Objects as Session-Typed Processes

Stephanie Balzer and Frank Pfenning
Computer Science Department
Carnegie Mellon University

Abstract

A key idea in object-oriented programming is that objects encapsulate state and interact with each other by message exchange. This perspective suggests a model of computation that is inherently concurrent (to facilitate simultaneous message exchange) and that accounts for the effect of message exchange on an object's state (to express valid sequences of state transitions). In this paper we show that such a model of computation arises naturally from session-based communication. We introduce an object-oriented programming language that has processes as its *only* objects and employs linear session types to express the protocols of message exchange and to reason about concurrency and state. Based on various examples we show that our language supports the typical patterns of object-oriented programming (e.g., encapsulation, dynamic dispatch, and subtyping) while guaranteeing session fidelity in a concurrent setting. In addition, we show that our language facilitates new forms of expression (e.g., type-directed reuse, internal choice), which are not available in current object-oriented languages. We have implemented our language in a prototype compiler.

Categories and Subject Descriptors D.1 [*Programming Techniques*]: Concurrent Programming—Parallel programming; D.1 [*Programming Techniques*]: Object-oriented Programming

Keywords object, session types, linear types, process, protocol

1. Introduction

Since its inception in the 1960s [19, 20] and 1970s [30], object-oriented programming has become a ubiquitous programming model. A multitude of object-oriented languages have emerged since, each with their own characteristics. In an extensive survey of the object-oriented literature, Armstrong [8] identifies 39 concepts generally associated with object-oriented programming, such as object, encapsulation, inheritance, message passing, information hiding, dynamic dispatch, reuse, modularization, etc., out of which she distills the "quarks" of object-orientation, which are: object and class, encapsulation and abstraction, method and message passing, dynamic dispatch and inheritance.

These findings are consistent with the concepts supported by the protagonists of object-oriented languages: Simula [19, 20] introduced the notions of an object and a class and substantiated the idea to encapsulate the operations and the data on which they operate in an object. Smalltalk [30] put forward the message-passing aspect of object-orientation by viewing computation as message exchange, whereby messages are dispatched dynamically according to the actual receiver object. The Actor model implemented this idea in a distributed and concurrent context where message-passing is the sole means of exchange between actors [6, 34].

Viewing *computation as the exchange of messages between stateful objects* seems to be a key idea in object-oriented programming. Alan Kay phrased this view in a publicly available email exchange on the meaning of object-oriented programming as follows: "*OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things. It can be done in Smalltalk and in LISP. There are possibly other systems in which this is possible, but I'm not aware of them.*" ([42]). This view suggests computation to be inherently *concurrent*, as an object may simultaneously exchange messages with several objects, and the expression of the *protocols* that govern message exchange.

Concurrency support in mainstream object-oriented languages is the subject of an active area of research. Mainstream object-oriented languages typically support concurrency by purely operational synchronization idioms (e.g., synchronized statements/methods), but leave it to the programmer to ensure thread-safety of a class and absence of data races. To alleviate the burden put on programmers, type system extensions have been suggested that, for example, use ownership types [18] to guarantee absence of data races and deadlocks [13] or the notion of an interval to make the or-

5th International Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!) 2015, October 26, 2015, Pittsburgh, PA, USA.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

dering between threads explicit for the prevention of data races [44]. Of particular concern are also high-level data races, whose prevention is possible by means of a static analysis [59]. Moreover, a wide range of tools have been developed for static [49] and dynamic [25] data race detection.

Considerable effort has been devoted to protocol expression in object-oriented languages. Work on typestate [11, 22] allows programmers to annotate methods with the receiver object's typestate in pre- and post-conditions. To check adherence to the protocols expressed in this way, static program analysis techniques are employed [12, 22] that use some means to control aliasing, such as fractional permissions [14]. Another line of research adopts session types [36, 37] for protocol expression in a concurrent context and incorporates session types into existing object-oriented languages [23, 39, 40] based on a linear treatment of channels.

In this paper, we take a different approach to supporting concurrency and expressing protocols. Rather than extending existing object-oriented languages with concepts necessary to address the challenges, we derive a new model of object-oriented programming that internalizes the seminal idea of object-orientation that computation is concurrent message exchange between stateful objects. Our model takes linear session types as its foundation. This choice is motivated by the observation that object-oriented programming arises naturally from session-typed communication and facilitates program reasoning, because linear session types guarantee session fidelity and freedom of data races and deadlocks.

