Celsius: A Model for Safe Initialization of Objects

FENGYUN LIU, EPFL
AGGELOS BIBOUDIS, EPFL
PAOLO G. GIARRUSSO, Delft University of Technology
MARTIN ODERSKY, EPFL

Accessing uninitialized data during object initialization is a common and subtle programming error. This error is either not prevented by mainstream languages, like in Java, C++, Scala, or it is prevented by greatly restricting initialization patterns, like in Swift. In this paper, we propose a model called *Celsius* for safe and modular initialization of objects, and prove its soundness. We extend the model and implement a prototype in Scala. The experiments on several real-world Scala projects show that the design requires few programmer annotations.

Additional Key Words and Phrases: Object initialization, Scala

1 INTRODUCTION

In object-oriented programming, using not-yet-initialized fields during object construction is a common source of subtle programming errors. Such errors are usually not detected by compilers. For example, in the Java code below, the field `a` in class `Parent` is incorrectly initialized with unassigned field `b` from the class `Child`. Therefore, the field `a` will get the value `0` instead of `20`. As a result, the intended invariant is violated. The C++ or Scala versions of the code expose the same problem.

```
abstract class Parent {
  int a;
  Parent() {
    a = init();
  }
  abstract int init();
}
```

```
class Child extends Parent {
  int b;
  Child() {
    super(); b = 10;
  }
  int init() { return 2 * b; }
}
```

The readers might wonder, does it suffice to disallow the usage of `this` until all fields of the object are initialized? Indeed, avoiding escape of `this` inside the constructor is a good practice in programming [Bloch 2008]. Escape of `this` happens when (1) we call a method on `this`; (2) pass `this` as an argument to method calls or new expressions to create new objects; (3) assign `this` to the field of another object, to a static field or store in an array; (4) capture `this` in an anonymous function or in an inner class instance.

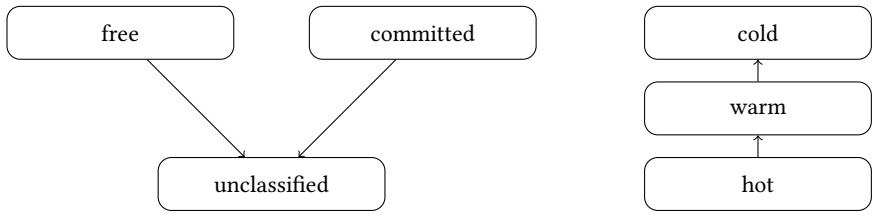
The Swift language disallows escape of `this` during its initialization — `this` only becomes available when all fields of the object are assigned. The scheme is simple and safe, however it is not expressive enough. First, it does not allow creating circular data structures without resorting to `null`, which is demonstrated by the following Scala code example:

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

```
class Child(parent: Parent) {
  val tag = 10
}
```

Authors' addresses: Fengyun Liu, EPFL; Aggelos Biboudis, EPFL; Paolo G. Giarrusso, Delft University of Technology; Martin Odersky, EPFL.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/



(a) The Freedom Model (b) The Celsius Model

Fig. 1. The comparison of the two models. The arrow means whether a value in the source state is a value in the target state.

In the code above, in order to create a mutual reference between `Parent` and `Child`, we need to escape `this` in the expression `new Child(this)`.

Second, it does not support calling methods in parent class during initialization. Such initialization patterns cannot be simply dismissed as a code smell, as the following code demonstrates, which looks reasonable and safe:

```
1  abstract class TokensCommon {
2    private val map: mutable.Map[Int, String] = mutable.Map.empty
3    def enter(k: Int, v: String) = map(k) = v
4  }
5  object Tokens extends TokensCommon {
6    final val WITH = 26;          enter(WITH, "with")
7    final val CASE = 28;          enter(CASE, "case")
8  }
```

Designing a system for safe initialization is an art which tries to strike a balance between usability and expressiveness. There has been a lot of work done on safe initialization, some leaning towards expressiveness [Qi and Myers 2009], and others towards usability [Summers and Müller 2011; Zibin et al. 2012]. The former usually require significant annotation overhead, thus are not used in practice and only serve theoretical purposes. The latter usually achieve simplicity by sacrificing expressiveness. However, as long as common programming patterns are supported, the latter are more practical.

Our work is directly inspired by *Freedom before Commitment* [Summers and Müller 2011], which is currently the state of the art in the latter group and implemented both in Spec# and the Checker Framework [Papi et al. 2008]. We call the model presented by Summers and Müller [2011] *the Freedom model*, and ours *the Celsius model*; both are illustrated in Figure 1. The models look simple, but proving soundness of the models is non-trivial, as it involves reasoning about properties of memory at runtime.

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 Celsius model, however, three states are used to describe the lifetime of an object, enabling us to capture the nature of object initialization more precisely without sacrificing practicality.

- **cold**: some fields of the object might not have been assigned a value.
- **warm**: all fields have been assigned a value which in turn might be *cold*, *warm* or *hot*.
- **hot**: the object is fully initialized and its fields only point to *hot* values.

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

- In Celsius, hot is a subtype of cold, whereas in Freedom, committed is not a subtype of free.
- Celsius introduces warm, which has no correspondent in Freedom.

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, while both models support the parent-child example, there is a subtle difference. Freedom supports it by annotating the `parent` parameter of the `Child` class with `@free` :

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

```
class Child(parent: Parent @free) {
  val tag = 10
}
```

Celsius supports this with the annotation `@cold` and `@warm`:

```
class Parent {
  val child: Child @warm = new Child(this)
}
```

```
class Child(parent: Parent @cold) {
  val tag = 10
}
```

The difference lies in the way the expression `new Child(this)` is handled. In Celsius the expression returns a warm value, while in Freedom it returns a free value. As a result, the selection `child.tag` will be safe in Celsius, but Freedom cannot tell whether `child.tag` is initialized or not. Meanwhile, the Celsius model supports the `TokensCommon` example, whereas the Freedom model does not. As we will see in our empirical evaluation (Figure 6), 30.7% warnings we get from real-world projects are about calling parent methods as in the `TokensCommon` example; in the Dotty project [Odersky et al. 2013] the percentage is 83.6%. Without the concept *warm*, we would be unable to support such use cases.

We believe the Checker Framework [Papi et al. 2008] extension of the Freedom model is motivated by similar real world programming patterns. However, the extended model is not formalized and is a little complex. It requires four different annotations, two of which are further extended with additional parameters. Our work can be seen as an attempt to capture the essence of the extension in a simplified model.

