Order Types: Static Reasoning about Message Races in Asynchronous Message Passing Concurrency

Mehdi Bagherzadeh
Oakland University, USA
mbagherzadeh@oakland.edu

Hridesh Rajan
Iowa State University, USA
hridesh@iastate.edu

Abstract

Asynchronous message passing concurrency with higher level concurrency constructs including activities, asynchronous method invocations and future return values is gaining increased popularity, as an alternative to shared memory concurrency with lower level threads and locks. However, similar to data races in shared memory concurrency, message races in asynchronous message passing concurrency can make a program incorrect and cause bugs that are hard to find and fix. This paper presents order types, a novel type system for static reasoning about message races in asynchronous message passing concurrency. Order types are local, causal and polymorphic types with the following features. First, order types encode both communication and flow behaviors and their happens-before relations. Second, order types are designed for an imperative calculus with concurrent activities, asynchronous method invocations, future return values, wait-by-necessity synchronizations on futures, first-class activities and futures, recursion and dynamic creation of activities. Third, order types are polymorphic and introduce universally quantified type variables to encode unknown values of variables. Fourth, the order type of a module can be inferred modularly using only its implementation and independent of implementations of other modules in the program. Order types complement previous work on static reasoning about races in asynchronous message passing concurrency.

CCS Concepts • **Software and its engineering → Formal software verification**; **Concurrent programming languages**;

Keywords Order types, message races, asynchronous message passing concurrency, messaging behavior, flow behavior, happens-before relation, static reasoning

AGERE'17, October 23, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5516-2/17/10…$15.00
https://doi.org/10.1145/3141834.3141837

ACM Reference Format:
Mehdi Bagherzadeh and Hridesh Rajan. 2017. Order Types: Static Reasoning about Message Races in Asynchronous Message Passing Concurrency . In *Proceedings of 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE'17).* ACM, New York, NY, USA, 10 pages.
https://doi.org/10.1145/3141834.3141837

1 Introduction

The concurrency revolution [45] has renewed interest in asynchronous message passing concurrency. Asynchronous message passing concurrency with higher level concurrency constructs, including activities, asynchronous method invocations and future return values [51], is an alternative to shared memory concurrency with lower-level constructs, including threads and locks. In imperative asynchronous message passing concurrency, a program is a set of sequential activities that run concurrently, communicate via asynchronous invocations of their methods and synchronize on their future return values. In response to a method invocation, an activity can change state, invoke a method of another activity or create a new activity.

Concurrent activities with asynchronous method invocations are similar to actors [3, 4] in ActorFoundry [6], Rosette [48], HAL [23] and THAL [27], active objects in asynchronous sequential processes (ASP) [11], concurrent objects in abstract behavioral specification (ABS), coboxes in JCoBox [43] and capsules in Panini [7, 8, 41, 42]. Asynchronous method invocations [6, 11] are type-safe message sends and share similarities with future messages in ABCL/1 [53] and remote procedure calls (RPCs) in THAL.

The use of the phrase activity [11] allows referring to a broad range of active entities with a unified nomenclature. It also allows distinguishing between active and passive entities. An entity, such as object, is active if it encapsulates its thread of execution.

Similar to data races in shared memory concurrency, message races in asynchronous message passing concurrency can make a program incorrect and cause bugs that are hard to find and fix [32, 33, 46, 47]. Two messages (asynchronous method invocations) are racy if they arrive (are invoked) at an activity with no happens-before relation [31] between them. Happens-before is a partial order that denotes the execution ordering among events of a program. However, static

reasoning about message races in an asynchronous message passing concurrency program is challenging because for each activity of the program such reasoning requires the understanding of:

1. All messages (invocations) sent by the activity directly or indirectly, and
2. All synchronizations on future return values of these invocations, and
3. All happens-before relations between message sends and future synchronizations.

This paper presents *order types*, a novel type system for static reasoning about message races in imperative message passing concurrency. Order types guarantee that a well-typed program is free from message races.

Order types are local, causal [40] and polymorphic types with the following novel features.

Encode both communication and flow behaviors and their happens-before relations The order type of an activity encodes both its communication and flow behaviors and their happens-before relations. Communication behaviors include message sends via method invocations and synchronizations on their future return values. Flow behaviors include flows of activity instances and future values into variables. Happens-before relations encode ordering between communication and flow behaviors.

Allow static reasoning about message races in imperative asynchronous message passing concurrency Order types allow static reasoning about message races in programs in an imperative calculus with concurrent activities, asynchronous method invocations, future return values, wait-by-necessity synchronizations [10, 11], first-class activities and futures, mutually recursive method invocations and dynamic creation of activities [28]. In wait-by-necessity synchronization, an invoking activity synchronizes with the invocation only when it needs its return value. First-class activities and futures can be stored in a state of an activity, and passed and returned to and from invocations of its methods.

Use polymorphism to encode unknown values The order type of an activity introduces a universally quantified type variable to encode the unknown value of a variable, during the type inference phase. This makes the order type polymorphic. The quantified type variable is eliminated when its value is known, during the type closure phase. The type closure propagates the flow information among activities of a program to construct its order graph. The order graph of the program containts its communication and flow behaviors and their happens-before relations. The order graph is later analyzed for absence of message races.

Modular inference Use of polymorphism to encode unknown values of an activity (module) allows the type of the activity to be inferred modularly, using only its implementation and independent of implementations of other activities in the program.

1.1 Overview of Related Work

Among previous work on static reasoning about races in asynchronous message passing concurrency, closest to our work are Honda et al.'s global session types [21, 22] for functional session programming on top of channel programming in π calculus, Igarashi and Kobayashi's generic type system for functional channel programming in π calculus and Crafa's behavioral types [13] for session programming on top of standard functional actors. These works allow for static reasoning about channel races (linearity). However, they are not directly applicable to static reasoning about message races in an imperative calculus with concurrent activities, asynchronous method invocations, future return values, wait-by-necessity synchronizations, first-class activities and futures, mutually recursive invocations and dynamic creation of activities with no support for channel and session programming. Order types complement previous work on static reasoning about races in asynchronous message passing concurrency. Detailed comparison with previous work can be found in §4.

2 Program Syntax

Figure 1 shows the program syntax for a core imperative asynchronous message passing calculus. The calculus is inspired by previous work [6, 8, 11, 26, 43] and is simplified to focus on message races. In Figure 1, the superscript notation $\overline{term}$ denotes a sequence of zero or more *term*s.

$prog ::= \overline{decl}$	program
$decl ::= \textbf{activity}\ C\ \{\ \overline{state}\ \overline{meth}\ \}$	activity
$state ::= T\ x;$	state
$meth ::= T\ m(\overline{T\ x})\{\ e\ \}$	method
$e ::=$	*expression:*
$\mid x.^{[\alpha]}\ m(\overline{x})$	invocation
$\mid \textbf{new}^{[\overline{\alpha}]}\ C(\overline{x})$	activity instantiation
$\mid \textbf{future}^{[\alpha]}\ (x)$	future creation
$\mid \textbf{wait}\ x$	synchronization
$\mid e;e$	sequence
$\mid x = e$	assign
$\mid x$	variable
$\mid \textbf{self}$	self
$\mid \textbf{null}$	null
$T ::=$	*value type:*
$\mid C$	activity
$\mid \textbf{Fut}(T)$	future

Figure 1. Program syntax for a core imperative concurrent asynchronous message passing calculus.

