OOlong: An Extensible Concurrent Object Calculus

Elias Castegren
Tobias Wrigstad
elias.castegren@it.uu.se
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), and a mechanised version for doing rigorous proofs in Coq.

KEYWORDS

Object Calculi, Semantics, Mechanisation, Concurrency

ACM Reference format:
Elias Castegren and Tobias Wrigstad. 2018. OOlong: An Extensible Concurrent Object Calculus. In *Proceedings of SAC 2018: Symposium on Applied Computing , Pau, France, April 9–13, 2018 (SAC 2018),* 8 pages.
https://doi.org/10.1145/3167132.3167243

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 [12]. 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, 15, 21].

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 [20], and all type 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.

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 (§ 2–5).
- We provide a mechanised version of the full semantics and soundness proof, written in Coq (§ 6).
- We provide Ott sources for easily extending the paper version of the semantics and generating type rules in LaTeX (§ 7).
- We give two examples of how OOlong can be extended; support for assertions, and more fine-grained locking based on regions (§ 8).

2 RELATED WORK

The main source of inspiration for OOlong is Welterweight Java by Östlund and Wrigstad [15], a concurrent core calculus for Java with ease of reuse as an explicit goal. Welterweight 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 [16], 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.

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
SAC 2018, April 9–13, 2018, Pau, France
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/3167132.3167243

	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. The original formulation of Featherweight Java was not mechanised, but later extensions have been mechanised in Coq [14].

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

Figure 2: 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.

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 [11] or traits [19] becomes easier.

The smallest proposed candidate for a core Java calculus is probably Featherweight Java [12], 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, 14, 18]). The original formulation of Featherweight Java was not mechanised, but a later variation omitting casts and introducing assignment was mechanised in Coq (∼2300 lines) [14]. When developing mixins, Flatt *et al.* define `ClassicJava` [11], an imperative core Java calculus with classes and interfaces. It has been extended several times (*e.g.,* [8, 22]). Flanagan and Freund later added concurrency and locks to `ClassicJava` in `ConcurrentJava` [10], 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 also 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 [21], 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) [9].

Last, some language models go beyond the surface language and execution. One such model is Jinja by Klein and Nipkow [13], 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 (§ 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.

3 STATIC SEMANTICS OF OOLONG

In this section, we describe the formal 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 one are outlined in § 6.

Figure 2 shows the syntax of OOlong. 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.

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 parameter (multiple parameters can be simulated through object indirection). 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 **let** $\_ = e_1$ **in** e_2 (due to eager evaluation of e_1). Parallel threads are spawned with the expression **finish**{**async**{e_1} **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 **lock**(x) **in** e locks the object pointed to by x for the duration of e. While an expression locking ι is executed in the dynamic semantics, it appears as $\mathbf{locked}_\iota\{e\}$. 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.

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)

A well-formed program 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. 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)

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).

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

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

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

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

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

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

Figure 3: Well-formedness of classes and interfaces. The helper function msigs is defined in the appendix (*cf.* § A.3).

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

3.3 Expression Typing (Figure 5)

Most typing rules for expressions are simple. 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 **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*).

4 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 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.

[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.

$\boxed{\vdash t}$ *(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
$$\frac{\begin{array}{c}\forall\, \gamma : t \in \Gamma_2.\Gamma_1(\gamma) = t \\ \forall\, \gamma : t \in \Gamma_3.\Gamma_1(\gamma) = t \\ (\textbf{vardom}\,(\Gamma_2) \cap \textbf{vardom}\,(\Gamma_3)) \equiv \emptyset\end{array}}{\Gamma_1 = \Gamma_2 + \Gamma_3}$$

Figure 4: Typing, subtyping, typing environment and the frame rule. In the latter, γ abstracts over variables x and locations ι to reduce clutter. The helper function vardom extracts the set of variables from an environment (*cf.* § A.3).

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

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

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

Figure 6: Run-time constructs of OOlong. $\mathcal{L}$ is a set of locations whose locks are held by the current thread.

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**. The current semantics only supports the **NullPointerException**.

