fieldType(C, f)$, $methodType(C, m)$ and $constrType(C)$, to look up in class table Ξ the type, respectively, of field $C.f$, of method $C.m$ and of the constructor of C.

Expression Typing $\boxed{\Gamma; C \vdash e : D \,!\, (\Phi, \Pi)}$

$$\frac{x : D \in \Gamma}{\Gamma; C \vdash x : D \,!\, (\emptyset, \emptyset)} \quad \text{(T-VAR)}$$

$$\Gamma; C \vdash this : C \,!\, (\emptyset, \{C.this\}) \quad \text{(T-THIS)}$$

$$\frac{\Gamma; C \vdash e : D \,!\, (\Phi, \Pi) \qquad (\Phi', \Pi') = select(\Pi, f) \qquad E = fieldType(D, f)}{\Gamma; C \vdash e.f : E \,!\, (\Phi \cup \Phi', \Pi')} \quad \text{(T-SEL)}$$

$$\frac{\Gamma; C \vdash e_0 : E_0 \,!\, (\Phi, \Pi) \qquad \overline{\Gamma; C \vdash e_i : E_i \,!\, (\Phi_i, \Pi_i)} \qquad (\overline{x_i{:}E_i}, D) = methodType(E_0, m) \qquad (\Phi', \Pi') = call(\Pi, m)}{\Gamma; C \vdash e_0.m(\overline{e}) : D \,!\, (\Phi \cup \overline{\Phi_i} \cup \overline{\Pi_i\uparrow} \cup \Phi', \Pi')} \quad \text{(T-CALL)}$$

$$\frac{\overline{\hat{f}_i{:}E_i} = constrType(C) \qquad \overline{\Gamma; C \vdash e_i : E_i \,!\, (\Phi_i, \Pi_i)} \qquad (\Phi', \Pi') = init(C, \overline{\hat{f}_i = \Pi_i})}{\Gamma; C \vdash new\ C(\overline{e}) : C \,!\, (\cup\overline{\Phi_i} \cup \Phi', \Pi')} \quad \text{(T-NEW)}$$

$$\frac{\Gamma; C \vdash e_0 : E_0 \,!\, (\Phi_0, \Pi_0) \qquad E_1 = fieldType(E_0, f) \qquad \Gamma; C \vdash e_1 : E_1 \,!\, (\Phi_1, \Pi_1) \qquad \Gamma; C \vdash e_2 : E_2 \,!\, (\Phi_2, \Pi_2)}{\Gamma; C \vdash e_0.f = e_1; e_2 : E_2 \,!\, (\Phi_0 \cup \Phi_1 \cup \Pi_1\uparrow \cup \Phi_2, \Pi_2)} \quad \text{(T-BLOCK)}$$

Figure 6.2 – Expression Typing

In the typing rule T-VAR, the effects are empty, because accessing a variable cannot cause any initialization error. The potential is empty because the design of the system ensures that variables are transitively initialized, thus they do not need to be tracked in the system.

In the typing rule T-THIS, the effect is empty, and the potential is $C.this$, as it aliases *this* in class C.

The typing rule T-SEL first computes the effects Φ and potentials Π of the expression e. Then it calls $select(\Pi, f)$ to produce the final potentials Π' and accompanied effects Φ'.

The typing rule T-CALL first checks the receiver e_0 and the arguments e_i. Then it calls the helper function $call(\Pi, m)$.

Note that in the current system, method arguments must be transitively initialized. This fact is expressed in the method $call$ by promoting all potentials of the arguments as effects.

To type check $new-$expressions, the typing rule T-NEW first type checks all arguments, then it calls the helper method $init(C, \overline{\hat{f}_i = \Pi_i})$.

Finally, to type check a block expression $e_0.f = e_1; e_2$, the typing rule T-BLOCK first type checks e_0, e_1 and e_2 separately. Then in the final effect, it promotes the potentials Π_1 for e_1, which ensures that the value of e_1 is transitively initialized. This is how monotonicity of initialization is enforced in the system.

6.4.4 Definition Typing

Program Typing $\boxed{\vdash \mathcal{P}}$

$$\frac{\Xi = \overline{C \mapsto \mathcal{C}} \quad \overline{\Xi(D) = class\ D\ \{\ def\ main : T = e\ \}} \quad \overline{\Xi \vdash \mathcal{C} \,!\, (\Delta_c, \mathcal{S}_c)} \quad \mathcal{E} = \overline{C \mapsto (\Delta_c, \mathcal{S}_c)} \quad \overline{\emptyset; D \vdash e : T \,!\, (\Phi, \Pi)} \quad \overline{\Xi; \mathcal{E} \vdash \mathcal{C}}}{\vdash (\overline{\mathcal{C}}, D)} \text{ (T-PROG)}$$

Effect Checking $\boxed{\Xi; \mathcal{E} \vdash \mathcal{C}}$

$$\frac{(\Delta, \_) = \mathcal{E}(C) \quad \overline{(\Phi, \_) = \Delta(f_i) \quad \mathcal{E}; C^{\{\ f_1, \cdots, f_{i-1}\ \}} \vdash \Phi}}{\Xi; \mathcal{E} \vdash class\ C(\overline{\hat{f}{:}T})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \}} \text{ (T-CHECK)}$$

Class Typing $\boxed{\Xi \vdash \mathcal{C} \,!\, (\Delta, \mathcal{S})}$

$$\frac{\overline{\Xi; C \vdash \mathcal{F}_i \,!\, (\Phi_i, \Pi_i)} \quad \Delta = \overline{f_i \mapsto (\Phi_i, \Pi_i)} \quad \overline{\Xi; C \vdash \mathcal{M}_i \,!\, (\Phi_i, \Pi_i)} \quad \mathcal{S} = \overline{m_i \mapsto (\Phi_i, \Pi_i)}}{\Xi \vdash class\ C(\overline{\hat{f}{:}T})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \} \,!\, (\Delta, \mathcal{S})} \text{ (T-CLASS)}$$

Field Typing $\boxed{\Xi; C \vdash \mathcal{F} \,!\, (\Phi, \Pi)}$

$$\frac{\emptyset; C \vdash e : D \,!\, (\Phi, \Pi)}{\Xi; C \vdash var\ f : D = e \,!\, (\Phi, \Pi)} \text{ (T-FIELD)}$$

Method Typing $\boxed{\Xi; C \vdash \mathcal{M} \,!\, (\Phi, \Pi)}$

$$\frac{\overline{x{:}D}; C \vdash e : E \,!\, (\Phi, \Pi)}{\Xi; C \vdash\ def\ m(\overline{x{:}D}) : E = e \,!\, (\Phi, \Pi)} \text{ (T-METHOD)}$$

Figure 6.3 – Definition Typing

Definition typing (Figure 6.3) defines how programs, classes, fields and methods are checked. The checking happens in two phases:

(1) *first phase*: conventional type checking is performed and effect summaries are computed;

(2) *second phase*: effect checking is performed to ensure initialization safety.

The two-phase checking is reflected in the typing rule T-PROG. To type check a program $(\overline{\mathcal{C}}, e)$, first each class is type checked separately for well-typing and the effect summary for fields Δ_c and methods $\mathcal{S}_c$ is computed. Then effect checking is performed modularly on each class with the help of the effect table $\mathcal{E}$. The typing rule T-PROG also checks that the entry expression e is well-typed with the empty environment and the type $Null$ for this. The usage of $Null$ for

the type of $this$ for the entry expression e unifies the semantics and typing rules for top-level expressions and expressions inside classes, which simplifies the meta-theory.

When type checking a class, the rule T-CLASS checks that the body fields and methods are well-typed, and the associated effects and potentials are computed. The effects and potentials associated with a field are the effects and potentials of its initializer (the right-hand-side expression). The effects and potentials associated with a method are the effects and potentials of the body expression of the method. The effect summaries are used during the second phase in T-CHECK, which checks that given the already initialized fields, the effects on the right-hand-side of each field are allowed.

The typing rule T-FIELD checks the right-hand-side expression e in an empty typing environment, as there are no variables in a class body (class parameters are fields of their defining class). In the typing rule T-METHOD, the method parameters $\overline{x : D}$ are used as the typing environment to check the method body. They correspond to the semantics for field initialization and method calls respectively.

6.4.5 Effect Checking

The effect checking judgment $\mathcal{E}; C^{\Omega} \vdash \Phi$ (Figure 6.4) means that the effects Φ are permitted inside class C when the fields in Ω are initialized. It first checks that there is no promotion of *this* in the closure of the effects, as the underlying object is not transitively initialized, the promotion thus is illegal. Then it checks that each accessed field is in the set Ω, i.e., only initialized fields are used.

The closure of effects and potentials is presented in a declarative style for clarity, but it has a straight-forward algorithmic interpretation: it just propagates the effects or potentials recursively until a fixed-point is reached. The fixed-point always exists as the domain of effects and potentials is finite for any given program.

The main step in fixed-point computation is the propagation of effects and potentials. In effect propagation $\mathcal{E} \vdash \phi \rightsquigarrow \Phi$, field access $\beta.f!$ is an atomic effect, thus it propagates to the empty set. For a promotion effect $\pi\uparrow$, we first propagate the potential π to a set of potentials Π, and then promote each potential in Π. For a method call effect $C.this.m\Diamond$, it looks up the effects associated with the method from the effect table.

In potential propagation $\mathcal{E} \vdash \pi \rightsquigarrow \Pi$, root potentials $C.this$ propagate to the empty set, as they do not contain *proxy* aliasing information in the effect table. For a field potential like $C.this.f$, it just looks up the potentials associated with the field f from the effect table. For a method potential $C.this.m$, it looks up the potentials associated with the method m from the effect table.

6.5 Discussions

6.5.1 Why restrict the length of potentials?

The reader might ask, why restrict the length of potentials, thus effects? Let us look at the following program:

Propagate Potentials $\boxed{\mathcal{E} \vdash \pi \rightsquigarrow \Pi}$

$$\mathcal{E} \vdash \beta \rightsquigarrow \emptyset$$

$$\frac{(\Delta, \_) = \mathcal{E}(C) \qquad (\_, \Pi) = \Delta(f)}{\mathcal{E} \vdash C.this.f \rightsquigarrow \Pi} \qquad \frac{\mathcal{E} \vdash C.this.f \rightsquigarrow \Pi \qquad \Pi' = [C.this \mapsto warm[C]]\Pi}{\mathcal{E} \vdash warm[C].f \rightsquigarrow \Pi'}$$

$$\frac{(\_, \mathcal{S}) = \mathcal{E}(C) \qquad (\_, \Pi) = \mathcal{S}(m)}{\mathcal{E} \vdash C.this.m \rightsquigarrow \Pi} \qquad \frac{\mathcal{E} \vdash C.this.m \rightsquigarrow \Pi \qquad \Pi' = [C.this \mapsto warm[C]]\Pi}{\mathcal{E} \vdash warm[C].m \rightsquigarrow \Pi'}$$

Propagate Effects $\boxed{\mathcal{E} \vdash \phi \rightsquigarrow \Phi}$

$$\mathcal{E} \vdash \beta.f! \rightsquigarrow \emptyset$$

$$\frac{(\_, \mathcal{S}) = \mathcal{E}(C) \qquad (\Phi, \_) = \mathcal{S}(m)}{\mathcal{E} \vdash C.this.m\Diamond \rightsquigarrow \Phi}$$

$$\frac{\mathcal{E} \vdash \pi \rightsquigarrow \Pi}{\mathcal{E} \vdash \pi\uparrow \rightsquigarrow \Pi\uparrow} \qquad \frac{\mathcal{E} \vdash C.this.m\Diamond \rightsquigarrow \Phi \qquad \Phi' = [C.this \mapsto warm[C]]\Phi}{\mathcal{E} \vdash warm[C].m\Diamond \rightsquigarrow \Phi'}$$

Closure

$$\frac{\Phi \subseteq \Phi' \qquad \forall \phi \in \Phi'. \mathcal{E} \vdash \phi \rightsquigarrow \Phi'' \Longrightarrow \Phi'' \subseteq \Phi'}{\Phi^c = \Phi'}$$

$$\frac{\Pi \subseteq \Pi' \qquad \forall \pi \in \Pi'. \mathcal{E} \vdash \pi \rightsquigarrow \Pi'' \Longrightarrow \Pi'' \subseteq \Pi'}{\Pi^c = \Pi'}$$

Check $\boxed{\mathcal{E}; \Omega; C \vdash \Phi}$

$$\frac{\beta\uparrow \notin \Phi^c \qquad \forall C.this.f! \in \Phi^c.\ f \in \Omega}{\mathcal{E}; C^\Omega \vdash \Phi}$$

Figure 6.4 – Effect Checking

```
1 class A {
2   var a: A = this.g
3   def g: A = this.g.g
4 }
```

Suppose we do not have the restriction. When checking the effect $A.this.g\Diamond$, the expansion would encounter the effect $A.this.g.g\Diamond$. To check such an effect, the natural approach is to first expand the prefix potential $A.this.g$, and then call the method g on the expanded potential. The potential $A.this.g$ always expands to $A.this.g.g$ in one step. As a result, the process leads to the

following infinite sequence:

$$\begin{array}{rl} & A.this.g\Diamond \\ \Longrightarrow & A.this.g.g\Diamond \\ \Longrightarrow & A.this.g.g.g\Diamond \\ \Longrightarrow & A.this.g.g.g.g\Diamond \\ \Longrightarrow & \ldots \end{array}$$

The example shows that non-termination may happen with simple programs if the abstract domain is infinite. We have to restrict the length of potentials to terminate the checking process.

6.5.2 Why the cold annotation?

The reader might be wondering, can we do better without the `cold` annotation on class parameters? For example, with the *cold* abstraction, the following code does not type check:

```
1 class C {
2   val name = "c"
3   val d : D = new D(this)
4 }
5
6 class D(c: C @cold) {
7   println(c.name)          // error: parent is cold
8 }
```

A natural idea is, instead of saying that `new D(this)` has the potential `warm[D]`, can we invent a potential like `warm[D, c=C.this]`? The idea looks promising. However, to follow the path, we will also need the object construction effect `warm[D, c=C.this].init`, to denote the possible effects in the constructor of the class `D`. Now effect checking for the class `C` works as follows:

$$\begin{array}{rl} & warm[D, c = C.this].init \\ \Longrightarrow & warm[D, c = C.this].c.name! \\ \Longrightarrow & C.this.name! \end{array}$$

As the field `name` is already initialized at the point, the code above will be accepted.

However, naive extension of the system will make effect checking non-terminating, as the following example shows:

```
1 class C(c: C @cold) { val c2 = new C(this) }
```

The non-termination can be seen from the expansion:

$$\begin{array}{rl} & warm[C, c = C.this].init \\ \Longrightarrow & warm[C, c = warm[C, c = C.this]].init \\ \Longrightarrow & warm[C, c = warm[C, c = warm[C, c = C.this]]].init \\ \Longrightarrow & \ldots \end{array}$$

In the above, we have an infinite sequence of effects of the form $\pi_i.init$, where $\pi_0 =$

$warm[C, c = C.this]$ and $\pi_i = warm[C, c = \pi_{i-1}]$.

We have to resort to a standard technique in abstract interpretation, widening [34]. As any potential can be regarded as cold, it suffices to widen a dependent potential to $cold$ if it exceeds some size limit, e.g.:

$$\begin{aligned} & warm[C, c = C.this].init \\ \Longrightarrow\ & warm[C, c = warm[C, c = C.this]].init \\ \Longrightarrow\ & warm[C, c = warm[C, c = cold]].init \\ \Longrightarrow\ & warm[C, c = warm[C, c = cold]].init \end{aligned}$$

This guarantees that the expansion of effects always reaches a fixed point. We will take advantage of this insight in the implementation.

6.6 Extension: Functions

Nowadays most languages combine object-oriented programming with functional programming, such as Java, Scala, Swift. To support functions, we add a new potential $\lambda(\Phi, \Pi)$, where Φ is the set of effects to be triggered when the function is called, while Π is the set of potentials for the result of the function call. The effect domain is still finite, as the set of function potentials is constrained by the number of function literals in a given program.

The addition improves expressiveness. For example, it enables the following code, which is rejected in Swift:

```
class Rec {
  val even = (n: Int) => n == 0 || odd(n - 1)
  val odd = (n: Int) => n == 1 || even(n - 1)
  val flag: Boolean  = odd(6)
}
```

In functional programming, the recursive binding construct $letrec$ may introduce similar initialization patterns as the code above. With the latest checker [3], OCaml still does not support the code below in the construct $let\ rec$.

```
let rec even n = n = 0 || odd (n - 1)
    and odd n = n = 1 || even (n - 1)
    and flag = odd 3
```

The OCaml compiler (version 4.06.0) complains that:

```
File "./code/letrect.ml", line 9, characters 13-18:
Error: This kind of expression is not allowed as right-hand side of ‘let rec’
```

6.7 Related Work

Lucassen and Gifford first introduced type-and-effect systems [36]. To the best of our knowledge, we are the first to introduce the concept of *potentials* to control aliasing in type-and-effect systems.

Qi and Myers [16] introduce a flow-sensitive type-and-effect system for initialization based on masked types. In the system, methods and constructors have effects, which are essentially the mappings of initialization status of `this` before and after the call. Their system does not introduce a concept like *potentials* to track aliasing. The system suffers from the problem of typestate polymorphism, i.e. it lacks parametric polymorphism over masks to support simple use cases. The system depends on unspecified type-and-effect inference to cut down its syntactic verbosity.

Summers and Müller [12] show that initialization of cyclic data structures can be supported in a light-weight, flow-insensitive type system with three modifiers (*free*, *committed* and *unclassified*). To call a method on `this`, annotations are required on the method. The system depends on an unspecified dataflow analysis to support usage of already initialized fields.

The language X10 employs an inter-procedural analysis to ensure safe initialization [11], which removes the annotation burden required when calling final or private methods on `this`. However, the analysis algorithm is not presented in the paper, nor studied formally. To call virtual methods on `this`, the annotation `@NoThisAccess` is required on the method definition. They do not propose a way to handle aliasing, nor do they support cyclic data structures.

All the existing work depends on some unspecified inference or analysis to cut down syntactic overhead [16, 12, 11]. We are the first to present a formal inference system on the problem of safe initialization.

The Swift language features a two-phase initialization scheme [1]. The scheme requires that the fields of a class are initialized first, then the super constructor is called to initialize fields of super classes. Before the super constructor call, the usage of `this` is forbidden. After the call, `this` can be used freely, as all fields of the object are initialized. This scheme does not address the need for usage of `this` during the initialization of fields. Meanwhile, it is incompatible with the initialization of cyclic data structures, as in this case an object is not transitively initialized even when all fields are initialized. Moreover, the scheme does not work with diamond-like inheritance structures, such as multiple inheritance or traits.

6.8 Conclusion

In this chapter, we presented a simple type-and-effect system that can significantly cut down the syntactic overhead of the basic model. One novelty of the system is that it controls aliasing based on the abstraction *potentials*. The system performs *two-phase checking* to handle recursive methods, and it resorts to *full-construction analysis* to deal with inheritance, traits, and properties. The system can be easily extended to handle first-class functions.