Declarations A program is a set of activity declarations with the activity *Main* as the entry point to the program. An activity has a name and includes sets of state and method

declarations. A state declaration includes the value type of the state and its name. A method declaration includes its return type, name, set of parameter declarations and body. The activity *Main* has no state and declares a single method *main* with no parameters.

Value types A value type is either the name of an activity declaration or a future type. All types in the programmer syntax are value types. Order types are inferred automatically and are not present in the programmer syntax.

Expressions An expression is an asynchronous method invocation, activity instantiation, future creation, future synchronization, sequence, assign, variable, pseudo-variable ***self*** and value ***null***. A loop is implemented using mutually recursive method invocations. A synchronous method invocation can be implemented using an asynchronous invocation and an immediate synchronization on its future return value.

Programmer syntax vs annotated syntax An annotation α is a type variable that denotes the return value of a method invocation, activity instantiation and future creation. Type variables are part of an automatically annotated syntax used to infer order types are not written by programmers.

2.1 Dynamic Semantics

An asynchronous message passing program is a set of activity instances that run concurrently. An activity instance encapsulates a single thread of execution and is sequential. An activity instance communicates with another instance via asynchronous invocations and future return values.

An activity enqueues an asynchronous invocation of its methods in its queue and returns with a future return value without waiting for its execution. The activity dequeues an invocation from its queue, executes it to the finish and updates its future return value with the result of the invocation. The execution of an invocation in the queue starts only after the execution of the previously dequeued invocation terminates. Futures allow for wait-by-necessity synchronization [11] where the caller synchronizes with the invocation only when it needs the return value. Future and activity values are first-class values that can be stored in a state of an activity, and passed and returned to and from an invocation of its methods. In response to an invocation, an activity can update its state, invoke another method or create a new activity.

Features of our calculus are present in other asynchronous message passing calculi and languages including ActorFoundry [6], HAL [23], THAL [27], ASP [11], ABS [26], JCoBox [43] and Panini [8]. For simplicity and without loss of generality, our calculus does not include passive objects present in some of these calculi and languages. A passive object is a regular object owned by an active object (activity) with no thread and queue and with synchronous invocation of its methods. Message races manifest and can be studied independent of passive objects [32, 47].

2.2 Message Race Example

Figure 2 illustrates a message race and the incorrect behavior and the bug it can cause. The figure shows a utility program inspired by a real world message race bug from previous work [47]. The code declares a utility activity *Utl*, a proxy activity *Pr* and a server activity *Sr*. *Utl* is responsible to save its client work in a server *s*, via a proxy *r*, and shut down the server afterwards. The proxy *r* is responsible for preprocessing the utility's save request, including authentication (not shown), before relaying it to the server.

```
1  activity Utl{
2    Sr s; Pr r; Fut(Sr) f;
3    Fut(Sr) save(){
4      s = new Sr();
5      r = new Pr(s);
6      f = r.save();
7      wait f;
8      s.kill(); } }
```

```
racy alternative
6      f = r.save();
7      s.kill();
8      wait f; }
```

```
9  activity Pr{
10   Sr s;
11   Pr (Sr _s){
12     s = _s;
13     self; }
14   Fut(Sr) save(){
15     s.save(); }
16 }
```

```
17 activity Sr{
18   Fut(Sr) save(){
19     future(self); }
20   Fut(Sr) kill(){
21     future(null); }
22 }
```

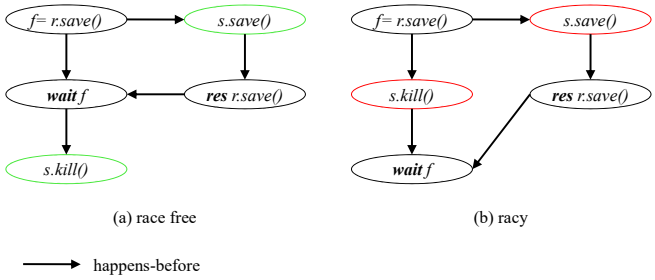

Figure 2. Race-free utility and its racy alternative in gray.

The activity *Utl* declares its states *s*, *r* and *f* of types *Sr*, *Pr* and ***Fut***(*Sr*). To save the client's work, the method *save* of *Utl* after instantiating its states *(1)* invokes the method *save* on its proxy instance *r* to save the work in the server *s* *(2)* assigns the future return value of the invocation to *f* *(3)* blocks on the future *f* to ensure that save is complete before *(4)* it invokes the method *kill* of the server *s* to shut down the server. The proxy *Pr* declares its state *s* of type *Sr*, a constructor method *Pr* that initializes the value of *s* and a method *save* that forwards the save request to its server instance *s* and returns the future result of this invocation. The server activity *Sr* declares two methods *save* and *kill* that save the client's work in the server and shut down the server.

Free from message race In Figure 2, *Utl* in the white background is free from message races and correctly saves the client's work in the server and shuts down the server afterwards. *Utl* is not racy because, as the program's order graph (a) shows, there is a happens-before relations between

the message *s.save()* that *Utl* sends indirectly via the proxy *r* and the message *s.kill()* that it sends directly. This happens-before relation means that *s.save()* arrives at the server before *s.kill()* which in turn allows for saving the client's work before the server is shut down. Happens-before relation is transitive.

An order graph shows message sent directly and indirectly by an activity, synchronizations on their future return values and happens-before relations between them. In the order graph (a) , *r.save()* denotes a message send (invocation), whereas ***res*** *r.save()* denotes the end of the execution of the invocation and resolution of its future result. A future must be resolved before any blocking wait on the future can be unblocked. That is, ***res*** *r.save()* happens-before ***wait*** *f* where *f* is the return value of *r.save()*.

Suffer from message race However, *Utl* in the gray background suffers from a message race and could incorrectly shut down the server before saving the client's work. This is because, as the order graph (b) shows, there is no happens-before relation between *s.save()* and *s.kill()*. Therefore, *s.kill()* can arrive at the server before *s.save()* shutting down the server before saving client's work.

For message delivery and processing ordering, this example assumes first-in-first-out (FIFO) where messages are processed in the same order they are delivered.

3 Order Types

Type checking of a program using order types involves the following three phases:

- ***Type inference*** To automatically and modularly infer the order type of an expression, method and activity of the program.
- ***Type closure*** To propagate the flow information among activities of the program and construct the order graph of the program.
- ***Causality analysis*** To analyze the order graph of the program for message races.

3.1 Order Type Syntax

Figure 3 shows the syntax for order types, inspired by [19, 30]. The type judgement denotes that an expression e has the order type $\alpha\backslash\sigma$ in the typing environment Γ. The typing environment is a map from a variable name to its value. The order type encodes the value of the expression in the type variable α and its communication and flow behaviors and their happens-before relations in the behavior σ. A value is an activity value γ or a future value $\alpha(\alpha)$. A behavior σ is an empty behavior $\bullet$, a communication behavior κ, a flow behavior ϕ or a happens-before behavior $\sigma \lhd \sigma$.