To be fair, there are code examples that type check in Freedom, but fail to type check in Celsius. The following is one such example:

```
class A(b: B) {
  var c: C = new C(this)
  b.m(this)
}
```

```
class B {
  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, as `new C(a)` is a value under initialization (it holds a reference to a free value `a`). In Celsius, it is only possible to assign hot values

to fields of cold objects[1], while in Freedom 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. Summers and Müller [2011] reached the same conclusion in their empirical studies:

> "*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.*"

Thus, we think this loss of expressiveness is a justified design choice. The main contributions of this paper are the following:

1. We propose *Celsius*, a new model for safe and modular initialization based on three initialization states of objects. The model improves the state of the art [Summers and Müller 2011], and it applies to class-based languages with a static type system (Section 2).
2. We formalize the model in a first-order class-based language and *prove its soundness*. To our knowledge, we are the first to give a simple and formal characterization of a common phenomenon in the initialization of mutually-dependent values which we call *collective maturity*. Despite the additional expressiveness, the formalization and meta-theory is simpler than that of Summers and Müller [2011] (Section 4).
3. We extend and prototype the Celsius model in the new Scala compiler[2], Dotty, and we evaluate the implementation based on real-world Scala projects. The experiments show that few annotations are required to migrate legacy codebases (Section 5).

Organization. We first present an overview of our model in Section 2 and show how to use the model by examples in Section 3. Then, we formalize the model in Section 4. We show experimental results in Section 5. Finally, we discuss the related work in Section 6.

2 OVERVIEW OF CELSIUS

To ensure modular and safe initialization, we propose a thermal model called *Celsius*, which we survey in this section. In the model, each object can be in one of the three initialization states:

- **cold**: some fields of the object might not have been assigned a value;
- **warm**: all fields have been assigned a value, but those values might *cold*, *warm* or *hot*;
- **hot**: the object is fully initialized and its fields only point to *hot* values.

We also call the initialization states *thermal states* or *modes* in the paper. The thermal value of an object *changes* during initialization: at the beginning of its constructor, the object is *cold*; after all fields of the object are assigned, the object becomes *warm*; when all objects it refers to become hot, it becomes hot.

The type systems of typical object-oriented languages can be augmented with initialization states (e.g., in the form of annotations) to implementation safe and modular initialization. Typically, the type T in the original type system now becomes (μ, T) in the augmented type system, where μ denotes the initialization state of the value.

If we think of thermal states in terms of maturity levels, then the states form a subtyping lattice: hot < warm < cold. The ordering is employed to check whether a value conforms to the expected thermal state of a parameter in a method call. 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.

[1]However, initializing fields with non-hot values is supported. Initialization and assignment are handled differently in our model.

[2]The prototype is implemented in *–url hidden for anonymization purposes–*. The test suite alongside with all examples from this paper are in *–url hidden for anonymization purposes–*.

2.1 Rules

The thermal state of an object specifies what operations we can perform on this object safely. The rules for interacting with thermal objects are summarized in Table 1.

o : C	o.f		o.f = e	o.m($\overline{e}$)	
	allow?	result	allow?	allow?	result
hot	✓	hot	e : hot	✓	$\overline{\text{e : hot}}$? hot : $\mathcal{R}$(C, m)
warm	✓	$\mathcal{F}$(C, f)	e : hot	$\mathcal{M}$(C, m) $\sqsupseteq$ warm	$\mathcal{R}$(C, m)
cold	✗	-	e : hot	$\mathcal{M}$(C, m) = cold	$\mathcal{R}$(C, m)

Table 1. An overview of the thermal state of objects. The columns represent the thermal value with respect to the receiver object (field selection, assignment and method calls). For field selection a checkmark means that all fields have the corresponding thermal value, and a cross means none.

We assume some helper functions below:

- $\mathcal{F}$(C, f): the thermal state of the field `f` in class `C`.
- $\mathcal{M}$(C, m): the thermal state of `this` inside the method `m` in class `C`.
- $\mathcal{R}$(C, m): the thermal state of the return result of the method `m` in class `C`.

2.2 Hot Values

For hot values, accessing all fields and calling all methods is allowed. Accessing a field always returns a hot value, as hot values only point to hot values. The thermal state of the return value of a method call depends on the thermal states of its arguments. If all arguments are hot, the thermal state of the return value is hot , irregardless of the declared thermal state of the method return. Otherwise, the method call returns a value with the thermal state indicated by its return type. This achieves a kind of polymorphism that produces more precise types when the arguments are hot. It is only possible to assign hot values to hot objects—we consider the invariant that all fields of a hot object are hot.

2.3 Warm Values

For warm values, it is possible to access all fields. A field access returns a value with the thermal state indicated by the field type. For methods, only *warm methods* can be called on a *warm object*. This is consistent with the definition that a *warm method* means the receiver, `this`, inside the method is *warm*. Warm values can be used as cold values. Finally, it is only possible to assign hot values to warm objects.

2.4 Cold Values

If an object is cold: none of its fields can be accessed (as they might not be initialized), and only its *cold methods* can be called. This is consistent with the definition that a *cold method* means the receiver `this` inside the method is *cold*. It is only possible to assign hot values to cold objects.

2.5 Assignment

The rule for assignment `o.f = e` may look too conservative at first: a reader might think that it is safe to assign a warm value to a warm field of a warm object. However, this breaks soundness. The reason is that a warm or cold value could in fact also be hot. If we weaken the rule to allow

assigning a non-hot value to a warm value, then it is possible that a hot value points to a non-hot value, which breaks the invariant that *hot objects only point to hot values*.

In contrast, assignment of free values to fields of free values is allowed in the Freedom model [Summers and Müller 2011]. As we mentioned in the introduction, Summers and Müller [2011] reported they have not found any cases where an object escapes from its constructor and has its fields written (with hot or non-hot values). This finding is confirmed by our evaluation of real-world projects. Thus we think the actual expressiveness of our model is not affected by this design decision.

2.6 New Expression

A new expression `new C(args)` returns a *warm* value if any of the arguments is *cold* or *warm*. Otherwise, it returns a hot value. Observe the example below:

```
1   class Parent {
2     val child: Child @warm = new Child(this)
3     child.tag         // ok
4     child.parent      // ok
5     child.parent.name // error
6     val name = "Celsius"
7   }
8
9   class Child(parent: Parent @cold) {
10    val tag = 10
11  }
```

At line 2, the `new`-expression creates a warm value, as `this` is cold. At line 3 and 4, we can access `child.tag` and `child.parent`, because all fields of a warm object are assigned. At line 5, `child.parent.name` is disallowed because `child.parent` is a cold value and selection on a cold value is forbidden.

This contrasts our model to the Freedom model. In that model, the field `child` at line 2 would be typed as *free*, which is roughly *cold* in Celsius. In the Freedom model, selection on a free value, e.g. `child.parent`, is permitted, but it returns a value of the type `unclassified C?` instead of `C`, where `unclassified` means the value may be free or committed, and `C?` means the value may be `null`. Programers have to test whether `child.parent` is `null` or not, while in Celsius it is guaranteed to be not `null`. In the Freedom model, it cannot tell whether `child.tag` is initialized or not. Even if the usage of an uninitialized integer field will not cause null-pointer exceptions, it is still a semantic error to do so.

2.7 Flow-Sensitive Analysis

The Celsius model is paired with a flow-sensitive analysis in the constructor. For instance, without such an analysis, we would have to treat `this` as a cold value until all fields have been assigned. In the following snippet, `this` would be considered cold when selecting `this.size` on line 3, and the checker would have to report a spurious error, even though `size` has been initialized in the previous line:

```
1   class Store {
2     var size = 100
3     var buffer = new Array(this.size)
4   }
```

Instead of reporting such spurious errors, we use flow-sensitive analysis to record what fields of `this` are assigned; selection is only allowed on those fields. When `this` escapes the analysis boundary, e.g. passed as a parameter to an external method call, it should be promoted to one of the thermal states and type checked according to the thermal model.

The Celsius model and the analysis are orthogonal. While the Celsius model is generally applicable, the analysis may vary in granularity (intra-procedural or inter-procedural), context-sensitivity [Smaragdakis et al. 2011] for different languages or even for the same language. For simplicity, in the formalization (Section 4) we only use the simplest intra-procedural flow-sensitive analysis. In our implementation, we perform intra-class analysis which achieves a balance between modularity and expressiveness.

We would like to point out that the Freedom model is also paired with definite assignment analysis in the constructor to ensure that all fields of a class are assigned in its constructor.

3 EXAMPLES

In this section, we demonstrate the usage of the Celsius model on examples of a few initialization patterns.

3.1 Parent-Child Interaction

Our system supports interactions between child and parent classes. Child classes can call parent methods annotated with `@warm`, as the following example shows:

```
import scala.collection.mutable
abstract class TokensCommon {
  private val map: mutable.Map[Int, String] = mutable.Map.empty
  @warm def enter(k: Int, v: String) = map(k) = v
}
object Tokens extends TokensCommon {
  final val WITH = 26;          enter(WITH, "with")
  final val CASE = 28;          enter(CASE, "case")
}
```

The `@warm` annotation on the method `enter` declares that `this` is warm inside the method. The compiler will check the method body of `enter` to make sure it can be safely called from child classes, assuming `this` is warm. In the code above, we can call the method `enter` in the subclass `Tokens` because (1) `super` is assumed to be warm after the super constructor call, and (2) it is allowed to call a warm method on a warm value.

It is also possible to call an abstract method during the initialization of an object. Suppose we have the following definition of the class `AbstractFile`:

```
abstract class AbstractFile {
  @cold def name: String
  val extension: String = Path.extension(name)
}
```

In the code above, the annotation `@cold` on the method `name` constrains that `this` must be cold inside the scope of the method. Now the compiler will reject the following code:

```
class RemoteFile(url: String) extends AbstractFile {
  val localFile: String = url.hashCode + ".tmp"
  @cold def name: String = localFile // error
}
```

Semantically, there is an initialization error in the code above. The reason is that when the method `name` is called during initialization of the field `extension`, the field `localFile` is not initialized yet. Thus, the method `name` in class `RemoteFile` should not use the field `localFile`. In Celsius, the code is rejected without the need for such complex semantic reasoning. We only use a simple rule: `this` is cold in the method `name`, and selecting fields on a cold value is disallowed!

3.2 Circular Data Structures

The initialization of circular data structures is supported in a similar way as the Freedom model. Consider the following example:

```
1   class Parent {
2     val child: Child @warm = new Child(this)
3     child.tag         // ok
4     child.parent      // ok
5     child.parent.name // error
6
7     val name = "Celsius"
8   }
9
10  class Child(parent: Parent @cold) {
11    val tag = 10
12    parent.name       // error
13  }
```

The annotation `@cold` on the class parameter of the class `Child` makes it possible to safely pass `this` to another class at line 2 before it is fully initialized. The selection `child.parent.name` at line 5 is rejected because Celsius recognizes that `child.parent` is cold and selecting fields from cold values is forbidden. In the class `Child`, the selection `parent.name` is rejected, because it is not allowed to select fields of cold values.

3.3 Inner-Outer Interaction

Next, we exemplify how to use the thermal model to check interactions between outer classes and inner classes:

```
1   class Foo {
2     @warm val bar = new Bar(this)
3     @warm val inner = new bar.Inner  // ok
4     val list = List(1, 2, 3)
5
6     @warm class Inner { val len = list.size }
7   }
8   class Bar(val foo: Foo @cold) {
9     val inner = new foo.Inner        // error
10    val name = "Bar"
11
12    @warm class Inner { val message = "hello, " + name }
13  }
```

In this example we annotate inner classes, similarly to methods. Recall that inner class constructors receive an "outer pointer" to an instance of the external class. The annotation `@warm` on the two

inner classes means that the outer of the class is assumed to be warm when creating an instance of the inner class.

The code above is problematic because the expression `new foo.Inner` at line 9 will indirectly touch the uninitialized field `list` in the class `Foo`. The checker rejects the code `new foo.Inner` at line 9 with a simple rule: *it is disallowed to instantiate an inner class marked as* `@warm` *on a cold value.* In contrast, the expression `new bar.Inner` at line 3 is accepted due to the fact that `bar` is a warm value.

The following case demonstrates the combination of inner-outer interaction and parent-child interaction:

```
1   class Parent {
2     val list = List(3, 5, 6)
3     def foo() = 4
4     @warm class Inner1 { val len = list.size }
5     class Inner2 { val len = foo() }
6   }
7   class Child extends Parent {
8     val a: Inner1 @warm = new Inner1
9     val b: Inner2 @warm = new Inner2 // error
10    val c: Int = 30
11    override def foo() = c
12  }
```

The expression `new Inner2` at line 9 is problematic as it indirectly touches the uninitialized field `c` in the class `Child`. Our prototype rejects this code using a simple rule: `super` is warm[3] and it is illegal to instantiate an inner class marked with @hot (which is the default) on a warm value.

4 FORMALIZATION

In this section, we formalize the thermal model Celsius in a class-based language with mutations. The type system enforces the rules of the thermal model and implements a simple flow-sensitive analysis. We prove that the system is sound, hence in particular enjoys initialization safety.

4.1 Language

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

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

$\mathsf{x}, \mathsf{y}, \mathsf{z} \in \textbf{Variable}$ variable names
$\mathsf{m} \in \textbf{MethodName}$ method names
$\mathsf{C}, \mathsf{D}, \mathsf{E} \in \textbf{ClassName}$ class names

A program is a pair of a list of class definitions and a term representing the execution entry point. Types T pair a mode and a class name; we write types as the pair (μ, C) in the meta-theory.

[3] In Scala, super constructors are always called before executing the current class body, thus it is safe to assume that `super` is warm. In languages like Java, `super` may be assumed to be warm after the point where the super constructor is called.

A class definition contains class parameters (x:T), field definitions (var x:T = e) and method definitions. Class parameters are also fields of the class, just that their values come from outside in contrast to field definitions. All class fields are mutable.

We sidestep the category of statements by introducing blocks e.x = e; e as an expression. In a block, the first part is an assignment, the second part is an expression, the value of the latter is the value of the block.

A method definition is decorated with a mode declaration, which means the initialization state of `this` inside the method. We restrict method body to just expressions. This choice simplifies the meta-theory without loss of expressivity thanks to block expressions.

Note that in the language, there are no constructors for classes, like in Scala. Instead, the constructor is replaced by class parameters and class fields. Compared to Featherweight Java [Igarashi et al. 2001], the syntax of our language is more succinct. The constructors in Featherweight Java can only assign constructor parameters to class fields, which is necessary to avoid initialization errors. We make our language closer to Scala by allowing fields like var x:T = e in the class body, the right-hand side of which can be any expression. This enables the usage of `this` during initialization, which may result in errors.

Like in *Freedom before Commitment* [Summers and Müller 2011], we do not model inheritance formally. On the one hand, the essence of the thermal model is orthogonal to inheritance. On the other hand, we believe it is more valuable to have a minimal calculus, so that it can be easily studied, shared and extended. Our formalization and its meta-theory is simpler than Summers and Müller [2011] in that we never need to reason about stack frames.

4.2 Semantics

In this section, we discuss the operational semantics of our language. At a high-level, our evaluation is call-by-value, deterministic, and with left-to-right evaluation order. We formalize it as a definitional interpreter [Amin and Rompf 2017]. This interpreter can fail or loop, and we'll need to turn it into a total function for our soundness proof. We discuss how we make it total in Section 4.5.

The following constructs are used in defining the semantics:

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

A class table is a partial function mapping class names to class definitions. An environment is a partial function mapping names to values, representing values of method parameters. The only values are references to objects, and objects pair a class name with a mappings from field names to values.

Note that non-initialized fields are represented by missing keys in the object, instead of a `null` value. As we will see, newly initialized objects have no fields, and new fields are gradually inserted during initialization, until all fields defined by the class have been assigned.

Our definitional interpreter appears in Figure 2; we explain it in detail next.

When evaluating a program $(\overline{C}, \mathsf{e})$, the interpreter first turns the lists of class definitions into a map class table, and then evaluates the top-level expression e in an empty environment. As receiver, the interpreter provides a dummy instance of class `Null`. The type system ensures that the class `Null` is defined and has no fields or methods.

Program evaluation $\boxed{[\![(\overline{C}, e)]\!] = (v, \sigma)}$

$$[\![(\overline{C}, e)]\!] = [\![e]\!]\,(ct, l \mapsto (\mathsf{Null}, \emptyset), \emptyset, l)$$
where $ct = \overline{C \to C}$ and l is a fresh location

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

$$[\![x]\!]\,(ct, \sigma, \rho, v) = (\rho(x), \sigma)$$
$$[\![\mathsf{this}]\!]\,(ct, \sigma, \rho, v) = (v, \sigma)$$
$$[\![e.x]\!]\,(ct, \sigma, \rho, v) = (o(x), \sigma_1) \text{ where } (v_0, \sigma_1) = [\![e]\!]\,(ct, \sigma, \rho, v) \text{ and } (\_, o) = \sigma_1(v_0)$$
$$[\![e_0.m(\overline{e})]\!]\,(ct, \sigma, \rho, v) = [\![e_1]\!]\,(ct, \sigma_2, \rho_1, v_0)$$
where $(v_0, \sigma_1) = [\![e_0]\!]\,(ct, \sigma, \rho, v)$
and $(C, \_) = \sigma_1(v_0)$
and $\mathsf{lookup}(ct, C, m) = \mathsf{def}\ m(\overline{x{:}T}) : T = e_1$
and $(\overline{v}, \sigma_2) = [\![\overline{e}]\!]\,(ct, \sigma_1, \rho, v)$
and $\rho_1 = \overline{x \mapsto v}$

$$[\![\mathsf{new}\ C(\overline{e})]\!]\,(ct, \sigma, \rho, v) = (l, \sigma_3)$$
where $(\overline{v}, \sigma_1) = [\![\overline{e}]\!]\,(ct, \sigma, \rho, v)$
and $\sigma_2 = l \mapsto (C, \emptyset); \sigma_1$ where l is fresh
and $\sigma_3 = \mathsf{init}(l, \overline{v}, C, ct, \sigma_2)$

$$[\![e_1.x = e_2; e]\!]\,(ct, \sigma, \rho, v) = [\![e]\!]\,(ct, \sigma_3, \rho, v)$$
where $(v_1, \sigma_1) = [\![e_1]\!]\,(ct, \sigma, \rho, v)$
and $(v_2, \sigma_2) = [\![e_2]\!]\,(ct, \sigma_1, \rho, v)$
and $\sigma_3 = \mathsf{assign}(v_1, x, v_2, \sigma_2)$

Initialization

$$\mathsf{init}(l, \overline{v}, C, ct, \sigma) = [\![\overline{\mathcal{F}}]\!]\,(ct, \sigma_1, l)$$
where $\mathsf{lookup}(ct, C) = \mathsf{class}\ C(\overline{x{:}T})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \}$
and $\sigma_1 = \mathsf{assign}(l, \overline{x}, \overline{v}, \sigma)$

$$[\![\mathsf{var}\ x : T = e]\!]\,(ct, \sigma, v) = \mathsf{assign}(v, x, v_1, \sigma_1) \text{ where } (v_1, \sigma_1) = [\![e]\!]\,(ct, \sigma, \emptyset, v)$$

Helpers

$$[\![\overline{e}]\!]\,(ct, \sigma, \rho, v) = \mathsf{fold}\ \overline{e}\ (\mathsf{Nil}, \sigma)\ f \text{ where}$$
$f\ (vs, \sigma_1)\ e = \mathsf{let}\ (v, \sigma_2) = [\![e]\!]\,(ct, \sigma_1, \rho, v)\ \mathsf{in}\ (v :: vs, \sigma_2)$

$$[\![\overline{\mathcal{F}}]\!]\,(ct, \sigma, v) = \mathsf{fold}\ \overline{\mathcal{F}}\ \sigma\ f \text{ where } f\ \sigma_1\ \mathcal{F} = [\![\mathcal{F}]\!]\,(ct, \sigma_1, v)$$
$$\mathsf{assign}(v_0, x, v, \sigma) = [v_0 \mapsto (C, [x \mapsto v]\mathsf{fields})]\sigma \text{ where } (C, \mathsf{fields}) = \sigma(v_0)$$
$$\mathsf{assign}(v_0, \overline{x}, \overline{v}, \sigma) = [v_0 \mapsto (C, [\overline{x \mapsto v}]\mathsf{fields})]\sigma \text{ where } (C, \mathsf{fields}) = \sigma(v_0)$$

Fig. 2. Semantics, defined as a definitional interpreter.

Evaluating an expression requires the quadruple (ct, σ, ρ, v), consisting of a class table ct, a store σ, an environment ρ and a receiver value v for `this`. The evaluation returns the result value and the updated store.

Evaluating a variable x just looks up the value of the variable in the environment, and keeps the store unchanged. Evaluating `this` just returns the receiver and the store.

To evaluate a field selection e.x, the interpreter first evaluates the expression e to the value v_0 and the updated store σ_1, and then looks up the field x on the object referred by the value v_0.

To evaluate a method call $e_0.m(\overline{e})$, the interpreter first evaluates the expression e_0 to the value v_0 and the updated store σ_1. It then evaluates the arguments $\overline{e}$ from left to right. Finally, it sets up the parameters and evaluates the body of the method, which is looked up from the class table based on the class tag of the receiver and the method name.

When evaluating a new expression new $C(\overline{e})$, the interpreter first evaluates the arguments $\overline{e}$ from left to right. It then creates an empty object with the class tag C and put it in the store. Next, it looks up the code of the class C from the class table, updates the empty object by assigning argument values to class parameters. Finally, it evaluates the field definitions in the class body from top to bottom. For each field definition var x : T = e, it first evaluates the expression e to a value, and then assigns the value to the field x of the object. The store is updated accordingly, and the environment is kept empty in evaluating field definitions because there are no local variables available.

When evaluating a block expression $e_1.x = e_2; e$, it first executes the assignment, then evaluates the expression e.

4.3 Type System

In addition to the types T that appear in the source code, we introduce *analysis types*, defined as follows:

$$\begin{aligned} T \in \text{Type} &= \text{mode} \times \text{ClassName} \\ R \in \text{Fields} &= \mathcal{P}(\text{Name}) \\ A \in \text{AnalysisType} &= \text{Type} \mid \text{Fields} \times \text{ClassName} \end{aligned}$$

An analysis type A is either a type T or a pair (R, C) of a set of initialized fields R and a class name C. The latter is used to type `this` in the class body; it tracks the fields of the object that are already initialized, and selection is only possible for those fields. These analysis types cannot be written by the user, but only generated by the simple flow-sensitive analysis we motivated in Section 2.7.

Lattice and Subtyping. The mode lattice and subtyping rules are presented in Figure 3. 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. The rule S-Widening widens the type of `this` (R, C) to the source-level type (cold, C), which enables safe escape of `this` during its initialization.

Definition Typing. The typing rules for definitions are presented in Figure 4. When type checking a program $(\overline{C}, e)$ with the rule T-Prog, it checks that every class is well-typed, and the top-level expression e is well-typed with the empty environment and the type Null for `this`. It also checks that the class Null is defined appropriately.

When type checking a class, the rule T-Class first checks the field definitions, with the assigned fields of `this` being the class parameters $\overline{x}$ initially, which are accumulated for each field definition. Then it checks that each method is well-typed.

When type checking a field definition `var x:T = e`, the rule T-Field ensures that the expression e can be typed as T in an empty environment. Finally, the rule augments the initialized field set R with x.

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 $\overline{x{:}T}$ and assuming `this` to take the mode of the method.

Types

$$\begin{array}{lcl} T & ::= & (\mu, C) \\ R & ::= & \{x, y, \cdots\} \\ A & ::= & T \mid (R, C) \end{array}$$

Mode Lattice

$$\text{hot} \sqsubseteq \text{warm} \sqsubseteq \text{cold}$$

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

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

Subtyping $\boxed{A_1 <: A_2}$

$$A <: A \qquad \text{(S-Refl)}$$

$$\frac{A_1 <: A_2 \qquad A_2 <: A_3}{A_1 <: A_3} \qquad \text{(S-Trans)}$$

$$\frac{\mu_1 \sqsubseteq \mu_2}{(\mu_1, C) <: (\mu_2, C)} \qquad \text{(S-Mode)}$$

$$(R, C) <: (\text{cold}, C) \qquad \text{(S-Widening)}$$

Fig. 3. Lattice and Subtyping

Expression Typing. The typing rules for expressions are presented in Figure 5. The typing judgements for expressions have the form $CT; \Gamma; A_1 \vdash e : A_2$, which means that the expression e can take the type A_2 assuming the class table CT, the environment Γ, and the type A_1 as the type of `this` .

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

The rule T-Sub is standard in type systems with subtyping. The rule T-Var looks up the type of a variable from the environment, and the rule T-This assumes the type A 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. The rule T-SelObj says that only initialized fields of `this` can be selected during its initialization, and the result takes the type of the field.

When check 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 $\mu = (\sqcup\ \mu_i) \sqcap \text{warm}$. Note that this rule is more expressive than the following rule:

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

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

$$\frac{CT = \overline{C \rightarrow \mathcal{C}} \qquad CT(Null) = \text{class Null} \qquad CT; \emptyset; (hot, Null) \vdash e : T \qquad \overline{CT \vdash \mathcal{C}}}{\vdash (\overline{C}, e)} \quad \text{(T-Prog)}$$

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

$$\frac{R_0 = \overline{x} \qquad CT; (R_i, C) \vdash \mathcal{F}_i; R_{i+1} \qquad \overline{CT; C \vdash \mathcal{M}}}{CT \vdash \text{class } C(\overline{x{:}T}) \{ \overline{\mathcal{F}}\ \overline{\mathcal{M}} \}} \quad \text{(T-Class)}$$

Field Typing $\boxed{CT; \Gamma; (R_1, C) \vdash \mathcal{F}; R_2}$

$$\frac{CT; \emptyset; (R, C) \vdash e : T}{CT; (R, C) \vdash \text{var } x : T = e; R \cup \{x\}} \quad \text{(T-Field)}$$

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

$$\frac{CT; \overline{x{:}T}; (\mu, C) \vdash e : S}{CT; C \vdash @\mu \text{ def } m(\overline{x{:}T}) : S = e} \quad \text{(T-Method)}$$

Fig. 4. Definition Typing

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

When checking a block expression $e_1.x = e_2; e$, 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.x 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 refers to a variable before it is declared; (2) it enforces modularity of initialization, as all fields of a class have to be assigned inside the class.

4.4 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 }
```

```
1 class Child(parent: Parent @cold) {
2   var tag: Int = 10
3 }
```

