OOlong: A Concurrent Object Calculus for Extensibility and Reuse

Elias Castegren
KTH Royal Institute of Technology
Kista, Sweden
eliasca@kth.se

Tobias Wrigstad
Uppsala University
Uppsala, Sweden
tobias.wrigstad@it.uu.se

ABSTRACT

We present OOlong, an object calculus with interface inheritance, structured concurrency and locks. The goal of the calculus is extensibility and reuse. The semantics are therefore available in a version for LaTeX typesetting (written in Ott), a mechanised version for doing rigorous proofs in Coq, and a prototype interpreter (written in OCaml) for typechecking an running OOlong programs.

CCS Concepts

•**Theory of computation** → **Operational semantics;** *Concurrency;* Interactive proof systems; •**Software and its engineering** → *Object oriented languages; Concurrent programming structures; Interpreters;*

Keywords

Object Calculi; Semantics; Mechanisation; Concurrency

1. INTRODUCTION

When reasoning about object-oriented programming, object calculi are a useful tool for abstracting away many of the complicated details of a full-blown programming language. They provide a context for prototyping in which proving soundness or other interesting properties of a language is doable with reasonable effort.

The level of detail depends on which concepts are under study. One of the most used calculi is Featherweight Java, which models inheritance but completely abstracts away mutable state [14]. The lack of state makes it unsuitable for reasoning about any language feature which entails object mutation, and many later extensions of the calculus re-adds state as a first step. Other proposals have also arisen as contenders for having "just the right level of detail" [3, 18, 26].

This paper introduces OOlong, a small, imperative object calculus for the multi-core age. Rather than modelling a specific language, OOlong aims to model object-oriented programming in general, with the goal of being extensible and reusable. To keep subtyping simple, OOlong uses interfaces and omits class inheritance and method overriding. This avoids tying the language to a specific model of class inheritance (*e.g.,* Java's), while still maintaining an object-oriented style of programming. Concurrency is modeled in a finish/async style, and synchronisation is handled via locks.

The semantics are provided both on paper and in a mechanised version written in Coq. The paper version of OOlong is defined in Ott [25], and all typing rules in this paper are generated from this definition. To make it easy for other researchers to build on OOlong, we are making the sources of both versions of the semantics publicly available. We also provide a prototype interpreter written in OCaml.

With the goal of extensibility and re-usability, we make the following contributions:

- We define the formal semantics of OOlong, motivate the choice of features, and prove type soundness (Sections 2–5).
- We provide a mechanised version of the full semantics and soundness proof, written in Coq (Section 6).
- We provide Ott sources for easily extending the paper version of the semantics and generating typing rules in LaTeX (Section 7).
- We give three examples of how OOlong can be extended; support for assertions, more fine-grained locking based on regions, and type-level tracking of null references (Section 8).
- We present the implementation of a simple prototype interpreter to allow automatic type checking and evaluation of OOlong programs. It provides a starting point for additional prototyping of extensions (Section 9).

This paper is an extended version of previous work [7]. Apart from minor additions and improvements, the section on the mechanised semantics has been expanded (Section 6). The extension to track null references through types is new (Section 8.3), and the prototype OOlong interpreter has not been described before (Section 9).

2. RELATED WORK

The main source of inspiration for OOlong is Welterweight Java by Östlund and Wrigstad [18], a concurrent core calculus for Java with ease of reuse as an explicit goal. Welterweight

	FJ	ClJ	ConJ	MJ	LJ	WJ	**OOlong**
State		×	×	×	×	×	×
Statements				×	×	×	
Expressions	×	×	×	×			×
Class Inheritance	×	×	×	×	×	×	
Interfaces		×					×
Concurrency			×			×	×
Stack				×		×	
Mechanised	×*				×		×
LaTeX sources					×	×	×

Figure 1: A comparison between Featherweight Java, `ClassicJava`, `ConcurrentJava`, Middleweight Java, Lightweight Java, Welterweight Java and OOlong.

Java is also defined in Ott, which facilitates simple extension and LaTeX typesetting, but only exists as a calculus on paper. There is no online resource for accessing the Ott sources, and no published proofs except for the sketches in the original treatise. OOlong provides Ott sources and is also fully mechanised in Coq, increasing reliability. Having a proof that can be extended along with the semantics also improves re-usability. Both the Ott sources and the mechanised semantics are publicly available online [5]. OOlong is more lightweight than Welterweight Java by omitting mutable variables and using a single flat stack frame instead of modelling the call stack. Also, OOlong is expression-based whereas Welterweight Java is statement-based, making the OOlong syntax more flexible. We believe that all these things make OOlong easier to reason and prove things about, and more suitable for extension than Welterweight Java.

Object calculi are used regularly as a means of exploring and proving properties about language semantics. These calculi are often tailored for some special purpose, *e.g.,* the calculus of dependent object types [1], which aims to act as a core calculus for Scala, or OrcO [19], which adds objects to the concurrent-by-default language Orc. While these calculi serve their purposes well, their tailoring also make them fit less well as a basis for extension when reasoning about languages which do not build upon the same features. OOlong aims to act as a calculus for common object-oriented languages in order to facilitate reasoning about extensions for such languages.

2.1 Java-based Calculi

There are many object calculi which aim to act as a core calculus for Java. While OOlong does not aim to model Java, it does not actively avoid being similar to Java. A Java programmer should feel comfortable looking at OOlong code, but a researcher using OOlong does not need to use Java as the model. Figure 1 surveys the main differences between different Java core calculi and OOlong. In contrast to many of the Java-based calculi, OOlong ignores inheritance between classes and instead uses only interfaces. While inheritance is an important concept in Java, we believe that subtyping is a much more important concept for object-oriented programming in general. Interfaces provide a simple way to achieve subtyping without having to include concepts like overriding. With interfaces in place, extending the calculus to model other inheritance techniques like mixins [12] or traits [24] becomes easier.

The smallest proposed candidate for a core Java calculus is probably Featherweight Java [14], which omits all forms of assignment and object state, focusing on a functional core of Java. While this is enough for reasoning about Java's type system, the lack of mutable state precludes reasoning about object-oriented programming in a realistic way. Extensions of this calculus often re-add state as a first step (*e.g.,* [2, 17, 23]). The original formulation of Featherweight Java was not mechanised, but a later variation omitting casts and introducing assignment was mechanised in Coq (∼2300 lines) [17]. When developing mixins, Flatt *et al.* define `ClassicJava` [12], an imperative core Java calculus with classes and interfaces. It has been extended several times (*e.g.,* [9, 27]). Flanagan and Freund later added concurrency and locks to `ClassicJava` in `ConcurrentJava` [11], but omitted interfaces. To the best of our knowledge, neither `ClassicJava` nor `ConcurrentJava` have been mechanised.

Bierman *et al.* define Middleweight Java [3], another imperative core calculus which also models object identity, null pointers, constructors and Java's block structure and call stack. Middleweight Java is a true subset of Java, meaning that all valid Middleweight Java programs are also valid Java programs. The high level of detail however makes it unattractive for extensions which are not highly Java-specific. To the best of our knowledge, Middleweight Java was never mechanised. Strniša proposes Lightweight Java as a simplification of Middleweight Java [26], omitting block scoping, type casts, constructors, expressions, and modelling of the call stack, while still being a proper subset of Java. Like Welterweight Java it is purely based on statements, and does not include interfaces. Like OOlong, Lightweight Java is defined in Ott, but additionally uses Ott to generate a mechanised formalism in Isabelle/HOL. A later extension of Lightweight Java was also mechanised in Coq (∼800 lines generated from Ott, and another ∼5800 lines of proofs) [10].

Last, some language models go beyond the surface language and execution. One such model is Jinja by Klein and Nipkow [16], which models (parts of) the entire Java architecture, including the virtual machine and compilation from Java to byte code. To handle the complexity of such a system, Jinja is fully mechanised in Isabelle/HOL. The focus of Jinja is different than that of calculi like OOlong, and is therefore not practical for exploring language extensions which do not alter the underlying runtime.

2.2 Background

OOlong started out as a target language acting as dynamic semantics for a type system for concurrency control [6]. The proof schema for this system involved translating the source language into OOlong, establishing a mapping between the types of the two languages, and reasoning about the behaviour of a running OOlong program. In this context, OOlong was extended with several features, including assertions, readers–writer locks, regions, destructive reads and mechanisms for tracking which variables belong to which stack frames (Section 8 outlines the addition of assertions and regions). By having a machine checked proof of soundness for OOlong that we could trust, the proof of progress and preservation of the source language followed from showing that translation preserves well-formedness of programs.

$$
\begin{array}{lcll}
P & ::= & Ids\ Cds\ e & \textit{(Programs)} \\
Id & ::= & \textbf{interface}\ I\ \{Msigs\} & \textit{(Interfaces)} \\
 & | & \textbf{interface}\ I\ \textbf{extends}\ I_1, I_2 & \\
Cd & ::= & \textbf{class}\ C\ \textbf{implements}\ I\ \{Fds\ Mds\} & \textit{(Classes)} \\
Msig & ::= & m(x : t_1) : t_2 & \textit{(Signatures)} \\
Fd & ::= & f : t & \textit{(Fields)} \\
Md & ::= & \textbf{def}\ Msig\ \{e\} & \textit{(Methods)} \\
e & ::= & v \mid x \mid x.f \mid x.f = e & \textit{(Expressions)} \\
 & | & x.m(e) \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{new}\ C \mid (t)\ e & \\
 & | & \textbf{finish}\{\textbf{async}\{e_1\}\ \textbf{async}\{e_2\}\};\ e_3 & \\
 & | & \textbf{lock}(x)\ \textbf{in}\ e \mid \boxed{\textbf{locked}_\iota\{e\}} & \\
v & ::= & \textbf{null} \mid \boxed{\iota} & \textit{(Values)} \\
t & ::= & C \mid I \mid \textbf{Unit} & \textit{(Types)} \\
\Gamma & ::= & \epsilon \mid \Gamma, x : t \mid \boxed{\Gamma, \iota : C} & \textit{(Typing environment)}
\end{array}
$$

Figure 2: The syntax of OOlong.