Communication behaviors The communication behavior includes the invocation (message send), synchronization and resolution behaviors. The invocation behavior $\alpha._{[\alpha'']}m(\overline{\alpha'})$ denotes the asynchronous invocation of a method m on a receiver α with arguments $\overline{\alpha'}$ and return value α''. The synchronization behavior ***wait*** α denotes synchronization on a future return value α. The resolution behavior ***res*** $\alpha._{[\alpha'']}m(\overline{\alpha'})$ denotes the end of the execution of the invocation $\alpha._{[\alpha'']}m(\overline{\alpha'})$ and the resolution of its future value α''.

Flow behaviors The flow behavior $\alpha \preceq \alpha'$ denotes the flow of an activity or future value α to another value α'. The instantiation behavior $[\overline{\alpha}]^C \preceq \alpha$ denotes the flow of an instantiated activity $[\overline{\alpha}]^C$ to a value α. $[\alpha, \overline{\alpha'}]^C$ denotes an activity value α of type C with states $\overline{\alpha'}$.

Happens-before behaviors A happens-before behavior $\sigma \lhd \sigma'$ denotes that the behaviors in σ happen before the behaviors in σ'.

σ ::=	*behavior:*
$\mid \bullet$	empty
$\mid \kappa$	communication
$\mid \phi$	flow
$\mid \sigma \lhd \sigma$	happens-before
κ ::=	*communication:*
$\mid \alpha._{[\alpha]}m(\overline{\alpha})$	invocation
$\mid$ ***wait*** α	synchronization
$\mid$ ***res*** $\alpha._{[\alpha]}m(\overline{\alpha})$	resolution
ϕ ::=	*flow:*
$\mid \alpha \preceq \alpha$	value
$\mid [\overline{\alpha}]^C \preceq \alpha$	instantiation
α ::=	*type variable:*
$\mid \gamma$	activity
$\mid \alpha(\alpha)$	future
$\Gamma \vdash e : \alpha\backslash\sigma$	judgement
Γ ::= $\epsilon \mid \Gamma,\ x : \alpha$	environment

Figure 3. Syntax of order types.

3.2 Type Inference

The type inference automatically infers the order type of an expression, method and activity of the program. Type inference operates on an automatically annotated program in which fresh type variable annotations denote the return value of a method invocation, activity instantiation and future creation, as Figure 1 shows.

3.3 Inference Rules

Figure 4 shows the inference rules for order types, inspired by [19, 30, 50]. Figure 9 shows the auxiliary functions used in the type inference rules.

3.3.1 Expressions

In (T-Inv), the value of the invocation expression $x._{[\alpha'']}m(\overline{x'})$ is its return value α'' and its behavior is the invocation behavior $\alpha._{[\alpha'']}m(\overline{\alpha'})$ where α and $\overline{\alpha'}$ are values of the receiver

$$\text{(T-Inv)}\ \frac{\Gamma \vdash x : \alpha\backslash\bullet \qquad \forall x' \in \overline{x'}.\ \Gamma \vdash x' : \alpha'\backslash\bullet \qquad \alpha = \gamma}{\Gamma \vdash\ x._{[\alpha'']} m(\overline{x'}) : \alpha''\backslash\alpha._{[\alpha'']} m(\overline{\alpha'})}$$

$$\text{(T-Wait)}\ \frac{\Gamma \vdash x : \alpha(\alpha')\backslash\bullet}{\Gamma \vdash \textbf{\textit{wait}}\ x : \alpha'\backslash\textbf{\textit{wait}}\ \alpha(\alpha')}$$

$$\text{(T-Fut)}\ \frac{\Gamma \vdash x : \alpha'\backslash\bullet}{\Gamma \vdash \textbf{\textit{future}}_{[\alpha]}(x) : \alpha(\alpha')\backslash\bullet}$$

$$\text{(T-New)}\ \frac{\forall x' \in \overline{x'}.\ \Gamma \vdash x' : \alpha''\backslash\bullet}{\Gamma \vdash \textbf{\textit{new}}_{[\alpha,\overline{\alpha'}]} C\ (\overline{x'}) : \alpha\backslash([\alpha,\overline{\alpha'}]^C \preceq \alpha) \lhd \alpha._{[\alpha,\overline{\alpha'}]} C(\overline{\alpha''}) \lhd \textbf{\textit{wait}}\ \alpha}$$

$$\text{(T-Var)}\ \frac{\Gamma(x) = \alpha}{\Gamma \vdash x : \alpha\backslash\bullet} \qquad \text{(T-Null)}\ \frac{\mathit{fresh}(\alpha)}{\Gamma \vdash \textbf{\textit{null}} : \alpha\backslash\bullet} \qquad \text{(T-Asgn)}\ \frac{\Gamma \vdash x : \alpha'\backslash\bullet \qquad \Gamma \vdash e : \alpha\backslash\sigma}{\Gamma \vdash x = e : \alpha\backslash\sigma \lhd (\alpha \preceq \alpha')}$$

$$\text{(T-Seq)}\ \frac{\Gamma \vdash e' : \alpha'\backslash\sigma' \qquad \Gamma \vdash e'' : \alpha''\backslash\sigma''}{\Gamma \vdash e; e' : \alpha''\backslash\sigma' \lhd \sigma''}$$

$$\text{(T-Meth)}\ \frac{\Gamma, \overline{x' : \alpha'} \vdash e : \alpha\backslash\sigma \qquad \mathit{fresh}(\alpha'') \qquad \mathit{fresh}(\overline{\alpha'}) \qquad \mathcal{A} = \{\alpha'', \overline{\alpha'}\} \qquad \mathcal{A}' = \mathit{labels}(e)}{\Gamma \vdash T\ m(\overline{T'\ x'})\{e\} : \forall\mathcal{A}.\ \forall\mathcal{A}'.\ \sigma \lhd (\alpha \preceq \alpha'')}$$

$$\text{(T-Actv)}\ \frac{\forall\mathcal{A}.\Gamma = \mathit{states}(C) \qquad \forall T\ m(\overline{T'\ x'})\{e\} \in \overline{\mathit{meth}}.\ \Gamma \vdash T\ m(\overline{T'\ x'})\{e\} : \forall\mathcal{A}'.\ \forall\mathcal{A}''.\ \sigma}{\vdash \textbf{\textit{activity}}\ C\ \{\overline{\mathit{state}}\ \overline{\mathit{meth}}\}\backslash\ \overline{m : \forall\mathcal{A}.\ \forall\mathcal{A}'.\ \forall\mathcal{A}''.\ \sigma}}$$

$$\text{(T-Prog)}\ \frac{\begin{array}{l}\forall\ \textbf{\textit{activity}}\ C\ \{\overline{\mathit{state}}\ \overline{\mathit{meth}}\} \in \overline{\mathit{decl}}.\ \vdash \textbf{\textit{activity}}\ C\ \{\overline{\mathit{state}}\ \overline{\mathit{meth}}\}\backslash\overline{m : \forall\mathcal{A}.\ \forall\mathcal{A}'.\ \forall\mathcal{A}''.\ \sigma} \\ \mathit{fresh}(\beta_0) \qquad \Omega_0 = \left([\beta_0]^{Main} \preceq \beta_0\right) \lhd \beta_{0\cdot[\beta_0]} main() \qquad \Omega_0 \overset{\epsilon}{\hookrightarrow} \Omega \qquad \Omega \models \diamond\end{array}}{\vdash \overline{\mathit{decl}} : \overline{C : \overline{m : \forall\mathcal{A}.\ \forall\mathcal{A}'.\ \forall\mathcal{A}''.\ \sigma}}}$$

Figure 4. Modular type inference rules for order types.

x and arguments $\overline{x}$. (T-Inv) assumes that the receiver of an invocation is an activity value γ and not a future value.