The main steps of type derivation for type checking the class `Parent` are given below (we use `P` for `Parent` and `C` for `Child`):

Expression Typing $\boxed{\mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{e} : \mathsf{A}}$

$$\frac{\mathsf{CT}; \Gamma; \mathsf{A}_1 \vdash \mathsf{e} : \mathsf{A}_2 \qquad \mathsf{A}_2 <: \mathsf{A}_3}{\mathsf{CT}; \Gamma; \mathsf{A}_1 \vdash \mathsf{e} : \mathsf{A}_3} \quad \text{(T-Sub)}$$

$$\frac{\mathsf{x} : \mathsf{T} \in \Gamma}{\mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{x} : \mathsf{T}} \quad \text{(T-Var)}$$

$$\mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{this} : \mathsf{A} \quad \text{(T-This)}$$

$$\frac{\mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{e} : (\mathsf{hot}, \mathsf{D}) \qquad (\_, \mathsf{C}) = \mathsf{fieldType}(\mathsf{CT}, \mathsf{D}, \mathsf{x})}{\mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{e.x} : (\mathsf{hot}, \mathsf{C})} \quad \text{(T-SelHot)}$$

$$\frac{\mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{e} : (\mathsf{warm}, \mathsf{D}) \qquad \mathsf{T} = \mathsf{fieldType}(\mathsf{CT}, \mathsf{D}, \mathsf{x})}{\mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{e.x} : \mathsf{T}} \quad \text{(T-SelWarm)}$$

$$\frac{\mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{e} : (\mathsf{R}, \mathsf{D}) \qquad \mathsf{x} \in \mathsf{R} \qquad \mathsf{T} = \mathsf{fieldType}(\mathsf{CT}, \mathsf{D}, \mathsf{x})}{\mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{e.x} : \mathsf{T}} \quad \text{(T-SelObj)}$$

$$\frac{\begin{array}{c}\overline{\mathsf{T}} = \mathsf{constrType}(\mathsf{CT}, \mathsf{C}) \qquad \mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{e}_\mathsf{i} : (\mu_\mathsf{i}, \mathsf{C}_\mathsf{i}) \\ (\mu_\mathsf{i}, \mathsf{C}_\mathsf{i}) <: \mathsf{T}_\mathsf{i} \qquad \mu = (\sqcup\, \mu_\mathsf{i}) \sqcap \mathsf{warm}\end{array}}{\mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{new}\ \mathsf{C}(\overline{\mathsf{e}}) : (\mu, \mathsf{C})} \quad \text{(T-New)}$$

$$\frac{\begin{array}{c}\mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{e} : (\mu_0, \mathsf{C}) \qquad (\mu_\mathsf{m}, \overline{\mathsf{T}}, (\mu_\mathsf{r}, \mathsf{D})) = \mathsf{methodType}(\mathsf{CT}, \mathsf{C}, \mathsf{m}) \\ \mu_0 \sqsubseteq \mu_\mathsf{m} \qquad \mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{e}_\mathsf{i} : (\mu_\mathsf{i}, \mathsf{D}_\mathsf{i}) \qquad (\mu_\mathsf{i}, \mathsf{D}_\mathsf{i}) <: \mathsf{T}_\mathsf{i} \qquad \mu = \mathsf{if}(\sqcup\, \mu_\mathsf{i} = \mathsf{hot})\ \mathsf{hot}\ \mathsf{else}\ \mu_\mathsf{r}\end{array}}{\mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{e.m}(\overline{\mathsf{e}}) : (\mu, \mathsf{D})} \quad \text{(T-Invoke)}$$

$$\frac{\mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{e}_1\mathsf{.x} : (\_, \mathsf{C}) \qquad \mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{e}_2 : (\mathsf{hot}, \mathsf{C}) \qquad \mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{e} : \mathsf{A}_1}{\mathsf{CT}; \Gamma; \mathsf{A} \vdash \mathsf{e}_1\mathsf{.x} = \mathsf{e}_2; \mathsf{e} : \mathsf{A}_1} \quad \text{(T-Block)}$$

Fig. 5. Expression Typing

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathsf{CT}; \emptyset; (\emptyset, \mathsf{P}) \vdash \mathsf{this} : (\emptyset, \mathsf{P}) \qquad (\emptyset, \mathsf{P}) <: \mathsf{P@cold}}{\mathsf{CT}; \emptyset; (\emptyset, \mathsf{P}) \vdash \mathsf{this} : \mathsf{P@cold}}\ \text{(T-Sub)} \qquad \mu = (\sqcup\mathsf{cold}) \sqcap \mathsf{warm} = \mathsf{warm}}{\mathsf{CT}; \emptyset; (\emptyset, \mathsf{P}) \vdash \mathsf{new}\ \mathsf{C}(\mathsf{this}) : \mathsf{P@warm}}\ \text{(T-New)}}{\mathsf{CT}; \emptyset; (\emptyset, \mathsf{P}) \vdash \mathsf{var}\ \mathsf{c} : \mathsf{C@warm} = \mathsf{new}\ \mathsf{C}(\mathsf{this}); \{\mathsf{c}\}}\ \text{(T-Field)}}{\mathsf{CT} \vdash\ \mathsf{class}\ \mathsf{P}\ \{\ \mathsf{var}\ \mathsf{c} : \mathsf{C@warm} = \mathsf{new}\ \mathsf{C}(\mathsf{this})\ \}}\ \text{(T-Class)}$$

4.5 Soundness

To show that the type system is sound, we follow the approach of definitional interpreters [Amin and Rompf 2017]. In order to reason about the definitional interpreter formally as a function, it has to be total. To make it total, we follow the approach by Amin and Rompf [2017] to (a) introduce a fuel k and a purely functional way to represent timeouts; and (b) use an `option` monad to allow for evaluation errors.

Therefore, we modify the return type of interpreter functions from `T` to `nat ->option (option T)`. Functions are made structurally recursive on the fuel. When receiving no fuel, no recursive calls are possible, and functions return `None` instead to signal failure. Evaluation errors are signaled by returning `Some None`. Finally, successful evaluation results are signaled by returning `Some (Some r)` for some evaluation result `r`. In Coq, the general structure of the adapted interpreter would look as follows:

```
Fixpoint evalP (p: Prog) (k: nat): option (option (Value, Store)) :=
  eval (expOf p) (toCT prog) emptyEnv initStore topThis k

Fixpoint eval (e: Expr) (ct: CT) (env: Env) (s: Store) (v: Value) (k: nat):
option (option (Value, Store)) :=
  match k with
  | Z => None
  | S n' =>
    (* Evaluate e; recursive calls must use n' as fuel *)
    (* For error, return*) Some None
    (* For success, return*) Some (Some (v, s))
  end.
```

We have to refactor the definitional interpreter defined in Figure 2 to thread the errors and the fuel. We omit the straightforward details here and refer readers to Amin and Rompf [2017].