[1] https://docs.swift.org/swift-book/LanguageGuide/Initialization.html

Chapter 7

Meta-Theory: The Inference System

The logical picture of the facts is the thought.

— Ludwig Wittgenstein, *Tractatus Logico-Philosophicus*

In this chapter, we study the meta-theory of the type-and-effect system.

7.1 Definitions

We take advantage of the definitions and results of the basic model. In particular, we reuse the definition of modes, types and subtyping. For completeness, the definitions are reproduced in Figure 7.1. Recall that a type C^{Ω} is well-formed only if Ω are fields of the class C.

Modes and Types

$$\begin{array}{rcl} \Omega & = & \{\ f_1, f_2, \dots\ \} \\ \mu & = & cold \mid warm \mid hot \mid \Omega \\ T & = & C^{\mu} \end{array}$$

Mode Ordering

$$hot \sqsubseteq \mu \qquad warm \sqsubseteq \Omega \qquad \Omega_1 \cup \Omega_2 \sqsubseteq \Omega_1 \qquad \mu \sqsubseteq cold$$

Subtyping

$$T <: T \qquad \frac{T_1 <: T_2 \qquad T_2 <: T_3}{T_1 <: T_3} \qquad \frac{\mu_1 \sqsubseteq \mu_2}{C^{\mu_1} <: C^{\mu_2}}$$

Figure 7.1 – Subtyping

We also reuse the semantic result of scopability developed in the meta-theory of the basic model, thanks to the fact that the languages share the same semantics.

Store typing $\boxed{\Sigma \vDash \sigma}$

$$\frac{dom(\Sigma) = dom(\sigma) \qquad \forall\, l \in dom(\Sigma).\ \Sigma; l \vDash \sigma(l) : \Sigma(l)}{\Sigma \vDash \sigma}$$

Monotonicity $\boxed{\Sigma_1 \preccurlyeq \Sigma_2}$

$$\frac{\forall\, l \in dom(\Sigma_1).\ \Sigma_2(l) <: \Sigma_1(l)}{\Sigma_1 \preccurlyeq \Sigma_2}$$

Authority $\boxed{\Sigma_1 \rhd \Sigma_2}$

$$\frac{\forall l \in dom(\Sigma_1).\Sigma_1(l) = C^{\Omega} \Longrightarrow \Sigma_2(l) = C^{\Omega}}{\Sigma_1 \rhd \Sigma_2}$$

Stackability $\boxed{\Sigma_1 \ll \Sigma_2}$

$$\frac{\forall l \in dom(\Sigma_2).\Sigma_2 \vDash l : C^{warm} \bigvee l \in dom(\Sigma_1)}{\Sigma_1 \ll \Sigma_2}$$

Value typing $\boxed{\Sigma \vDash l : T}$

$$\frac{\Sigma(l) = T_1 \qquad T_1 <: T_2}{\Sigma \vDash l : T_2}$$

Environment typing $\boxed{\Gamma; \Sigma \vDash \rho}$

$$\emptyset; \Sigma \vDash \emptyset \qquad\qquad \frac{\Gamma; \Sigma \vDash \rho \qquad \Sigma \vDash l : C^{hot}}{\Gamma, x{:}C; \Sigma \vDash \rho, x{:}l}$$

Convenience Definitions

$$\frac{\Sigma \vDash l : D^{\mu}}{\Sigma \vDash l : D} \qquad \frac{\Sigma \vDash l : D^{\mu}}{\Sigma \vDash l : \mu} \qquad \frac{\forall l \in L.\Sigma \vDash l : \mu}{\Sigma \vDash L : \mu}$$

Figure 7.2 – Store typing, environment typing, value typing

For expression soundness, we need also reuse the definition of monotonicity ($\Sigma_1 \preceq \Sigma_2$), authority ($\Sigma_1 \rhd \Sigma_2$), stackability ($\Sigma_1 \ll \Sigma_2$). Value typing stays the same as before, while in environment typing, we require all values to be hot. For completeness, the definitions are reproduced in Figure 7.2.

The semantic typing for objects, presented in Figure 7.3, has changed. The typing judgment

Object Typing $\boxed{\Sigma; l \vDash o : C^{\mu}}$

$$\frac{(C,\omega) = o \qquad classParameters(C) \subseteq dom(\omega) \\ \forall f \in dom(\omega).\ D = fieldType(C,f) \Longrightarrow \Sigma; l \vDash \omega(f) : potential(C,f)}{\Sigma; l \vDash o : C^{dom(\omega)}}$$

$$\frac{(C,\omega) = o \qquad \forall f \in fields(C).\ D = fieldType(C,f) \Longrightarrow \Sigma; l \vDash \omega(f) : potential(C.this, f)}{\Sigma; l \vDash o : C^{warm}}$$

$$\frac{(C,\omega) = o \qquad \forall f \in fields(C).\ D = fieldType(C,f) \Longrightarrow \Sigma \vDash \omega(f) : D^{hot}}{\Sigma; l \vDash o : C^{hot}}$$

Field Potential

$$\begin{array}{lcl} potential(\beta, \tilde{f}) & = & \{\ cold\ \} \\ potential(\beta, \hat{f}) & = & \emptyset \\ potential(\beta, f) & = & \{\ \beta.f\ \} \end{array}$$

Figure 7.3 – Object typing

$\Sigma; l \vDash o : C^{\mu}$ says that the object may take the type C^{μ} given that the object address is l. The address of the object is required to give semantic meaning to potentials, which represent aliasing information. If $\mu = hot$, the typing rule checks that each field can be typed as hot. Otherwise, when $\mu = warm$ or $\mu = \overline{f}$, it checks that each field has the right potential, which represents aliasing of values under initialization.

Note that objects are never typed as C^{cold}, though object references may be typed as C^{cold} by subtyping. It is harmless to add an object typing rule for C^{cold}, as it is done in the meta-theory for the basic model. We prefer simplicity here and thus exclude the rule.

The semantics for potentials and effects are defined in Figure 7.4. Note that both potential typing and effect typing are defined relative to the value ψ for *this* — that is partly expected, as potentials and effects are in essence tracking aliasing and field accesses relative to *this*.

In potential typing $\Sigma; \psi \vDash l : \Pi$, a location l may take any set of potentials Π if l is hot. In particular, Π can be an empty set. Otherwise, there should exist a potential β in $[\![\Pi]\!]^C$, such that $\Sigma; \psi \vDash l : \beta$. There are three cases:

- $\beta = C.this$. In this case, the rule checks that $l = \psi$.
- $\beta = warm[C]$. The rule checks that l can be typed as C^{warm}.
- $\beta = cold$. The rule checks that the value is well-typed.

An immediate result of this definition is that if $\Sigma; \psi \vDash l : \Pi$, then either l is hot or there exists β such that $\Sigma; \psi \vDash l : \beta$. This is exactly what we want: if an expression may take a value that is not transitively initialized, then the potentials of the expression should include the potential β, which means the value may not be fully initialized.

Potential Typing $\boxed{\Sigma;\psi \vDash l : \Pi}$

$$\frac{\Sigma \vDash l : hot}{\Sigma;\psi \vDash l : \Pi} \qquad \frac{\exists \pi \in \Pi^c.\Sigma;\psi \vDash l : \pi}{\Sigma;\psi \vDash l : \Pi}$$

$$\frac{l = \psi}{\Sigma;\psi \vDash l : C.this} \qquad \frac{\Sigma \vDash l : C^{warm}}{\Sigma;\psi \vDash l : warm[C]} \qquad \frac{\Sigma \vDash l : D^{\mu}}{\Sigma;\psi \vDash l : cold}$$

Effect Typing $\boxed{\Sigma;\psi \vDash \Phi}$

$$\frac{\forall \phi \in \Phi^c.\ \Sigma;\psi \vDash \phi}{\Sigma;\psi \vDash \Phi} \qquad \frac{\Sigma \vDash \psi : C^{\overline{f}} \qquad f \in \overline{f}}{\Sigma;\psi \vDash C.this.f!}$$

$$\Sigma;\psi \vDash warm[C].f!$$

$$\frac{\pi = \beta \Longrightarrow \Sigma \vDash \psi : hot}{\Sigma;\psi \vDash \pi\uparrow} \qquad \Sigma;\psi \vDash \beta.m\Diamond$$

Figure 7.4 – Definition of potential typing and effect typing

In an effect typing judgment $\Sigma;\psi \vDash \Phi$, we ensure that all effects Φ are allowed given the value ψ for *this* and the store typing Σ. This is enforced by checking that each effect ϕ in Φ^c, which is the closure of Φ, is allowed. There are several cases depending on ϕ:

- $\phi = \pi\uparrow$: if π is $C.this$, $warm[C]$ or $cold$, then ψ must be hot.
- $\phi = C.this.f!$: f must be an initialized field of ψ.
- $\phi = \beta.m\Diamond$: placeholder effects are ignored.

7.2 Monotonicity Lemmas

We inherit most lemmas from the meta-theory for the basic model. Here we only state new lemmas related to effects and potentials, and lemmas related to object typing, as they are changed.

LEMMA 7.1 (Potential Typing Monotonicity)**.** *For all* $\Sigma, \Sigma', \Pi, \psi, l$*, if* $\Sigma \preccurlyeq \Sigma'$ *and* $\Sigma, \psi \vDash l : \Pi$*, then* $\Sigma', \psi \vDash l : \Pi$.

Proof. By the definition of potential typing there are two cases.

- **case** $\Sigma \vDash l : C^{hot}$.
  By Lemma 5.3, we have $\Sigma' \vDash l : C^{hot}$. Using the definition of potential typing we have $\Sigma';\psi \vDash : \Pi$.
- **case** $\Sigma;\psi \vDash l : \pi$ where $\pi \in \Pi^c$.
  There are three cases depending on the shape of π:
  - **case** $\pi = C.this$
    We have $l = \psi$, using the same rule we get $\Sigma';\psi \vDash l : C.this$. Now use the definition of potential typing.

– **case** $\pi = warm[C]$
We have $\Sigma \vDash l : C^{warm}$. Using Lemma 5.3, we have $\Sigma' \vDash l : C^{warm}$. Now use the definition of potential typing.
– **case** $\pi = cold$
We have $\Sigma \vDash l : D^{\mu}$. Using Lemma 5.3, we have $\Sigma' \vDash l : C^{\mu}$. Now use the definition of potential typing.

□

LEMMA 7.2 (Object Typing Monotonicity). *For all* $\Sigma, \Sigma', T, \Pi, l$ *and object* o*, if* $\Sigma \preccurlyeq \Sigma'$ *and* $\Sigma; l \vDash o : C^{\mu}$*, then* $\Sigma'; l \vDash o : C^{\mu}$.

Proof. By definition of object typing, Lemma 5.3 and Lemma 7.1. □

LEMMA 7.3 (Effect Typing Monotonicity). *For all* Σ, Σ', Φ *and value* ψ*, if* $\Sigma \preccurlyeq \Sigma'$ *and* $\Sigma, \psi \vDash \Phi$*, then* $\Sigma', \psi \vDash \Phi$.

Proof. By the definition of effect typing and Lemma 5.3. □

LEMMA 7.4 (Environment Regularity). *If* $\Gamma; \Sigma \vDash \rho$ *and* $\Gamma; C \vdash x{:}D$*, then* $\Sigma \vDash \rho(x) : D^{hot}$.

Proof. By induction on the typing derivation $\Gamma; \Sigma \vDash \rho$. □

7.3 Closure Lemmas

The following lemmas are about the properties of the closure of effects and potentials.

LEMMA 7.5 (Closure Distribution). *The closure operation is distributive:*
- $(\Pi_1 \cup \cdots \cup \Pi_n)^c = \Pi_1^c \cup \cdots \cup \Pi_n^c$
- $(\Phi_1 \cup \cdots \cup \Phi_n)^c = \Phi_1^c \cup \cdots \cup \Phi_n^c$

Proof. Let us consider the case for potentials, the case for effects is similar. First, it is obvious that $\Pi_1 \cup \cdots \cup \Pi_n \subseteq \Pi_1^c \cup \cdots \cup \Pi_n^c$ due to that $\Pi_1 \subseteq \Pi_1^c$ and so on. Second, for any $\pi \in \Pi_1^c \cup \cdots \cup \Pi_n^c$, there must exist i such that $\pi \in \Pi_i^c$. Now by the definition of Π_i^c, if $\mathcal{E} \vdash \pi \rightsquigarrow \Pi'$, then $\Pi' \subseteq \Pi_i^c$. Therefore, $\Pi' \subseteq \Pi_1^c \cup \cdots \cup \Pi_n^c$. The conclusion follows by definition. □

LEMMA 7.6 (Closure Extensivity). *The closure operation is extensive:*
- $\Pi \subseteq \Pi^c$
- $\Phi \subseteq \Phi^c$

Proof. Immediate from the definition. □

LEMMA 7.7 (Closure Monotonicity). *The closure operation is monotone:*
- $\Pi_1 \subseteq \Pi_2 \Longrightarrow \Pi_1^c \subseteq \Pi_2^c$
- $\Phi_1 \subseteq \Phi_2 \Longrightarrow \Phi_1^c \subseteq \Phi_2^c$

Proof. Immediate from the Lemma 7.5. □

LEMMA 7.8 (Closure Idempotency). *The closure operation is idempotent:*

- $\Pi^c = (\Pi^c)^c$
- $\Phi^c = (\Phi^c)^c$

Proof. Immediate from the definition. □

LEMMA 7.9 (Closure Prefix Constancy). *If* $D.this \in \{\ C.this.f\ \}^c$, *then* $D = C$.

Proof. By the definition of field typing, method typing and expression typing, we know if any potential for fields or methods of a class C has the form $D.this$, then we must have $D = C$. Based on the fact, now it suffices to perform induction on the potential expansion rules to prove the following lemma:

$$prefix(\Pi^c) \subseteq prefix(\Pi)$$

where $prefix$ is defined as follows:

$$\begin{array}{lcl} prefix(\{\ C.this\ \} \cup \Pi) & = & \{\ C.this\ \} \cup prefix(\Pi) \\ prefix(\{\ C.this.f\ \} \cup \Pi) & = & \{\ C.this\ \} \cup prefix(\Pi) \\ prefix(\{\ C.this.m\ \} \cup \Pi) & = & \{\ C.this\ \} \cup prefix(\Pi) \\ prefix(\{\ \pi\ \} \cup \Pi) & = & prefix(\Pi) \end{array}$$

The result immediately follows from the lemma above. □

LEMMA 7.10 (Prefix Substitution). *The following propositions hold*

- $([C.this \mapsto warm[C]]\Pi)^c = [C.this \mapsto warm[C]](\Pi^c)$
- $([C.this \mapsto warm[C]]\Phi)^c = [C.this \mapsto warm[C]](\Phi^c)$

Proof. By induction on the size of Π and case analysis on potential expansion rules. □

LEMMA 7.11 (Closure Expansion Regularity). *The following propositions hold*

1. *If* $C.this \in \{C.this.f\}^c$, *then* $warm[C] \in \{warm[C].f\}^c$.
2. *If* $warm[D] \in \{C.this.f\}^c$, *then* $warm[D] \in \{warm[C].f\}^c$.
3. *If* $C.this \in \{C.this.m\}^c$, *then* $warm[C] \in \{warm[C].m\}^c$.
4. *If* $warm[D] \in \{C.this.m\}^c$, *then* $warm[D] \in \{warm[C].m\}^c$.

Proof. Immediate from Lemma 7.10. □

7.4 Potential Lemmas

LEMMA 7.12 (Potential Typing Regularity). *If* $\Sigma; \psi \vDash l : \Pi$, *then* $\Sigma \vDash l : hot$ *or* $\beta \in \Pi^c$.

Proof. By the definition of potential typing, there are two cases. If $\Sigma \vDash l : hot$, which completes the proof goal immediately. Otherwise, there exist $\pi \in \Pi^c$ such that $\Sigma; \psi \vDash l : \pi$. There are three cases, for each case we have $\pi = \beta$. □

LEMMA 7.13 (Empty Potential Regularity). *If* $\Sigma; \psi \vDash l : \emptyset$, *then* $\Sigma \vDash l : hot$.

Proof. By definition of potential typing, the only possibility is $\Sigma \vDash l : hot$. □

LEMMA 7.14 (Potential Weakening). *If* $\Sigma; \psi \vDash l : \Pi$ *and* $\Pi \subseteq \Pi_0$, *then* $\Sigma; \psi \vDash l : \Pi_0$.

Proof. From the definition of potential typing, there are two cases: either $\Sigma \vDash l : hot$ or $\exists \pi \in \Pi^c.\Sigma; \psi \vDash l : \pi$. In the first case, applying the definition of potential typing, we get the result immediately. We only need to consider the second case, where $\Sigma; \psi \vDash l : \pi$.

From $\pi \in \Pi^c$, $\Pi \subseteq \Pi_0$ and Lemma 7.7, we have $\pi \in \Pi_0^c$. The result follows immediately from the fact that $\Sigma; \psi \vDash l : \pi$ and $\pi \in \Pi_0^c$. □

LEMMA 7.15 (Potential Strengthening - Closure). *If* $\Sigma; \psi \vDash l : \Pi^c$, *then* $\Sigma; \psi \vDash l : \Pi$.

Proof. From the definition of potential typing, there are two cases. In the first case, we know l is hot, the conclusion follows trivially. In the second case, the conclusion follows from the fact that $(\Pi^c)^c = \Pi^c$. □

LEMMA 7.16 (Potential View Change - Field). *If* $\Sigma; l_0 \vDash l : \{C.this.f\}$ *and* $\Sigma; \psi \vDash l_0 : warm[C]$, *then* $\Sigma; \psi \vDash l : \{warm[C].f\}$.

Proof. From the definition of potential typing, there are two cases: either $\Sigma \vDash l : D^{hot}$ or there exists π in the closure $\{\ C.this.f\ \}^c$ such that $\Sigma; \psi \vDash l : \pi$. In the first case, applying the definition of potential typing, we get the result immediately. We only need to consider the second case. There are following possibilities for π:

- $\pi = C.this$
  We have $l = l_0$ by the definition of potential typing. As $C.this \in \{C.this.f\}^c$, we must have $warm[C] \in \{warm[C].f\}^c$ from Lemma 7.11. The result follows from $\Sigma; \psi \vDash l : warm[C]$.

- $\pi = warm[E]$
  We have $\Sigma \vDash l : E^{warm}$. As $warm[E] \in \{C.this.f\}^c$, we must have $warm[E] \in \{warm[C].f\}^c$ from Lemma 7.11. The result follows from $\Sigma; \psi \vDash l : warm[E]$.

- $\pi = cold$
  We have $\Sigma \vDash l : D^{\mu}$. The result follows from the potential typing for *cold*.

□

LEMMA 7.17 (Potential View Change - Method). *If* $\Sigma; l_0 \vDash l : \{C.this.m\}$ *and* $\Sigma; \psi \vDash l_0 : warm[C]$, *then* $\Sigma; \psi \vDash l : \{warm[C].m\}$.