Delayed invocation contour In a polymorphic type system like order types, similar to let polymorphism, the type of the body of an invoked method should be refreshed before it is integrated into the type of its invocation. However, in the presence of mutually recursive method invocations, the naive refreshing of the type of the method body could lead to nontermination in type inference. To address this issue, (T-Inv) uses a delayed contour $\alpha._{[\alpha'']} m(\overline{\alpha'})$ to represent the behavior of the invocation [30, 50]. Later, as discussed in §3.4, the type closure phase integrates the invocation contour with the refreshed type of the body of the invoked method, but in the presence of a detection mechanism that can detect mutual recursion and therefore avoid the nontermination.

In (T-Wait), the value of a synchronization expression ***wait*** x is α' where α' is the unwrapping of the value $\alpha(\alpha')$ for the future x. The behavior of the wait expression is a wait behavior ***wait*** $\alpha(\alpha')$. In (T-Fut), the value of a future creation expression $\textbf{\textit{future}}_{[\alpha]}(x)$ is the future value $\alpha(\alpha')$ which is the wrapping of the value α' of its parameter x into its return value α. The behavior of the future expression is the empty behavior $\bullet$.

In (T-New), the value of a new expression $\textbf{\textit{new}}_{[\alpha,\overline{\alpha'}]} C\ (\overline{x'})$ is the value α. The behavior of the new expression is a happens-before behavior including the following three behaviors. The flow behavior $[\alpha,\overline{\alpha'}]^C \preceq \alpha$ to instantiate the activity instance α with states $\overline{\alpha'}$ where $\overline{\alpha'}$ are values for fields of α. The invocation behavior $\alpha._{[\alpha,\overline{\alpha'}]} C(\overline{\alpha''})$ to invoke the constructor C on activity α with parameters $\overline{\alpha''}$. The wait behavior ***wait*** α to wait on the future return value of the constructor invocation. The new expression is encoded as a synchronous method invocation.

In (T-Var), the value of a variable x is its value α in the typing environment Γ and its behavior is empty. In (T-Null) the value of ***null*** is a fresh value α and its behavior is empty. Auxiliary function *fresh* ensures that a type variable is unique.

In (T-Asgn), the value of an assign expression $x = e$ is the value α of its expression e. The behavior of the expression is a happens-before behavior in which the behavior σ of the expression e happens-before the flow behavior $\alpha \preceq \alpha'$. The flow behavior encodes the flowing of the value α of e to the value α' of x.

In (T-Seq), the value of a sequence expression $e'; e''$ is the value α'' of e''. The behavior of the expression is a happens-before behavior $\sigma' \lhd \sigma''$ in which the behavior σ' of e' happens-before the behavior σ'' of e''.

3.3.2 Declarations

A declaration does not have a value and therefore its order type only encodes the behavior of the declaration.

In the rule (T-Meth), the behavior of a method declaration $T\ m(\overline{T'\ x'})\{e\}$ is a happens-before behavior in which the behavior σ of the method body e happens-before the flow behavior $\alpha \preceq \alpha''$ in which the value α of the body flows into the return value α'' of the method. The auxiliary function *labels* returns the type variable annotations of an expression.

Universal quantifications of sets of type variables $\mathcal{A}$ and $\mathcal{A}'$ denote that values of these type variables are unknown.

In (T-Actv), an activity declaration maps a name m of its methods to its order type. The type of the method of an activity is inferred in a type environment Γ that maps the pseudo-variable ***self*** and states of the activity to unique type variables. The auxiliary function *states* constructs this typing environment using function *tvar*.

Modular inference The use of the type environment Γ with type variables with unknown values allows for modular inference of the order type of an activity in (T-Actv) using only implementation of its methods and independent of implementations of other activities in a program.

In (T-Prog), a program declaration maps a name C of an activity to its order type. (T-Prog) is the end of the modular type inference phase for the program and the start of its type closure phase.

Type closure The type closure constructs the order graph of the program and is defined by the relation $\overset{\Delta}{\hookrightarrow}$ where Δ denotes a call sequence. In (T-Prog), type closure constructs the order graph Ω of the program from the initial order graph Ω_0 and the initial empty call sequence ϵ. In Ω_0, the flow behavior $[\beta_0]^{Main} \preceq \beta_0$ represents the instantiation of activity β_0 of type *Main* and the invocation behavior $\beta_{0\cdot[\beta_0]}main()$ represents the invocation of method *main* on β_0. Recall that the activity *Main* is the entry point to the program with a single method *main* with no parameters and no states. The return type variable β_0 in this invocation behavior can be any arbitrary type variable. Unlike inference phase that uses type variables α with unknown values, the type closure phase uses type variables β with known values.

Causality analysis After the construction of the order graph Ω of a program, (T-Prog) uses a causality analysis of the order graph to ensure there are no message races. The relation $\Omega \models \diamond$ encodes this causaility analysis as the wellformedness of the order graph.

3.3.3 Illustration of Type Inference

Figure 5 illustrates the inference of order types for the proxy *Pr* in the utility program in Figure 2.

In (T-Act), the order type of the activity *Pr* and its methods is inferred using the type environment Γ that maps values of its variable ***self*** and state s to arbitrary but unique type variables α_{200} and α_{201}. In (T-Meth), the order type θ_{Pr} for the constructor *Pr* is inferred after mapping its return value $\boldsymbol{ret}_{pr}$ and parameter s to unique type variables α_{202} and α_{203}. θ_{Pr} is a happens-before relation in which the behavior $((\alpha_{203} \preceq \alpha_{201}) \triangleleft \bullet)$ of the body of the constructor happens-before the behavior $\alpha_{200} \preceq \alpha_{202}$. The flow behavior $\alpha_{200} \preceq \alpha_{202}$ denotes the flow of the value α_{200} of the body of the constructor *Pr* to its return value α_{202}. The type of the body of the constructor is inferred using (T-Seq). θ_{12} and θ_{13} show the order types of the expressions in the body of the constructor, inferred

$\Gamma = \boldsymbol{self} : \alpha_{200}, s : \alpha_{201}$

```
9  activity Pr {
10   Sr s;
```

$\boldsymbol{ret}_{Pr} : \alpha_{202}, \_s : \alpha_{203}$

$\theta_{Pr} = \forall\alpha_{202}, \alpha_{203}.\forall.\ ((\alpha_{203} \preceq \alpha_{201}) \triangleleft \bullet) \triangleleft (\alpha_{200} \preceq \alpha_{202})$

```
11   Pr (Sr _s) {
12     s = _s;
13     self; }
```

$\theta_{12} = \alpha_{203} \backslash \alpha_{203} \preceq \alpha_{201}$

$\theta_{13} = \alpha_{200} \backslash \bullet$

$\boldsymbol{ret}_{save} : \alpha_{204}(\alpha_{205})$

$\theta_{save} = \forall...\forall... \ \alpha_{201\cdot[\alpha_{206}]}save() \triangleleft (\alpha_{206} \preceq \alpha_{204}(\alpha_{205}))$

```
14   Fut(Sr) save(){
15     s.[α206]save(); }
16   }
```

$\theta_{15} = \alpha_{206} \backslash \alpha_{201\cdot[\alpha_{206}]}save()$

Figure 5. Type inference for the proxy activity *Pr* in Figure 2.

using (T-Asgn) and (T-Var). Similarly, the order type θ_{save} for the method *save* is inferred using (T-Inv).