The soundness proof depends on the following definitions. We let Σ range over store typings, that is finite maps from locations to types:

$$\Sigma \in \text{StoreTyping} \quad = \quad \text{Loc} \rightharpoonup \text{Type}$$

We next define typing judgements on the runtime state. For simplicity, we omit the class table, which is threaded unchanged throughout these judgments as needed.

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

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

Store typing ordering $\boxed{\Sigma_1 \preccurlyeq \Sigma_2}$

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

Value typing $\boxed{\Sigma \vDash v : A}$

$$\frac{\Sigma(v) = A_1 \qquad A_1 <: A_2}{\Sigma \vDash v : A_2}$$

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

$$\emptyset \mid \Sigma \vDash \emptyset$$

$$\frac{\Gamma \mid \Sigma \vDash \rho \qquad \Sigma \vDash v : A}{\Gamma, x{:}A \mid \Sigma \vDash \rho, x{:}v}$$

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

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

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

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

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

Reachable $\boxed{\mathsf{reachable}(\sigma, l_1, l_2)}$

$$\frac{l \in \sigma}{\mathsf{reachable}(\sigma, l, l)}$$

$$\frac{l_1 \in \sigma \qquad l_2 \in \sigma \qquad (\_, o) = \sigma(l_1) \qquad \exists f.\ o(f) = l_2}{\mathsf{reachable}(\sigma, l_1, l_2)}$$

$$\frac{\mathsf{reachable}(\sigma, l_1, l_2) \qquad \mathsf{reachable}(\sigma, l_2, l_3)}{\mathsf{reachable}(\sigma, l_1, l_3)}$$

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 typings would run into cycles, and introducing store typings is a standard solution [Pierce 2002, 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* relative to the store typing ordering (Lemmas 4.1, 4.2 and 4.3). As usual, if $\Sigma_1 \preccurlyeq \Sigma_2$, then the updated store typing Σ_2 can have more entries than Σ_1. Unlike in Pierce [2002], we must also allow types to be updated to types that are smaller according to subtyping: this is needed to allow for objects to become hot, as we'll see later.

The value typing judgement $\Sigma \vDash v : A$ 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 $\Gamma \mid \Sigma \vDash \rho$ 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.

Finally we get to object typing. The object typing judgement $\Sigma \vDash o : A$ checks that the object o is well-typed according to the semantics of thermal states. For example, to check that an object is hot, it verifies that every field of the object is hot.

Proofs start with monotonicity of store typing.

Lemma 4.1 (Value Typing Monotonicity). *For all* Σ_1, Σ_2, A *and value* v, *if* $\Sigma_1 \preccurlyeq \Sigma_2$ *and* $\Sigma_1 \vDash v : A$, *then* $\Sigma_2 \vDash v : A$.

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

Lemma 4.2 (Environment Typing Monotonicity). *For all* $\Sigma_1, \Sigma_2, \Gamma$ *and environment* ρ*, if* $\Sigma_1 \preccurlyeq \Sigma_2$ *and* $\Gamma \mid \Sigma_1 \vDash \rho$*, then* $\Gamma \mid \Sigma_2 \vDash \rho$.

Proof. We use induction on $\Gamma \mid \Sigma_1 \vDash \rho$ to apply value typing monotonicity (Lemma 4.1) to each environment entry. □

Lemma 4.3 (Object Typing Monotonicity). *For all* Σ, Σ', A *and object* o*, if* $\Sigma \preccurlyeq \Sigma'$ *and* $\Sigma \vDash o : A$*, then* $\Sigma' \vDash o : A$.

Proof. By inversion on the given derivation of object typing, the definition of store typing ordering, and value typing monotonicity (Lemma 4.1). □

Lemma 4.4 (Hot Transitivity). *If* $\Sigma \vDash l : (\text{hot}, C)$ *and* $\Sigma \vDash \sigma$*, then for all* l'*,* $\text{reachable}(\sigma, l, l')$ *implies* $\Sigma(l') = (\text{hot}, \_)$.

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

The next lemma says that, if all references reachable from a warm value are warm, then all values reachable from the value are hot, thus the value itself is hot as well.

Lemma 4.5 (Collective Maturity). *For all* Σ, σ, l, C*, if the following conditions hold:*

(1) $\Sigma \vDash l : (\text{warm}, C)$,
(2) $\Sigma \vDash \sigma$,
(3) if for all l', $\text{reachable}(\sigma, l, l')$ *implies* $\Sigma \vDash l' : (\text{warm}, \sigma(l').\text{fst}))$,

then there exists Σ' *such that:*

(1) $\Sigma' \vDash \sigma$ *and* $\Sigma \preccurlyeq \Sigma'$ *and*
(2) for all l', $\text{reachable}(\sigma, l, l')$ *implies* $\Sigma' \vDash l' : (\text{hot}, \sigma(l').\text{fst}))$.

This lemma reflects an interesting phenomenon in object initialization: the full initialization point of a group of interdependent values is delayed until the last object in the group becomes *warm*. This is better illustrated by the following code:

```
1  class Bar(foo: Foo)
2  class Foo {
3    val bar: Bar = new Bar(this)
4  }
5  val foo = new Foo
```

In the code above, the object `foo` and `bar` are interdependent, each holds a reference to the other. The maturity point when `bar` becomes hot is delayed until all fields of `foo` are assigned, then both become hot at the same time. We call this phenomenon *collective maturity*. Depending on the program, there is no limit on how big this collection can be.

The phenomenon of collective maturity is ubiquitous, not only in object-oriented programming, but also in functional programming. In both Scheme and OCaml, there is a recursive bind construct `letrec`, which enables writing code similar to the following in Scala:

```
1  val f = (x: Int) => if (x > 0) g(x - 1) else 10
2  val g = (x: Int) => if (x > 0) f(x - 2) else 20
```

In the code above, the function values `f` and `g` are mutually dependent. They become fully initialized at the same time. Premature usage of them before collective maturity may lead to initialization problems.

This phenomenon is also noticed in Summers and Müller [2011]. We believe they must have a similar result in their meta-theory, though we cannot find an explicit formulation in neither the paper nor technical report [Summers and Mueller 2011]. We think the importance of the lemma should not be underestimated by the simplicity of its proof, but should be appreciated by its central role in a large proof and the insights it sheds on explaining common facts about the initialization of objects.

Proof. Define Σ' as follows:

$$\Sigma'(l') = \begin{cases} (\text{hot}, \sigma(l').\text{fst})) & \text{if reachable}(\sigma, l, l'), \\ \Sigma(l') & \text{otherwise} \end{cases}$$

It is straightforward to show that $\Sigma \preccurlyeq \Sigma'$ and that all values reachable from l can be typed as hot. We must then prove $\Sigma' \vDash \sigma$, that is, $\text{dom}(\sigma) = \text{dom}(\Sigma')$ (immediate) and $\forall l \in \sigma.\Sigma' \vDash \sigma(l) : \Sigma'(l)$. In words, we must show that all objects in σ are well-typed in Σ'. For any location l' that is not reachable from l, we have $\Sigma'(l') = \Sigma(l')$, so we need $\Sigma' \vDash \sigma(l') : \Sigma'(l')$, which follows from object typing monotonicity (Lemma 4.3).

If instead reachable(σ, l, l'), we must show $\Sigma' \vDash \sigma(l') : \Sigma'(l)$. That follows because:

- By the definition of Σ', all values l' reachable from l have the type $(\text{hot}, \sigma(l').\text{fst}))$, thus the fields of the object referred by l' are hot.
- From premises, the value l' is warm, thus all fields that are defined in its class are assigned.