3. STATIC SEMANTICS OF OOlong

In this section, we describe the static semantics of OOlong. The semantics are also available as Coq sources, together with a full soundness proof. The main differences between the paper version and the mechanised semantics are outlined in Section 6.

Figure 2 shows the syntax of OOlong. *Ids*, *Cds*, *Fds*, *Mds* and *Msigs* are sequences of zero or more of their singular counterparts. Terms in grey boxes are not part of the surface syntax but only appear during evaluation. The meta-syntactic variables are x, y and **this** for variable names, f for field names, C for class names, I for interface names, and m for method names. For simplicity we assume that all names are unique. OOlong defines objects through classes, which implement some interface. Interfaces are in turn defined either as a collection of method signatures, or as an "inheriting" interface which joins two other interfaces. There is no inheritance between classes, and no overriding of methods. A program is a collection of interfaces and classes together with a starting expression e. An example of a full OOlong program (extended to handle integers) can be found in Figure 10.

Most expressions are standard: values (**null** or abstract object locations ι), variables, field accesses, field assignments, method calls, object instantiation and type casts. For simplicity, targets of field and method lookups must be variables, and method calls have exactly one argument (multiple arguments can be simulated through object indirection, and an empty argument list by passing **null**). We also use **let**-bindings rather than sequences and variables. Sequencing can be achieved through the standard trick of translating $e_1; e_2$ into $\textbf{let}\ \_ = e_1\ \textbf{in}\ e_2$ (due to eager evaluation of e_1). Parallel threads are spawned with the expression $\textbf{finish}\{\textbf{async}\{e_1\}\ \textbf{async}\{e_2\}\};\ e_3$, which runs e_1 and e_2 in parallel, waits for their completion, and then continues with e_3.

The expression $\textbf{lock}(x)\ \textbf{in}\ e$ locks the object pointed to by x for the duration of e. While an expression locking the object at location ι is executed in the dynamic semantics, it appears as $\textbf{locked}_\iota\{e\}$. This way, locks are automatically released at the end of the expression e. It also allows tracking which field accesses are protected by locks and not.

$$
\boxed{\vdash P : t \quad \vdash Id \quad \vdash Cd \quad \vdash Fd \quad \vdash Md} \ \textit{(Well-formed program)}
$$

WF-PROGRAM
$$
\frac{\forall\, Id \in Ids.\ \vdash Id \qquad \forall\, Cd \in Cds.\ \vdash Cd \qquad \epsilon \vdash e : t}{\vdash Ids\ Cds\ e : t}
$$

WF-INTERFACE
$$
\frac{\forall\, m(x : t) : t' \in Msigs.\ \vdash t \wedge \vdash t'}{\vdash \textbf{interface}\ I\ \{\ Msigs\ \}}
$$

WF-INTERFACE-EXTENDS
$$
\frac{\vdash I_1 \qquad \vdash I_2}{\vdash \textbf{interface}\ I\ \textbf{extends}\ I_1, I_2}
$$

WF-CLASS
$$
\frac{\begin{array}{c}\forall\, m(x : t) : t' \in \textbf{msigs}\,(I).\textbf{def}\, m(x : t) : t'\ \{\ e\ \} \in Mds \\ \forall\, Fd \in Fds.\ \vdash Fd \qquad \forall\, Md \in Mds.\textbf{this} : C \vdash Md\end{array}}{\vdash \textbf{class}\ C\ \textbf{implements}\ I\ \{\ Fds\ Mds\ \}}
$$

WF-FIELD
$$
\frac{\vdash t}{\vdash f : t}
$$

WF-METHOD
$$
\frac{\textbf{this} : C, x : t \vdash e : t'}{\textbf{this} : C \vdash \textbf{def}\, m(x : t) : t'\ \{\ e\ \}}
$$

Figure 3: Well-formedness of classes and interfaces.

Types are class or interface names, or **Unit** (used as the type of assignments). The typing environment Γ maps variables to types and abstract locations to classes.

3.1 Well-Formed Program

Figure 3 shows the definition of a well-formed program, which consists of well-formed interfaces and well-formed classes, plus a well-typed starting expression. A non-empty interface is well-formed if its method signatures only mention well-formed types (WF-INTERFACE), and an inheriting interface is well-formed if the interfaces it extends are well-formed (WF-INTERFACE-EXTENDS). A class is well-formed if it implements all the methods in its interface (the helper function **msigs** is defined in the appendix, *cf.,* Section A.3). Further, all fields and methods must be well-formed (WF-CLASS). A field is well-formed if its type is well-formed (WF-FIELD). A method is well-formed if its body has the type specified as the method's return type under an environment containing the single parameter and the type of the current **this** (WF-METHOD).

3.2 Types and Subtyping

Figure 4 shows the rules relating to typing, subtyping, and the typing environment Γ. Each class or interface in the program corresponds to a well-formed type (T-WF-*). Subtyping is transitive and reflexive, and is nominally defined by the interface hierarchy of the current program (T-SUB-*). A well-formed environment Γ has variables of well-formed types and locations of valid class types (WF-ENV). Finally, the frame rule splits an environment Γ_1 into two sub-environments Γ_2 and Γ_3 whose variable domains are disjoint (but which may share locations ι). The meta-syntactic variable γ abstracts over variables x and locations ι (to reduce clutter), and the helper function **vardom** extracts the set of variables from an environment (*cf.,* Section A.3). The frame rule is used

$\boxed{\vdash t}$ *(Well-formed types)*

T-WF-CLASS
$$\frac{\textbf{class}\ C\ \textbf{implements}\ I\ \{\ \_\ \} \in P}{\vdash C}$$

T-WF-INTERFACE
$$\frac{\textbf{interface}\ I\ \{\ \_\ \} \in P}{\vdash I}$$

T-WF-INTERFACE-EXTENDS
$$\frac{\textbf{interface}\ I\ \textbf{extends}\ I_1, I_2 \in P}{\vdash I}$$

T-WF-UNIT
$$\frac{}{\vdash \textbf{Unit}}$$

$\boxed{t_1 <: t_2}$ *(Subtyping)*

T-SUB-CLASS
$$\frac{\textbf{class}\ C\ \textbf{implements}\ I\ \{\ \_\ \} \in P}{C <: I}$$

T-SUB-INTERFACE-LEFT
$$\frac{\textbf{interface}\ I\ \textbf{extends}\ I_1, I_2 \in P}{I <: I_1}$$

T-SUB-INTERFACE-RIGHT
$$\frac{\textbf{interface}\ I\ \textbf{extends}\ I_1, I_2 \in P}{I <: I_2}$$

T-SUB-TRANS
$$\frac{t_1 <: t_2 \qquad t_2 <: t_3}{t_1 <: t_3}$$

T-SUB-EQ
$$\frac{\vdash t}{t <: t}$$

$\boxed{\vdash \Gamma}$ *(Well-formed environment)*

WF-ENV
$$\frac{\forall\, x : t \in \Gamma.\ \vdash t \qquad \forall\, \iota : C \in \Gamma.\ \vdash C}{\vdash \Gamma}$$

$\boxed{\Gamma_1 = \Gamma_2 + \Gamma_3}$ *(Frame Rule)*

WF-FRAME
$$\frac{\begin{array}{c}\forall\, \gamma : t \in \Gamma_2.\Gamma_1(\gamma) = t \\ \forall\, \gamma : t \in \Gamma_3.\Gamma_1(\gamma) = t \\ (\textbf{vardom}\,(\Gamma_2) \cap \textbf{vardom}\,(\Gamma_3)) = \emptyset\end{array}}{\Gamma_1 = \Gamma_2 + \Gamma_3}$$

Figure 4: Typing, subtyping, and the typing environment.

when spawning new threads to prevent them from sharing variables[1].

3.3 Expression Typing

Figure 5 shows the typing rules for expressions, most of which are straightforward. Variables are looked up in the environment (WF-VAR) and introduced using **let** bindings (WF-LET). Method calls require the argument to exactly match the parameter type of the method signature (WF-CALL). We require explicit casts, and only support upcasts (WF-CAST). Fields are looked up with the helper function

[1]Since variables are immutable in OOlong, this kind of sharing would not be a problem in practice, but for extensions requiring mutable variables, we believe having this in place makes sense.