3.4 Type Closure

The type closure phase propagates the flow information among activities of a program to construct its order graph. The order graph of a program is a graph with its communication and flow behaviors as its nodes and their happens-before relations as its edges.

Closure copy rule Type closure uses a variation of Morgan's copy rule [35] to propagate flow information via method invocations. Unlike the original copy rule that replaces a method invocation with functional specification of its method body, the closure copy rule relates the order type of a method invocation with the order type of the method body. The closure copy rule says that the type of the invocation happens before the type of the method body, after substitution of method's parameters with invocation's arguments in the type of the method body. Type closure repeatedly applies the copy rule to method invocations in the order graph until it reaches a fixpoint.

The closure copy rule allows the type closure to eliminate the universal quantification of a type variable α with an unknown value in the body of a method and replace it with a type variable β with a known value.

Type closure is a call sensitive and object sensitive relation [34, 50].

3.5 Order Graph Syntax

Figure 6 shows the syntax for order graphs. Order graph syntax is similar to the order type syntax in Figure 1 except for the use of type variables β with known values instead of type variables α with unknown values and the addition of contextual behavior $\langle\omega\rangle^{\delta}$ which encodes a behavior ω that happens in the context of an invocation (call) δ.

$\Omega \overset{\Delta}{\hookrightarrow} \Omega$	type closure
$\omega, \Omega ::=$	*order graph:*
$\mid \bullet$	empty
$\mid \varkappa$	communication
$\mid \varphi$	flow
$\mid \omega \lhd \omega$	happens-before
$\mid \langle \omega \rangle^{\delta}$	contextual
$\beta ::=$	*type variable:*
$\mid \nu$	activity
$\mid \beta(\beta)$	future
$\varkappa ::=$	*communication:*
$\mid \beta._{[\beta]} m(\overline{\beta})$	invocation (contour)
$\mid$ ***wait*** β	synchronization
$\mid$ ***res*** $\beta._{[\beta]} m(\overline{\beta})$	resolution
$\varphi ::=$	*flow:*
$\mid \beta \preceq \beta$	activity
$\mid [\overline{\beta}]^{C} \preceq \beta$	new
$\Delta ::= \delta :: \Delta$	call sequence
$\delta ::= (\beta, C, m)$	call

Figure 6. Order graph syntax.

3.6 Type Closure Rules

Type closure is a transition from one (partial) order graph to another under the call sequence. Figure 7 shows the type closure rules for order types.

In (C-Hapn-Inv), the closure copy rule replaces a method invocation behavior contour $\beta._{[\beta'']} m(\overline{\beta'})$ with a behavior in which the invocation behavior happens before the behavior ω'' of its body. Similar to let polymorphism, the behavior of the method body should be refreshed before it can be related to the behavior of its invocation. Substitution of unknown type variables $\mathcal{A}''$ that annotate the body of the method in its behavior σ with fresh known type variables $\mathcal{B}''$ refreshes the behavior of the method body. The auxiliary function *generate* takes type variables annotating the body and a call sequence and generates fresh type variables. In addition to refreshing, (C-Hapn-Inv) substitutes in σ the unknown type variables for the ***self*** and its states as well as method parameters and return value, with corresponding known type variables β''', $\overline{\beta''''}$, $\overline{\beta'}$ and β'', respectively. The flow judgement $\Omega \vdash [\overline{\beta'}]^{C} \preceq \beta$ tracks the receiver β to its instantiation site to know its value and values of its fields. After substitution, (C-Hapn-Inv) augments the behavior ω' of the method body with a resolution behavior ***res*** $\beta._{[\beta'']} m(\overline{\beta'})$ that marks the end of the execution of the method body and resolution of its future return value. The augmented behavior is wrapped in a contextual behavior to record the invocation δ that lead to the behavior of the method body. *mtype* returns the order type of a method.

Recursion detection In the presence of mutually recursive invocations, naive refreshing of the body of a method invocation can lead to nontermination of the type closure. To prevent nontermination, the body of a mutually recursive method invocation is refreshed only if the method invocation is not present in the call sequence. The function *generate* uses *collapse* to adapt mutually recursive invocations by generating the same type variables for the body of a mutually recursive invocation, if the invocation already exists in the call sequence. In other words, type variables for the previous invocations are reused. For a mutually recursive invocation, *generate* unfolds the invocation twice before reusing the type variables. Twice unfolding to represent loops and recursive invocations is inspired by previous work [22].

Circular order graph A mutually recursive invocation and reuse of type variables leads to a backedge happens-before edge and a consequent cycle in the order graph. This makes the order graph ill-formed because the happens-before relation is not circular. To address this issue, *generate* returns ***true*** if the type variables it generates are reuse of previous type variables and otherwise returns ***false***. This allows for detection of backedges and their proper handling, as discussed later. (C-Hapn-Inv-Cir) records backedges in a set $\mathcal{L}$. (C-Hapn-Inv-Cir) is similar to (C-Hapn-Inv) except that the invocation in (C-Hapn-Inv) is not mutually recursive.

In (C-Hapn-Wait), a synchronization behavior ***wait*** β''' is reduced to a behavior in which the resolution behavior ***res*** $\beta._{[\beta'']} m(\overline{\beta'})$ of a method invocation happens before the future synchronization behavior where its return value β'' of the invocation flows into the future β'''.

(C-Hapn-Redc) allows for the reduction of a happens-before behavior and encodes its reduction order. (C-Ctxt-Redc) allows for the reduction of a contextual behavior.

Flow and happens-before behaviors Flow behaviors form a reflexive transitive relation, denoted by (C-Flow-Reflx) and (C-Flow-Trans). Happens-before behaviors form a transitive, irreflexive and antisymmetric relation, denoted by the rule (C-Hapn-Trans). In a happens-before behavior $\omega \lhd \omega'$, all behaviors in ω happen before all behaviors in ω'. (C-Hapn-Comp) encodes this by relating the last elements of ω' to the first elements of ω in a happens-before relation. All other behaviors in ω and ω' are in happens-before relations. *first* and *last* return the first and last elements of a behavior.