Proof. Similar as the Lemma 7.16 above. □

7.5 Effect Lemmas

LEMMA 7.18 (Effect Weakening). *If* $\Sigma; \psi \vDash \Phi$ *and* $\Phi_0 \subseteq \Phi$, *then* $\Sigma; \psi \vDash \Phi_0$.

Proof. From the definition of effect typing, we know $\forall \phi \in \Phi^c.\Sigma; \psi \vDash \phi$. From $\Phi_0 \subseteq \Phi$, we have $\Phi_0^c \subseteq \Phi^c$ from Lemma 7.7. Thus it follows that $\forall \phi \in \Phi_0^c.\Sigma; \psi \vDash \phi$. The result follows immediately from the definition of effect typing. □

LEMMA 7.19 (Effect Strengthening - Closure)**.** *If* $\Sigma; \psi \vDash \Phi$*, then* $\Sigma; \psi \vDash \Phi^c$.

Proof. From the definition of effect typing, we know $\forall \phi \in \Phi^c . \Sigma; \psi \vDash \phi$. The result follows immediately from the fact that $(\Phi^c)^c = \Phi^c$. □

LEMMA 7.20 (Hot Effect Regularity)**.** *If* $\Sigma \vDash \psi : C^{hot}$*, then* $\Sigma; \psi \vDash \Phi$.

Proof. By the definition of effect typing. □

LEMMA 7.21 (Field Check Regularity)**.** *If* $\mathcal{E}; C^{\Omega} \vdash \Phi$ *and* $\Sigma \vDash \psi : C^{\Omega}$*, then* $\Sigma; \psi \vDash \Phi$.

Proof. We need to prove that for all $\phi \in \Phi^c$, we have $\Sigma; \psi \vDash \phi$. If $\Phi^c = \emptyset$, then the conclusion holds trivially by the definition of effect typing.

The only cases that do not hold trivially are when $\phi = \pi\uparrow$ or $\phi = C.this.f!$. In the first case, from the definition of $\mathcal{E}; C^{\Omega} \vdash \Phi$, we know $\beta\uparrow \notin \Phi^c$. Thus, $\pi \neq \beta$, $\Sigma; \psi \vDash \pi\uparrow$ holds trivially. In the second case $\phi = C.this.f!$, and from effect checking we have $f \in \Omega$. Now use the definition of effect typing, the result follows immediately. □

LEMMA 7.22 (Effect View Change - Method)**.** *If* $\Sigma; \psi \vDash warm[C].m\Diamond$*, and* $\Sigma \nvDash \psi : hot$*, and* $\Sigma; \psi \vDash l_0 : warm[C]$*, then* $\Sigma; l_0 \vDash C.this.m\Diamond$.

Proof. For any $\phi \in \{ C.this.m\Diamond \}^c$, . We perform case analysis on ϕ.

- **case** $\phi = C.this.f!$
  We have $\Sigma \vDash l_0 : warm$, thus $\Sigma; l_0 \vDash C.this.f!$ holds trivially.

- **case** $\phi = warm[E].f!$
  $\Sigma; l_0 \vDash \phi$ holds trivially by definition.

- **case** $\phi = \pi\uparrow$
  If $\pi \neq \beta$, $\Sigma; l_0 \vDash \phi$ holds trivially. Otherwise, $[C.this \mapsto warm[C]]\beta\uparrow \in \{ warm[C].m\Diamond \}^c$ by Lemma 7.10. We have $\Sigma; \psi \vDash [warm[C] \mapsto C.this]\beta\uparrow$, which requires $\Sigma \vDash \psi : hot$, a contradiction with the premise that ψ is not hot.

- **case** $\phi = \beta.m\Diamond$
  $\Sigma; l_0 \vDash \pi$ holds trivially by definition.

□

7.6 Local Reasoning

LEMMA 7.23 (Effect Potential Cancellation)**.** *Given*

- $\Sigma; \psi \vDash \Pi\uparrow$
- $\Sigma'; \psi \vDash l : \Pi$
- $\Sigma \vDash \psi : hot \implies \Sigma' \vDash l : hot$

Then we have

- $\Sigma' \vDash l : hot$

Proof. From Lemma 7.12, we have either $\beta \in \Pi^c$ or $\Sigma'; \psi \vDash l : hot$. The case l is hot is trivial, we only need to consider $\beta \in \Pi^c$. From the definition of $\Sigma; \psi \vDash \Pi\uparrow$, we know $\Sigma; \psi \vDash \beta\uparrow$. From the definition of effect typing, we have $\Sigma \vDash \psi : hot$. Now using the premise, we get $\Sigma' \vDash l : hot$. □

7.7 Selection Lemmas

LEMMA 7.24 (Hot Selection). *If* $\Sigma \vDash l : C^{hot}$, $\Sigma \vDash \sigma$, *and* $fieldType(C, f) = D$, *then for all* ψ *and* Π *we have* $\Sigma; \psi \vDash field(\sigma, l, f) : \Pi$.

Proof. By the definition of object typing, we have $\Sigma; \psi \vDash field(\sigma, l, f) : hot$. Now using the definition of potential typing, the conclusion follows immediately. □

LEMMA 7.25 (This Selection). *If* $\Sigma; \psi \vDash l : C.this$, *and* $\Sigma \vDash \sigma$, *and* $potential(C.this, f) \subseteq \Pi$, *and* $fieldType(C, f) = D$, *then we have* $\Sigma; \psi \vDash field(\sigma, l, f) : \Pi$.

Proof. By the definition of object typing, we have $\Sigma; \psi \vDash field(\sigma, l, f) : potential(C.this, f)$. Now using Lemma 7.14, the conclusion follows immediately. □

LEMMA 7.26 (Warm Selection). *If* $\Sigma; \psi \vDash l : warm[C]$, *and* $\Sigma \vDash \sigma$, *and* $potential(warm[C], f) \subseteq \Pi$, *and* $fieldType(C, f) = D$, *then we have* $\Sigma; \psi \vDash field(\sigma, l, f) : \Pi$.

Proof. By the definition of object typing, we have $\Sigma; l \vDash field(\sigma, l, f) : potential(C.this, f)$. There are three cases for $potential(C.this, f)$:

- **Case** $potential(C.this, f) = \emptyset$
  In this case we know $field(\sigma, l, f)$ is hot, thus the result holds trivially.

- **Case** $potential(C.this, f) = \{\ cold\ \}$
  The result holds by Lemma 7.14.

- **Case** $potential(C.this, f) = \{\ C.this.f\ \}$
  By definition of $potential$, we have
  - (A1) $potential(warm[C], f) = \{\ warm[C].f\ \}$

  Using Lemma 7.16, we have the following
  - (B1) $\Sigma; \psi \vDash field(\sigma, l, f) : \{\ warm[C].f\ \}$

  Now using Lemma 7.14, the conclusion follows immediately.

□

LEMMA 7.27 (Potential Selection). *If* $\beta \neq cold$, $\beta \in \Pi_0^c$ *and* $(\_, \Pi_1) = select(\Pi_0, f)$, *then we have* $potential(\beta, f) \subseteq \Pi_1^c$.

Proof. By induction on the size of Π_0 and case analysis on the definition of $select$. □

LEMMA 7.28 (Cold Selection). *If* $cold \in \Pi^c$ *and* $(\Phi, \_) = select(\Pi, f)$, *then we have* $cold\uparrow \in \Phi^c$.

Proof. By induction on the size of Π and case analysis on the definition of $select$. □

7.8 Method Call Lemmas

LEMMA 7.29 (Potential Invoke)**.** *If* $\beta \neq cold$, $\beta \in \Pi_0^c$ *and* $(\_, \Pi_1) = call(\Pi_0, m)$, *then we have* $\beta.m \subseteq \Pi_1^c$.

Proof. By induction on the size of Π_0 and case analysis on the definition of $call$. □

LEMMA 7.30 (Cold Invoke)**.** *If* $cold \in \Pi^c$ *and* $(\Phi, \_) = call(\Pi, m)$, *then we have* $cold{\uparrow} \in \Phi^c$.

Proof. By induction on the size of Π and case analysis on the definition of $call$. □

7.9 Expression Soundness

With the definitions above, we will be able to prove the following lemma:

LEMMA 7.31 (Expression Soundness)**.** *Given*

(1) $\Gamma; C \vdash e : D \,!\, (\Phi, \Pi)$
(2) $\Gamma; \Sigma \vDash \rho$
(3) $\Sigma \vDash \sigma$
(4) $\Sigma \vDash \psi : C$
(5) $\Sigma; \psi \vDash \Phi$
(6) $[\![e]\!](\rho, \sigma, \psi)(k) = Some\ result$

then there exist l, σ', Σ' *such that:*

(a) $result = Some\ (l, \sigma')$
(b) $\Sigma \preccurlyeq \Sigma', \Sigma \ll \Sigma', \Sigma \rhd \Sigma'$ *and* $\Sigma' \vDash \sigma'$
(c) $\Sigma' \vDash l : D$
(d) $\Sigma'; \psi \vDash l : \Pi$
(e) $\Sigma \vDash \psi : hot \Longrightarrow \Sigma' \vDash l : hot$

Proof. The proof follows the same structure as the expression soundness for the basic model. The proof for $\Sigma \ll \Sigma'$, $\Sigma \rhd \Sigma'$ and $\Sigma' \vDash l : D$ are almost the same, thus we omit the details. The result $\Sigma \vDash \psi : hot \Longrightarrow \Sigma' \vDash l : hot$ follows from Lemma 5.15 and Theorem 3.5. It is not strictly necessary, but it simplifies the usage of the Lemma 7.23.

We focus on how effects and potentials are handled in the proof.

- **case T-VAR.** $e = x$
  From the typing rule T-VAR, we have the following:
  - (A1) $\Phi = \emptyset$
  - (A2) $\Pi = \emptyset$

  Choose $l = \rho(x), \sigma' = \sigma, \Sigma' = \Sigma$. From Lemma 7.4, we have
  - (B1) $\Sigma \vDash l : hot$

  Now using the definition of potential typing, we have $\Sigma; \psi \vDash l : \emptyset$.

- **case T-THIS.** $e = this$
  From the typing rule T-THIS, we have the following:
  - (A1) $\Phi = \emptyset$
  - (A2) $\Pi = \{C.this\}$
  - (A3) $D = C$

  Choose $l = \psi, \sigma' = \sigma, \Sigma' = \Sigma$. The result $\Sigma'; \psi \vDash \psi : \{\ C.this\ \}$ holds by definition of potential typing.

- **case T-SEL.** $e = e_0.f$
  From the typing rule T-SEL, we have the following:
  - (A1) $\Sigma; C \vdash e_0 : E\ !\ (\Phi_0, \Pi_0)$
  - (A2) $D = fieldType(E, f)$
  - (A3) $(\Phi', \Pi) = select(\Pi_0, f)$
  - (A4) $\Phi = \Phi_0 \cup \Phi'$

  From the induction hypothesis on e_0, we know there exist l_0, σ' and Σ' such that:
  - (B1) $\Sigma \preccurlyeq \Sigma'$, $\Sigma \ll \Sigma'$, $\Sigma \rhd \Sigma'$ and $\Sigma' \vDash \sigma'$
  - (B2) $\Sigma' \vDash l_0 : E$
  - (B3) $\Sigma'; \psi \vDash l_0 : \Pi_0$
  - (B4) $\Sigma \vDash \psi : hot \Longrightarrow \Sigma' \vDash l_0 : hot$
  - (B5) $(E, \omega) = \sigma'(l_0)$

  Now case analysis on the typing derivation for (B3):
  - **case** $\Sigma' \vDash l_0 : hot$
    Choose $\omega(f)$, σ' and Σ'. We have $\Sigma' \vDash \omega(f) : \Pi'$ because $\omega(f)$ is hot.

  - **case** $\exists \pi \in \Pi_0^c.\Sigma'; \psi \vDash l_0 : \pi$
    There are three cases:
    - **case** $\pi = E.this$
      By definition of potential typing and object typing, we have
      - (a1) $l_0 = \psi$
      - (a2) $E = C$
      - (a3) $\forall f \in dom(\omega).\ D = fieldType(C, f) \Longrightarrow \Sigma'; \psi \vDash \omega(f) : \{C.this.f\}$

      From (A3) and Lemma 7.27, we have
      - (b1) $potential(C.this, f) \subseteq \Pi^c$

      Now using the Lemma 7.25 and Lemma 7.15, we have
      - (b1) $\Sigma'; \psi \vDash \omega(f) : \Pi$

    - **case** $\pi = warm[E]$

Similar as the case above with the Lemma 7.26.

* **case** $\pi = cold$
  From Lemma 7.28, we have
  · (a1) $cold\uparrow \in \Phi'^{c}$

  From monotonicity of effect typing, Lemma 7.19 and Lemma 7.18, we have
  · (b1) $\Sigma; \psi \vDash cold\uparrow$

  From the definition of effect typing, we know
  · (c1) $\Sigma \vDash \psi : hot$

  From (B4), we know l_0 is hot. Now use Lemma 5.16.

- **case T-CALL.** $e = e_0.m(\overline{e})$
  From the typing rule T-CALL, we have the following:
  – (A1) $\Sigma; C \vdash e_0 : E_0 \,!\, (\Phi_0, \Pi_0)$
  – (A2) $\overline{\Sigma; C \vdash e_i : E_i \,!\, (\Phi_i, \Pi_i)}$
  – (A3) $(\overline{x_i{:}E_i}, D) = methodType(E_0, m)$
  – (A4) $(\Phi', \Pi) = call(E_0.m, \Pi_0)$
  – (A5) $\Phi = \Phi_0 \cup \overline{\Phi_i} \cup \overline{\Pi_i\uparrow} \cup \Phi'$

  From the induction hypothesis on e_0, we know there exist l_0, σ_0, Σ_0 such that:
  – (B1) $\Sigma \preccurlyeq \Sigma_0$, $\Sigma \ll \Sigma_0$, $\Sigma \rhd \Sigma_0$ and $\Sigma_0 \vDash \sigma_0$
  – (B2) $\Sigma_0 \vDash l_0 : E_0$
  – (B3) $\Sigma_0; \psi \vDash l_0 : \Pi_0$
  – (B4) $\Sigma \vDash \psi : hot \Longrightarrow \Sigma_0 \vDash l_0 : hot$

  In order to use induction hypothesis on the arguments, we need to check that the preconditions hold. It follows trivially from monotonicity, thus we have:
  – (C1) $\Gamma; \Sigma_0 \vDash \rho$
  – (C2) $\Sigma_0 \vDash \psi : C$
  – (C3) $\Sigma_0; \psi \vDash \Phi$

  By using induction hypothesis and monotonicity repeatedly on all arguments, we get l_i, σ_i and Σ_i such that
  – (D1) $\Sigma_{i-1} \preccurlyeq \Sigma_i$, $\Sigma_{i-1} \ll \Sigma_i$, $\Sigma_{i-1} \rhd \Sigma_i$ and $\Sigma_i \vDash \sigma_i$
  – (D2) $\Sigma_i \vDash l_i : E_i$
  – (D3) $\Sigma_i; \psi \vDash l_i : \Pi_i$
  – (D4) $\Sigma_{i-1} \vDash \psi : hot \Longrightarrow \Sigma_i \vDash l_i : hot$

  From monotonicity of effect typing, we have:

– (E1) $\Sigma_i; \psi \vDash \Pi_i\uparrow$

Let $\Sigma' = \Sigma_n$, using Lemma 7.23 and Lemma 5.3, we have
– (F1) $\Sigma' \vDash l_i : hot$

The method $E_0.m$ is well typed, thus we have
– (G1) $\overline{x_i{:}E_i}; E_0 \vDash e_m : D \,!\, (\Phi_m, \Pi_m)$

We prepare the environment ρ' and ψ' for the method call as follows:
– (H1) $\rho' = \overline{x_i{:}l_i}$
– (H2) $\psi' = l_0$

Now do case analysis on the potential derivation for (B3):
– **case** $\Sigma_0 \vDash l_0 : hot$
  As both the receiver and method parameters are hot, we have:
  * (a1) $\Sigma'; l_0 \vDash \Phi_m$ ▷ Lemma 7.20

  Now we can use the induction hypothesis for (G1):
  * (b1) $\Sigma' \preccurlyeq \Sigma_m$, $\Sigma' \ll \Sigma_m$, $\Sigma' \rhd \Sigma_m$ and $\Sigma_m \vDash \sigma_m$
  * (b2) $\Sigma_m \vDash l_m : D$
  * (b3) $\Sigma_m; l_0 \vDash l_m : \Pi_m$
  * (b4) $\Sigma' \vDash l_0 : hot \Longrightarrow \Sigma_m \vDash l_m : hot$

  Choose l_m, σ_m and Σ_m. As l_m is hot, it may take any potential by the definition of potential typing.

– **case** $\exists \pi \in \Pi_0^c.\Sigma_0 \vDash l_0 : \pi$
  We only consider $\Sigma_0 \nvDash l_0 : hot$ in this case. Otherwise, it suffices to follow the case above. There are three cases for π:
  * **case** $\pi = C.this$
    By definition of potential typing for l_0, we have
    · (a1) $E_0 = C$
    · (b1) $l_0 = \psi$

    From $\Phi_m \subseteq \{\ C.this.m\Diamond\ \}^c \subseteq \Phi'^c \subseteq \Phi^c$, Lemma 7.18, and Lemma 7.3, and Lemma 7.19, we know
    · (b1) $\Sigma'; \psi \vDash \Phi_m$

    Now we can use the induction hypothesis for (G1):
    · (c1) $\Sigma' \preccurlyeq \Sigma_m$, $\Sigma' \ll \Sigma_m$, $\Sigma' \rhd \Sigma_m$ and $\Sigma_m \vDash \sigma_m$
    · (c2) $\Sigma_m \vDash l_m : D$
    · (c3) $\Sigma_m; \psi \vDash l_m : \Pi_m$

· (c4) $\Sigma' \vDash \psi : hot \Longrightarrow \Sigma_m \vDash l_m : hot$

From Lemma 7.29 , we have
· (d1) $C.this.m \in \Pi^c$

From (c3), $\Pi_m \subseteq \{\ C.this.m\ \}^c \subseteq \Pi^c$, and Lemma 7.14, and Lemma 7.15, we have
· (e3) $\Sigma_m; \psi \vDash l_m : \Pi$

∗ **case** $\pi = warm[E_0]$
From $warm[C].m\Diamond \in \Phi'^c \subseteq \Phi^c$, Lemma 7.18, and Lemma 7.3, and Lemma 7.19, we know
· (a1) $\Sigma'; \psi \vDash warm[C].m\Diamond$

Now use Lemma 7.22:
· (b1) $\Sigma'; l_0 \vDash C.this.m\Diamond$

From $\Phi_m \subseteq \{\ C.this.m\Diamond\ \}^c$ and Lemma 7.18, and Lemma 7.19, we know
· (c1) $\Sigma'; l_0 \vDash \Phi_m$

Now we can use the induction hypothesis for (G1):
· (d1) $\Sigma' \preccurlyeq \Sigma_m$, $\Sigma' \ll \Sigma_m$, $\Sigma' \rhd \Sigma_m$ and $\Sigma_m \vDash \sigma_m$
· (d2) $\Sigma_m \vDash l_m : D$
· (d3) $\Sigma_m; l_0 \vDash l_m : \Pi_m$
· (d4) $\Sigma' \vDash l_0 : hot \Longrightarrow \Sigma_m \vDash l_m : hot$

From (d3) and $\Pi_m \subseteq \{\ C.this.m\ \}^c$ and Lemma 7.14 and Lemma 7.15, we have
· (e1) $\Sigma_m; l_0 \vDash l_m : \{\ C.this.m\ \}$

From Lemma 7.17 we have
· (f1) $\Sigma_m; \psi \vDash l_m : \{\ warm[C].m\ \}$

From (A4) and Lemma 7.29, we have
· (g1) $\{\ warm[C].m\ \}^c \subseteq \Pi^c$

Using Lemma 7.14, and Lemma 7.15, we have
· (e3) $\Sigma_m; \psi \vDash l_m : \Pi$

∗ **case** $\pi = cold$
From Lemma 7.30, we have
· (a1) $cold\uparrow \in \Phi'^c$

From monotonicity of effect typing, Lemma 7.19 and Lemma 7.18, we have

· (b1) $\Sigma; \psi \vDash cold\uparrow$

From the definition of effect typing, we know
· (c1) $\Sigma \vDash \psi : hot$

Now using (B4), we know l_0 is hot, thus it may follow the same proof steps as the first case.