$\boxed{\Gamma \vdash e : t}$ *(Typing Expressions)*

WF-VAR
$$\frac{\vdash \Gamma \qquad \Gamma(x) = t}{\Gamma \vdash x : t}$$

WF-LET
$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x : t_1 \vdash e_2 : t}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : t}$$

WF-CALL
$$\frac{\Gamma \vdash x : t_1 \qquad \Gamma \vdash e : t_2 \qquad \textbf{msigs}\,(t_1)(m) = y : t_2 \to t}{\Gamma \vdash x.m(e) : t}$$

WF-CAST
$$\frac{\Gamma \vdash e : t' \qquad t' <: t}{\Gamma \vdash (t)e : t}$$

WF-SELECT
$$\frac{\Gamma \vdash x : C \qquad \textbf{fields}\,(C)(f) = t}{\Gamma \vdash x.f : t}$$

WF-UPDATE
$$\frac{\Gamma \vdash x : C \qquad \Gamma \vdash e : t \qquad \textbf{fields}\,(C)(f) = t}{\Gamma \vdash x.f = e : \textbf{Unit}}$$

WF-NEW
$$\frac{\vdash \Gamma \qquad \vdash C}{\Gamma \vdash \textbf{new}\ C : C}$$

WF-LOC
$$\frac{\vdash \Gamma \qquad \Gamma(\iota) = C \qquad C <: t}{\Gamma \vdash \iota : t}$$

WF-NULL
$$\frac{\vdash \Gamma \qquad \vdash t}{\Gamma \vdash \textbf{null} : t}$$

WF-FJ
$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma_1 \vdash e_1 : t_1 \qquad \Gamma_2 \vdash e_2 : t_2 \qquad \Gamma \vdash e : t}{\Gamma \vdash \textbf{finish}\ \{\ \textbf{async}\ \{\ e_1\ \}\ \textbf{async}\ \{\ e_2\ \}\ \};\ e : t}$$

WF-LOCK
$$\frac{\Gamma \vdash x : t_2 \qquad \Gamma \vdash e : t}{\Gamma \vdash \textbf{lock}(x)\,\textbf{in}\ e : t}$$

WF-LOCKED
$$\frac{\Gamma \vdash e : t \qquad \Gamma(\iota) = t_2}{\Gamma \vdash \textbf{locked}_\iota\{e\} : t}$$

Figure 5: Typing of expressions

fields (WF-SELECT). Fields may only be looked up in class types (as interfaces do not define fields). Field updates have the **Unit** type (WF-UPDATE). Any class in the program can be instantiated (WF-NEW). Locations can be given any super type of their class type given in the environment (WF-LOC). The constant **null** can be given any well-formed type, including **Unit** (WF-NULL). Forking new threads requires that the accessed variables are disjoint, which is enforced by the frame rule $\Gamma = \Gamma_1 + \Gamma_2$ (WF-FJ). Locks can be taken on any well-formed target (WF-LOCK*).

Section 9 introduces a bidirectional version of the typing rules which are entirely syntax-directed (meaning they can be directly implemented by a type checker) and which handle implicit upcasts, *e.g.,* for arguments to method calls.

4. DYNAMIC SEMANTICS OF OOlong

In this section, we describe the dynamic semantics of OOlong. Figure 6 shows the structure of the run-time constructs of OOlong. A configuration $\langle H; V; T\rangle$ contains a heap H, a variable map V, and a collection of threads T. A heap H maps abstract locations to objects. Objects store their class, a map F from field names to values, and a lock status L which is either **locked** or **unlocked**. A stack map V maps variable names to values. As variables are never updated, OOlong could use a simple variable substitution scheme

$$
\begin{array}{llll}
cfg & ::= & \langle H; V; T\rangle & \textit{(Configuration)} \\
H & ::= & \epsilon \mid H, \iota \mapsto obj & \textit{(Heap)} \\
V & ::= & \epsilon \mid V, x \mapsto v & \textit{(Variable map)} \\
T & ::= & (\mathcal{L}, e) \mid T_1 \,||\, T_2 \rhd e \mid \textbf{EXN} & \textit{(Threads)} \\
obj & ::= & (C, F, L) & \textit{(Objects)} \\
F & ::= & \epsilon \mid F, f \mapsto v & \textit{(Field map)} \\
L & ::= & \textbf{locked} \mid \textbf{unlocked} & \textit{(Lock status)} \\
\textbf{EXN} & ::= & \textbf{NullPointerException} & \textit{(Exceptions)}
\end{array}
$$

Figure 6: Run-time constructs of OOlong.

instead of tracking the values of variables in a map. However, the current design gives us a simple way of reasoning about object references on the stack as well as on the heap, and makes it easier to later add support for assignable variables.

A thread collection T can have one of three forms: $T_1||T_2 \rhd e$ denotes two parallel asyncs T_1 and T_2 which must reduce fully before evaluation proceeds to e. $(\mathcal{L}, e)$ is a single thread evaluating expression e. $\mathcal{L}$ is a set of locations of all the objects whose locks are currently being held by the thread. The initial configuration is $\langle \epsilon; \epsilon; (\emptyset, e)\rangle$, where e is the initial expression of the program. A thread can also be in an exceptional state **EXN**, which is a well-formed but "crashed" state that cannot be recovered from. The current semantics only supports the **NullPointerException**.

4.1 Well-Formedness Rules

Figure 7 shows the definition of a well-formed OOlong configuration. A configuration is well-formed if its heap H and stack V are well-formed, its collection of threads T is well-typed, and the current lock situation in the system is well-formed (WF-CFG). Note that well-formedness of threads is split into two sets of rules regarding expression typing and locking respectively A heap H is well-formed under a Γ if all locations in Γ correspond to objects in H, all objects in the heap have an entry in Γ, and the fields of all objects are well-formed under Γ (WF-HEAP). The fields of an object of class C are well-formed if each name of the static fields of C maps to a value of the corresponding type (WF-FIELDS). A stack V is well-formed under a Γ if each variable in Γ maps to a value of the corresponding type in V, and each variable in V has an entry in Γ (WF-VARS). A well-formed thread collection requires all sub-threads and expressions to be well-formed (WF-T-*). An exceptional state can have any well-formed type (WF-T-EXN).

The current lock situation is well-formed for a thread if all locations in its set of held locks $\mathcal{L}$ correspond to objects whose lock status is **locked**. Two instances of $\textbf{locked}_\iota$ in e must refer to different locations ι (captured by $\textit{distinctLocks}(e)$, *cf.,* Section A.3), and for each $\textbf{locked}_\iota$ in e, ι must be in the set of held locks $\mathcal{L}$. The parallel case propagates these properties, and additionally requires that two parallel threads do not hold the same locks in their respective $\mathcal{L}$. Any locks held in the continuation e must be held by the first thread of the async. This represents the fact the first thread is the one that will continue execution after the threads join (WF-L-ASYNC). Exceptional states are always well-formed with respect to locking (WF-L-EXN).

$\boxed{\Gamma \vdash \langle H; V; T\rangle : t}$ *(Well-formed configuration)*

WF-CFG
$$\frac{\Gamma \vdash H \qquad \Gamma \vdash V \qquad \Gamma \vdash T : t \qquad H \vdash_{\text{lock}} T}{\Gamma \vdash \langle H; V; T\rangle : t}$$

WF-HEAP
$$\frac{\forall \iota : C \in \Gamma. H(\iota) = (C, F, L) \wedge \Gamma; C \vdash F \qquad \forall \iota \in \textbf{dom}(H). \iota \in \textbf{dom}(\Gamma) \qquad \vdash \Gamma}{\Gamma \vdash H}$$

WF-FIELDS
$$\frac{\textbf{fields}(C) \equiv f_1 : t_1, .., f_n : t_n \qquad \Gamma \vdash v_1 : t_1, .., \Gamma \vdash v_n : t_n}{\Gamma; C \vdash f_1 \mapsto v_1, .., f_n \mapsto v_n}$$

WF-VARS
$$\frac{\forall x : t \in \Gamma. V(x) = v \wedge \Gamma \vdash v : t \qquad \forall x \in \textbf{dom}(V). x \in \textbf{dom}(\Gamma) \qquad \vdash \Gamma}{\Gamma \vdash V}$$

WF-T-ASYNC
$$\frac{\Gamma \vdash T_1 : t_1 \qquad \Gamma \vdash T_2 : t_2 \qquad \Gamma \vdash e : t}{\Gamma \vdash T_1 \,||\, T_2 \rhd e : t}$$

WF-T-THREAD
$$\frac{\Gamma \vdash e : t}{\Gamma \vdash (\mathcal{L}, e) : t}$$

WF-T-EXN
$$\frac{\vdash t \qquad \vdash \Gamma}{\Gamma \vdash \textbf{EXN} : t}$$

WF-L-THREAD
$$\frac{\forall \iota \in \mathcal{L}. H(\iota) = (C, F, \textbf{locked}) \qquad \textit{distinctLocks}(e) \qquad \forall \iota \in \textbf{locks}(e). \iota \in \mathcal{L}}{H \vdash_{\text{lock}} (\mathcal{L}, e)}$$

WF-L-ASYNC
$$\frac{\textbf{heldLocks}(T_1) \cap \textbf{heldLocks}(T_2) = \emptyset \qquad \forall \iota \in \textbf{locks}(e). \iota \in \textbf{heldLocks}(T_1) \qquad \textit{distinctLocks}(e) \qquad H \vdash_{\text{lock}} T_1 \qquad H \vdash_{\text{lock}} T_2}{H \vdash_{\text{lock}} T_1 \,||\, T_2 \rhd e}$$

WF-L-EXN
$$\frac{}{H \vdash_{\text{lock}} \textbf{EXN}}$$

Figure 7: Well-formedness rules.