Message ordering and processing (C-Hapn-FIFO) encodes first-in-first-out (FIFO) message delivery and processing in which messages are delivered in the same order they appear in program text and are processed in the same order they are delivered [22].

3.7 Causality Analysis for Race Detection

Causality analysis checks the order graph of a program for message races. Two messages are racy if they are sent to the same activity and there is no happens-before relation

$$\text{(C-Hapn-Inv)}\quad \frac{\delta = (\beta, C, m) \qquad mtype(C,m) = \forall\mathcal{A}.\,\forall\mathcal{A}'.\,\forall\mathcal{A}''.\,\sigma \qquad \Delta' = \delta :: \Delta \qquad (\mathcal{B}'', \textbf{\textit{false}}) = generate\left(\Delta', \mathcal{A}''\right) \\ \Omega \vdash [\beta''', \overline{\beta''''}]^{C} \preceq \beta \qquad \omega' = \sigma\left[\{\beta''', \overline{\beta''''}\}/\mathcal{A}, \{\beta'', \overline{\beta'}\}/\mathcal{A}', \mathcal{B}''/\mathcal{A}''\right] \qquad \omega'' = \left\langle \omega' \lhd \textbf{\textit{res}}\ \beta_{\cdot[\beta'']m(\overline{\beta'})}\right\rangle^{\delta}}{\beta_{\cdot[\beta'']m(\overline{\beta'})} \overset{\Delta}{\hookrightarrow} \beta_{\cdot[\beta'']m(\overline{\beta'})} \lhd \omega''}$$

$$\text{(C-Hapn-Inv-Cir)}\quad \frac{\delta = (\beta, C, m) \\ mtype(C,m) = \forall\mathcal{A}.\,\forall\mathcal{A}'.\,\forall\mathcal{A}''.\,\sigma \qquad \Delta' = \delta :: \Delta \qquad (\mathcal{B}'', \textbf{\textit{true}}) = generate\left(\Delta', \mathcal{A}''\right) \qquad \Omega \vdash [\beta''', \overline{\beta''''}]^{C} \preceq \beta \\ \omega' = \sigma\left[\{\beta''', \overline{\beta''''}\}/\mathcal{A}, \{\beta'', \overline{\beta'}\}/\mathcal{A}', \mathcal{B}''/\mathcal{A}''\right] \qquad \omega'' = \left\langle \omega' \lhd \textbf{\textit{res}}\ \beta_{\cdot[\beta'']m(\overline{\beta'})}\right\rangle^{\delta} \qquad \mathcal{L} = \mathcal{L} \cup (\beta_{\cdot[\beta'']m(\overline{\beta_a})} \lhd \omega'')}{\beta_{\cdot[\beta'']m(\overline{\beta'})} \overset{\Delta}{\hookrightarrow} \beta_{\cdot[\beta'']m(\overline{\beta'})} \lhd \omega''}$$

$$\text{(C-Hapn-Wait)}\quad \frac{\Omega \vdash \beta'' \preceq \beta'''}{\textbf{\textit{wait}}\ \beta''' \overset{\Delta}{\hookrightarrow} \textbf{\textit{res}}\ \beta_{\cdot[\beta'']m(\overline{\beta'})} \lhd \textbf{\textit{wait}}\ \beta'''} \qquad \text{(C-Hapn-Redc)}\quad \frac{\omega' \overset{\Delta}{\hookrightarrow} \omega''}{\omega \lhd \omega' \overset{\Delta}{\hookrightarrow} \omega \lhd \omega''} \qquad \text{(C-Ctxt-Redc)}\quad \frac{\omega \overset{\delta::\Delta}{\hookrightarrow} \omega'}{\langle\omega\rangle^{\delta} \overset{\Delta}{\hookrightarrow} \langle\omega'\rangle^{\delta}}$$

$$\text{(C-Flow-Reflx)}\quad \Omega \vdash \beta \preceq \beta$$

$$\text{(C-Flow-Trans)}\quad \frac{\Omega \vdash \beta \preceq \beta' \qquad \Omega \vdash \beta' \preceq \beta''}{\Omega \vdash \beta \preceq \beta''} \qquad \text{(C-Hapn-Trans)}\quad \frac{\Omega \vdash \omega \lhd \omega' \qquad \Omega \vdash \omega' \lhd \omega''}{\Omega \vdash \omega \lhd \omega''} \qquad \text{(C-Hapn-Comp)}\quad \frac{\Omega \vdash \omega \lhd \omega'}{\Omega \vdash last\,(\omega) \lhd first\,(\omega')}$$

$$\text{(C-Hapn-FIFO)}\quad \frac{\Omega \vdash \beta_{\cdot[\beta_r]m(\overline{\beta_a})} \lhd \beta_{\cdot[\beta_{r'}]m'(\overline{\beta'_a})} \qquad \beta_{\cdot[\beta_r]m(\overline{\beta_a})} \overset{\Delta}{\hookrightarrow} \beta_{\cdot[\beta_r]m(\overline{\beta_a})} \lhd \omega' \qquad \beta_{\cdot[\beta_{r'}]m'(\overline{\beta'_a})} \overset{\Delta}{\hookrightarrow} \beta_{\cdot[\beta_{r'}]m'(\overline{\beta'_a})} \lhd \omega''}{\Omega \vdash \omega' \lhd \omega''}$$

Figure 7. Type closure for order types.

between them. Figure 8 shows the wellformedness of the order graph in the absence of message races.

In (W-Wellformed), an order graph Ω is wellformed if it is free from message races. That is, for any two messages (invocations) $\omega_1 = \beta_{1\cdot[\beta''_1]}m_1(\overline{\beta'_1})$ and $\omega_2 = \beta_{2\cdot[\beta''_2]}m_2(\overline{\beta'_2})$ if the receiver β_1 flows to the receiver β_2 or vice versa, then there is a happens-before relation between ω_1 and ω_2 where ω_1 happens-before ω_2 or vice versa. The function *peel* removes contextual information of a behavior.

Before performing causality analysis, circular backedges are removed from the order graph. The function *uncircle* removes backedges from the order graph.

$$\text{(W-Wellformed)}\quad \frac{\Omega' = uncircle\,(\Omega) \qquad \omega_1, \omega_2 \in \Omega' \\ peel\,(\omega_1) = \beta_{1\cdot[\beta''_1]}m_1(\overline{\beta'_1}) \qquad peel\,(\omega') = \beta_{2\cdot[\beta''_2]}m_2(\overline{\beta'_2}) \\ \Omega' \vdash (\beta_1 \preceq \beta_2) \vee (\beta_2 \preceq \beta_1) \qquad \Omega' \vdash (\omega_1 \lhd \omega_2) \vee (\omega_2 \lhd \omega_1)}{\Omega \models \diamond}$$

Figure 8. Wellformedness for the absence of message races.

3.8 Termination and Soundness

Termination Detection of mutually recursive invocations and prevention of their nontermination could guarantee that the type closure reaches a fixpoint and terminates.

Soundness Progress and preservation arguments [52] could guarantee that static order types are a sound approximation of the runtime communication and flow behaviors of the program and their happens-before relation and therefore can be used for static reasoning about message races.

Formalization of termination and soundness of order types and their proofs are out of the scope of this work and will be provided as part of the future work.