□

Lemma 4.6 (Subsumption for Value Typing). *If* $\Sigma \vDash l : A_1$ *and* $A_1 <: A_2$, *then* $\Sigma \vDash l : A_2$.

Proof. Immediate: by inversion on value typing and transitivity of subtyping. □

Lemma 4.7 (Environment Regularity). *If* $\Gamma \mid \Sigma \vDash \rho$ *and* $CT; \Gamma; A \vdash x{:}T$, *then* $\Sigma \vDash \rho(x) : T$.

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

Lemma 4.8 (Hot Selection). *If* $\Sigma \vDash \sigma$ *and* $\Sigma \vDash l : (\text{hot}, C)$, *and* $\text{fieldType}(C, f) = (\_, D)$, *then* $\Sigma \vDash \sigma(l).f : (\text{hot}, D)$.

Proof. By the definition of value typing. □

Lemma 4.9 (This Selection). *If* $\Sigma \vDash \sigma$ *and* $\Sigma \vDash l : (R, C)$, $f \in R$ *and* $\text{fieldType}(C, f) = T$, *then* $\Sigma \vDash \sigma(l).f : T$.

Proof. By the definition of value typing. □

Lemma 4.10 (Warm Selection). *If* $\Sigma \vDash \sigma$ *and* $\Sigma \vDash l : (\text{warm}, C)$, *and* $\text{fieldType}(C, f) = T$, *then* $\Sigma \vDash \sigma(l).f : T$.

Proof. By the definition of value typing. □

Lemma 4.11 (Expression Soundness). *For all free variables below, if the following conditions hold:*

(1) $\forall C \in \text{dom}(CT).CT \vdash CT(C)$
(2) $CT; \Gamma; A \vdash e : A_1$
(3) $\Gamma \mid \Sigma \vDash \rho$
(4) $\Sigma \vDash \sigma$
(5) $\Sigma \vDash \psi : A$
(6) $\text{eval } e\ CT\ \rho\ \sigma\ \psi\ k = \text{Some result}$

then we have the following:

(a) result = Some (l, σ')
(b) *there exists* $\Sigma', \Sigma \preccurlyeq \Sigma'$ *and* $\Sigma' \vDash \sigma'$
(c) $\Sigma' \vDash l : A_1$
(d) *for all* l_1, *if* reachable(σ', l, l_1) *and* $\Sigma' \nvDash l_1 : (\text{warm}, \sigma'(l_1).\text{fst}))$, *then* $\exists l_2 \in \text{codom}(\rho) \cup \{\psi\}$ *such that* reachable(σ, l_2, l_1).

While this lemma is standard, the main insight is to have (d) in the lemma, which is essential to justify the typing rules T-New and T-Invoke for new-expressions and method calls. Intuitively, the proposition (d) says that any actual cold values (not warm, not hot) reachable from the resulting value of an expression must pre-exist in the heap and be reachable from the environment for evaluating the expression.

Proof. By induction on k. The case k = 0 trivially holds. For the case k = n + 1, perform case analysis on the typing judgement (2).

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 *eval e CT* ρ σ ψ *k* itself does not time out.

Also in each case, evaluating subexpression e_i updates the store typing Σ_i to Σ_{i+1} (where $\Sigma_0 = \Sigma$), so we must use typing monotonicity (Lemmas 4.1, 4.2) to show premises (3) and (5) still hold in the updated Σ_{i+1}.

- **T-Sub**. (c) follows from Lemma 4.6, other goals follow from induction hypothesis.
- **T-Var**. (c) follows from Lemma 4.7. (d) holds by chosing $l_2 = \rho(x)$.
- **T-This**. (c) follows from (5) and the case for 'this' in 'eval'. (d) holds by chosing $l_2 = \psi$.
- **T-SelHot**. (c) follows from Lemma 4.8. (d) holds by chosing l_2 to be the same value for e in e.x.
- **T-SelWarm**. similar as the case above.
- **T-SelObj**. (c) follows from Lemma 4.9. (d) holds by chosing l_2 to be the same value for e in e.x.
- **T-Block**. The key here is to argue that after the assignment $e_1.x = e_2$, we have $\Sigma' \vDash \sigma'$ for the updated store typing Σ' and store σ'. This is guaranteed by the fact that e_2 is hot, and a hot value can take any type corresponding to its class. (c) and (d) follows trivially from the induction hypothesis for the sub-expression e.
- **T-Invoke**. From the well-typing of args and e in e.m(args), we can apply the induction hypothesis repeatedly to get the appropriate $\Sigma', \rho', \sigma', \psi'$ (Lemma 4.6 is used for subtyping of ψ'). From the well-typing of the method m, we can apply the induction hypothesis for the method body again. (d) holds by tracing back the existentials.
  The subtlety is to argue that *the result* l *is hot if the receiver and arguments are hot.* First, as $\text{codom}(\rho) \cup \{\psi\}$ are all hot, by Lemma 4.4 and (d) we know that all values reachable from l (including itself) are not cold (or at least warm). Now use Lemma 4.5 we conclude that the result value l is hot.
- **T-New**. First, if the result of new C(args) is (warm, C), (c) follows from the well-typing of class fields — all fields are required to be assigned by T-Class.
  The complexity here is to justify the result (hot, C) when args are all hot. Note that the evaluation of new C(args) requires the execution of the n field assignments at the start of the class body, i.e. $\overline{\text{var f : T = e}}$. The key insight is that *in each of the field assignment* var f : T = e, *all objects reachable from* this *are warm except itself.* The proof is by induction on the number of fields. If num(fields) = 0, it trivially holds, as this only point to hot values and all values reachable from hot values are hot (Lemma 4.4). If num(fields) = n + 1, the evaluation for e

in var f : T = e uses an empty environment and `this`, thus all non-warm values reachable from the value of e will be reachable from `this`. By induction hypothesis, we know the only reachable non-warm value is `this` itself, so after the field assignment the invariant continues to hold. Now, when the last field assignment is executed, we know `this` is warm, so all reachable values from `this` are warm. Now using Lemma 4.5, we conclude the new object value is hot in a suitable Σ'.

□

Corollary 4.12 (Soundness). *If* $\vdash \mathcal{P}$, *and for any k, if* evalP $\mathcal{P}$ k = Some r, *then* r ≠ None.

Proof. It follows trivially from Lemma 4.11. □

5 EXPERIMENTS

We extended and implemented a prototype of the Celsius model as a compiler phase in Dotty, the next generation compiler for Scala. We evaluated the prototype implementation on several real-world Scala projects, the result is presented in Table 2.

Project	KLOC	Warnings (w/o inf.)	Warnings (inf.)	+/- (LOC)	@unchecked
Dotty	68.2K	1165	383	120	31
ScalaPB	30.5K	19	20	24	8
ScalaLib	31.9K	24	11	7	0
Squants	14.0K	0	0	0	0
minitest	0.8K	0	0	0	0
better-files	3.2K	0	0	0	0
algebra	3.3K	6	6	6	6
ScalaCheck	5.9K	81	3	3	2
ScalaTest	394K	105	54	32	13
scopt	1.9K	2	0	0	0
fastparse	16.7K	8	0	0	0

Table 2. Experiment Results. With inference of annotations, warnings are reduced significantly. The column +/- shows the changes required to suppress all the warnings. The last column shows the number of @uncheck annotations used. All projects are well known in the Scala community, ScalaLib refers to the new Scala collection.

The version of prototype without annotation inference produces a large amount of warnings (1165) for the Dotty project, of which 83.6% are due to unsafe parent method calls. The distribution of warnings by category is shown in Figure 6. From the figure, we can see the largest five category of warnings are: (1) call parent methods; (2) partially initialized fields (due to capturing) that cannot be safely used from child classes; (3) dynamic method call; (4) leak of `this`; (5) leak of inner class instances. The version of prototype with annotation inference removes a lot of warnings from the first three categories.