- **case T-New.** $e = new\ D(\overline{e})$
  From the typing rule T-NEW, we have the following:
  - (A1) $\overline{\hat{f}_i{:}E_i} = constrType(D)$
  - (A2) $\overline{\Gamma; C \vdash e_i : E_i\ !\ (\Phi_i, \Pi_i)}$
  - (A3) $(\Phi', \Pi') = init(D, \overline{\hat{f}_i = \Pi_i})$
  - (A4) $\Phi = \cup\overline{\Phi_i} \cup \Phi'$
  - (A5) $\Pi = \Pi'$

  Let $\Sigma_0 = \Sigma, \sigma_0 = \sigma$, use induction hypothesis on arguments consecutively:
  - (B1) $\Sigma_{i-1} \preccurlyeq \Sigma_i$, $\Sigma_{i-1} \ll \Sigma_i$, $\Sigma_{i-1} \rhd \Sigma_i$ and $\Sigma_i \vDash \sigma_i$
  - (B2) $\Sigma_i \vDash l_i : E_i$
  - (B3) $\Sigma_i; \psi \vDash l_i : \Pi_i$
  - (B4) $\Sigma_{i-1} \vDash \psi : hot \Longrightarrow \Sigma_i \vDash l_i : hot$

  We define σ'_0 and Σ'_0 with a fresh location l:
  - (C1) $\sigma'_0 = \sigma_n \cup \{l \mapsto (D, \overline{\hat{f}_i = l_i})\}$
  - (C2) $\Sigma'_0 = \Sigma_n \cup \{l \mapsto D^{\{\overline{\hat{f}_i}\}}\}$ ▷ class parameters

  The fields $\tilde{f}$ of the value l type check because the potential typing for $C.this.\tilde{f}$ allows any value, other fields type check because they are hot.

  Now we may repeat the proof steps in the basic model to arrive at σ' and Σ' such that
  - (D1) $\Sigma' \vDash l : D^{warm}$
  - (D2) $\Sigma' \vDash \sigma'$
  - (D3) $\Sigma \preceq \Sigma'$
  - (D4) $\Sigma \ll \Sigma'$
  - (D5) $\Sigma \rhd \Sigma'$

  From the definition of $init$, there are two cases:
  - **case** $\Pi = \{warm[C]\}$
    The conclusion $\Sigma'; \psi \vDash l : \Pi$ follows from (D1) and the definition of potential typing.

  - **case** $\Pi = \emptyset$
    In this case, we know all arguments $\overline{l_i}$ are hot. By the same reasoning in the basic

model, we know l is hot as well. The conclusion $\Sigma'; \psi \models l : \Pi$ follows immediately from the definition of potential typing.

- **case T-Block.** $e = (e_0.f = e_1; e_2)$
  From the typing rule T-BLOCK, we have the following:
  - (A1) $\Gamma; C \vdash e_0 : E \,!\, (\Phi_0, \Pi_0)$
  - (A2) $B = fieldType(E, f)$
  - (A3) $\Gamma; C \vdash e_1 : B \,!\, (\Phi_1, \Pi_1)$
  - (A4) $\Gamma; C \vdash e_2 : D \,!\, (\Phi_2, \Pi)$
  - (A5) $\Phi = \Phi_0 \cup \Phi_1 \cup \Pi_1 \uparrow \cup \Phi_2$

  From the induction hypothesis on e_0, we know there exist l_0, σ_0 and Σ_0 such that:
  - (B1) $\Sigma \preccurlyeq \Sigma_0$, $\Sigma \ll \Sigma_0$, $\Sigma \rhd \Sigma_0$ and $\Sigma_0 \models \sigma_0$
  - (B2) $\Sigma_0 \models l_0 : E$
  - (B3) $\Sigma_0; \psi \models l_0 : \Pi_0$
  - (B4) $\Sigma \models \psi : hot \Longrightarrow \Sigma_0 \models l_0 : hot$
  - (B5) $(E, \omega_0) = \sigma_0(l_0)$

  From the induction hypothesis on e_1, we know there exist l_1, σ_1 and Σ_1 such that:
  - (C1) $\Sigma_0 \preccurlyeq \Sigma_1$, $\Sigma_0 \ll \Sigma_1$, $\Sigma_0 \rhd \Sigma_1$ and $\Sigma_1 \models \sigma_1$
  - (C2) $\Sigma_1 \models l_1 : B$
  - (C3) $\Sigma_1; \psi \models l_1 : \Pi_1$
  - (C4) $\Sigma_0 \models \psi : hot \Longrightarrow \Sigma_1 \models l_1 : hot$

  From the premise $\Sigma; \psi \models \Phi$, (A5) and monotonicity of effect typing, we have
  - (D1) $\Sigma_1; \psi \models \Pi_1 \uparrow$

  From (C3), (C4), (D1) and Lemma 7.23, we have
  - (E1) $\Sigma_1 \models l_1 : B^{hot}$

  We define σ_1' as follows
  - (F1) $\sigma_1' = \sigma_1 \cup \{l \mapsto [f \mapsto l_1]\sigma_1(l_0)\}$
  - (F2) $\Sigma_1 \models \sigma_1'$ $\qquad\rhd$ by (E1) and definition

  Now using induction hypothesis on e_2 in (A4), we have
  - (G1) $\Sigma_1 \preccurlyeq \Sigma_2$, $\Sigma_1 \ll \Sigma_2$, $\Sigma_1 \rhd \Sigma_2$ and $\Sigma_2 \models \sigma_2$
  - (G2) $\Sigma_2 \models l_2 : D$
  - (G3) $\Sigma_2; \psi \models l_2 : \Pi$
  - (G4) $\Sigma_1 \models \psi : hot \Longrightarrow \Sigma_2 \models l_2 : hot$

  The result follows immediately.

□

COROLLARY 7.1 (Soundness). *If* $\vdash \mathcal{P}$, *then* $\forall k.\ evalProg(\mathcal{P})(k) = Some(r) \Longrightarrow r \neq None$.

7.10 Discussion

The most important insight is the semantic interpretation of potentials and effects. While the semantics of effects is relatively straight-forward, the semantics of potentials is subtle.

Our first attempt was to store potentials directly in the store typing Σ. This leads us nowhere, as a potential does not have a meaning on its own. Potentials represent aliasing information, they are relational in nature. This means that the potential of a value may only be defined relative to another value.

As the potentials of a class field are defined relative to $this$, it is natural to check that the field value of an object always has the potential $\{\ C.this.f\ \}$ relative to the value of $this$, where C is the class of the object.

Meanwhile, potentials are an over-approximation of values that are possibly under initialization. Consequently, if a value is hot, we may disregard its potentials, i.e. it may take any potentials. Otherwise, if the value is not hot, then it should take one of the root potentials (β) directly or indirectly, which represents a value under initialization.

With the insights above, we arrive at the definitions presented in Figure 7.3 and Figure 7.4.

7.11 Conclusion

In this chapter, we proved the soundness of the type-and-effect system. The meta-theory heavily depends on the meta-theory developed for the basic model, which evidences the foundational role it plays in developing more complex algorithms and their meta-theories.

Chapter 8

Implementation and Evaluation

Thoughts without content are empty,
intuitions without concepts are blind.

— Immanuel Kant

The inference system presented in Chapter 6 works on an experimental language with neither inheritance nor inner classes. Does it scale to real-world programming languages? We answer the question affirmatively by showing concretely how to scale the technique to handle inner classes and inheritance. Based on the extended system, we implement an initialization checker for the Scala programming language.

8.1 Motivation

We have presented an inference system for a simple experimental language in Chapter 6. Real-world programming languages, however, have many more language features than the experimental language. One particular challenge is related to inner classes [30]. Though inner classes are flattened as part of compilation, there are several compelling reasons for the checker to work directly with inner classes before the flattening:

- Flattening will create a lot of synthetic code, e.g. outer accessors, and the names and positions for the synthetic code are sometimes meaningless for programmers. Consequently, it is difficult to report user-friendly error messages.

- The full-construction analysis requires the source code of parent constructors, which could be located in another separately-compiled library. In the Scala 3 compiler, the code is available in an intermediate representation before any lowering, such as flattening or closure conversion. Therefore, the checker cannot avoid inner classes that are present in libraries.

- The inference system in chapter 6 is not expressive enough to support interactions between inner classes and outer class during initialization.

The last point can be seen from the example below:

```
class Trees {
  private var counter = 0
  class ValDef { counter += 1 }
  val theEmptyValDef = new ValDef
}
```

The code above is semantically equivalent to the following after lowering:

```
class Trees {
  var counter: Int = 0
  val theEmptyValDef = new ValDef(this)
}
class ValDef(outer: Trees) { outer.counter += 1 }
```

If we depend on flattening of inner classes and then resort to the inference system in Chapter 6 to check initialization of classes, the code above would be rejected, even if we automatically mark $outer$ as $@cold$. This is because we cannot access the field $outer.counter$ of the cold object $outer$.

The combination of inheritance and inner classes may create complex initialization code, as the following example shows:

```
abstract class NameInfo
abstract class NameKind(val tag: Int) { self =>
  class Info extends NameInfo {
    def kind = self
  }
}
object SignedName extends NameKind(SIGNED) {
  case class SignedInfo(sig: Signature) extends Info {
    override def toString: String = s"$kind $sig"
  }
  val signedInfo = SignedInfo(Signature.NotAMethod)
  val signedTxt = signedInfo.toString
}
```

Inheritance and inner classes may also combine to create some subtle code patterns:

```
class Outer { outer =>
  class Inner extends Outer {
    val x = 5 + outer.n
  }
  val inner = new Inner
  val n = 6
}
```

We will show how to implement a type-and-effect inference system that is capable of handling the complex features above. For the example above, it reports the following error message:

```
-- Error: code/inner-loop.scala:6:6 ----------------------------------
6 |  val n = 6
  |      ^
  |      Access non-initialized field n. Calling trace:
  |       -> val inner = new Inner [ inner-loop.scala:5 ]
  |        -> val x = 5 + outer.n [ inner-loop.scala:3 ]
```

The careful reader will find that the program actually loops and causes stack overflow, instead of accessing the uninitialized field `n`. The checker soundly over-approximates the concrete semantics, it is thus justified to report an error here.

8.2 Design

The main challenge here is how to deal with inner classes. Given the following code example:

```
class Trees {
  class ValDef {
    counter += 1
    def isEmpty = theEmptyValDef == this
  }
  class EmptyValDef extends ValDef
  val theEmptyValDef = new EmptyValDef
  private var counter = 0                  // error
}
```

The first question is, what are the potentials and effects of the expression `new EmptyValDef`?

The potential is not empty, as the object holds a hidden reference to the outer `this`, which enables it to access the field `counter` and `theEmptyValDef`. A tentative answer is `warm`, as all fields of the object are initialized and it is in accord with the potential of the expression after flattening. For the system of chapter 6, we know that all outer `this` should be regarded as `cold` for soundness. This means it is impossible to support the following code:

```
class Trees {
  private var counter = 0
  class ValDef {
    counter += 1                           // Trees.this is cold!
    def isEmpty = theEmptyValDef == this
  }
  class EmptyValDef extends ValDef
  val theEmptyValDef = new EmptyValDef
}
```

For practicality of the system, we find it important to support the code patterns above that involve the interaction between inner classes and outer classes. A natural idea is to add more information to `warm` to remember the potentials for the outer. This idea gives us a potential like $warm(C, \pi)$, where C is the concrete class of the object, and π is the outer for the class C. In the code above, the potential for the new-expression would be `warm(EmptyValDef, Trees.this)`.

The reader may have a doubt in mind: an object may span several different classes in the inheritance chain, possibly with a different outer for each class. The potential $warm(C, \pi)$ seems insufficient to represent the outers for all classes in the inheritance chain. Here, the crucial insight is that *the outers for parent classes are decided by the outer for the most concrete class*. This can be seen from the following example:

```
class Parent {
  class Base
}
class Child(p: Parent) extends Parent {
  class A extends p.Base
  class B extends this.Base
}
def test(c1: Child, c2: Child) = {
  val a = new c1.A
  val b = new c2.B
}
```

In the code above, we create two objects of the class `A` and `B` in the method `test`. For the object `a`, the outer for `Base` is `c1.p`. For the object `b`, the outer for `Base` is `c2`. Both of them can be derived from the definitions of the class and the outer for the most concrete class. This insight will be made concrete in the semantics.

Therefore, the potential $warm(C, \pi)$ is a good choice. In the context of *two-phase checking*, the summarization phase will create potentials like $warm(C, \pi)$ for new-expressions `new p.C(args)`. In the checking phase, we compute the outers for parent classes from the potential $warm(C, \pi)$.

Another complexity concerns deeply nested inner classes, where an inner class may be enclosed by several outer classes, as the following code shows:

```
class A {
  class B {
    class C {
      B.this
      A.this
    }
  }
}
```

Does it suffice to just store the potentials of the immediate outer π in $warm(C, \pi)$? The answer is affirmative, as non-immediate outers are determined by the immediate outer π. In the above, the non-immediate outer `A.this` for class `C` can be computed as the immediate outer of `B.this`, which is in turn the immediate outer of `C.this`. The formal semantics will take advantage of this insight.

Now we turn to the other half of the question: what are the effects of the expression `new EmptyValDef`? Semantically, the expression will execute the constructor of the class `EmptyValDef`, which in turn will call its super constructor. This is similar to method call effects, we represent them as $warm(C, \pi).init(C)$. Why do we choose the prefix $warm(C, \pi)$? First, what we care about

in this effect is possible accesses of uninitialized fields of the outer π, thus it has to be reflected in the prefix. Second, initialization errors related to uninitialized fields of the inner class will be checked separately, we thus assume all fields of the inner class are initialized in the effect.

8.3 Formalization

We start by introducing the syntax and semantics of our experimental language.

8.3.1 Syntax and Semantics

Our language resembles a subset of Scala with inner classes, immutable fields and methods:

$$
\begin{array}{rcl}
\mathcal{P} \in Program & ::= & class\ C\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \overline{\mathcal{C}}\ \} \\
\mathcal{C} \in Class & ::= & class\ C(\overline{\hat{f}{:}T})\ extends\ p.D(\overline{e})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \overline{\mathcal{C}}\ \}\ \mid \\
 & & class\ C(\overline{\hat{f}{:}T})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \overline{\mathcal{C}}\ \} \\
\mathcal{F} \in Field & ::= & val\ f{:}T = e \\
e \in Exp & ::= & x \mid C.this \mid e.f \mid e.m(\overline{e}) \mid new\ p.C(\overline{e}) \\
p \in Path & ::= & C.this \mid p.f \mid x \\
T \in Type & ::= & C \\
\mathcal{M} \in Method & ::= & def\ m(\overline{x{:}T}) : T = e
\end{array}
$$

A program is just a parameter-less class, which may contain deeply nested classes in its class body. Classes have two variants, those that extend other classes, and those that do not extend any class. We do not assume a universal `Object` class in the language. We assume class names are unique in a program.

As expected, now $this$ has to be prefixed with a class name to indicate the class that it refers to, due to the existence of inner classes. Meanwhile, to create an instance of a class, we need to specify its outer, which is required to be a path p, as in $new\ p.C(\overline{e})$.

Why restrict the outer to be a path? Theoretically, it can be any expression, and it is exactly what Java supports. A superficial answer is that the restriction resembles path-dependent types in Scala. The deep semantic reason is that the restriction enables us to only store one immediate outer for an object, and to derive the outers for all classes in the inheritance chain. This property simplifies the representation of objects. Otherwise, we would have to store multiple outers for an object, one for each class in the inheritance chain.

The following constructs are used in defining the semantics:

$$
\begin{array}{rcl}
\Xi \in ClassTable & = & ClassName \rightharpoonup (ClassName \times Class) \\
\sigma \in Store & = & Loc \rightharpoonup Obj \\
\rho \in Env & = & Name \rightharpoonup Value \\
o \in Obj & = & ClassName \times Loc \times (Name \rightharpoonup Value) \\
l, \psi \in Value & = & Loc
\end{array}
$$

We assume a global class table Ξ, which maps a class name to its definition and the name of its immediate outer class. For the top-level class, we assume its outer class is itself. The information will never be used, thus it is not a problem.

An object stores its class name, its outer, and a mapping of field values, which includes all fields defined in the inheritance chain. We assume that field names are unique in the class inheritance hierarchy.

An immediate problem in the semantics is, what is the meaning of $C.this$, encountered in a class D, where the value for $D.this$ is ψ? Note that the concrete class of ψ may be a subclass of D. The resolution is represented by $resolveThis(C, \psi, \sigma, D)$, which is defined in Figure 8.1.

In the simplest case, $C = D$, thus the result is just ψ. Otherwise, C must be an outer class of D. Suppose the concrete class of ψ is E, the immediate outer of E for ψ is l, we may compute the immediate outer of D as l_0 with $resolveOuter(E, l, \sigma, D)$. Then we call $resolveThis(C, l_0, \sigma, owner(D))$ recursively.