4.1 Well-Formedness Rules (Figure 7)

An OOlong 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). 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**. Locks must be taken at most once in e (captured by *distinctLocks*(e), *cf.* § A.3), and for each $\textbf{locked}_\iota$ in the current expression, 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

$\boxed{\Gamma \vdash \langle H; V; T\rangle : t}$ *(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
$$\frac{\forall \iota : C \in \Gamma. H(\iota) = (C, F, L) \wedge \Gamma; C \vdash F \qquad \forall \iota \in \mathbf{dom}(H). \iota \in \mathbf{dom}(\Gamma) \qquad \vdash \Gamma}{\Gamma \vdash H}$$

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

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

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
$$\frac{\vdash t \qquad \vdash \Gamma}{\Gamma \vdash \mathbf{EXN} : t}$$

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

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

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

Figure 7: Well-formedness rules. Note that well-formedness of threads is split into two sets of rules regarding expression typing and locking respectively.

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).

4.2 Evaluation of Expressions (Figure 8)

OOlong uses a small-step dynamic semantics, with the standard technique of evaluation contexts 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 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

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

$\boxed{cfg_1 \hookrightarrow cfg_2}$ *(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
$$\frac{x' \,\mathbf{fresh} \qquad V' = V[x' \mapsto v] \qquad e' = e[x \mapsto x']}{\langle H; V; (\mathcal{L}, \mathbf{let}\, x = v \,\mathbf{in}\, e)\rangle \hookrightarrow \langle H; V'; (\mathcal{L}, e')\rangle}$$

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

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

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

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

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

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

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

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

Figure 8: Dynamic semantics (1/2). Expressions. The evaluation context E is defined as
$E[\bullet] ::= x.f = \bullet \mid x.m(\bullet) \mid \mathbf{let}\, x = \bullet \,\mathbf{in}\, e \mid (t)\ \bullet \mid \mathbf{locked}_\iota\{\bullet\}$

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

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

DYN-EVAL-SPAWN-CONTEXT

$$\frac{\langle H; V; (\mathcal{L}, e) \rangle \hookrightarrow \langle H; V; (\mathcal{L}, e_1) \mid\mid (\emptyset, e_2) \rhd e_3 \rangle}{\langle H; V; (\mathcal{L}, E[e]) \rangle \hookrightarrow \langle H; V; (\mathcal{L}, e_1) \mid\mid (\emptyset, e_2) \rhd E[e_3] \rangle}$$

DYN-EVAL-ASYNC-JOIN

$$\frac{}{\langle H; V; (\mathcal{L}, v) \mid\mid (\mathcal{L}', v') \rhd e \rangle \hookrightarrow \langle H; V; (\mathcal{L}, e) \rangle}$$

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

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 evaluate to **null** (DYN-EVAL-UPDATE). We have omitted constructors from this treatise. 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.* § 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.* § A.2).

4.3 Concurrency (Figure 9)

OOlong models concurrency as non-deterministic choice between 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 $\rhd$ 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 $\rhd$ (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.* § 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.* § A.1).

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

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

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)*

WF-SUBSUMPTION

$$\frac{\forall\, \gamma : t \in \Gamma . \Gamma'(\gamma) = t}{\Gamma \subseteq \Gamma'}$$

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.*

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

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. □

6 MECHANISED SEMANTICS

We have fully mechanised the semantics of OOlong in Coq, including the proofs of soundness. The source code weighs in at ~4700 lines of Coq, ~1100 of which are definitions and ~3600 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 [17], as well as the `crush` tactic [7]. 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 need to be updated.

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

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 8.6.1, the latest version at the time of writing.

7 TYPESETTING OOLONG

The paper version of OOlong is written in Ott [20], which lets a user define the grammar and type 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].

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 and region based locking to the semantics. 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 available as examples on how to extend the 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 type rule for assertions states that the two variables are of the same type. The type of an assertion is **Unit**.

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

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

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

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

Note that the rules for exceptions already handle exception propagation, regardless of the kind of exception (*cf.* § 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 ("$\mathbf{lock}(x, r)$ **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:

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

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:

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

The changes can mostly be summarised as adding one extra level of indirection each time a lock status is looked up on the heap. This extension increases the mechanised semantics by ∼130 lines.

9 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.

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.

Acknowledgments We thank the anonymous reviewers for helpful feedback and suggestions.

REFERENCES

[1] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World*. Springer.

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

[3] Gavin M Bierman, MJ Parkinson, and AM Pitts. 2003. *MJ: An imperative core calculus for Java and Java with effects*. Technical Report. University of Cambridge, Computer Laboratory.