4.2 Evaluation of Expressions

Figure 8 shows the single-threaded execution of an OOlong program. OOlong uses a small-step dynamic semantics, with the standard technique of evaluation contexts (the definition of E in the top of the figure) to decide the order of evaluation and reduce the number of rules (DYN-EVAL-CONTEXT). We use a single stack frame for the entire program and employ renaming to make sure that variables have unique names[2]. Evaluating a variable simply looks it up in the stack (DYN-EVAL-VAR). A **let**-expression introduces a fresh variable that it substitutes for the static name (DYN-EVAL-LET). Similarly, calling a method introduces two new fresh variables—one for **this** and one for the parameter of the method. The method is dynamically dispatched on the type

[2]This sacrifices reasoning about properties of the stack size in favour of simpler dynamic semantics.

$E[\bullet] ::= x.f = \bullet \mid x.m(\bullet) \mid \textbf{let}\ x = \bullet\ \textbf{in}\ e \mid (t)\ \bullet \mid \textbf{locked}_\iota\{\bullet\}$

$\boxed{cfg_1 \hookrightarrow cfg_2}$ *(Evaluation of expressions)*

DYN-EVAL-CONTEXT
$$\frac{\langle H; V; (\mathcal{L}, e)\rangle \hookrightarrow \langle H'; V'; (\mathcal{L}', e')\rangle}{\langle H; V; (\mathcal{L}, E[e])\rangle \hookrightarrow \langle H'; V'; (\mathcal{L}', E[e'])\rangle}$$

DYN-EVAL-VAR
$$\frac{V(x) = v}{\langle H; V; (\mathcal{L}, x)\rangle \hookrightarrow \langle H; V; (\mathcal{L}, v)\rangle}$$

DYN-EVAL-LET
$$\frac{x'\ \textbf{fresh} \qquad V' = V[x' \mapsto v] \qquad e' = e[x \mapsto x']}{\langle H; V; (\mathcal{L}, \textbf{let}\ x = v\ \textbf{in}\ e)\rangle \hookrightarrow \langle H; V'; (\mathcal{L}, e')\rangle}$$

DYN-EVAL-CALL
$$\frac{\begin{array}{c} V(x) = \iota \qquad H(\iota) = (C, F, L) \\ \textbf{methods}\,(C)(m) = y : t_2 \to t, e \\ \textbf{this}'\ \textbf{fresh} \qquad y'\ \textbf{fresh} \\ V' = V[\textbf{this}' \mapsto \iota][y' \mapsto v] \\ e' = e[\textbf{this} \mapsto \textbf{this}'][y \mapsto y'] \end{array}}{\langle H; V; (\mathcal{L}, x.m(v))\rangle \hookrightarrow \langle H; V'; (\mathcal{L}, e')\rangle}$$

DYN-EVAL-CAST
$$\frac{}{\langle H; V; (\mathcal{L}, (t)v)\rangle \hookrightarrow \langle H; V; (\mathcal{L}, v)\rangle}$$

DYN-EVAL-SELECT
$$\frac{\begin{array}{c} V(x) = \iota \qquad H(\iota) = (C, F, L) \\ \textbf{fields}\,(C)(f) = t \qquad F(f) = v \end{array}}{\langle H; V; (\mathcal{L}, x.f)\rangle \hookrightarrow \langle H; V; (\mathcal{L}, v)\rangle}$$

DYN-EVAL-UPDATE
$$\frac{\begin{array}{c} V(x) = \iota \qquad H(\iota) = (C, F, L) \\ \textbf{fields}\,(C)(f) = t' \qquad H' = H[\iota \mapsto (C, F[f \mapsto v], L)] \end{array}}{\langle H; V; (\mathcal{L}, x.f = v)\rangle \hookrightarrow \langle H'; V; (\mathcal{L}, \textbf{null})\rangle}$$

DYN-EVAL-NEW
$$\frac{\begin{array}{c} \textbf{fields}\,(C) \equiv f_1 : t_1, .., f_n : t_n \\ F \equiv f_1 \mapsto \textbf{null}, .., f_n \mapsto \textbf{null} \\ \iota\ \textbf{fresh} \qquad H' = H[\iota \mapsto (C, F, \textbf{unlocked})] \end{array}}{\langle H; V; (\mathcal{L}, \textbf{new}\ C)\rangle \hookrightarrow \langle H'; V; (\mathcal{L}, \iota)\rangle}$$

DYN-EVAL-LOCK
$$\frac{\begin{array}{c} V(x) = \iota \qquad H(\iota) = (C, F, \textbf{unlocked}) \qquad \iota \notin \mathcal{L} \\ H' = H[\iota \mapsto (C, F, \textbf{locked})] \qquad \mathcal{L}' = \mathcal{L} \cup \{\iota\} \end{array}}{\langle H; V; (\mathcal{L}, \textbf{lock}(x)\ \textbf{in}\ e)\rangle \hookrightarrow \langle H'; V; (\mathcal{L}', \textbf{locked}_\iota\{e\})\rangle}$$

DYN-EVAL-LOCK-REENTRANT
$$\frac{V(x) = \iota \qquad H(\iota) = (C, F, \textbf{locked}) \qquad \iota \in \mathcal{L}}{\langle H; V; (\mathcal{L}, \textbf{lock}(x)\ \textbf{in}\ e)\rangle \hookrightarrow \langle H; V; (\mathcal{L}, e)\rangle}$$

DYN-EVAL-LOCK-RELEASE
$$\frac{\begin{array}{c} H(\iota) = (C, F, \textbf{locked}) \qquad \mathcal{L}' = \mathcal{L} \backslash \{\iota\} \\ H' = H[\iota \mapsto (C, F, \textbf{unlocked})] \end{array}}{\langle H; V; (\mathcal{L}, \textbf{locked}_\iota\{v\})\rangle \hookrightarrow \langle H'; V; (\mathcal{L}', v)\rangle}$$

Figure 8: Dynamic semantics (1/2). Expressions.

$\boxed{cfg_1 \hookrightarrow cfg_2}$ *(Concurrency)*

DYN-EVAL-ASYNC-LEFT
$$\frac{\langle H; V; T_1\rangle \hookrightarrow \langle H'; V'; T_1'\rangle}{\langle H; V; T_1 \mid\mid T_2 \rhd e\rangle \hookrightarrow \langle H'; V'; T_1' \mid\mid T_2 \rhd e\rangle}$$

DYN-EVAL-ASYNC-RIGHT
$$\frac{\langle H; V; T_2\rangle \hookrightarrow \langle H'; V'; T_2'\rangle}{\langle H; V; T_1 \mid\mid T_2 \rhd e\rangle \hookrightarrow \langle H'; V'; T_1 \mid\mid T_2' \rhd e\rangle}$$

DYN-EVAL-SPAWN
$$\frac{e = \textbf{finish}\ \{\ \textbf{async}\ \{\ e_1\ \}\ \textbf{async}\ \{\ e_2\ \}\ \};\ e_3}{\langle H; V; (\mathcal{L}, e)\rangle \hookrightarrow \langle H; V; (\mathcal{L}, e_1)\ ||(\emptyset, e_2) \rhd e_3\rangle}$$

DYN-EVAL-SPAWN-CONTEXT
$$\frac{\langle H; V; (\mathcal{L}, e)\rangle \hookrightarrow \langle H; V; (\mathcal{L}, e_1)\ ||(\emptyset, e_2) \rhd e_3\rangle}{\langle H; V; (\mathcal{L}, E[e])\rangle \hookrightarrow \langle H; V; (\mathcal{L}, e_1)\ ||(\emptyset, e_2) \rhd E[e_3]\rangle}$$

DYN-EVAL-ASYNC-JOIN
$$\frac{}{\langle H; V; (\mathcal{L}, v)\ ||(\mathcal{L}', v') \rhd e\rangle \hookrightarrow \langle H; V; (\mathcal{L}, e)\rangle}$$

Figure 9: Dynamic semantics (2/2). Concurrency.

of the target object (DYN-EVAL-CALL).

Casts will always succeed and are therefore no-ops dynamically (DYN-EVAL-CAST). Adding support for downcasts is possible with the introduction of a new exceptional state for failed casts. Fields are looked up in the field map of the target object (DYN-EVAL-SELECT). Similarly, field assignments are handled by updating the field map of the target object. Field updates, which are always typed as **Unit**, evaluate to **null** (DYN-EVAL-UPDATE). We have omitted constructors from this treatise (Section 8.3 discusses how they can be added). A new object has its fields initialised to **null** and is given a fresh abstract location on the heap (DYN-EVAL-NEW).

Taking a lock requires that the lock is currently available and adds the locked object to the lock set $\mathcal{L}$ of the current thread. It also updates the object to reflect its locked status (DYN-EVAL-LOCK). The locks in OOlong are reentrant, meaning that grabbing the same lock twice will always succeed (DYN-EVAL-LOCK-REENTRANT). Locking is structured, meaning that a thread can not grab a lock without also releasing it sooner or later (modulo getting stuck due to deadlocks). The **locked** wrapper around e records the successful taking of the lock and is used to release the lock once e has been fully reduced (DYN-EVAL-LOCK-RELEASE). Note that a thread that cannot take a lock gets stuck until the lock is released. We define these states formally to distinguish them from unsound stuck states (*cf.,* Section A.1)

Dereferencing **null**, *e.g.,* using a **null** valued argument when looking up a field or calling a method, results in a **NullPointerException**, which crashes the program. These rules are unsurprising and are therefore relegated to the appendix (*cf.,* Section A.2).