Despite the fact that the number of changes required to suppress warnings are relatively small, we do need to resort to the backdoor annotation `@unchecked` in several projects. A major use of `@unchecked` in Dotty is that the following code pattern is quite common:

```
1  class RecType(parentExp: RecType => Type) {
2    val parent = parentExp(this)
3  }
```

For safety, we could change the type of `parentExp` to `RecType @cold => Type`, however, then we would have to change class parameter of most types to accept a `@cold` value, which is tedious. In ScalaTest, we have to use `@unchecked` for three GUI classes, which call parent methods defined in an external GUI library to specify the graphical design. In ScalaPB, it is mainly due to leaks of partially initialized objects. In algebra, it is about calling parent methods defined in an external library.

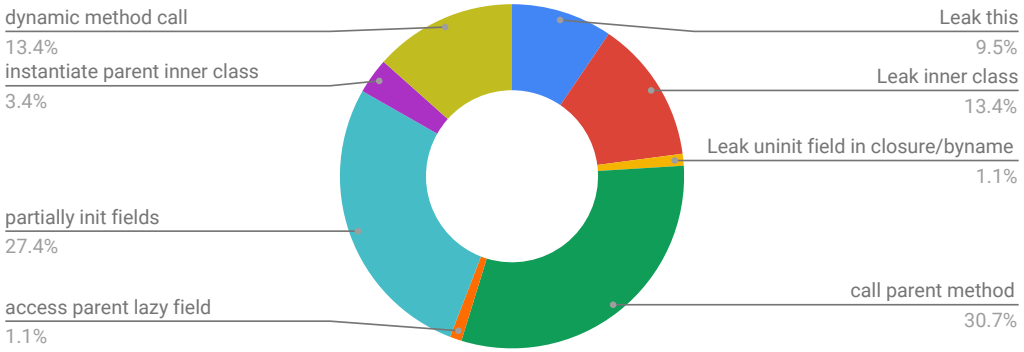


Fig. 6. Distribution of Warnings. The statistics is based on data from the experiment projects excluding Dotty, as the latter is atypically dominated by parent method calls (83.6%).

Our prototype detected 4 instances of a bug in the ScalaTest project, where an uninitialized boolean field is used in a parent method. One reason the error is not caught by the extensive test set of ScalaTest (> 300KLOC) is that accessing an uninitialized boolean field will not cause runtime exception[4].

6 RELATED WORK

In the previous sections, we compared our model to the Freedom model. We discuss more related work here as well as an in-depth discussion about the Freedom model.

Masked Types. As far as we know, masked types [Qi and Myers 2009] is the most expressive approach to safe initialization. It works by tracking uninitialized fields of objects in the type system. The system is formulated in the style of an effect system, where the initialization effects of methods are annotated explicitly. This involves verbose type annotations and it incurs cognitive burden for programmers due to concepts such us *conditionally masked types* and *abstract masks*.

Freedom before Commitment. As mentioned in the introduction, our thermal model is inspired by *Freedom before Commitment* [Summers and Müller 2011]. In the meta-theory of Summers and Müller [2011], it 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. And in their soundness proof there is a similar proposition like the proposition d in our Lemma 4.11. To some extent, this correspondence evidences that *warm* is an important concept from a semantic perspective. Our work shows it is worth introducing this concept to programmers.

The concept *deep_init* in their meta-theory means that all objects reachable from the current object are *locally initialized*, which is related to our Lemma 4.5. We believe there is a similar result like our Lemma 4.5 in their meta-theory, though it is not explicitly formulated as a lemma.

The Checker Framework. The Checker Framework enables pluggable type systems for Java [Papi et al. 2008]. The framework has enabled many useful checkers for various properties of

[4]The bug report can be found at: *URL removed for anonymity.*

Java programs [Ernst and Ali 2010]. In particular, it implements the Freedom model; the annotation `@Initialized` corresponds to *committed*, `@UnderInitialization` corresponds to *free* and `@UnknownInitialization` corresponds to *unclassified* [5]. It extends the Freedom model with annotations like `@UnderInitialization(C.class)` and `@UnknownInitialization(C.class)` to indicate that all fields of class `C` (and its super classes) are initialized. For soundness, a field annotation `@NotOnlyInitialized` is introduced to mean that the field may hold a value that is not fully initialized. The semantics of `@UnknownInitialization(C.class)` is similar to *warm* in the Celsius model. The extended model is expressive, but it is a little complex and not yet formalized. From the available information in the manual of Checker Framework, it is unclear to us whether the extended model supports composability of *warm*: a field selection on a warm value may return another warm value. Nevertheless, our work can be seen as an attempt to capture the essence of the extension in a simplified model. We believe an extension based on our model will make the system simpler and more expressive.

X10. X10 [Zibin et al. 2012] uses inter-procedural analysis to check safe initialization. The analysis follows final or private method calls from the constructor and dynamic-dispatching calls are allowed for methods annotated with `@NoThisAccess`. X10 forbids instantiation of inner classes, capture of `this` in closures or leaking `this` out of the constructor during object initialization. While that design makes sense in the setting of X10 which targets concurrent and distributed computation, it is overly restrictive for a general-purpose programming language.

Raw Types and Delayed Types. Raw types [Fähndrich and Leino 2003] is a simple model that has only one modifier *raw*, which roughly corresponds to *cold* in our model. For example, `raw(C)` means the fields declared in `C` may not be assigned. However, raw types are not expressive enough, which motivated the work on delayed types [Fähndrich and Xia 2007]. Delayed types introduce delayed scopes `delay t { B }` to mark the initialization duration of fresh objects. As mentioned in Summers and Müller [2011], the system presented in the paper is too complex and in the compiler a simple but unsound version is implemented. The situation motivated the Freedom model.

The Billion-Dollar Fix. Servetto et al. [2013] introduces a new language construct called *placeholders* for safe initialization of mutually dependent objects. They give operational semantics for placeholders as well as introduce *placeholder types* to ensure that placeholders are used safely. Placeholder types basically forbids usage of placeholder values until it is statically known that they are replaced by references to actual objects. This work is orthognoal to ours and that of Summers and Müller [2011], as we are not aiming to introduce new constructs to programming languages.

REFERENCES

Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *POPL*.

Joshua Bloch. 2008. *Effective Java (2nd Edition) (The Java Series)* (2 ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

Michael D. Ernst and Mahmood Ali. 2010. Building and using pluggable type systems. In *SIGSOFT FSE*.

Manuel Fähndrich and K. Rustan M. Leino. 2003. Declaring and checking non-null types in an object-oriented language. In *OOPSLA*.

Manuel Fähndrich and Songtao Xia. 2007. Establishing object invariants with delayed types. In *OOPSLA*.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23 (2001), 396–450.

Martin Odersky et al. 2013. Dotty Compiler: A Next Generation Compiler for Scala. https://web.archive.org/web/20170325001401/http://dotty.epfl.ch/.

Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jeff H. Perkins, and Michael D. Ernst. 2008. Practical pluggable types for Java. In *ISSTA*.

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). MIT Press.

Xin Qi and Andrew C. Myers. 2009. Masked Types for Sound Object Initialization. In *POPL (POPL '09)*. ACM, 53–65.

[5] https://checkerframework.org/manual/#initialization-checker

Marco Servetto, Julian Mackay, Alex Potanin, and James W Noble. 2013. The Billion-Dollar Fix - Safe Modular Circular Initialisation with Placeholders and Placeholder Types. In *ECOOP*.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *POPL*.

Alexander J. Summers and Peter Mueller. 2011. Freedom Before Commitment: Simple Flexible Initialisation for Non-Null Types *(Technical Report 716)*. ETH Zurich.

Alexander J. Summers and Peter Müller. 2011. Freedom Before Commitment: A Lightweight Type System for Object Initialisation. In *OOPSLA (OOPSLA '11)*. ACM, New York, NY, USA, 1013–1032. https://doi.org/10.1145/2048066.2048142

Yoav Zibin, David Cunningham, Igor Peshansky, and Vijay Saraswat. 2012. Object initialization in X10. In *European Conference on Object-Oriented Programming*. Springer, 207–231.