This Resolution

$$
\begin{array}{rcl}
resolveThis(C, \psi, \sigma, C) & = & \psi \\
resolveThis(C, \psi, \sigma, D) & = & resolveThis(C, l_o, \sigma, owner(D)) \\
 & & \text{where } (E, l, \_) = \sigma(\psi) \\
 & & \text{and } l_o = resolveOuter(E, l, \sigma, D)
\end{array}
$$

Outer Resolution

$$
\begin{array}{rcl}
resolveOuter(C, l_o, \sigma, C) & = & l_o \\
resolveOuter(E, l_o, \sigma, C) & = & resolveOuter(D, l_p, \sigma, C) \\
 & & \text{where } lookup(E) = class\ E(\overline{\hat{f}{:}T})\ extends\ p.D(\overline{e})\ \{\ \ldots\ \} \\
 & & \text{and } l_p = [\![p]\!]\ (\sigma, \emptyset, l_o, E).1
\end{array}
$$

Method Resolution

$$
\begin{array}{rcl}
resolveMethod(C, m) & = & (C, meth) \quad \text{if } Some(meth) = lookup(C, m) \\
resolveMethod(E, m) & = & resolveMethod(D, m) \quad \text{if } None = lookup(E, m) \\
 & & \text{where } lookup(E) = class\ E(\overline{\hat{f}{:}T})\ extends\ p.D(\overline{e})\ \{\ \ldots\ \}
\end{array}
$$

Owner Resolution

$$
owner(C) = D \quad \text{where } (D, \_) = \Xi(C)
$$

Figure 8.1 – Semantic Helper Methods

The method $resolveOuter(E, l_0, \sigma, C)$ computes the immediate outer of the class C, given the immediate outer of class E is l_0. Note that as a prerequisite, E must be a subclass of C. In the simplest case, if $E = C$, then the result is just l_0. Otherwise, we need to go up the inheritance chain. If class E extends another class D with the prefix p, we may compute the outer for class D as l_p, then call $resolveOuter(D, l_p, \sigma, C)$ recursively.

Here we see concretely why the restriction of the outer to be a path matters: *we may derive the outer from p by just querying the heap.* If the path is an arbitrary expression or it refers to a

mutable variable, it will be impossible to derive the outer. As the values for outers are immutable, it makes sense to cache them in an interpreter or generate additional fields for them in the compiler to avoid recomputation.

With the helper methods, now we may define semantics of the language, which is presented in Figure 8.2. The semantics of a program is defined by creating an instance of the top-level class. The top-level object does not have an outer, so we set its outer to itself.

The evaluation of an expression has the form $[\![e]\!]\,(\sigma, \rho, \psi, C) = (l, \sigma')$, which means that the expression inside the class C evaluates to l and the heap σ', given the heap σ, the environment ρ, and the value ψ for $C.this$. Without the additional parameter C, it would be impossible to give semantics to the expression $E.this$. Note that ψ might be a subclass of C, thus we cannot derive C from ψ. The following code example demonstrates the semantic subtlety:

```
1  class Outer {
2    val inner = new Inner
3
4    inner.foo
5    inner.bar
6
7    class Inner extends Outer {
8      def foo = Outer.this
9    }
10
11   def bar = Outer.this
12 }
```

In the code above, the method call `inner.foo` and `inner.bar` will return different results, though they contain the same code and share the same value for `this`. The reason is that the meaning of `Outer.this` depends on where it is located. In the method `bar`, `Outer.this` is the same as the receiver. In contrast, it refers to the outer object in the method `foo`. This example shows that the following tempting optimization of the semantics is incorrect:

$$[\![D.this]\!]\,(\sigma, \rho, \psi, C) = \psi \text{ if } classOf\,(\psi) <: D$$

Another noticeable change in the semantics is that in instantiating a class, we need to follow the inheritance chain and accumulate all fields. There is no need to store the outer for each class in the inheritance chain, because they can be derived from the outer for the most concrete class.

8.3.2 Effects and Potentials

The definition of effects and potentials are defined in Figure 8.3. Compared to the system in chapter 6, now a warm potential takes the form $warm(C, \pi)$, where C is the concrete class of the object, π is the potential for the immediate outer of C. The potential $\pi.outer(C)$ refers to the immediate outer of the class C, where $C.this$ takes the potential π. It is used as a proxy to give meaning to outer $this$ references, whose concrete meaning will be resolved during effect checking. We also introduce the effect $\pi.init(C)$, which denotes the effects of the constructor of

Program evaluation $\boxed{[\![class\ C\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \overline{\mathcal{C}}\ \}]\!] = (l, \sigma)}$

$[\![class\ C\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \overline{\mathcal{C}}\ \}]\!] = (l, \sigma)$
where $\sigma = init(l, \emptyset, C, \{\ l \mapsto (C, l, \emptyset)\ \}, l)$
and l is a fresh location, and $\Xi = \overline{C \mapsto (D, \mathcal{C})}$

Expression evaluation $\boxed{[\![e]\!]\,(\sigma, \rho, \psi, C) = (l, \sigma')}$

$[\![x]\!]\,(\sigma, \rho, \psi, C) = (\rho(x), \sigma)$

$[\![E.this]\!]\,(\sigma, \rho, \psi, C) = (resolveThis(E, \psi, \sigma, C), \sigma)$

$[\![e.f]\!]\,(\sigma, \rho, \psi, C) = (\omega(f), \sigma_1)$ where $(l_0, \sigma_1) = [\![e]\!]\,(\sigma, \rho, \psi, C)$
and $(\_, \_, \omega) = \sigma_1(l_0)$

$[\![e_0.m(\overline{e})]\!]\,(\sigma, \rho, \psi, C) = [\![e_1]\!]\,(\sigma_2, \rho_1, \psi, C)$
where $(l_0, \sigma_1) = [\![e_0]\!]\,(\sigma, \rho, \psi, C)$
and $(D, \_, \_) = \sigma_1(l_0)$
and $resolveMethod(D, m) = (E, def\ m(\overline{x{:}T}) : T = e_1)$
and $(\overline{l}, \sigma_2) = [\![\overline{e}]\!]\,(\sigma_1, \rho, \psi, E)$
and $\rho_1 = \overline{x \mapsto l}$

$[\![new\ p.D(\overline{e})]\!]\,(\sigma, \rho, \psi, C) = (l, \sigma_3)$
where $l_p = [\![p]\!]\,(\sigma, \emptyset, \psi, C).1$
and $(\overline{l}, \sigma_1) = [\![\overline{e}]\!]\,(\sigma, \rho, \psi, C)$
and $\sigma_2 = [l \mapsto (D, l_p, \emptyset)]\sigma_1$ where l is fresh
and $\sigma_3 = init(l, \overline{l}, D, \sigma_2, l_p)$

Initialization

$init(\psi, \overline{l}, C, \sigma, l_o) = init(\psi, \overline{l_u}, D, \sigma_3, l_p)$
if $lookup(C) = class\ C(\overline{\hat{f}{:}T})\ extends\ p.D(\overline{e})\{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \overline{\mathcal{C}}\ \}$
where $\sigma_1 = assign(\psi, \overline{\hat{f}}, \overline{l}, \sigma)$
and $(\overline{l_u}, \sigma_2) = [\![\overline{e}]\!]\,(\sigma_1, \emptyset, l_o, owner(C))$
and $l_p = [\![p]\!]\,(\sigma_2, \emptyset, l_o, owner(C)).1$
and $\sigma_3 = [\![\overline{\mathcal{F}}]\!]\,(\sigma_2, \psi, C)$

$init(\psi, \overline{l}, C, \sigma, l_o) = \sigma_2$ if $lookup(C) = class\ C(\overline{\hat{f}{:}T})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \overline{\mathcal{C}}\ \}$
where $\sigma_1 = assign(\psi, \overline{\hat{f}}, \overline{l}, \sigma)$
and $\sigma_2 = [\![\overline{\mathcal{F}}]\!]\,(\sigma_1, \psi, C)$

$[\![val\ f : T = e]\!]\,(\sigma, \psi, C) = assign(\psi, f, l_1, \sigma_1)$ where $(l_1, \sigma_1) = [\![e]\!]\,(\sigma, \emptyset, \psi, C)$

Helpers

$[\![\overline{e}]\!]\,(\sigma, \rho, \psi, C) = fold\ \overline{e}\ (Nil, \sigma)\ f$ where
$f\ (ls, \sigma_1)\ e = let\ (l, \sigma_2) = [\![e]\!]\,(\sigma_1, \rho, \psi, C)\ in\ (l :: ls, \sigma_2)$

$[\![\overline{\mathcal{F}}]\!]\,(\sigma, \psi, C) = fold\ \overline{\mathcal{F}}\ \sigma\ f$ where $f\ \sigma_1\ \mathcal{F} = [\![\mathcal{F}]\!]\,(\sigma_1, \psi, C)$

$assign(\psi, f, l, \sigma) = [\psi \mapsto (C, l_o, [f \mapsto l]\omega)]\sigma$ where $(C, l_o, \omega) = \sigma(\psi)$

$assign(\psi, \overline{f}, \overline{l}, \sigma) = [\psi \mapsto (C, l_o, [\overline{f \mapsto l}]\omega)]\sigma$ where $(C, l_o, \omega) = \sigma(\psi)$

Figure 8.2 – Big-step semantics, defined as a definitional interpreter.

class C, where $C.this$ has the potential π.

The outer potential is created by outer selection. The helper function out is also parameterized by the length limit of potentials, so that it is possible to tweak how much interaction between inner classes and outer classes are allowed when initializing an instance of the outer class.

The potential $warm(C, \pi)$ and effect $\pi.init(C)$ are used in the helper function $init$. The system requires all method parameters and class parameters to be fully initialized, only the outer may be under initialization. In the latter case, the resulting potential is warm, and the resulting effects contain a constructor call effect.

8.3.3 Expression Typing

Expression typing is presented in Figure 6.2. The expression typing judgment takes the form $\Gamma; C \vdash e : D \,!\, (\Phi, \Pi)$, which means that the expression e in class C under the environment Γ, can be typed as D, it produces effects Φ and has the potential Π.

There are two noticeable changes. First, in the typing rule T-THIS, for $C.this$ to be well-typed, C must be visible from the current class D, or equivalently, D must be enclosed inside the class C. This is enforced by the helper method $resolveThis$, which traverses the owner classes of the current class D, and generates a potential of the form $\pi.outer(E)$ relative to $this$. If the outer class is too many levels away beyond the length limit of potentials, the result potentials will be empty, and the effects will contain a promotion of the outer $this$.

Second, due to the support of inheritance in the source language, now the type system has to introduce subtyping. As a result, we need the standard rule T-SUB. The subtyping system is straight-forward, we thus omit detailed explanation.

8.3.4 Definition Typing

Definition typing defines how programs, classes, fields and methods are checked. The checking happens in two phases:

(1) *first phase*: conventional type checking is performed and effect summaries are computed;
(2) *second phase*: effect checking is performed to ensure initialization safety.

In type checking a class, it checks that field definitions, method definitions and the optional parent class call are well-typed. The effects for the constructor are accumulated as Φ_{init}, and stored in the method summary map $\mathcal{S}$. Note that the checking does not go to parent classes nor inner classes, as each class is type checked and summarized separately.

Note that for field typing, we assume that field names are unique in the inheritance hierarchy, thus they are not checked. In contrast, for methods, we only assume method names are unique within a class. Due to overriding, we have to check that the overriding is valid, i.e. the method parameter types are contra-variant, while the return type is co-variant.

Effect checking of a class takes the form $\Xi; \mathcal{E}; B^{\Omega} \vdash C \rightsquigarrow B^{\Omega'}$, which means that given an object of class B with fields in Ω initialized, initializing the fields of class $\mathtt{C}$ ensures that fields in Ω' of the object are initialized. As a precondition, B must be a subclass of C. It first checks parent classes, then checks the effects of each field initialization in the current class.

Potentials and Effects

$$
\begin{array}{rcll}
T & ::= & C \mid D \mid E \mid \cdots & \text{type} \\
\beta & ::= & this \mid warm(C,\pi) \mid cold & \text{root} \\
\pi & ::= & \beta \mid \pi.f \mid \pi.m \mid \pi.outer(C) & \text{potential} \\
\Pi & ::= & \{\, \pi_1, \pi_2, \cdots \} & \text{potentials} \\
\phi & ::= & \pi\uparrow \mid \pi.f! \mid \pi.m\Diamond \mid \pi.init(C) & \text{effect} \\
\Phi & ::= & \{\, \phi_1, \phi_2, \cdots \} & \text{effects} \\
\Omega & ::= & \{\, f_1, f_2, \cdots \} & \text{fields} \\
\Delta & ::= & \overline{f_i \mapsto (\Phi_i, \Pi_i)} & \text{field summary} \\
\mathcal{S} & ::= & \overline{m_i \mapsto (\Phi_i, \Pi_i)} & \text{method summary} \\
\mathcal{E} & ::= & \overline{C \mapsto (\Delta, \mathcal{S})} & \text{effect table}
\end{array}
$$

Select

$$
\begin{array}{rcl}
select(\Pi, f) & = & \Pi.map(\pi \Rightarrow select(\pi, f)).reduce(\oplus) \\
select(\pi, \hat{f}) & = & (\emptyset, \emptyset) \\
select(cold, f) & = & (\{cold\uparrow\}, \emptyset) \\
select(\pi, f) & = & (\{\pi.f!\}, \{\pi.f\}) \text{ if } length(\pi) < LIMIT \\
select(\pi, f) & = & (\{\pi\uparrow\}, \emptyset) \text{ otherwise}
\end{array}
$$

Call

$$
\begin{array}{rcl}
call(\Pi, m) & = & \Pi.map(\pi \Rightarrow call(m, \pi)).reduce(\oplus) \\
call(cold, m) & = & (\{cold\uparrow\}, \emptyset) \\
call(\pi, m) & = & (\{\pi.m\Diamond\}, \{\pi.m\}) \text{ if } length(\pi) < LIMIT \\
call(\pi, m) & = & (\{\pi\uparrow\}, \emptyset) \text{ otherwise}
\end{array}
$$

Outer Selection

$$
\begin{array}{rcl}
out(\Pi, C) & = & \Pi.map(\pi \Rightarrow out(m, \pi)).reduce(\oplus) \\
out(cold, C) & = & (\{cold\uparrow\}, \emptyset) \\
out(\pi, C) & = & (\emptyset, \{\pi.outer(C)\}) \text{ if } length(\pi) < LIMIT \\
out(\pi, C) & = & (\{\pi\uparrow\}, \emptyset) \text{ otherwise}
\end{array}
$$

Init

$$
\begin{array}{rcl}
init(C, \overline{\pi_i}, \overline{\hat{f}_j = \Pi_j}) & = & (\cup\overline{\Pi_j\uparrow} \cup \overline{warm(C,\pi_i).init(C)}, \overline{warm(C,\pi_i)}) \\
init(C, \emptyset, \overline{\hat{f}_i = \Pi_i}) & = & (\cup\overline{\Pi_i\uparrow}, \emptyset)
\end{array}
$$

Helpers

$$
\begin{array}{rcl}
\Pi\uparrow & = & \{\, \pi\uparrow \mid \pi \in \Pi \,\} \\
(A_1, A_2) \oplus (B_1, B_2) & = & (A_1 \cup B_1, A_2 \cup B_2)
\end{array}
$$

Figure 8.3 – Potential and Effect Definition

Expression Typing $\boxed{\Gamma; C \vdash e : D \,!\, (\Phi, \Pi)}$

$$\frac{x : D \in \Gamma}{\Gamma; C \vdash x : D \,!\, (\emptyset, \emptyset)} \quad \text{(T-VAR)}$$

$$\frac{(\Phi, \Pi) = resolveThis(C, \emptyset, \{\ this\ \}, D)}{\Gamma; D \vdash C.this : C \,!\, (\Phi, \Pi)} \quad \text{(T-THIS)}$$

$$\frac{\Gamma \vdash e : D \,!\, (\Phi, \Pi) \qquad (\Phi', \Pi') = select(\Pi, f) \qquad E = fieldType(D, f)}{\Gamma; C \vdash e.f : E \,!\, (\Phi \cup \Phi', \Pi')} \quad \text{(T-SEL)}$$

$$\frac{\begin{array}{c}\Gamma; C \vdash e_0 : E_0 \,!\, (\Phi, \Pi) \qquad \overline{\Gamma; C \vdash e_i : E_i \,!\, (\Phi_i, \Pi_i)} \\ \overline{(x_i{:}E_i}, D) = methodType(E_0, m) \qquad (\Phi', \Pi') = call(\Pi, m)\end{array}}{\Gamma; C \vdash e_0.m(\overline{e}) : D \,!\, (\Phi \cup \overline{\Phi_i} \cup \overline{\Pi_i{\uparrow}} \cup \Phi', \Pi')} \quad \text{(T-CALL)}$$

$$\frac{\begin{array}{c}\overline{\hat{f}_i{:}E_i} = constrType(C) \qquad \overline{\Gamma; E \vdash e_i : E_i \,!\, (\Phi_i, \Pi_i)} \\ \Gamma; C \vdash p : D \,!\, (\Phi_p, \Pi_p) \qquad D <: owner(C) \qquad (\Phi', \Pi') = init(C, \Pi_p, \overline{\hat{f}_i = \Pi_i})\end{array}}{\Gamma; E \vdash new\ p.C(\overline{e}) : C \,!\, (\cup\overline{\Phi_i} \cup \Phi_p \cup \Phi', \Pi')} \quad \text{(T-NEW)}$$

$$\frac{\Gamma; C \vdash e : D \,!\, (\Phi, \Pi) \qquad D <: E}{\Gamma; C \vdash e : E \,!\, (\Phi, \Pi)} \quad \text{(T-SUB)}$$

Subtyping $\boxed{C <: D}$

$$C <: C \quad \text{(S-REFL)}$$

$$\frac{D <: C \qquad C <: B}{D <: B} \quad \text{(S-TRANS)}$$

$$\frac{lookup(D) = class\ D(\overline{\hat{f}{:}T})\ extends\ p.C(\overline{e})\ \{\ \ldots\ \}}{D <: C} \quad \text{(S-EXTEND)}$$

This Resolution

$$\begin{array}{lcl} resolveThis(C, \Phi, \Pi, C) & = & (\Phi, \Pi) \\ resolveThis(C, \Phi, \Pi, D) & = & resolveThis(C.this, \Phi' \cup \Phi, \Pi', owner(D)) \\ & & \text{where } (\Phi', \Pi') = out(\Pi, D) \end{array}$$

Figure 8.4 – Expression Typing

Program Typing $\boxed{\vdash \mathcal{P}}$

$$\frac{\Xi = \overline{C \mapsto (D, \mathcal{C})} \qquad \overline{\Xi; D \vdash \mathcal{C} \,!\, (\Delta_c, \mathcal{S}_c)} \qquad \mathcal{E} = \overline{C \mapsto (\Delta_c, \mathcal{S}_c)} \qquad \Xi; \mathcal{E}; C^{\overline{\hat{f}}} \vdash C \rightsquigarrow C^{\Omega}}{\vdash \mathit{class}\ C\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \overline{\mathcal{C}}\ \}} \quad \text{(T-PROG)}$$

Effect Checking $\boxed{\Xi; \mathcal{E}; B^{\Omega} \vdash C \rightsquigarrow B^{\Omega'}}$

$$\frac{\mathit{lookup}(C) = \mathit{class}\ C(\overline{\hat{f}{:}T})\ [\ \mathit{extends}\ p.D(\overline{e})\ ]\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \overline{\mathcal{C}}\ \} \qquad \Xi; \mathcal{E}; B^{\Omega \cup \overline{\hat{f}_D}} \vdash D \rightsquigarrow B^{\Omega'} \qquad (\Delta, \_) = \mathcal{E}(C) \qquad (\Phi_f, \_) = \Delta(f_i) \qquad \mathcal{E}; B^{\Omega' \cup \{f_1, \cdots, f_{i-1}\}} \vdash \Phi_f}{\Xi; \mathcal{E}; B^{\Omega} \vdash C \rightsquigarrow B^{\Omega' \cup \overline{f}_i}} \quad \text{(T-CHECK)}$$