4.3 Concurrency

Figure 9 shows the semantics of concurrent execution in OO-

long. Concurrency is modeled as non-deterministic choice of what thread to evaluate (DYN-EVAL-ASYNC-LEFT/RIGHT). Finish/async spawns one new thread for the second async and uses the current thread for the first. This means that the first async holds all the locks of the spawning thread, while the second async starts out with an empty lock set (DYN-EVAL-SPAWN). The evaluation context rule, needed because DYN-EVAL-CONTEXT does not handle spawning, forces the full reduction of the parallel expressions to the left of $\triangleright$ before continuing with e_3, which is the expression placed in the hole of the evaluation context (DYN-EVAL-SPAWN-CONTEXT). When two asyncs have finished, the second thread is removed along with all its locks[3], and the first thread continues with the expression to the right of $\triangleright$ (DYN-EVAL-ASYNC-JOIN).

5. TYPE SOUNDNESS OF OOlong

We prove type soundness as usual by proving progress and preservation. This section only states the theorems and sketches the proofs. We refer to the mechanised semantics for the full proofs (*cf.,* Section 6).

Since well-formed programs are allowed to deadlock, we must formulate the progress theorem so that this is handeled. The *Blocked* predicate on configurations is defined in the appendix (*cf.,* Section A.1).

PROGRESS. *A well-formed configuration is either done, has thrown an exception, has deadlocked, or can take one additional step:*

$$\begin{array}{l}\forall \Gamma,\ H,\ V,\ T,\ t\ .\ \Gamma \vdash \langle H;V;T\rangle : t \Rightarrow \\ \quad T = (\mathcal{L}, v) \vee T = \mathbf{EXN} \vee \mathit{Blocked}(\langle H;V;T\rangle) \vee \\ \quad \exists \mathit{cfg}', \langle H;V;T\rangle \hookrightarrow \mathit{cfg}'\end{array}$$

PROOF SKETCH. Proved by induction over the thread structure T. The single threaded case is proved by induction over the typing relation over the current expression.

To show preservation of well-formedness we first define a subsumption relation $\Gamma_1 \subseteq \Gamma_2$ between environments. Γ_2 subsumes Γ_1 if all mappings $\gamma : t$ in Γ_1 are also in Γ_2:

$\boxed{\Gamma_1 \subseteq \Gamma_2}$ *(Environment Subsumption)*

$$\frac{\text{WF-SUBSUMPTION}\quad \forall \gamma : t \in \Gamma . \Gamma'(\gamma) = t}{\Gamma \subseteq \Gamma'}$$

PRESERVATION. *If $\langle H;V;T\rangle$ types to t under some environment Γ, and $\langle H;V;T\rangle$ steps to some $\langle H';V';T'\rangle$, there exists an environment subsuming Γ which types $\langle H';V';T'\rangle$ to t.*

$$\begin{array}{l}\forall \Gamma,\ H,\ H',\ V,\ V',\ T,\ T',\ t. \\ \quad \Gamma \vdash \langle H;V;T\rangle : t \wedge \langle H;V;T\rangle \hookrightarrow \langle H';V';T'\rangle \Rightarrow \\ \quad\quad \exists \Gamma' . \Gamma' \vdash \langle H';V';T'\rangle : t \wedge \Gamma \subseteq \Gamma'\end{array}$$

PROOF SKETCH. Proved by induction over the thread structure T. The single threaded case is proved by induction over the typing relation over the current expression. There are also a number of lemmas regarding locking that needs proving (*e.g.,* that a thread can never steal a lock held by another thread). We refer to the mechanised proofs for details.

[3]In practice, since locking is structured these locks will already have been released.

6. MECHANISED SEMANTICS

We have fully mechanised the semantics of OOlong in Coq, including the proofs of soundness. The source code weighs in at ∼4100 lines of Coq, ∼900 of which are definitions and ∼3200 of which are properties and proofs. In the proof code, ∼300 lines are extra lemmas about lists and ∼200 lines are tactics specific to this formalism used for automating often re-occurring reasoning steps. The proofs also make use of the LibTactics library [20], as well as the `crush` tactic [8]. We use Coq bullets together with Aaron Bohannon's "Case" tactic to structure the proofs and make refactoring simpler; when a definition changes and a proof needs to be rewritten, it is immediately clear which cases fail and therefore need to be updated.

The mechanised semantics are the same as the semantics presented here, modulo uninteresting representation differences such as modelling the typing environment Γ as a function rather than a sequence. It explicitly deals with details such as how to generate fresh names and separating static and dynamic constructs (*e.g.,* when calling a method, the body of the method will not contain any dynamic expressions, such as $\mathbf{locked}_\iota\{e\}$). It also defines helper functions like field and method lookup.

The Coq sources are available in a public repository so that the semantics can be easily obtained and extended [5]. The source files compile under Coq 8.8.2, the latest version at the time of writing.

As a comparison between the Coq definitions and the paper versions, here are the mechanised formulations of progress and preservation:

```
Theorem progress :
  forall P t' Gamma cfg t,
    wfProgram P t' ->
    wfConfiguration P Gamma cfg t ->
      cfg_done cfg \/ cfg_exn cfg \/
      cfg_blocked cfg \/
      exists cfg', P / cfg ==> cfg'.

Theorem preservation :
  forall P t' Gamma cfg cfg' t,
    wfProgram P t' ->
    wfConfiguration P Gamma cfg t ->
    P / cfg ==> cfg' ->
    exists Gamma',
      wfConfiguration P Gamma' cfg' t /\
      wfSubsumption Gamma Gamma'.
```

Other than some notational differences (*e.g.,* using the name `cfg` instead of spelling out $\langle H;V;T\rangle$), the biggest noticeable difference is that the program `P` is threaded through all the definitions, among other things to be able to do field and method lookups. For example, the proposition `P / cfg ==> cfg'` should be read as "`cfg` steps to `cfg'` when executing in program `P`". There is also an explicit requirement

that this program is well-formed (`wfProgram P t'`, where `t'` is the type of the starting expression).

As another example, here is the lemma that states that if two threads have disjoint lock sets, stepping one of them will not cause the lock sets to overlap:

```
Lemma stepCannotSteal :
  forall P H H' V V' n n' T1 T1' T2,
    wfLocking H T1 ->
    wfLocking H T2 ->
    disjointLocks T1 T2 ->
    P / (H, V, n, T1) ==> (H', V', n', T1') ->
    disjointLocks T1' T2.
```

The propositions `disjointLocks T1 T2` and `wfLocking H T` correspond to $\textbf{heldLocks}(T_1) \cap \textbf{heldLocks}(T_2) = \emptyset$ and $H \vdash_{\text{lock}} T$ (*cf.,* Figure 7). The extra element `n` in the configuration `(H, V, n, T)` is an integer used to generate fresh variable names.

Finally, we first show how the evaluation context E (*cf.,* Figure 8) is expressed in Coq. An evaluation context is a function taking a single expression to another expression:

```
Definition econtext := expr -> expr.
```

Each case of E is represented by a Coq function of type `econtext`, for example:

```
Definition ctx_call (x : _) (m : _) : econtext :=
  (fun e => ECall x m e).
```

To capture which functions are valid evaluation contexts, we define a proposition `is_econtext`, which is used by all definitions which reason about evaluation contexts (the snippet below shows the dynamic rule DYN-EVAL-CONTEXT):

```
Inductive is_econtext : econtext -> Prop :=
  | EC_Call :
      forall x m,
        is_econtext (ctx_call x m)
  | ...

Inductive step (P : program) :
  configuration -> configuration -> Prop :=
  | ...
  | EvalContext :
      forall H H' V V' n n' E e e' Ls Ls',
        is_econtext E ->
        P / (H, V, n, T_Thread Ls e) ==>
          (H', V', n', T_Thread Ls' e') ->
        P / (H, V, n, T_Thread Ls (E e)) ==>
          (H', V', n', T_Thread Ls' (E e'))
  | ...
```

When performing case analysis over which `step` rules are applicable, Coq sometimes generates absurd cases where the an invalid evaluation context is applied. To handle these cases automatically, we define tactics for unfolding (applying) all evaluation contexts in scope and finding impossible equalities in the assumptions (`context[e]` is the Coq notation for matching any term with `e` in it):

```
Ltac unfold_ctx :=
  match goal with
    | [H: context[ctx_call] |- _] =>
      unfold ctx_call in H
    | [_ : _ |- context[ctx_call]] =>
      unfold ctx_call
    | ...
  end.

Ltac malformed_context :=
  match goal with
    | [Hctx : is_econtext ?ctx |- _] =>
      inv Hctx; repeat(unfold_ctx);
        try(congruence)
    | _ =>
      fail 1 "could not prove malformed context"
  end.
```

The tactic `malformed_context` tries to find an instance of `is_econtext` in the current assumptions, inverts it (`inv H` is defined as `(inversion H; subst; clear H)`), unfolds all evaluation contexts in scope and then uses `congruence` to dismiss all subgoals with absurd equalities in the assumptions. In proofs where case-analysis of the step relation is needed, the tactic `inv Hstep; try malformed_context` is used to only keep the sane cases around.