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

[5] E. Castegren. 2017. Coq and Ott sources for OOlong. https://github.com/EliasC/oolong. (2017). GitHub repository.

[6] E. Castegren and T. Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2016.5

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

[8] David G Clarke, John M Potter, and James Noble. 1998. Ownership types for flexible alias protection. In *ACM SIGPLAN Notices*, Vol. 33. ACM, 48–64.

[9] Benjamin Delaware, William R Cook, and Don Batory. 2009. Fitting the pieces together: a machine-checked model of safe composition. In *ESEC-FSE*. ACM.

[10] Cormac Flanagan and Stephen N Freund. 2000. Type-based race detection for Java. In *ACM SIGPLAN Notices*, Vol. 35. ACM, 219–232.

[11] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1998. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 171–183.

[12] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS* 23, 3 (2001).

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

[14] Julian Mackay, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas Cameron. 2012. Encoding Featherweight Java with assignment and immutability using the Coq proof assistant. In *FTfJP*.

[15] Johan Östlund and Tobias Wrigstad. 2010. Welterweight Java. In *TOOLS*. Springer.

[16] Arthur Michener Peters, David Kitchin, John A. Thywissen, and William R. Cook. 2016. OrcO: a concurrency-first approach to objects. In *OOPSLA*.

[17] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hritcu, Vilhelm Sjöberg, and Brent Yorgey. 2017. *Software Foundations*. Electronic textbook. Version 5.0.

[18] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, Vol. 46. ACM.

[19] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. 2003. Traits: Composable Units of Behaviour. In *ECOOP 2003*, Luca Cardelli (Ed.). Lecture Notes in Computer Science, Vol. 2743. Springer Berlin Heidelberg.

[20] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. 2007. Ott: Effective Tool Support for the Working Semanticist. In *ICFP*.

[21] R. Strňisa. 2010. *Formalising, improving, and reusing the Java Module System*. Ph.D. Dissertation. St. John's College, United Kingdom.

[22] Marko van Dooren and Wouter Joosen. 2009. A modular type system for first-class composition inheritance. (2009).

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.

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

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

BLOCKED-DEADLOCK
$$\frac{Blocked(\langle H; V; T_1\rangle) \qquad Blocked(\langle H; V; T_2\rangle)}{Blocked(\langle H; V; T_1 \mid\mid T_2 \rhd e\rangle)}$$

BLOCKED-LEFT
$$\frac{Blocked(\langle H; V; T_1\rangle)}{Blocked(\langle H; V; T_1 \mid\mid (\mathcal{L}, v) \rhd e\rangle)}$$

BLOCKED-RIGHT
$$\frac{Blocked(\langle H; V; T_2\rangle)}{Blocked(\langle H; V; (\mathcal{L}, v) \mid\mid T_2 \rhd e\rangle)}$$

BLOCKED-CONTEXT
$$\frac{Blocked(\langle H; V; (\mathcal{L}, e)\rangle)}{Blocked(\langle H; V; (\mathcal{L}, E[e])\rangle)}$$

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**.

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

DYN-NPE-SELECT
$$\frac{V(x) = \textbf{null}}{\langle H; V; (\mathcal{L}, x.f)\rangle \hookrightarrow \langle H; V; \textbf{NPE}\rangle}$$

DYN-NPE-UPDATE
$$\frac{V(x) = \textbf{null}}{\langle H; V; (\mathcal{L}, x.f = v)\rangle \hookrightarrow \langle H; V; \textbf{NPE}\rangle}$$

DYN-NPE-CALL
$$\frac{V(x) = \textbf{null}}{\langle H; V; (\mathcal{L}, x.m(v))\rangle \hookrightarrow \langle H; V; \textbf{NPE}\rangle}$$

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

DYN-EXCEPTION-ASYNC-LEFT
$$\frac{}{\langle H; V; \textbf{EXN} \mid\mid T_2 \rhd e\rangle \hookrightarrow \langle H; V; \textbf{EXN}\rangle}$$

DYN-EXCEPTION-ASYNC-RIGHT
$$\frac{}{\langle H; V; T_1 \mid\mid \textbf{EXN} \rhd e\rangle \hookrightarrow \langle H; V; \textbf{EXN}\rangle}$$

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

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].

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

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

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

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

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

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

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