Class Typing $\boxed{\Xi; D \vdash \mathcal{C} \,!\, (\Delta, \mathcal{S})}$

$$\frac{\dfrac{\emptyset; E \vdash p : B \,!\, (\Phi_p, \Pi_p)}{\emptyset; E \vdash e : T_j \,!\, (\Phi_j, \Pi_j)} \qquad \dfrac{\overline{T_j} = \mathit{constrType}(D)}{\overline{\Xi; C \vdash \mathcal{F}_i \,!\, (\Phi_i, \Pi_i)}} \qquad \dfrac{B <: \mathit{owner}(D)}{\overline{\Xi; C \vdash \mathcal{M}_i \,!\, (\Phi_i, \Pi_i)}} \qquad \Phi_{init} = \Phi_p \cup \overline{\Phi_j} \cup \overline{\Phi_i} \cup \overline{\Pi_j{\uparrow}} \cup \mathit{this.init}(D) \qquad \Delta = \overline{f_i \mapsto (\Phi_i, \Pi_i)} \qquad \mathcal{S} = \overline{m_i \mapsto (\Phi_i, \Pi_i)} \cup \{\ \mathit{init} \mapsto (\Phi_{init}, \emptyset)\ \}}{\Xi; E \vdash \mathit{class}\ C(\overline{\hat{f}{:}T})\ [\ \mathit{extends}\ p.D(\overline{e})\ ]\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \overline{\mathcal{C}}\ \} \,!\, (\Delta, \mathcal{S})} \quad \text{(T-CLASS)}$$

Field Typing $\boxed{\Xi; C \vdash \mathcal{F} \,!\, (\Phi, \Pi)}$

$$\frac{\emptyset; C \vdash e : D \,!\, (\Phi, \Pi)}{\Xi; C \vdash \mathit{val}\ f : D = e \,!\, (\Phi, \Pi)} \quad \text{(T-FIELD)}$$

Method Typing $\boxed{\Xi; C \vdash \mathcal{M} \,!\, (\Phi, \Pi)}$

$$\frac{\overline{x{:}T}; C \vdash e : E \,!\, (\Phi, \Pi) \qquad \mathit{superClass}(C) = B \wedge \mathit{methoType}(B, m) = (\overline{x : S}, D) \Longrightarrow \overline{S <: T} \wedge E <: D}{\Xi; C \vdash\ \mathit{def}\ m(\overline{x{:}T}) : E = e \,!\, (\Phi, \Pi)} \quad \text{(T-METHOD)}$$

Figure 8.5 – Definition Typing

Note that we do not check the effects in the parent call $p.D(\overline{e})$. The reason is that when we check each class separately, we assume its outers are fully initialized, thus no initialization effects are observable. The outers for parent classes are determined by the outer of the most concrete class, as the latter is fully initialized, so are the former. In the case where the outer is not fully initialized, it is handled in the effect checking of the outer class with the effect $warm(Inner, \pi).init(Inner)$. As expected, the effects of class constructors contain the effects of parent calls, which can be seen in the typing rule T-CLASS.

8.3.5 Effect Checking

Effect checking judgments take the form $\mathcal{E}; C^{\Omega} \vdash \phi$, which means that given fields Ω of the object of class C are initialized, the effects ϕ are allowed. We use the notation $\mathcal{E}; C^{\Omega} \vdash \Phi$ to mean that we check each effect in Φ separately.

We explain the rules below:

- C-ACC1
  The effect $this.f!$ accesses the field f on the current object, thus it suffices to check that $f \in \Omega$.

- C-ACC2
  All fields of warm objects are initialized, thus the effect $warm(D, \pi).f!$ is always safe.

- C-ACC3
  In this case, π might be an outer potential of the form $\pi'.outer(D)$ or $\pi'.f$ or $\pi'.m$. In such cases, we just propagate the potential π, and check that field accesses on the propagated potentials are safe.

- C-INV1
  For a member call on the current object, we first resolve the virtual method call by using the concrete class B of the current object. Next, we look up the effects of the method m, which we denote as Φ. Finally, we check that the effects in Φ are safe.

- C-INV2
  Similar as above, but the view change uses $warm(D, \pi)$ to replace $this$.

- C-INV3
  Similar to C-ACC3.

- C-UP1
  To promote the potential $warm(D, \pi)$ to be fully initialized, it suffices to check that the outer π may be promoted.

- C-UP2
  Similar to C-INV3. Note that π may not be $cold$ or $this$, as there are no rules to propagate

potentials for them.

- C-INIT.
  Similar to C-INV2 Note that our system ensures that the prefix π of an initialization effect $\pi.init(E)$ always has the form $warm(D, \pi')$.

8.3.6 Potential Propagation

Potential propagation judgments take the form $\mathcal{E}; C \vdash \pi \rightsquigarrow (\Phi, \Pi)$, which means that the potential π propagates to the potentials Π and effects Φ, given that $this$ refers to an object of the class C. We need the effects Φ as part of the propagation result, because the propagation may generate potentials that are beyond the length limit, in which case we promote the potentials to be fully initialized and return an empty set of potentials in order to terminate the effect checking.

We explain the rules below:

- P-ACC1.
  For a field access on the current object, we first resolve the field f to its defining class D. Next, we look up the potentials of the field f in D, which we denote as Π. Finally, we return Π as the result.

- P-ACC2.
  Similar as above, but the view change uses $warm(C, \pi)$ to replace $this$.

- P-ACC3
  In this case, π might be an outer potential of the form $\pi'.outer(D)$ or $\pi'.f$ or $\pi'.m$. In such cases, we just propagate the potential π, and select the field f on the propagated potentials. This is one of the places where we may generate effects due to the selection.

- P-INV1
  Similar to P-ACC1, but work on methods.

- P-INV2
  Similar to P-ACC2, but work on methods.

- P-INV3
  Similar to P-ACC3. This is another place where we may generate effects due to the selection.

- P-OUT1
  In this case, we are referring to the outer of the current class, which is fully initialized. Recall that when we check each class separately, we assume its outer is fully initialized. As a result, all outers for the object in the inheritance chain are fully initialized. The case where outer is not fully initialized is handled in checking the initialization of the outer class, not the inner class.

Effect Checking $\boxed{\mathcal{E}; C^{\Omega} \vdash \phi}$

$$\frac{f \in \Omega}{\mathcal{E}; C^{\Omega} \vdash this.f!} \quad \text{(C-Acc1)}$$

$$\mathcal{E}; C^{\Omega} \vdash warm(D, \pi).f! \quad \text{(C-Acc2)}$$

$$\frac{\mathcal{E}; C \vdash \pi \rightsquigarrow (\Phi, \Pi) \qquad (\Phi', \_) = select(\Pi, f) \qquad \mathcal{E}; C^{\Omega} \vdash \Phi' \cup \Phi}{\mathcal{E}; C^{\Omega} \vdash \pi.f!} \quad \text{(C-Acc3)}$$

$$\frac{(D, \_) = resolve(C, m) \qquad \Phi = \mathit{effectsOf}(D, m) \qquad \mathcal{E}; C^{\Omega} \vdash \Phi}{\mathcal{E}; C^{\Omega} \vdash this.m\Diamond} \quad \text{(C-Inv1)}$$

$$\frac{\begin{array}{c}(E, \_) = resolve(D, m) \qquad \Phi = \mathit{effectsOf}(E, m) \\ warm(D, \pi) \vdash \Phi \rightsquigarrow \Phi' \qquad \mathcal{E}; C^{\Omega} \vdash \Phi'\end{array}}{\mathcal{E}; C^{\Omega} \vdash warm(D, \pi).m\Diamond} \quad \text{(C-Inv2)}$$

$$\frac{\mathcal{E}; C \vdash \pi \rightsquigarrow (\Phi, \Pi) \qquad (\Phi', \_) = call(\Pi, m) \qquad \mathcal{E}; C^{\Omega} \vdash \Phi' \cup \Phi}{\mathcal{E}; C^{\Omega} \vdash \pi.m\Diamond} \quad \text{(C-Inv3)}$$

$$\frac{\mathcal{E}; C^{\Omega} \vdash \pi\uparrow}{\mathcal{E}; C^{\Omega} \vdash warm(D, \pi)\uparrow} \quad \text{(C-Up1)}$$

$$\frac{\mathcal{E}; C \vdash \pi \rightsquigarrow (\Phi, \Pi) \qquad \mathcal{E}; C^{\Omega} \vdash \Pi\uparrow \cup \Phi}{\mathcal{E}; C^{\Omega} \vdash \pi\uparrow} \quad \text{(C-Up2)}$$

$$\frac{\Phi = \mathit{effectsOf}(E, init) \qquad warm(D, \pi) \vdash \Phi \rightsquigarrow \Phi' \qquad \mathcal{E}; C^{\Omega} \vdash \Phi'}{\mathcal{E}; C^{\Omega} \vdash warm(D, \pi).init(E)} \quad \text{(C-Init)}$$

Figure 8.6 – Effect Checking

Propagate Potentials $\boxed{\mathcal{E}; C \vdash \pi \rightsquigarrow (\Phi, \Pi)}$

$$\frac{(D, \_) = resolve(C, f) \qquad \Pi = potentialsOf(D, f) \qquad D; C.this \vdash \Pi \rightsquigarrow \Pi'}{\mathcal{E}; C \vdash this.f \rightsquigarrow (\emptyset, \Pi')} \quad \text{(P-ACC1)}$$

$$\frac{(D, \_) = resolve(C, f) \qquad \Pi = potentialsOf(D, f) \qquad warm(C, \pi) \vdash \Pi \rightsquigarrow \Pi'}{\mathcal{E}; E \vdash warm(C, \pi).f \rightsquigarrow (\emptyset, \Pi')} \quad \text{(P-ACC2)}$$

$$\frac{\mathcal{E}; C \vdash \pi \rightsquigarrow (\Phi, \Pi) \qquad (\Phi', \Pi') = select(\Pi, f)}{\mathcal{E}; C \vdash \pi.f \rightsquigarrow (\Phi' \cup \Phi, \Pi')} \quad \text{(P-ACC3)}$$

$$\frac{(D, \_) = resolve(C, m) \qquad \Pi = potentialsOf(D, m)}{\mathcal{E}; C \vdash this.m \rightsquigarrow (\emptyset, \Pi)} \quad \text{(P-INV1)}$$

$$\frac{(D, \_) = resolve(B, m) \qquad \Pi = potentialsOf(D, m) \qquad warm(C, \pi) \vdash \Pi \rightsquigarrow \Pi'}{\mathcal{E}; E \vdash warm(C, \pi).m \rightsquigarrow (\emptyset, \Pi')} \quad \text{(P-INV2)}$$

$$\frac{\mathcal{E}; C \vdash \pi \rightsquigarrow (\Phi, \Pi) \qquad (\Phi', \Pi') = call(\Pi, m)}{\mathcal{E}; C \vdash \pi.m \rightsquigarrow (\Phi' \cup \Phi, \Pi')} \quad \text{(P-INV3)}$$

$$\mathcal{E}; C \vdash this.outer(D) \rightsquigarrow (\emptyset, \emptyset) \quad \text{(P-OUT1)}$$

$$\frac{\Pi = resolveOuter(E, \pi, D)}{\mathcal{E}; C \vdash warm(E, \pi).outer(D) \rightsquigarrow (\emptyset, \Pi)} \quad \text{(P-OUT2)}$$

$$\frac{\mathcal{E} \vdash \pi \rightsquigarrow (\Phi, \Pi) \qquad (\Phi', \Pi') = out(\Pi, D)}{\mathcal{E}; C \vdash \pi.outer(D) \rightsquigarrow (\Phi \cup \Phi', \Pi')} \quad \text{(P-OUT3)}$$

Outer Resolution

$$\begin{array}{lcl} resolveOuter(C, \pi, C) & = & \{\ \pi\ \} \\ resolveOuter(E, \pi, C) & = & \bigcup \overline{resolveOuter(D, \pi_i, C)} \\ & & \text{where } lookup(E) = class\ E(\overline{\hat{f}{:}T})\ extends\ p.D(\overline{e})\{\ \ldots\ \} \\ & & \text{and } \emptyset; owner(E) \vdash p : B\ !\ (\_, \Pi) \\ & & \text{and } \pi \vdash \Pi \rightsquigarrow \overline{\pi_i} \end{array}$$

Figure 8.7 – Potential Propagation

- P-OUT2
  In this case, we know the current object is $warm(E, \pi)$, we need to resolve the immediate outer for the class D. The system ensures that we must have $E <: D$. Note that the result is not π, because it could be that E is not D, but a subclass of D. The outer is resolved with the helper function $resolveOuter$. The function starts from the outer for E, and goes up the inheritance chain by computing the outer from the parent prefix p until it reaches the class D. The helper method looks similar to its semantic counterpart, but works on the abstract domain.

- P-OUT3
  Similar to P-ACC3.

8.3.7 View Change

View change judgments for potentials take the form $\pi \vdash \pi_1 \rightsquigarrow \pi_2$, which roughly means that the potential π_1 becomes π_2 if we replace $this$ with π. We need to perform view change when checking an effect like $warm(C, \pi).m\Diamond$, where we need to replace $this$ with $warm(C, \pi)$ in the effects for the method m.

We explain the rules below:

- AS-POT-THIS
  It simply returns π, which is intended to replace $this$.

- AS-POT-COLD
  The potential $cold$ stays unchanged.

- AS-POT-WARM1
  It substitutes the potential π in $warm(D, \pi)$ recursively. However, we changed the substitute from $warm(E, \pi_o)$ to $warm(E, cold)$ in order to make effect checking terminate. This is discussed in more detail in the next section.

- AS-POT-WARM2
  It just performs the substitution recursively.

- AS-POT-ACC
  The same as above.

- AS-POT-INV
  The same as above.

- AS-POT-OUT
  The same as above.

The view change rules for effect simply performs substitution for the potential part, we thus

View Change - Potentials $\boxed{\pi \vdash \pi \rightsquigarrow \pi}$

$$\pi \vdash this \rightsquigarrow \pi \qquad \text{(AS-POT-THIS)}$$

$$\pi \vdash cold \rightsquigarrow cold \qquad \text{(AS-POT-COLD)}$$

$$\frac{warm(E, cold) \vdash \pi \rightsquigarrow \pi'}{warm(E, \pi_o) \vdash warm(D, \pi) \rightsquigarrow warm(D, \pi'_i)} \qquad \text{(AS-POT-WARM1)}$$

$$\frac{\pi_0 \vdash \pi \rightsquigarrow \pi'}{\pi_0 \vdash warm(D, \pi) \rightsquigarrow warm(D, \pi'_i)} \qquad \text{(AS-POT-WARM2)}$$

$$\frac{\pi_0 \vdash \pi \rightsquigarrow \pi'}{\pi_0 \vdash \pi.f \rightsquigarrow \pi'.f} \qquad \text{(AS-POT-ACC)}$$

$$\frac{\pi_0 \vdash \pi \rightsquigarrow \pi'}{\pi_0 \vdash \pi.m \rightsquigarrow \pi'.m} \qquad \text{(AS-POT-INV)}$$

$$\frac{\pi_0 \vdash \pi \rightsquigarrow \pi'}{\pi_0 \vdash \pi.outer(D) \rightsquigarrow \pi'.outer(D)} \qquad \text{(AS-POT-OUT)}$$

View Change - Effects $\boxed{\pi \vdash \phi \rightsquigarrow \phi}$

$$\frac{\pi_0 \vdash \pi \rightsquigarrow \pi'}{\pi_0 \vdash \pi\uparrow \rightsquigarrow \pi'\uparrow} \qquad \text{(AS-EFF-UP)}$$

$$\frac{\pi_0 \vdash \pi \rightsquigarrow \pi'}{\pi_0 \vdash \pi.f! \rightsquigarrow \pi'.f!} \qquad \text{(AS-EFF-ACC)}$$

$$\frac{\pi_0 \vdash \pi \rightsquigarrow \pi'}{\pi_0 \vdash \pi.m\Diamond \rightsquigarrow \pi'.m\Diamond} \qquad \text{(AS-EFF-INV)}$$

$$\frac{\pi_0 \vdash \pi \rightsquigarrow \pi'}{\pi_0 \vdash \pi.init(D) \rightsquigarrow \pi'.init(D)} \qquad \text{(AS-EFF-INIT)}$$

Figure 8.8 – View Change

omit the detailed explanation.

8.3.8 Termination

In effect checking, we assume the maintenance of a set of already checked effects during the checking of each class. This gives us a simple way to avoid non-termination caused by recursive methods, which is essentially the same as computing the closure of effects for the type-and-effect system in chapter 6. Meanwhile, it improves performance by avoiding checking the same effect twice. To avoid cluttering the effect checking rules, we omit chaining the already checked set of effects in the rules.

In the system of chapter 6, the domain of effects and potentials is finite for a given program thanks to the limited length of potentials. In the current system, we also restrict the length of potentials. However, the domain is still infinite due to the existence of the potential $warm(C, \pi)$, which poses a problem for the termination of effect checking.

The non-termination of effect checking can be triggered with a simple program, if the effect checking rules are designed naively:

```
1 class B {
2   class C extends B
3   val c: C = new C
4 }
```

In the code above, the class C extends the class B, and we create an instance of the class C in the body of the class B. When checking the initialization of an instance of B, we would encounter the effect $\phi = warm(C, this).init(C)$. Note that ϕ is an effect in the constructor of class B, thus indirectly in the constructor of class C. When we are checking the effect $warm(C, \pi).init(C)$, we will encounter the effect $[this \mapsto warm(C, \pi)]\phi$, for any π. Thus, with naive effect checking, it will follow the infinite sequence below:

$$\begin{aligned} & warm(C, this).init(C) \\ \Longrightarrow\ & warm(C, warm(C, this)).init(C) \\ \Longrightarrow\ & warm(C, warm(C, warm(C, this))).init(C) \\ \Longrightarrow\ & \ldots \end{aligned}$$

The key to stop the sequence is the rule AS-POT-WARM1 in Figure 8.8, where we change the potential to $warm(E, cold)$ to avoid building up more complex nested potentials. With this rule, we will get a terminating sequence below:

$$\begin{aligned} & warm(C, this).init \\ \Longrightarrow\ & warm(C, warm(C, cold)).init \\ \Longrightarrow\ & warm(C, warm(C, cold)).init \end{aligned}$$

Actually, this is the place where we use the potential $cold$. This is another evidence that the abstractions in the basic model play a fundamental role in the design of initialization systems.

8.4 Implementation

In this section, we show the implementation of a practical initialization checker for the Scala programming language. The implementation is already integrated in the Scala 3 compiler and available to Scala programmers via the compiler option `-Ycheck-init`.