7. TYPESETTING OOlong

The paper version of OOlong is written in Ott [25], which lets a user define the grammar and typing rules of their semantics using ASCII-syntax. The rules are checked against the grammar to make sure that the syntax is consistent. Ott can then generate LaTeX code for these rules, which when typeset appear as in this paper. The Ott sources are available in the same repo as the Coq sources [5]. As an example, here is the Ott version of the rule WF-LET:

```
G |- e1 : t1
G, x : t1 |- e2 : t
---------------------------- :: let
G |- let x = e1 in e2 : t
```

The LaTeX rendering of `G` as Γ and `|-` as $\vdash$ is defined elsewhere in the Ott file, and the rule is rendered as:

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x : t_1 \vdash e_2 : t}{\Gamma \vdash \textbf{let}\, x = e_1\, \textbf{in}\, e_2 : t}\ \text{WF-LET}$$

It is also possible to have Ott generate LaTeX code for the grammar, but these tend to require more whitespace than one typically has to spare in an article. We therefore include LaTeX code for a more compact version of the grammar, as well as the definitions of progress and preservation [5]. Ott also supports generating Coq and Isabelle/HOL code from the same definitions that generate LaTeX code. We have not used this feature as we think it is useful to let the paper version of the semantics abstract away some of the details that a mechanised version requires.

8. EXTENSIONS TO THE SEMANTICS

This section demonstrates the extensibility of OOlong by adding assertions, region based locking, and type-based null reference tracking to the semantics. They are chosen as examples of adding new expressions, adding new runtime constructs, and extending the type system respectively. Here we only describe the additions necessary, but these features have also been added to the mechanised version of the semantics with little added complexity to the code. They are all available as examples on how to extend the mechanised semantics [5].

8.1 Supporting Assertions

Assertions are a common way to enforce pre- and postconditions and to fail fast if some condition is not met. We add support for assertions in OOlong by adding an expression **assert(x == y)**, which asserts that two variables are aliases (if we added richer support for primitives we could let the argument of the assertion be an arbitrary boolean expression). If an assertion fails, we throw an **AssertionException**. The typing rule for assertions states that the two variables are of the same type. The type of an assertion is **Unit**.

WF-ASSERT

$$\frac{\Gamma(x) = t \qquad \Gamma(y) = t}{\Gamma \vdash \mathbf{assert}\,(x == y) : \mathbf{Unit}}$$

In the dynamic semantics, we have two outcomes of evaluating an assertion: if successful, the program continues; if not, the program should crash.

DYN-EVAL-ASSERT

$$\frac{V(x) = V(y)}{\langle H; V; (\mathcal{L}, \mathbf{assert}\,(x == y))\rangle \hookrightarrow \langle H; V; (\mathcal{L}, \mathbf{null})\rangle}$$

DYN-EXN-ASSERT

$$\frac{V(x) \neq V(y)}{\langle H; V; (\mathcal{L}, \mathbf{assert}\,(x == y))\rangle \hookrightarrow \langle H; V; \mathbf{AssertionException}\rangle}$$

Note that the rules for exceptions already handle exception propagation, regardless of the kind of exception (*cf.,* Section A.2).

In the mechanised semantics, the automated tactics are powerful enough to automatically solve the additional cases for almost all lemmas. The additional cases in the main theorems are easily dispatched. This extension adds a mere ∼50 lines to the mechanisation.

8.2 Supporting Region-based Locking

Having a single lock per object prevents threads from concurrently updating disjoint parts of an object, even though this is benign from a data-race perspective. Many effect-systems divide the fields of an object into *regions* in order to reason about effect disjointness on a single object (*e.g.,* [4]). Similarly, we can add regions to OOlong, let each field belong to a region and let each region have a lock of its own. Syntactically, we add a region annotation to field declarations ("$f : t$ **in** r") and require that taking a lock specifies which region is being locked ("**lock**(x, r) **in** e"). Here we omit declaring regions and simply consider all region names valid. This means that the rules for checking well-formedness of fields do not need updating (other than the syntax).

Dynamically, locks are now identified not only by the location ι of their owning object, but also by their region r. Objects need to be extended from having one lock to having multiple locks, each with its own lock status. We model this by replacing the lock status of an object with a region map RL from region names to lock statuses. As an example, the dynamic rule for grabbing a lock for a region is updated thusly:

DYN-EVAL-LOCK-REGION

$$\frac{\begin{array}{c} V(x) = \iota \qquad H(\iota) = (C, F, RL) \qquad RL(\mathbf{r}) = \mathbf{unlocked} \\ (\iota, \mathbf{r}) \notin \mathcal{L} \qquad \mathcal{L}' = \mathcal{L} \cup \{(l, r)\} \\ H' = H[\iota \mapsto (C, F, RL[\mathbf{r} \mapsto \mathbf{locked}])] \end{array}}{\langle H; V; (\mathcal{L}, \mathbf{lock}(x, \mathbf{r})\,\mathbf{in}\,e)\rangle \hookrightarrow \langle H'; V; (\mathcal{L}', \mathbf{locked}_{(\iota,\mathbf{r})}\{e\})\rangle}$$

Similarly, the well-formedness rules for locking need to be updated to refer to region maps of objects instead of just objects. A region map must contain a mapping for each region used in the object:

WF-REGIONS

$$\frac{\forall f : t\ \mathbf{in}\ \mathbf{r} \in \mathbf{fields}\,(C).\mathbf{r} \in \mathbf{dom}(RL)}{C \vdash RL}$$

The changes can mostly be summarised as adding one extra level of indirection each time a lock status is looked up on the heap. The same is true for the mechanised semantics. For example, in the original mechanisation, the lemma showing that stepping a thread cannot cause it to take a lock that is already locked by another thread looks like this:

```
Lemma noDuplicatedLocks :
  forall P t' Gamma l H H' V V' n n' T T' t c F,
    wfProgram P t' ->
    heapLookup H l = Some (c, F, LLocked) ->
    ~ In l (t_locks T) ->
    wfConfiguration P Gamma (H, V, n, T) t ->
    P / (H, V, n, T) ==> (H', V', n', T') ->
    ~ In l (t_locks T').
```

The function `t_locks` extracts the locks held by a thread `T`. In the version extended with region locks, the same lemma instead looks like this:

```
Lemma noDuplicatedLocks : forall
  P t' Gamma l r H H' V V' n n' T T' t c F RL,
    wfProgram P t' ->
    heapLookup H l = Some (c, F, RL) ->
    RL r = Some LLocked ->
    ~ In (l, r) (t_locks T) ->
    wfConfiguration P Gamma (H, V, n, T) t ->
    P / (H, V, n, T) ==> (H', V', n', T') ->
    ~ In (l, r) (t_locks T').
```

Notice that instead of having a single taken lock, the object looked up on the heap has a region map `RL` whose lock related to region `r` is taken. The proof of the lemma is the same as before, except for one extra inversion and an additional call to `congruence`.

This extension increases the size of the mechanised semantics by ∼130 lines.

8.3 Supporting Nullable Types

Null pointers are famously referred to by Tony Hoare as his "billion dollar mistake", referring to the fact that accidentally dereferencing null pointers has been the cause of many bugs since their introduction in the 1960s [13]. One way to reduce the risk of these errors is to have the type system track which references may be **null** during runtime. This section introduces such "nullable types" to OOlong.

We start by extending the syntax of types:

$$t ::= C \mid I \mid C? \mid I? \mid \textbf{Unit}$$

The types $C?$ and $I?$ are given to references which could be **null** valued. In the mechanisation, class and interface types are extended with a boolean flag which tracks if the type is nullable or not:

```
Inductive ty : Type :=
  | TClass : class_id -> bool -> ty
  | TInterface : interface_id -> bool -> ty
  | TUnit : ty.
```

The subtyping rules are extended to allow non-nullable references to flow into nullable ones, but not the other way around:

$$\frac{t <: t'}{t? <: t'?}\ \text{T-SUB-N} \qquad \frac{t <: t'}{t <: t'?}\ \text{T-SUB-N-R}$$

Finally, the type checking rule for **null** is updated to disallow non-nullable types (**nullable**(t) is defined to be true for all types $t?$ and for **Unit**):

$$\frac{\vdash \Gamma \qquad \vdash t \qquad \textbf{nullable}\,(t)}{\Gamma \vdash \textbf{null} : t}\ \text{WF-NULLABLE}$$

For simplicity in this presentation, since there are no constructors in OOlong, we require that all field types are nullable (since fields are initialised to **null**). Lifting this restriction is straightforward, for example by providing a list of initial field values when creating new objects: **new** $C(e_1, \ldots, e_n)$.