4 Related Work

Channel races in session programming Previous work [21, 22] proposes session types to allow for static reasoning about communication safety including channel races (channel linearity) in session programming. A session is a conversation unit with designated channels to structure the communication among its participants. A session type is a global type that prescribes a communication protocol via specification of sends and receives on session channels and their orders. The projection of a session type into its participants produces local types for participants to abide by. A global session type can also be inferred from its local participant types by leveraging sessions they use to communicate [37]. Session programming constructs and session types are added to several programming languages and calculi including Java [24] Erlang [36], Moose [14] and standard actors [13, 39]. However, session types only work for session programming and are not directly applicable to asynchronous

message passing without sessions. Also, session types are global and prescriptive types whereas order types are local and descriptive.

$$
\begin{array}{lcl}
\mathit{labels}\,(x.{}_{[\alpha]}m(\overline{x})) & = & \{\alpha\} \\
\mathit{labels}\,(\textbf{\textit{new}}_{[\overline{\alpha}]}C(\overline{x})) & = & \overline{\alpha} \\
\mathit{labels}\,(\textbf{\textit{future}}_{[\alpha]}(x)) & = & \alpha \\
\mathit{labels}\,(\textbf{\textit{wait}}\ x) & = & \emptyset \\
\mathit{labels}\,(e';e'') & = & \mathit{labels}\,(e') \cup \mathit{labels}\,(e'') \\
\mathit{labels}\,(x = e) & = & \mathit{labels}\,(e) \\
\mathit{labels}\,(x) & = & \emptyset \\
\mathit{labels}\,(\textbf{\textit{null}}) & = & \emptyset
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{states}\,(C) & = & \forall \alpha, \overline{\alpha'}.\ \Gamma \\
 & & \textbf{\textit{if}}\ \Gamma = \textbf{\textit{self}} : \alpha,\ \overline{x : \alpha'}, \\
 & & \alpha = \mathit{tvar}\,(C\,), \overline{\alpha'} = \overline{\mathit{tvar}\,(T)} \\
 & & \textbf{\textit{activity}}\ C\ \{\overline{T\ x}\ \overline{meth}\} = AT\,(C) \\
\mathit{tvar}\,(C) & = & \alpha \\
 & & \textbf{\textit{if}}\ \mathit{fresh}\,(\alpha) \\
\mathit{tvar}\,(\textbf{\textit{Fut}}(T)) & = & \alpha(\ \mathit{tvar}\,(T)\ ) \\
 & & \textbf{\textit{if}}\ \mathit{fresh}\,(\alpha)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{generate}\,(\Delta, \mathcal{A}) & = & \mathit{gen}\,(\mathit{collapse}\,(\Delta), \mathcal{A}) \\
\mathit{gen}\,(\Delta, \lfloor, \mathcal{A}) & = & (\mathcal{B}, \lfloor) \\
 & & \textbf{\textit{if}}\ |\mathcal{A}| = |\mathcal{B}|,\ \mathit{fresh}\,(\mathcal{B}) \\
\mathit{collapse}\,(\epsilon) & = & (\epsilon, \textbf{\textit{false}}) \\
\mathit{collapse}\,(\delta :: \Delta) & = & (\delta :: \Delta'' :: \delta :: \Delta''', \textbf{\textit{true}}) \\
 & & \textbf{\textit{if}}\ \mathit{collapse}\,(\Delta) = (\Delta' :: \delta :: \Delta'' :: \delta :: \Delta''', \textbf{\textit{false}}) \\
\mathit{collapse}\,(\delta :: \Delta) & = & (\delta :: \Delta' :: \delta :: \Delta'', \textbf{\textit{false}}) \\
 & & \textbf{\textit{if}}\ \mathit{collapse}\,(\Delta) = (\Delta' :: \delta :: \Delta'', \textbf{\textit{false}}) \\
\mathit{collapse}\,(\delta :: \Delta) & = & (\delta :: \Delta', \textbf{\textit{false}}) \\
 & & \textbf{\textit{if}}\ \delta \notin \Delta' \\
 & & \Delta' = \mathit{collapse}\,(\Delta)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{last}\,(\bullet) & = & \emptyset \\
\mathit{last}\,(\kappa) & = & \{\kappa\} \\
\mathit{last}\,(\phi) & = & \{\phi\} \\
\mathit{last}\,(\sigma \lhd \sigma') & = & \mathit{last}\,(\sigma) \cup \mathit{last}\,(\sigma')
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{first}\,(\bullet) & = & \emptyset \\
\mathit{first}\,(\kappa) & = & \{\kappa\} \\
\mathit{first}\,(\phi) & = & \{\phi\} \\
\mathit{first}\,(\sigma \lhd \sigma') & = & \mathit{first}\,(\sigma) \cup \mathit{first}\,(\sigma')
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{peel}\,(\omega) & = & \omega' \\
 & & \textbf{\textit{if}}\ \omega = \langle \omega' \rangle^{\delta} \\
\mathit{peel}\,(\omega) & = & \omega \\
 & & \textbf{\textit{if}}\ \text{otherwise}
\end{array}
$$

Figure 9. Auxiliary functions.

Channel races in channel programming Prior work [25] proposes a generic type systems for static reasoning about a range of properties including channel races (linearity) in channel programming of π calculus. Types encode sends and receives on channels and their orders and are expressed as processes in the calculus of communicating systems (CCS). Another previous work [12] extends this work's process types with channel creation and hiding operators and use hiding to support assume-guarantee reasoning via model checking. Previous work [29] also proposes a substructural linear type system to guarantee the absence of channel races in π calculus. In standard π calculus and CCS, a program is a set of processes that run concurrently, communicate through channels and synchronize using synchronous handshakes where channels are first-class values in π calculus. However, these type systems are designed for functional π calculi with channel programming and are not directly applicable to an imperative message passing calculus with activities, asynchronous method invocations and future synchronizations.

Other previous work [18, 25] proposes translations between functional channel programming and actor [3, 4] and active object [11] programming. However, the translation requires all actors to be of the same type and does not translate between calculi with imperative constructs. Activities share similarities with actors particularly actors in languages like ActorFoundry [6] in which a message is sent via a method invocation.

Data races in shared memory concurrency Previous work [1, 2, 5, 9, 15–17, 20, 38, 49] allows for static reasoning about low level data races in shared memory concurrency with low level concurrency constructs including threads, locks and barriers. Two memory access are racy if one of them is a write and there is no order among them. These work track locksets that protect shared memory locations, happens-before relations between accesses to these locations and their ownership. However, previous work on reasoning about data races in shared memory concurrency with low level concurrency constructs including locks and threads is not directly applicable to reasoning about message races in asynchronous message passing concurrency with higher level concurrency constructs including activities, asynchronous message invocations and futures.

5 Conclusion and Future Work

This work presented order types, a novel local, causal and polymorphic type system for static reasoning about message races in an imperative calculus with concurrent activities, asynchronous method invocations, future return values, wait-by-necessity synchronizations, first-class activities and futures, recursion and dynamic creation of activities.