One advantage of the type-and-effect system is that it integrates well with the compiler without changing the core type system. In contrast, integrating a type-based system in the compiler poses an engineering challenge, as the following example demonstrates:

```
1 class Knot {
2   val self: Knot @cold = this
3 }
```

In the code above, the type of the field `self` depends on *when* we ask for its type. If it is queried during the initialization of the object, then it has the type `Knot @cold`. Otherwise, it has the type `Knot`. We do not see a principled way to implement the type-based solution in the Scala 3 compiler.

8.4.1 Lazy Summarization

The approach of two-phase checking will blindly summarize the effects and potentials of all methods, regardless of whether such methods are actually called or not during initialization. Given the following code, the summarization will summarize the method `doWork`, though it is not called during initialization:

```
1 class Worker(timeOut: Int) {
2   def doWork() = { /* 500 lines of code omitted */ }
3 }
```

In practice, most methods will not be called during initialization, so blind summarization is a waste of computation that harms compiler performance. In the implementation, we follow the strategy of *lazy summarization*: a method is summarized and cached the first time the summary is required.

8.4.2 Separate Compilation

The Scala 3 compiler provides a standard intermediate representation of Scala code, called TASTy. All libraries are distributed with TASTy, which can be loaded by the compiler together with symbol and type information.

Our implementation takes advantage of the existing compiler infrastructure to implement *full-construction analysis*. The analysis requires access to code in the parent class, which could be located in a separately compiled library.

8.4.3 Debuggability

Error reports should facilitate diagnosis and bug fix. The formal system only tells whether a program can be accepted or not, but it does not report why the program is incorrect when it is rejected. For example, given the following program:

```
class Greeting {
  val message: String = this.welcome()
  val name: String  = "Jack"
  def welcome()     = "Hello, " + name    // error
}
```

If the checker only reports that the field `name` is not initialized at the last line, the programmer still has no clue what the problem is. In a complex program, there can be many paths that lead to an initialization error. For diagnosis, showing the execution path that leads to an error is more helpful. The implemented checker reports the following error:

```
-- Error: code/greeting.scala:3:6 ----------------------------------------
3 |  val name: String  = "Jack"
  |      ^
  |Access non-initialized field name. Calling trace:
  | -> val message: String = this.welcome()   [ greeting.scala:2 ]
  |  -> def welcome()     = "Hello, " + name   [ greeting.scala:4 ]
```

The error message above contains the stack trace that leads to the error, which makes it much easier to figure out why the error happens.

Implementing this feature is easy: it suffices to maintain a stack of method calls as context for checking effects. When we check a method call effect, we augment the context with the method call, and use the augmented stack to check the effects inside the method.

8.4.4 Functions

We introduce the potential $Fun(\Phi, \Pi)$ to represent the potential of a function literal, where Φ is the set of effects to be triggered when the function is called, while Π is the set of potentials for the result of the function call. With the extension, we can easily support the following usage:

```
class Rec {
  val even = (n: Int) => n == 0 || odd(n - 1)
  val odd = (n: Int) => n == 1 || even(n - 1)
  val flag: Boolean  = odd(6)
}
```

8.4.5 Properties

In languages such as Scala and Kotlin, fields are actually properties, public field accesses are actually dynamic method calls, as the following code shows:

```
class A {
    val a = "Bonjour"
    val b: Int = a.size
}

class B extends A {
    override val a = "Hi"
}

new B
```

The code above will throw a null-pointer exception at runtime when initializing the field `A.b`, because the code `a.size` will access the field `B.a`, which is not yet initialized.

The checker treats field accesses as method calls, it can resolve dynamic-dispatching property access on `this` statically by the concrete type of `this`, thanks to the *full-construction analysis*. In the presence of properties, we think *full-construction analysis* is the only reasonable choice for safe initialization.

8.4.6 Traits

Traits are a key language feature of Scala. Unlike interfaces in Java, it is possible to define fields and methods in traits. The following example illustrates the subtlety related to initialization of traits:

```
trait TA { val x = "EPFL" }
trait TB { def x: String; val n = x.length }
class Foo extends TA with TB
class Bar extends TB with TA
new Foo // ok
new Bar // error
```

In the code above, the class `Foo` and class `Bar` only differ in the order in which the traits are mixed in. For the class `Foo`, the body of the trait `TA` is evaluated before the body of `TB`, thus the expression `new Foo` works as expected. In contrast, `new Bar` throws an exception, because the body of the trait `TB` is evaluated first, and at the time the field `x` in `TA` is not yet initialized when it is used in `TB`.

Formally, traits are initialized following a scheme called *linearization* [2]. The implementation follows the linearization semantics in initialization check as well as in the resolution of virtual method calls. The subtlety with traits is another justification for *full-construction analysis*.

8.4.7 Local Classes

Local classes are handled as if they were inner classes located in the closest enclosing classes. This approach is safe, because in the system all method parameters and local definitions are required to be *hot*. The only possible initialization effects that could be observed in a local class are the initialization effects of its enclosing class.

Project	KLOC	W/K	W	X1	X2	X3	X4	A	B	C	D	E	F	G	H
DOTTY	106.0	0.73	77	742	447	146	350	7	16	2	32	0	3	4	13
INTENT	1.8	39.53	71	10	290	0	1	0	0	0	71	0	0	0	0
ALGEBRA	1.3	4.70	6	1	6	0	0	0	0	0	0	0	0	6	0
STDLIB213	43.6	0.62	27	231	104	8	99	14	0	4	2	0	1	6	0
SCALACHECK	5.5	1.08	6	39	70	6	83	0	0	0	6	0	0	0	0
SCALATEST	378.9	0.39	149	1037	718	18	664	0	0	8	114	0	8	19	0
SCALAXML	6.8	0.15	1	36	13	0	0	0	0	0	0	0	0	1	0
SCOPT	0.3	0.00	0	6	4	0	0	0	0	0	0	0	0	0	0
SCALAP	2.2	5.43	12	62	57	2	108	0	0	0	7	5	0	0	0
SQUANTS	14.1	0.00	0	9	0	0	0	0	0	0	0	0	0	0	0
BETTERFILES	2.8	0.00	0	17	1	0	0	0	0	0	0	0	0	0	0
SCALAPB	16.2	0.31	5	28	10	0	6	4	0	0	1	0	0	0	0
SHAPELESS	2.5	0.79	2	5	0	0	0	0	0	0	0	2	0	0	0
EFFPI	5.7	0.53	3	15	5	0	12	0	0	0	3	0	0	0	0
SCONFIG	21.8	0.60	13	70	43	0	8	13	2	2	0	0	1	6	2
MUNIT	2.7	1.13	3	32	73	1	13	0	0	0	2	0	0	0	1
SUM	612.1	0.61	375	2340	1841	181	1344	38	18	16	238	7	13	42	16

Figure 8.9 – Experimental result. The column W/K is the number of warnings per KLOC, and the column W is the number of warnings issued for the corresponding project. Other columns are explained in the text.

8.5 Evaluation

We evaluate the implementation on a significant number of real-world projects, with zero changes to the source code.

8.5.1 Experimental Result

The experimental results are shown in Figure 8.9. The first three columns show the size of the projects and warnings reported for each project:

- **KLOC** - the number of lines of code (KLOC) in the project checked by the system
- **W/K** - the number of warnings issued by the system per KLOC
- **W** - the number of warnings issued by the system

We can see that for over 0.6 million lines of code, the system reports 375 warnings in total, the average is 0.61 warnings per KLOC. We can better interpret the data in conjunction with the following columns:

- **X1** - the number of field accesses on `this` during initialization
- **X2** - the number of method calls on `this` during initialization
- **X3** - the number of field accesses on *warm* objects during initialization
- **X4** - the number of method calls on *warm* objects during initialization

The data for the columns above are censused by the initialization checker, one per source location. Without type-and-effect inference, the system would have to issue one warning for each method call on `this` and warm objects [1], which contributes to more than 3K warnings, which is 8 times more warnings!

[1] If we forget that non-private field accesses are also method calls in Scala.

We manually analyzed all the warnings, and classified them into 8 categories:

- **A** - Use `this` as constructor arguments, e.g. `new C(this)`
- **B** - Use `this` as method arguments, e.g. `call(this)`
- **C** - Use inner class instance as constructor arguments, e.g. `new C(innerObj)`
- **D** - Use inner class instance as method arguments, e.g. `call(innerObj)`
- **E** - Use uninitialized fields as by-name arguments
- **F** - Access non-initialized fields
- **G** - Call external Java or Scala 2 methods
- **H** - others

The warnings in category **A** and **C** are related to the creation of cyclic data structures. From the theory, we know such code patterns can be supported by declaring a class parameter to be *cold*. The current implementation does not support any annotations yet, we plan to introduce explicit annotations in the next version of the system.

Most of the warnings lie in the category **D**, which refer to cases like the following:

```
1  object Foo {
2    case class Student(name: String, age: Int)
3    call(Student("Jack", 30)             // should be OK, currently a warning
4  }
```

For the code above, our system currently issues a warning, as it only knows that the object created by `Student("Jack", 20)` is warm, while method arguments are required to be hot. Checking whether an inner class instance may be safely promoted to hot or not can be expensive if the inner class contains many fields and methods. However, it suggests that the system could be improved for common use cases that only involve small classes, such as the example above.

The category **E** refers to cases like the following, which is not supported currently:

```
1  def foo(x: => Int) = new A(x)
2  class A(init: => Int)
3  class Foo {
4    val a: A = foo(b) // category E
5    val b: Int = 100
6  }
```

As an over-approximation, we expect the warnings in category **F** are all false positives. However, to our delight, the system actually finds 8 true positives in ScalaTest, and one true positive in the Scala standard library. It also discovers two bugs in the Scala 3 compiler and they are fixed already.

The category **G** involves method calls on `this` in the constructor, but the target method is compiled by Java or the Scala 2 compiler. The category **H** involves code that performs pattern matching on `this`, or calling methods on `cold` values.

The performance impact of the initialization checker on compilation is shown in Figure 8.10. Overall, the initialization checker takes about 10% to 30% of the compilation time. The performance is dependent on projects: the more complex the initialization code is, the more time it takes.

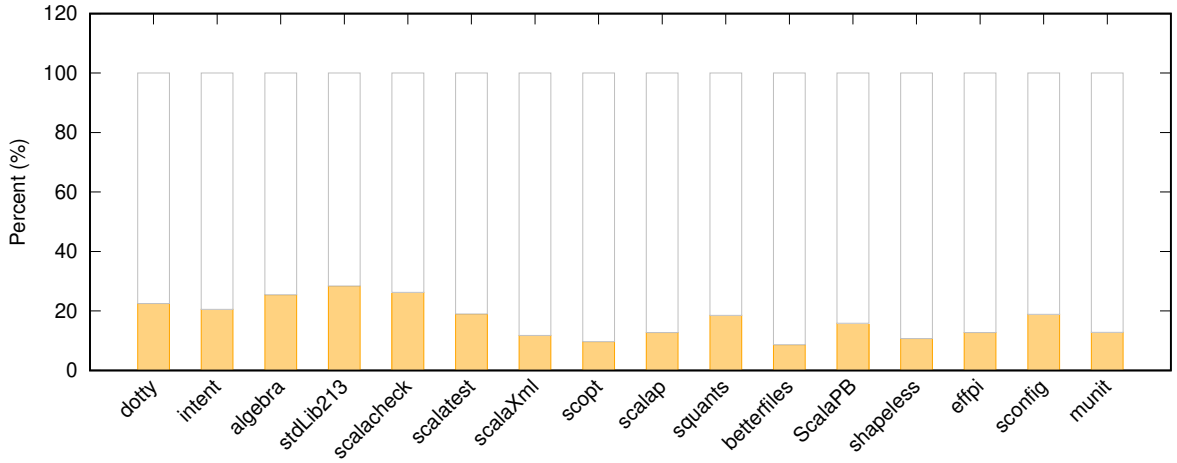


Figure 8.10 – Performance of the initialization checker. The numbers indicate the percentage of time for initialization check relative to the whole compilation time.

8.5.2 Discovered Bugs

As an over-approximation, we expect the warnings are all false positives. However, to our delight, our initialization system finds real bugs in high-quality projects, such as the Scala 3 compiler, Scala standard library and ScalaTest.

In the ScalaTest project, the checker reports 8 true positives [2]. The errors have similar forms, the following code demonstrates two of them:

```
1  sealed abstract class Fact {
2    val isVacuousYes: Boolean
3    val isYes: Boolean
4
5    final def stringPrefix: String =
6      if (isYes) {
7        if (isVacuousYes) "VacuousYes" else "Yes"
8      }
9      else "No"
10 }
11
12 class Binary_&(left: Fact, right: Fact) extends Fact {
13     val rawFactMessage: String = {
14         // ...
15         factDiagram(0)
16     }
17
18     val isYes: Boolean = left.isYes && right.isYes
19     val isVacuousYes: Boolean = isYes && (left.isVacuousYes || right.isVacuousYes)
20
21     override def factDiagram(level: Int): String = {
```

[2] https://github.com/scalatest/scalatest/issues/1481

```
22          stringPrefix
23      }
24  }
```

The problem with the code above is that when we create an instance of `Binary_&`, it will call `factDiagram`, which in turn calls `stringPrefix`, where the properties `isYes` and `isVacuousYes` are used before they are initialized in the class `Binary_&`. Such errors never cause null-pointer exceptions, and when they slip into a large code base, it will take significant efforts to debug.

The following code demonstrates a bug in the Scala 3 compiler [3]:

```
1  class Scanner(...) {
2    val indentSyntax = ...
3    // ...
4    nextToken()   // the call indirectly reach the property indentSyntax
5  }
6
7  class LookaheadScanner(indent: Boolean = false) extends Scanner(...) {
8    override val indentSyntax = indent
9    // ...
10 }
```

Our checker reports the following error:

```
1  [warn] -- Warning: dotty/compiler/src/dotty/tools/dotc/parsing/Scanners.scala:885:34
2  [warn] 885 |    override val indentSyntax = indent
3  [warn]     |                                ^
4  [warn]     |Access non-initialized field indentSyntax. Calling trace:
5  [warn]     | -> class LookaheadScanner(...) { [Scanners.scala:884 ]
6  [warn]     |  -> nextToken() [Scanners.scala:1323 ]
7  [warn]     |   -> if (isAfterLineEnd) handleNewLine(lastToken) [Scanners.scala:311 ]
8  [warn]     |    -> indentIsSignificant = indentSyntax [ Scanners.scala:484]
```

The problem is that when we create an instance of `LookaheadScanner`, the call `nextToken()` in the super class `Scanner` will reach the overridden property `indentSyntax`, which is not yet initialized in the sub-class.

The other bug found in the Scala 3 compiler is related to a subtle optimization of lazy value definitions in traits [4], which is not in accord with the language specification. Without the initialization checker, the bug would be latent longer in the compiler.

The bug in the Scala standard library [5] can be illustrated with the code below:

```
1  object Promise {
2    val Noop = new Transformation[Nothing, Nothing](...)
3
4    class Transformation[-F, T] (...) extends DefaultPromise[T]() with ... {
5      def this(...) = this(...)
```

[3] https://github.com/lampepfl/dotty/issues/7660
[4] https://github.com/lampepfl/dotty/issues/7434
[5] https://github.com/scala/bug/issues/11979

```
  }

  class DefaultPromise[T](initial: AnyRef) extends ... {
    def this() = this(Noop: AnyRef)
  }
}
```

The problem is that when we initialize the field `Noop`, it creates an instance of `Transformation`, which calls the super constructor in `DefaultPromise`, where `Noop` is accessed before initialization.

8.5.3 Challenging Examples

One design goal of the Scala 3 initialization system is to keep the core type system of the compiler intact. Consequently, we require that all arguments to methods are fully initialized, which is in line with good initialization practices. Otherwise, new types such as $T@cold$ must be introduced in the language to handle safe method overriding.

There are two problems related to changing the type system. First, integrating the types in a statically typed language poses engineering challenge. Second, type mismatches are usually reported as errors, while for initialization violations warnings are more appropriate.

The current implementation is based on a type-and-effect system. It elegantly lays on top of the type system, thus avoids the problems that a type-based solution would cause.

However, during the experiment we do encounter some reasonable code patterns that the current implementation does not support. The following code about `LazyList` construction is one such example:

```
trait LazyList[A] { ... }
implicit class Helper[A](l: => LazyList[A]) {
  def #:: [B >: A](elem: => B): LazyList[B] = ...
}
class Test {
  val a: LazyList[Int] = 5 #:: b
  val b: LazyList[Int] = 10 #:: a
}
```

In the code above, inside the class `Test`, we use `b` (before it is initialized) as a by-name argument to initialize the field `a`. Similar code patterns also appear in by-name implicits [6].

To support the example above, the system has to support passing objects under initialization as arguments to methods and constructors. There is a chance to support the usage above without complicating the type system if we *restrict that the methods are effectively final*. The restriction removes the burden of overriding checks. Class constructors are inherently final, thus this is not a problem.

However, the restriction cannot handle some use cases. The following code is a common pattern in the Scala 3 compiler to create cyclic type structures:

```
class RecType(parentExp: RecType => Type) {
```

[6] https://docs.scala-lang.org/sips/byname-implicits.html

```
2   val parent = parentExp(this)
3 }
```

A solution based on types would change the type of `parentExp` to something like `RecType @cold =>Type @cold`. The solution requires changes to the core type system, thus is not feasible as we discussed above.

We can make the field `parent` `lazy` to silence the warning about the escape of `this`. However, as compilers are performance-sensitive, we cannot do that due to the potential performance penalty with lazy fields. Currently, we have to resort to `@unchecked` for such cases.

Making a field `lazy` and adding the annotation `@unchecked` are currently the two ways to suppress warnings for complex initialization code. The lazy trick is a panacea with the slight danger of turning actual initialization errors into non-termination. On the other hand, drawing the line of when `@unchecked` should be used is a difficult language design decision. We expect the design principles of initialization will contribute to the decision process.

8.6 Open Challenges

Most object-oriented languages support static fields. safe initialization of static fields is a challenge that this thesis does not address.

Conceptually, we may regard static fields as fields of a dummy class encapsulating the module (i.e. project), then the technique for handling inner classes can be used. For it to be safe, module dependencies should be an acyclic graph, so that modules may be initialized in total order.

However, it is not the case in the Java world, where cycles are possible. Moreover, JVM does not have the concept of modules, and static fields are initialized along with the loading of class definitions.

We believe the proper handling of static fields will be to replace them with module fields and enforce that module dependencies are non-cyclic.

Another theoretical challenge is about the storage of partially constructed objects in generic data structures. From our empirical experience, such code patterns are rare. Nevertheless, it is challenging to support such use cases.

8.7 Related Work

Igarashi and Pierce are the first to study inner classes formally [30]. Their system sidesteps initialization problems by disallowing the usage of $this$ in the constructor, like that of Featherweight Java [29]. Their language supports the expression $new\ e.C(\overline{e})$, where the outer may be any expression. Our language only allows the outer to be a path, and we point out the semantic advantage of this restriction.