The mechanised semantics grows by ∼100 lines with the type-level additions shown above. The extension of the subtyping rules requires some proofs to be extended to handle these cases and some minor lemmas to be added (*e.g.,* that if $t_1 <: t_2$ and **nullable**(t_1), then **nullable**(t_2)). Allowing non-nullable types on the heap by adding constructors is possible, but is complicated by Coq's inability to generate effective induction principles for nested data types (*cf.,* [8], Chapter 3.8); the **new** expression would contain a list of expressions, for which no induction hypotheses are automatically generated during proofs by induction. Mechanising this extension with hand-written induction principles is left as future work.

```
interface Counter {
  add(x : int) : Unit
  get(tmp : int) : int
}
class Cell implements Counter {
  cnt : int
  def init(n : int) : Unit {
    this.cnt = n
  }
  def add(n : int) : Unit {
    let cnt = this.cnt in
      this.cnt = (cnt + n)
  }
  def get(tmp : int) : int {
    this.cnt
  }
}
let cell = new Cell in
let cell2 = (Counter) cell in // (†)
let tmp = cell.init(0) in     // (‡)
finish {
  async {
    lock(cell) in cell.add(1)
  }
  async {
    lock(cell2) in cell2.add(2)
  }
};
cell.get(0)
```

Figure 10: An OOlong program, with added integer support.

9. PROTOTYPE INTERPRETER

In addition the formalised semantics of OOlong, we have implemented a simple interpreter for the language. The purpose of this implementation is twofold: to provide an actual executable semantics, and to minimise the effort of prototyping a future extension of OOlong. It is provided together with the Ott and Coq sources in the OOlong repository [5].

The full implementation is ∼760 lines of OCaml, accompanied by ∼190 lines of comments and documentation. Lexing and parsing (∼90 lines) is implemented using `ocamllex` [15] and Menhir [22]. Type checking (∼190 lines) is based on a bidirectional version of the typing rules which makes typing entirely syntax-directed [21].

Figure 11 shows the bidirectional typing rules. Bidirectional type checking differentiates between *inferring* a type for an expression ($\Gamma \vdash e \Rightarrow t$) and *checking* that an expression has a given expression ($\Gamma \vdash e \Leftarrow t$). Most rules mirror the corresponding rules in Figure 5, but make explicit which types are inferred and which are checked against an existing type. The frame-rule in WF-FJ is exchanged for a requirement

$\boxed{\Gamma \vdash e \Rightarrow t \quad \Gamma \vdash e \Leftarrow t}$ *(Inference/Checking)*

BD-INFER-VAR
$$\frac{\vdash \Gamma \qquad \Gamma(x) = t}{\Gamma \vdash x \Rightarrow t}$$

BD-INFER-LET
$$\frac{\Gamma \vdash e_1 \Rightarrow t_1 \qquad \Gamma, x : t_1 \vdash e_2 \Rightarrow t}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Rightarrow t}$$

BD-INFER-CALL
$$\frac{\Gamma \vdash x \Rightarrow t_1 \qquad \Gamma \vdash e \Leftarrow t_2 \qquad \textbf{msigs}\,(t_1)(m) = y : t_2 \to t}{\Gamma \vdash x.m(e) \Rightarrow t}$$

BD-INFER-CAST
$$\frac{\Gamma \vdash e \Leftarrow t}{\Gamma \vdash (t)e \Rightarrow t}$$

BD-INFER-SELECT
$$\frac{\Gamma \vdash x \Rightarrow C \qquad \textbf{fields}\,(C)(f) = t}{\Gamma \vdash x.f \Rightarrow t}$$

BD-INFER-UPDATE
$$\frac{\Gamma \vdash x \Rightarrow C \qquad \Gamma \vdash e \Leftarrow t \qquad \textbf{fields}\,(C)(f) = t}{\Gamma \vdash x.f = e \Rightarrow \textbf{Unit}}$$

BD-INFER-NEW
$$\frac{\vdash \Gamma \qquad \vdash C}{\Gamma \vdash \textbf{new}\ C \Rightarrow C}$$

BD-INFER-LOCK
$$\frac{\Gamma \vdash x \Rightarrow t_2 \qquad \Gamma \vdash e \Rightarrow t}{\Gamma \vdash \textbf{lock}(x)\,\textbf{in}\ e \Rightarrow t}$$

BD-INFER-FJ
$$\frac{fv(e_1) \cap fv(e_2) = \emptyset \qquad \Gamma \vdash e_1 \Rightarrow t_1 \qquad \Gamma \vdash e_2 \Rightarrow t_2 \qquad \Gamma \vdash e \Rightarrow t}{\Gamma \vdash \textbf{finish}\ \{\ \textbf{async}\ \{\ e_1\ \}\ \textbf{async}\ \{\ e_2\ \}\ \};\, e \Rightarrow t}$$

BD-CHECK-NULL
$$\frac{\vdash \Gamma \qquad \vdash t}{\Gamma \vdash \textbf{null} \Leftarrow t}$$

BD-CHECK-SUB
$$\frac{\Gamma \vdash e \Rightarrow t' \qquad t' <: t \qquad e \neq \textbf{null}}{\Gamma \vdash e \Leftarrow t}$$

Figure 11: Bidirectional typing rules.

that the free variables in two parallel **async**s are disjoint (BD-INFER-FJ). Notably, the type of **null** is never inferred, but can be given any well-formed type (BD-CHECK-NULL). Checking any other expression against some type t amounts to inferring a type t' and seeing if t' is a subtype of t (BD-CHECK-SUB). We omit rules for syntax-directed subtyping, which is implemented as a simple traversal of the interface hierarchy. Since we only ever type check static programs, there are no rules for the dynamic expressions ι and $\textbf{locked}_\iota\{e\}$.

The actual evaluation of programs (∼290 lines) is a more or less direct translation of the dynamic rules in Section 4. The interpreter evaluates entire configurations one step a time, until the program terminates or deadlocks. Non-terminating programs (including programs with livelocks) are not detected by the interpreter, but will run forever. Non-deterministic choice is implemented using a "scheduler" function which decides whether to run the left or the right thread in a fork. This function is customisable, allowing for deterministic scheduling as well. There is no parallelism in the interpreter itself.

To allow for *slightly* more interesting programs to be written, we have also extended the interpreter with support for integers and addition. This extension adds ∼50 lines of code. Other than the obvious additions to parsing, type checking and evaluation, the typing rule BD-CHECK-NULL must also be updated with the premise $t \neq \text{int}$ so that **null** does not inhabit the integer type. Figure 10 shows an example program written with this extension. Ignoring the integers, it is also a syntactically correct OOlong program according to the formal semantics. It shows some of the ways that the implementation (and formalism) is kept simple. For example, a function must take exactly one argument, even when it is not used (*cf.,* method `get`). Similarly, there is no sequencing without variable binding, so the **Unit** result of `cell.init(0)` (‡) must be bound to a variable. Note also that the `Cell` object must be aliased (†) in order to be used by both threads in the subsequent fork (the upcast to `Counter` is not necessary, but is included to show all features of the language). All of these things are of course easily remedied, but the point of the interpreter is not to provide a smooth programmer experience, but to keep the *implementation* simple.

Interpreting the program in Figure 10 gives the following output:

```
Ran for 31 steps, resulting in
([], 3)
Heap:
{
 0 -> (Cell, {cnt -> 3}, Unlocked)
}
Variables:
{
 cell -> 0
 cell2 -> 0
 cnt -> 0
 cnt#4 -> 1
 n -> 0
 n#0 -> 1
 n#2 -> 2
 this -> 0
 this#1 -> 0
 this#3 -> 0
 this#6 -> 0
 tmp -> null
 tmp#5 -> 0
}
```

The result `([], 3)` represents a single thread with an empty set of held locks and the expression `3` (*cf.,* T in Figure 6). The heap maps integers ("addresses") to objects, in this case a single unlocked `Cell` object, with a field `cnt` of value 3. The "stack" is represented as a map from variables to values ("addresses", integers, or **null**). Since variable names are never recycled, fresh variable names are sometimes generated to avoid clashes (*e.g.,* `this#3`). These names may differ between different runs due to non-determinism in the scheduling of threads. For debugging purposes, the interpreter supports printing a representation of the state as above after every evaluation step.

The reason for implementing the interpreter in OCaml rather than in Coq, where correspondence to the formal semantics could be proven directly, is partly to be able to make the implementation simpler and partly to lower the threshold for a potential user. An OOlong program is not guaranteed to terminate, and since Coq requires all functions to be total, writing an interpreter would require tricks to work around this (*e.g.,* limiting evaluation to a maximum number of steps). We also believe that it is easier for someone without prior

experience to approach OCaml than it is to approach Coq. We leave implementing a formally verified interpreter in Coq for future work.

10. CONCLUSION

We have presented OOlong, an object calculus with concurrency and locks, with a focus on extensibility. OOlong aims to model the most important details of concurrent object-oriented programming, but also lends itself to extension and modification to cover other topics. A good language calculus should be both reliable and reusable. By providing a mechanised formalisation of the semantics, we reduce the leap of faith needed to trust the calculus, and also give a solid starting point for anyone wanting to extend the calculus in a rigorous way. Using Ott makes it easy to extend the calculus on paper and get usable LaTeX figures without having to spend time on manual typesetting. The prototype implementation offers an entry point for the more engineering-oriented researcher looking to experiment with new language features.

We have found OOlong to be a useful and extensible calculus, and by making our work available to others we hope that we will help save time for researchers looking to explore concurrent object-oriented languages in the future.

11. ACKNOWLEDGMENTS

We thank the anonymous reviewers for helpful feedback and suggestions for the original paper. The project is financially supported by the Swedish Foundation for Strategic Research (FFL15-0032), and by the Swedish Research Council Project 2014-05545, SCADA: Scalable Data for Pervasive Parallelism

12. REFERENCES

[1] N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World*. Springer, 2016.