We present our concurrent object-oriented language CLOO (Concurrent Linear Object-Orientation). In CLOO, *processes* are the *only objects*, and objects interact with each other by sending messages along channels. Message exchange is *asynchronous*, and an object is identified with the *channel* along which it exchanges messages with its clients. Objects (and their channels) are typed with *session types*, which define the protocol of message exchange. Protocol compliance is enforced by CLOO's type system, relying on a *linear* treatment of channels.

An important concern in the development of CLOO was to facilitate program reasoning while maintaining a genuine object-oriented programming style. Based on various programming examples, we show that CLOO supports the typical object-oriented concepts, such as encapsulation, dynamic dispatch, and subtyping. In addition, we show that CLOO enables new forms of program expression, such as *type-directed code reuse* and *internal choice*, concepts not existing in current object-oriented languages. Whilst type-directed reuse supports program extensibility in a similar way as inheritance, it does not suffer from the modularity issues caused by the latter, because type-directed reuse respects encapsulation. We have implemented a prototype compiler for CLOO, which supports most of the features presented in this paper.

Session Type	Behavior
$!\tau;\sigma$	Value output: send value v of type τ, continue as process of type σ.
$?\tau;\sigma$	Value input: receive value v of type τ, continue as process of type σ.
$!\sigma_1;\sigma_2$	Channel output: send channel of type σ_1, continue as process of type σ_2.
$?\sigma_1;\sigma_2$	Channel input: receive channel of type σ_1, continue as process of type σ_2.
$!\{l_1:\sigma_1,\ldots,l_n:\sigma_n\}$	Internal choice: send label l_i, continue as process of type σ_i.
$?\{l_1:\sigma_1,\ldots,l_n:\sigma_n\}$	External choice: receive label l_i, continue as process of type σ_i.
ϵ	Terminate process.
$\mu t.\,\sigma$	Recursive session.

Table 1. Linear session types.

Structure: Section 2 provides a short introduction to linear session types. Section 3 introduces CLOO and the basic correspondence between session-typed communication and object-oriented programming. Section 4 discusses the properties of session fidelity, data race freedom, and deadlock freedom. Section 5 elaborates on the new forms of expression available in CLOO. Section 6 provides a discussion of encapsulation and aliasing and an outlook on future work. Section 7 summarizes related work, and Section 8 concludes the paper.

2. Background

Session types [36, 37] prescribe the interaction behavior of processes that communicate along channels, connecting an offering process with its client process. A session type thus governs the *protocol* of message exchange. Table 1 provides an overview of the kinds of protocols that can be expressed using session types. For example, the session type $!\mathbf{int};?\mathbf{int};\epsilon$ requires sending an integer, then receiving an integer, and then terminating the session. Besides integers and other basic data values, processes can also send channels along channels (as in the π-calculus [46]) and can offer external and internal choices. An external choice $?\{l_1 : \sigma_1,\ldots,l_n : \sigma_n\}$ means to receive one of the labels l_i and then behave as prescribed by σ_i. Conversely, an internal choice $!\{l_1 : \sigma_1,\ldots,l_n : \sigma_n\}$ will send one of the labels l_i and continue as σ_i. Session types can be recursive, allowing for protocol repetition.

To guarantee that the interaction between a client process and an offering process indeed follows the protocol defined by the session type, aliasing of channels must be controlled. We treat channels *linearly* [29], making channels owned by their client process, with ownership transfer being possible by passing a channel to another process. For example, the typing rule for a channel output

$$\Sigma;\Delta,d:\sigma_1 \setminus \Delta \vdash \mathsf{send}(c,d) :: (c:!\sigma_1;\sigma_2) \setminus (c:\sigma_2)$$

indicates that the channel d of type σ_1 will no longer be available to the offering process once it has been sent along the channel c. The exact reading of the above typing rule is irrelevant for this paper, we refer the interested reader to the footnote[1]. Linearity requires a process to "consume" all the owned channels before terminating, either by passing them on to another process or by waiting for the offering process to terminate. The pattern of channel use enforced by linearity is thus colloquially referred to as an *exactly once* usage pattern.

3. Object-Oriented Programming as Session-Based Communication

In this section, we introduce our concurrent object-oriented language CLOO (Concurrent Linear Object-Orientation) and show, based on examples, that object-oriented programming arises naturally from linear session-based communication. The correspondence between the concepts of our language and the "quarks" of object-orientation are summarized in Table 2, which we detail in the following subsections. Section 4 focuses on CLOO's support for protocol expression and concurrency.