First avenue of future work is to mechanize formalization of order types in a proof assistant like Coq after providing formalizations and proofs of termination and soundness. Second is to investigate the causality relations, including causal precedence [44], that generalize and weaken the happens-before relation. Third avenue is to generalize order types to allow for a range of message processing and delivery strategies, in addition to first-in-first-out (FIFO).

Acknowledgements

Bagherzadeh was supported in part by Oakland University. Rajan was supported in part by the NSF grant CCF-08-46059.

References

[1] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. 2006. Types for Safe Locking: Static Race Detection for Java. *TOPLAS '06* 28, 2 (2006).

[2] Rahul Agarwal and Scott D. Stoller. 2004. Type Inference for Parameterized Race-Free Java. In *VMCAI '04*.

[3] Gul Agha. 1986. *Actors: a model of concurrent computation in distributed systems*. MIT Press.

[4] Gul Agha and Carl Hewitt. 1985. Concurrent Programming Using Actors: Exploiting large-Scale Parallelism. In *FSTTCS '85*.

[5] Alexander Aiken and David Gay. 1998. Barrier Inference. In *POPL '98*.

[6] M. Astley. 1999. The actor foundry: A Java-based actor programming environment. University of Illinois at Urbana-Champaign. (1999).

[7] Mehdi Bagherzadeh. 2016. *Toward a Concurrent Programming Model with Modular Reasoning*. Ph.D. Dissertation. Iowa State University.

[8] Mehdi Bagherzadeh and Hridesh Rajan. 2015. Panini: A Concurrent Programming Model for Solving Pervasive and Oblivious Interference. In *MODULARITY 2015*.

[9] Chandrasekhar Boyapati and Martin Rinard. 2001. A Parameterized Type System for Race-free Java Programs. In *OOPSLA '01*.

[10] Denis Caromel. 1993. Toward a Method of Object-oriented Concurrent Programming. *CACM '93* 36, 9 (1993).

[11] Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. 2009. Asynchronous Sequential Processes. *Inf. Comput. '09* 207, 4 (2009).

[12] Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. 2002. Types As Models: Model Checking Message-passing Programs. In *POPL '02*.

[13] Silvia Crafa. 2012. Behavioural Types for Actor Systems. *CoRR* abs/1206.1687 (2012).

[14] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. 2006. Session Types for Object-oriented Languages. In *ECOOP'06*.

[15] Matthew B. Dwyer and Lori A. Clarke. 1994. Data Flow Analysis for Verifying Properties of Concurrent Programs. *SIGSOFT Softw. Eng. Notes '94* 19, 5 (1994).

[16] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP '03*.

[17] Cormac Flanagan and Stephen N. Freund. 2000. Type-based Race Detection for Java. In *PLDI '00*.

[18] Simon Fowler, Sam Lindley, and Philip Wadler. 2017. Mixing Metaphors: Actors as Channels and Channels as Actors. In *ECOOP '17*.

[19] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. 2016. A framework for deadlock detection in core ABS. *SoSym '16* 15, 4 (2016).

[20] Dan Grossman. 2003. Type-safe Multithreading in Cyclone. In *TLDI '03*.

[21] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP '98*.

[22] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *POPL '08*.

[23] Christopher R. Houck and Gul Agha. 1992. HAL: A High-Level Actor Language and Its Distributed Implementation. In *ICPP '92*.

[24] Raymond Hu, Nobuko Yoshida, and Kohei Honda. 2008. Session-Based Distributed Programming in Java. In *ECOOP '08*.

[25] Atsushi Igarashi and Naoki Kobayashi. 2001. A Generic Type System for the Pi-calculus. In *POPL '01*.

[26] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2011. ABS: A Core Language for Abstract Behavioral Specification. In *FMCO '10*.

[27] WooYoung Kim. 1997. *ThAL: An Actor System for Efficient and Scalable Concurrent Computing*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[28] Naoki Kobayashi and Cosimo Laneve. 2017. Deadlock Analysis of Unbounded Process Networks. *Inf. Comput. '17* 252, C (2017).

[29] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1999. Linearity and the Pi-calculus. *TOPLAS '99* 21, 5 (1999).

[30] Aditya Kulkarni, Yu David Liu, and Scott F. Smith. 2010. Task Types for Pervasive Atomicity. In *OOPSLA '10*.

[31] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM '78* 21, 7 (1978).

[32] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. 2009. A Framework for State-Space Exploration of Java-Based Actor Programs. In *ASE '09*.

[33] Yuheng Long, Mehdi Bagherzadeh, Eric Lin, Ganesha Upadhyaya, and Hridesh Rajan. 2016. On Ordering Problems in Message Passing Software. In *MODULARITY 2016*.

[34] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *TSE '05* 14, 1 (2005).

[35] Carroll Morgan. 1988. Procedures, parameters, and abstraction: Separate concerns. *SCP '88* 11, 1 (1988).

[36] Dimitris Mostrous and Vasco T. Vasconcelos. 2011. Session Typing for a Featherweight Erlang. In *COORDINATION'11*.

[37] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. 2009. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *ESOP '09*.

[38] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *PLDI '06*.

[39] Rumyana Neykova and Nobuko Yoshida. 2014. Multiparty Session Actors. In *COORDINATION '14*.

[40] Flemming Nielson and Hanne Riis Nielson. 1999. Type and Effect Systems. In *Correct System Design: Recent Insights and Advances '99*.

[41] Hridesh Rajan. 2015. Capsule-oriented Programming. In *ICSE '15*.

[42] Hridesh Rajan, Steven M. Kautz, Eric Lin, Sean L. Mooney, Yuheng Long, and Ganesha Upadhyaya. 2014. *Capsule-oriented Programming in the Panini Language*. Technical Report 14-08. Iowa State University.

[43] Jan Schäfer and Arnd Poetzsch-Heffter. 2010. JCoBox: Generalizing Active Objects to Concurrent Components. In *ECOOP '10*.

[44] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *POPL '12*.

[45] Herb Sutter and James Larus. 2005. Software and the Concurrency Revolution. *Queue '05* 3, 7 (2005).

[46] Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. 2012. TransDPOR: A Novel Dynamic Partial-order Reduction Technique for Testing Actor Programs. In *FMOODS/FORTE '12*.

[47] S. Tasharofi, M. Pradel, Y. Lin, and R. Johnson. 2013. Bita: Coverage-guided, automatic testing of actor programs. In *ASE '13*.

[48] C. Tomlinson, W. Kim, M. Scheevel, V. Singh, B. Will, and G. Agha. 1988. Rosette: An Object-oriented Concurrent Systems Architecture. In *OOPSLA/ECOOP '88*.

[49] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: Static Race Detection on Millions of Lines of Code. In *ESEC-FSE '07*.

[50] Tiejun Wang and Scott F. Smith. 2001. Precise Constraint-Based Type Inference for Java. In *ECOOP '01*.

[51] Adam Welc, Suresh Jagannathan, and Antony Hosking. 2005. Safe Futures for Java. In *OOPSLA '05*.

[52] A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput. '94* 115, 1 (1994).

[53] Akinori Yonezawa (Ed.). 1990. *ABCL: An Object-oriented Concurrent System*. MIT Press, Cambridge, MA, USA.