[2] A. Banerjee and D. A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *CSFW*, volume 2, page 253, 2002.

[3] G. M. Bierman, M. Parkinson, and A. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical report, University of Cambridge, Computer Laboratory, 2003.

[4] R. Bocchino. An Effect System And Language For Deterministic-By-Default Parallel Programming, 2010. PhD thesis, University of Illinois at Urbana-Champaign.

[5] E. Castegren. Coq and Ott sources for OOlong. `https://github.com/EliasC/oolong`, 2018. GitHub.

[6] E. Castegren and T. Wrigstad. Reference Capabilities for Concurrency Control. In *ECOOP*, 2016.

[7] E. Castegren and T. Wrigstad. OOlong: An Extensible Concurrent Object Calculus. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1022–1029. ACM Press, 2018.

[8] A. Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

[9] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *ACM SIGPLAN Notices*, volume 33, pages 48–64. ACM, 1998.

[10] B. Delaware, W. R. Cook, and D. Batory. Fitting the pieces together: a machine-checked model of safe composition. In *ESEC-FSE*. ACM, 2009.

[11] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *ACM SIGPLAN Notices*, volume 35, pages 219–232. ACM, 2000.

[12] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183. ACM, 1998.

[13] T. Hoare. Null References: The Billion Dollar Mistake, 2009. Talk at QCon.

[14] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3), 2001.

[15] INRIA. Lexer and parser generators (ocamllex, ocamlyacc). `https://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html`, 2018. Tool tutorial.

[16] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *TOPLAS*, 28(4), 2006.

[17] J. Mackay, H. Mehnert, A. Potanin, L. Groves, and N. Cameron. Encoding Featherweight Java with assignment and immutability using the Coq proof assistant. In *FTfJP*, 2012.

[18] J. Östlund and T. Wrigstad. Welterweight Java. In *TOOLS*. Springer, 2010.

[19] A. M. Peters, D. Kitchin, J. A. Thywissen, and W. R. Cook. OrcO: a concurrency-first approach to objects. In *OOPSLA*, 2016.

[20] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hritcu, V. Sjöberg, and B. Yorgey. *Software Foundations*. Electronic textbook, 2017. Version 5.0.

[21] B. C. Pierce and D. N. Turner. Local Type Inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

[22] F. Pottier and Y. Régis-Gianas. What is Menhir? `http://gallium.inria.fr/~fpottier/menhir/`, 2018. Tool website.

[23] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46. ACM, 2011.

[24] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable Units of Behaviour. In L. Cardelli, editor, *ECOOP 2003*, volume 2743 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003.

[25] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective Tool Support for the Working Semanticist. In *ICFP*, 2007.

[26] R. Strňisa. *Formalising, improving, and reusing the Java Module System*. PhD thesis, St. John's College, United Kingdom, May 2010.

[27] M. van Dooren and W. Joosen. A modular type system for first-class composition inheritance, 2009.

APPENDIX

A. OMITTED RULES

This appendix lists the rules for deadlocked states, exception propagation, and the helper functions used in the main article. They should all be unsurprising but are included for completeness.

A.1 Blocking

The blocking property of a configuration holds if all its threads are either blocking on a lock or are done (*i.e.,* have reduced to a value). This property is necessary to distinguish deadlocks from stuck states.

$\boxed{\mathit{Blocked}(\mathit{cfg})}$ *(Configuration is blocked)*

$$\textsc{blocked-locked}\quad \frac{V(x) = \iota \qquad H(\iota) = (C, F, \textbf{locked}) \qquad \iota \notin \mathcal{L}}{\mathit{Blocked}(\langle H;\, V;\, (\mathcal{L}, \textbf{lock}(x)\,\textbf{in}\,e)\rangle)}$$

$$\textsc{blocked-deadlock}\quad \frac{\mathit{Blocked}(\langle H;\, V;\, T_1\rangle) \qquad \mathit{Blocked}(\langle H;\, V;\, T_2\rangle)}{\mathit{Blocked}(\langle H;\, V;\, T_1 \,||\, T_2 \rhd e\rangle)}$$

$$\textsc{blocked-left}\quad \frac{\mathit{Blocked}(\langle H;\, V;\, T_1\rangle)}{\mathit{Blocked}(\langle H;\, V;\, T_1 \,||(\mathcal{L}, v) \rhd e\rangle)}$$

$$\textsc{blocked-right}\quad \frac{\mathit{Blocked}(\langle H;\, V;\, T_2\rangle)}{\mathit{Blocked}(\langle H;\, V;\, (\mathcal{L}, v) \,||\, T_2 \rhd e\rangle)} \qquad \textsc{blocked-context}\quad \frac{\mathit{Blocked}(\langle H;\, V;\, (\mathcal{L}, e)\rangle)}{\mathit{Blocked}(\langle H;\, V;\, (\mathcal{L}, E[e])\rangle)}$$

A.2 Exceptions

Exceptions terminate the entire program and cannot be caught. The only rule that warrants clarification is the rule for exceptions in evaluation contexts which abstracts the nature of an underlying exception to avoid rule duplication (DYN-EXCEPTION-CONTEXT). For readability we abbreviate **NullPointerException** as **NPE**. When we don't care about the kind of exception we write **EXN**.

$\boxed{\mathit{cfg}_1 \hookrightarrow \mathit{cfg}_2}$ *(Exceptions)*

$$\textsc{dyn-npe-select}\quad \frac{V(x) = \textbf{null}}{\langle H;\, V;\, (\mathcal{L}, x.f)\rangle \;\hookrightarrow\; \langle H;\, V;\, \textbf{NPE}\rangle}$$

$$\textsc{dyn-npe-update}\quad \frac{V(x) = \textbf{null}}{\langle H;\, V;\, (\mathcal{L}, x.f = v)\rangle \;\hookrightarrow\; \langle H;\, V;\, \textbf{NPE}\rangle}$$

$$\textsc{dyn-npe-call}\quad \frac{V(x) = \textbf{null}}{\langle H;\, V;\, (\mathcal{L}, x.m(v))\rangle \;\hookrightarrow\; \langle H;\, V;\, \textbf{NPE}\rangle}$$

$$\textsc{dyn-npe-lock}\quad \frac{V(x) = \textbf{null}}{\langle H;\, V;\, (\mathcal{L}, \textbf{lock}(x)\,\textbf{in}\,e)\rangle \;\hookrightarrow\; \langle H;\, V;\, \textbf{NPE}\rangle}$$

$$\textsc{dyn-exception-async-left}\quad \frac{}{\langle H;\, V;\, \textbf{EXN} \,||\, T_2 \rhd e\rangle \;\hookrightarrow\; \langle H;\, V;\, \textbf{EXN}\rangle}$$

$$\textsc{dyn-exception-async-right}\quad \frac{}{\langle H;\, V;\, T_1 \,||\, \textbf{EXN} \rhd e\rangle \;\hookrightarrow\; \langle H;\, V;\, \textbf{EXN}\rangle}$$

$$\textsc{dyn-exception-context}\quad \frac{\langle H;\, V;\, (\mathcal{L}, e)\rangle \;\hookrightarrow\; \langle H';\, V';\, \textbf{EXN}\rangle}{\langle H;\, V;\, (\mathcal{L}, E[e])\rangle \;\hookrightarrow\; \langle H';\, V';\, \textbf{EXN}\rangle}$$

A.3 Helper Functions

This section presents the helper functions used in the formalism. Helpers **methods** and **fields** are analogous to **msigs**, and we refer to the mechanised semantics for details [5].

$$\textbf{vardom}(\Gamma) = \{x \mid x \in \textbf{dom}(\Gamma)\}$$

$$\textbf{msigs}(I) = \begin{cases} \mathit{Msigs} & \text{if } \textbf{interface } I\texttt{\{}\mathit{Msigs}\texttt{\}} \in P \\ \textbf{msigs}(I_1) \cup \textbf{msigs}(I_2) & \text{if } \textbf{interface } I \textbf{ extends } I_1, I_2 \in P \end{cases}$$

$$\textbf{msigs}(C) = \{\mathit{Msig} \mid \textbf{def } \mathit{Msig}\texttt{\{}e\texttt{\}} \in \mathit{Mds}\} \text{ if } \textbf{class } C ... \texttt{\{\_ }\mathit{Mds}\texttt{\}} \in P$$

$$\textbf{msigs}(t)(m) = x : t_1 \to t_2 \;\text{ if }\; m(x : t_1) : t_2 \in \textbf{msigs}(t)$$

$$\textbf{heldLocks}(T) = \begin{cases} \mathcal{L} & \text{if } T = (\mathcal{L}, e) \\ \textbf{heldLocks}(T_1) \cup \textbf{heldLocks}(T_2) & \text{if } T = T_1 \,||\, T_2 \rhd e \end{cases}$$

$$\textbf{locks}(e) = \{\iota \mid \textbf{locked}_\iota\texttt{\{}e'\texttt{\}} \in e\}$$

$$\mathit{distinctLocks}(e) \equiv |\textbf{locks}(e)| = |\textbf{lockList}(e)|$$
where $\textbf{lockList}(e) = [\iota \mid \textbf{locked}_\iota\texttt{\{}e'\texttt{\}} \in e]$