All the existing work on initialization depends on some unspecified inference or analysis to cut down syntactic overhead [16, 12, 11]. We are the first to present a formal inference system on the problem of safe initialization, as well as demonstrate in detail how it scales to language features like inheritance and inner classes.

8.8 Conclusion

This chapter shows that the inference system presented in Chapter 6 scales to complex language features, such as inner classes, functions, and inheritance.

We implement an initialization checker for Scala in the Scala 3 compiler. The experiments on several real-world projects show that the checker advances the state-of-the-art.

Bibliography

[1] Apple Inc. *Swift Language Guide: Initialization*. https://docs.swift.org/swift-book/LanguageGuide/Initialization.html. 2019.

[2] Martin Odersky. *Scala Language Specification*. https://scala-lang.org/files/archive/spec/2.13/. 2019.

[3] Alban Reynaud, Gabriel Scherer, and Jeremy Yallop. "A right-to-left type system for mutually-recursive value definitions". In: *arXiv preprint arXiv:1811.08134* (2018).

[4] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. "Lessons from building static analysis tools at Google". In: *Commun. ACM* 61 (2018), pp. 58–66.

[5] Nada Amin and Tiark Rompf. "Type soundness proofs with definitional interpreters". In: *POPL*. 2017.

[6] Andrej Bauer and Matija Pretnar. "Programming with algebraic effects and handlers". In: *Journal of logical and algebraic methods in programming* 84.1 (2015), pp. 108–123.

[7] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. "The Java Language Specification, Java SE 8 Edition". In: 2015.

[8] Dave Clarke, James Noble, and Tobias Wrigstad. "Aliasing in Object-Oriented Programming. Types, Analysis and Verification". In: *Lecture Notes in Computer Science*. 2013.

[9] Marco Servetto, Julian Mackay, Alex Potanin, and James W Noble. "The Billion-Dollar Fix - Safe Modular Circular Initialisation with Placeholders and Placeholder Types". In: *ECOOP*. 2013.

[10] Jeremy Siek. *Type Safety in Three Easy Lemmas*. http://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html. 2013.

[11] Yoav Zibin, David Cunningham, Igor Peshansky, and Vijay Saraswat. "Object initialization in X10". In: *European Conference on Object-Oriented Programming*. Springer. 2012, pp. 207–231.

[12] Alexander J. Summers and Peter Müller. "Freedom Before Commitment: A Lightweight Type System for Object Initialisation". In: *OOPSLA*. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 1013–1032. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048142.

[13] Joe Duffy. *On partially-constructed objects*. http://joeduffyblog.com/2010/06/27/on-partiallyconstructed-objects/. 2010.

[14] Joseph Gil and Tali Shragai. “Are We Ready for a Safer Construction Environment?” In: *ECOOP*. 2009.

[15] Tony Hoare. *Null References: The Billion Dollar Mistake*. `https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/`. 2009.

[16] Xin Qi and Andrew C. Myers. “Masked Types for Sound Object Initialization”. In: *POPL*. POPL ’09. ACM. Savannah, GA, USA, 2009, pp. 53–65. ISBN: 978-1-60558-379-2.

[17] Éric Tanter. “Beyond static and dynamic scope”. In: *ACM Sigplan Notices* 44.12 (2009), pp. 3–14.

[18] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN: 0321356683, 9780321356680.

[19] Manuel Fähndrich and Songtao Xia. “Establishing object invariants with delayed types”. In: *OOPSLA*. 2007.

[20] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. “Adding delimited and composable control to a production programming environment”. In: *ACM SIGPLAN Notices* 42.9 (2007), pp. 165–176.

[21] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P Black. “Traits: A Mechanism for Fine-grained Reuse”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28.2 (2006), pp. 331–388.

[22] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. “An Overview of the Scala Programming Language”. In: 2004.

[23] Manuel Fähndrich and K Rustan M Leino. “Heap monotonic typestates”. In: *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*. 2003.

[24] Manuel Fähndrich and K. Rustan M. Leino. “Declaring and checking non-null types in an object-oriented language”. In: *OOPSLA*. 2003.

[25] Manuel Fähndrich and Rustan Leino. “Heap Monotonic Typestate”. In: 2003.

[26] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. “Traits: Composable Units of Behaviour”. In: *European Conference on Object-Oriented Programming*. Springer. 2003, pp. 248–274.

[27] Benjamin C. Pierce. *Types and Programming Languages*. 1st. MIT Press, 2002.

[28] John C. Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures”. In: *LICS*. 2002.

[29] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. “Featherweight Java: a minimal core calculus for Java and GJ”. In: *ACM Trans. Program. Lang. Syst.* 23 (2001), pp. 396–450.

[30] Atsushi Igarashi and Benjamin C. Pierce. “On Inner Classes”. In: *Inf. Comput.* 177 (2000), pp. 56–89.

[31] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. “Principles of Program Analysis”. In: *Springer Berlin Heidelberg*. 1999.

[32] Luca Cardelli, Florian Matthes, and Martin Abadi. “Extensible syntax with lexical scoping”. In: (1994).

[33] John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard C. Holt. “The Geneva convention on the treatment of object aliasing”. In: *OOPS Messenger* 3 (1992), pp. 11–16.

[34] Patrick Cousot and Radhia Cousot. “Comparison of the Galois Connection and Widening/-Narrowing Approaches to Abstract Interpretation”. In: *JTASPEFT/WSA*. 1991.

[35] Olivier Danvy and Andrzej Filinski. “Abstracting control”. In: *Proceedings of the 1990 ACM conference on LISP and functional programming*. 1990, pp. 151–160.

[36] John M. Lucassen and David K. Gifford. “Polymorphic effect systems”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM. 1988, pp. 47–57.

[37] Robert E. Strom and Shaula Yemini. “Typestate: A programming language concept for enhancing software reliability”. In: *IEEE Transactions on Software Engineering* SE-12 (1986), pp. 157–171.

Appendix A

A Step-Indexed Interpreter in Coq

A.1 Syntax

```
Module Syntax.
  (** Mode lattice *)
  Inductive Mode: Type := hot | warm | cold.

  Definition CID := nat.   (* class name    *)
  Definition VID := nat.   (* variable name *)
  Definition FID := nat.   (* field name    *)
  Definition MID := nat.   (* method name   *)

  Definition Typ: Type := Mode * CID.

  Inductive Exp: Type :=
  | var (x: VID)                    (* Var 0 is this *)
  | new (cid: CID)(args: [Exp])
  | select(exp: Exp)(f: FID)
  | call (exp: Exp)(m: MID)(args: [Exp])
  | block (lhs: Exp)(f: FID)(rhs: Exp)(exp: Exp).

  Definition Field  :  Type := Typ * Exp.
  Definition Method :  Type := ([Typ]) * Typ * Exp.
  Definition Class  :  Type := ([Typ]) * ([Field]) * ([Method]).

  Parameter Ξ     : [Class].      (* class table    *)
  Parameter entry : CID.          (* entry class id *)

  (** Helper methods *)
  Definition classInfo (cid: CID): option Class :=
    nth_error Ξ cid.

  Definition classOfTyp (T: Typ): option Class :=
```

```
    match T with
    | (_, cid) => classInfo cid
    end.

  Definition fieldInfo (cid: CID)(fid: FID): option Field :=
    do (_, fields, _) ← classInfo cid;
    nth_error fields fid.

  Definition methodInfo (cid: CID)(mid: MID): option Method :=
    do (_, _, methods) ← classInfo cid;
    nth_error methods mid.
End Syntax.
```

A.2 Semantics

```
Module Semantics.
  Import Syntax.

  Definition Loc  : Type := nat.
  Definition Obj  : Type := CID * [Loc].
  Definition Heap : Type := [Obj].
  Definition Env  : Type := [Loc].

  Definition Res(T: Type):= option (option T).

  Definition update (A: Type)(lst: [A])(i: nat)(v': A): [A] :=
    let f v acc :=
      match acc with
      | (k, res) => (S k, (if i =? k then v' else v) :: res)
      end
    in
    match fold_right f (0, nil) lst with
    | (_, lst) => lst
    end.

  Definition assign (l1 : Loc)(f: FID)(l2: Loc)(σ: Heap): option Heap :=
    do (cid, ω) ← nth_error σ l1 ;
    do _ ← nth_error ω f;
    let ω2 := update Loc ω f l2 in
    let o' := (cid,  ω2) in
    Some (update Obj σ l1  o').

  Definition init_field (l1 : Loc)(f: FID)(l2: Loc)(σ: Heap): option Heap :=
    do (cid, ω) ← nth_error σ l1 ;
    if (length ω) =? f then
      let o' := (cid,  ω ++ (l2 :: nil)) in
      Some (update Obj σ l1  o')
```

```
33      else None.
34
35    Reserved Notation "'⟦' e '⟧' '(' σ ',' ρ ')' '(' n ')'"  (at level 80).
36
37    Fixpoint eval (exp: Exp)(σ: Heap)(ρ: Env)(n: nat): Res (Loc * Heap) :=
38      match n with
39      | 0 => TimeOut
40      | S n' =>
41        let evals (exps: [Exp])(σ: Heap): Res ([Loc] * Heap) :=
42          let f acc e :=
43              do (vs, σ1) ⇐ acc;
44              do (l, σ2) ⇐ ⟦e⟧(σ1, ρ)(n');
45              Success (vs ++ (l :: nil), σ2)
46          in fold_left f exps (Success (nil, σ))
47        in
48        let init (l: Loc)(ρ: Env)(cid: CID)(σ: Heap): Res Heap :=
49          do (_, fields, _) ← classInfo cid;
50          let fn acc field :=
51              match field with
52              | (_, e) =>
53                do (fid, σ1 ) ⇐ acc;
54                do (l1 , σ2) ⇐ ⟦e⟧(σ1, (l :: ρ))(n');
55                do σ3 ← init_field l fid l1  σ2;
56                Success (S fid, σ3)
57              end
58          in do (_, σ2) ⇐ fold_left fn fields (Success (0, σ));
59            Success σ2
60        in
61        match exp with
62        | var id =>
63          do l ← nth_error ρ id;
64          Success (l, σ)
65
66        | select e f =>
67          do (l, σ') ⇐ ⟦e⟧(σ, ρ)(n');
68          do (_, fs) ← nth_error σ' l;
69          do l' ← nth_error fs f;
70          Success (l', σ')
71
72        | call e m args =>
73          do (l, σ1) ⇐ ⟦e⟧(σ, ρ)(n');
74          do (cid, _) ← nth_error σ1  l;
75          do (_, _, methods) ← classInfo cid;
76          do (_, _, body) ← nth_error methods m;
77          do (argsV, σ2) ⇐ evals args σ1;
78          ⟦body⟧(σ2, (l :: argsV))(n')
79
80        | new cid args =>
```

```
        do (argsV, σ1) ⇐ evals args σ;
        let obj := (cid, nil) in
        let l   := length σ1  in
        let σ2  := σ ++ (obj :: nil) in
        do σ3 ⇐ init l argsV cid σ2;
        Success (l, σ3)

      | block lhs f rhs e =>
        do (l1 , σ1) ⇐ ⟦lhs⟧(σ, ρ)(n');
        do (l2, σ2) ⇐ ⟦rhs⟧(σ1, ρ)(n');
        do σ3 ← assign l1  f l2 σ2;
        ⟦e⟧(σ3, ρ)(n')
      end
    end

  where "'⟦' e '⟧' '(' σ ',' ρ ')' '(' n ')'"  := (eval e σ ρ n).

  Definition evalProg(n: nat): Res (Loc * Heap) :=
    match classInfo entry with
    | Some (nil, nil, (nil, _, e) :: nil) =>
      let o := (entry, nil) in
      ⟦e⟧((o :: nil), (0 :: nil))(n)

    | _ => None
    end.

End Semantics.
```

A.3 Typing

```
Module Typing.
  Import Syntax.

  Definition join (μ1 μ2: Mode): Mode :=
    match μ1, μ2 with
    | hot, _     => μ2
    | _, hot     => μ1
    | cold, _    => cold
    | _, cold    => cold
    | _, _       => warm
    end.

  Definition meet (μ1 μ2: Mode): Mode :=
    match μ1, μ2 with
    | hot, _     => hot
    | _, hot     => hot
    | cold, _    => μ2
```

```
18      | _, cold    => μ1
19      | _, _       => warm
20      end.
21
22    Inductive S_Mode: Mode -> Mode -> Prop :=
23    | s_mode_hot  : forall μ, S_Mode hot μ
24    | s_mode_cold : forall μ, S_Mode μ cold
25    | s_mode_warm : S_Mode warm warm.
26
27    Notation " m1 ⊓ m2 " := (meet m1 m2) (at level 40).
28    Notation " m1 ⊔ m2 " := (join m1 m2) (at level 40).
29
30    (** subtyping *)
31    Inductive S_Typ: Typ -> Typ -> Prop :=
32    | s_typ_ctor : forall C μ1 μ2, S_Mode μ1 μ2 ->
33                S_Typ (μ1, C) (μ2, C).
34
35    Notation " T1 <: T2 " := (S_Typ T1 T2) (at level 40).
36
37    Inductive S_Typs: ([Typ]) -> ([Typ]) -> Prop :=
38    | s_typs_nil : S_Typs nil nil
39    | s_typs_cons: forall T1 T2 ts1 ts2, S_Typs ts1 ts2 ->
40              T1 <: T2 ->
41              S_Typs (T1 :: ts1) (T2 :: ts2).
42
43    Definition P_hot (T: Typ): Prop :=
44      match T with
45      | (hot, _) => True
46      | _ => False
47      end.
48
49    Fixpoint P_hots (ts: [Typ]): Prop :=
50    match ts with
51    | nil => True
52    | T :: ts' => P_hot T ∧ P_hots ts'
53    end.
54
55    (** typing rules *)
56    Reserved Notation "Γ ⊢ e ∈ T" (at level 69).
57
58    Inductive T_Exp: ([Typ]) -> Exp -> Typ -> Prop :=
59    | t_var:
60      forall Γ vid T,
61      nth_error Γ vid = Some T -> T_Exp Γ (var vid) T
62
63    | t_new: forall Γ cid args paramTs fields methods,
64        classInfo cid = Some (paramTs, fields, methods) ->
65        T_Exps Γ args paramTs ->
```

```
66        Γ ⊢ (new cid args) ∈ (warm, cid)
67
68    | t_new_hot: forall Γ cid args argTs paramTs fields methods,
69       classInfo cid = Some (paramTs, fields, methods) ->
70        T_Exps Γ args argTs ->
71        P_hots argTs ->
72        S_Typs argTs paramTs ->
73        Γ ⊢ (new cid args) ∈ (warm, cid)
74
75    | t_select_hot: forall Γ e f C D μ init,
76        Γ ⊢ e ∈ (hot, C) ->
77        fieldInfo C f = Some ((μ, D), init) ->
78        Γ ⊢ (select e f) ∈ (hot, D)
79
80    | t_select_warm: forall Γ e f C T init,
81        Γ ⊢ e ∈ (warm, C) ->
82        fieldInfo C f = Some (T, init) ->
83        Γ ⊢ (select e f) ∈ T
84
85    | t_call: forall Γ e m args paramTs retT body thisT μ C,
86        Γ ⊢ e ∈ (μ, C) ->
87        (μ, C) <: thisT ->
88        methodInfo C m = Some (thisT :: paramTs, retT, body) ->
89        T_Exps Γ args paramTs ->
90        Γ ⊢ (call e m args) ∈ retT
91
92    | t_call_hot: forall Γ e m args argTs paramTs retT body thisT C,
93        Γ ⊢ e ∈ (hot, C) ->
94        (hot, C) <: thisT ->
95        methodInfo C m = Some (thisT :: paramTs, retT, body) ->
96        T_Exps Γ args argTs ->
97        P_hots argTs ->
98        S_Typs argTs paramTs ->
99        Γ ⊢ (call e m args) ∈ retT
100
101   | t_block: forall Γ e1  f e2 e3 C μ T,
102       Γ ⊢ (select e1  f) ∈ (μ, C) ->
103       Γ ⊢ e2 ∈ (hot, C) ->
104       Γ ⊢ e3 ∈ T ->
105       Γ ⊢ (block e1 f e2 e3) ∈ T
106
107   with
108   T_Exps: ([Typ]) -> ([Exp]) -> ([Typ]) -> Prop :=
109   | t_exps_nil: forall Γ, T_Exps Γ nil nil
110
111   | t_exps_cons: forall Γ Ts es T e, T_Exps Γ es Ts ->
112       Γ ⊢ e ∈ T ->
113       T_Exps Γ (e :: es) (T :: Ts)
```

```
114
115   where "Γ ⊢ e ∈ T" := (T_Exp Γ e T).
116
117   Definition T_Field (Γ: [Typ])(field: Field): Prop :=
118     match field with
119     | (T, e) => Γ ⊢ e ∈ T
120     end.
121
122   Fixpoint T_Fields (Γ: [Typ])(fields: [Field]): Prop :=
123     match fields with
124     | nil => True
125     | f :: fs => T_Field Γ f ∧ T_Fields Γ fs
126     end.
127
128   Definition T_Method (method: Method): Prop :=
129     match method with
130     | (Γ, retT, e) =>
131       Γ ⊢ e ∈ retT
132     end.
133
134   Fixpoint T_Methods (methods: [Method]): Prop :=
135     match methods with
136     | nil => True
137     | m :: ms => T_Method m ∧ T_Methods ms
138     end.
139
140
141   Definition T_Class (cid: CID)(class: Class): Prop :=
142     match class with
143     | (paramTs, fields, methods) =>
144       T_Fields ((cold, cid) :: paramTs) fields ∧
145       T_Methods methods
146     end.
147
148   Fixpoint T_Classes (rest: [Class])(cid: nat): Prop :=
149     match rest with
150     | nil            => True
151     | class :: rest' => (T_Class cid class) ∧ T_Classes rest' (cid + 1)
152     end.
153
154   Definition T_Prog: Prop :=
155     match classInfo entry with
156     | Some (nil, nil, (nil, T, e) :: nil) =>
157       ((hot, entry) :: nil) ⊢ e ∈ T ∧ T_Classes Ξ 0
158     | _ =>
159       False
160     end.
161 End Typing.
```

A.4 Properties

```
Module Properties.
  Import Semantics.
  Import Typing.

  Parameter well_typed: T_Prog.    (* the program is well typed *)

  Definition Soundness: Type := forall n, evalProg(n) <> Error.
End Properties.
```

A.5 Some Helpers

```
Notation "'Error'"     := (Some None).
Notation "'TimeOut'"   := None.
Notation "'Success' t" := (Some (Some t)) (at level 60).

Notation "'do' p ⇐ e1 ; e2" :=
  match e1 with
  | Some (Some p) => e2
  | Some _        => Error
  | None          => TimeOut
  end
  (right associativity, at level 60, p pattern).

Notation "'do' p ← e1 ; e2" :=
  match e1 with
  | Some p => e2
  | None   => None
  end
  (right associativity, at level 60, p pattern).

Notation "'[' X ']'" := (list X) (at level 40).
```