We have implemented a prototype compiler for CLOO, which supports most of the features presented in this paper. Our implementation builds on our compiler for the language C0 [10], a subset of C augmented with contracts. As such CLOO maintains a C-style syntax, a decision made out of convenience, but irrelevant to the concepts presented in this paper.

3.1 Objects as Processes

A CLOO program consist of a number of concurrently executing *processes* that communicate by exchanging *messages* with each other along *channels*. CLOO allows programmers to explicitly specify the *protocol* of message exchange in terms of a *session type*. Adherence to the protocol defined by the session type is enforced by the linear type system.

We use the example of a read-once counter to illustrate these concepts. A counter process can respond to the messages inc and val. If a counter process receives the message inc, it increments its current value and proceeds recursively. If a counter process receives the message val, it sends its current value and terminates. In terms of the notation of Section 2, this protocol can be expressed by the following session type:

$$ctr = ?\{inc : ctr, val : !\mathbf{int}; \epsilon\}$$

Figure 1 shows the corresponding session type declaration in CLOO. The struct-like notation choice $name$ $\{\ldots\}$;

[1] The typing rule uses the judgment $\Sigma; \Delta \setminus \Delta' \vdash s :: (c : \sigma) \setminus (c : \sigma')$ to type a process statement s. The judgment indicates that the process offers a session of type σ along channel c and transitions to a session of type σ' as a result of the interaction. The judgement employs the typing context Σ to type a process' local state and the linear context Δ to type the channels owned by the process. Δ denotes the available channels before the interaction, Δ' the available channels after the interaction.

```
typedef <?choice ctr> ctr; // external choice
choice ctr {
 <ctr>    Inc; // increment value, continue
 <!int; > Val; // send value, terminate
};
```

Figure 1. Session type `ctr` in CLOO.

declares the choice and the type definition denotes the actual communication direction (i.e., input or output). For example, the session type `ctr` is an external choice between `Inc` and `Val`. Upon receipt of the label `Inc`, a process implementing `ctr` increments its value and continues as a `ctr`, awaiting to receive any of the labels `Inc` and `Val`. Upon receipt of the label `Val`, a process implementing `ctr` sends its value and terminates. Unlike the original session type work [36, 37], we use a intuitionistic sequent-calculus-based formulation of session types [15], which discriminates between the offering and use site of a process service, eliminating the need to express a session type's dual. Message exchange in CLOO happens *asynchronously*, allowing processes to proceed in parallel without blocking for acknowledgement of message exchange.

Figure 2 shows a process implementation of the session type `ctr` in CLOO. The implementation consists of the two process declarations `bit` and `eps` that implement a counter in terms of a bit string. The bit string is represented as a sequence of `bit` processes, starting with the least significant bit and ending with the most significant bit, followed by an `eps` process that represents the high end of the bit string. We would spawn a bit string process with `ctr $c = eps()`, which binds the channel along which the newly spawned process offers its service to the channel variable `$c`. To distinguish channels from local variables, channels are preceded by a dollar sign `$`.

Process declarations indicate the type of the session they implement and the name of the channel along which they offer that session. For example, process `bit` offers along the channel `$lower` a session of type `ctr`. Processes can declare arguments, which are either local variables to capture internal state or channels to communicate with other processes. For example, process `bit` declares the local variable `b`, representing the bit's value, and the channel `$higher` along which the bit process communicates with its next more significant bit process.

In CLOO, we distinguish the internal state of a process from its externally observable state. The former amounts to a process' local variables, the latter amounts to a process' protocol state. Whereas local variables are altered by assignment, a process' protocol state is treated linearly and altered by message exchange. As such, local variables amount to the only state of a process that is shared and thus constitute the imperative part of CLOO. To guarantee encapsulation of local variables, they can only be read or written to in the body of the process that declares the variables. An assignment to

OO	Remarks	CLOO
Object	A process has state and identity. Its externally observable state amounts to the process' protocol state, its internal state to its local variables.	**Process**
Encapsulation	A process' local variables can only be read or written to by the declaring process, possibly in response to receipt of a message. A process' protocol state can only be altered by exchanging messages according to the protocol prescribed by the process' session type. Only a process' protocol state is externally observable, as it amounts to the process' session type.	**Encapsulation**
Message	An external choice's labels correspond to a class' methods. In addition, messages can encompass labels of internal choice, basic data values, and channels. Message exchange constitutes the sole means of externally observable computation.	**Label**
Reference	Communication happens always along a channel, a channel thus being the only means to "access" a process. Channels are bidirectional and owned by the client process. Ownership of a channel can be transferred by passing the channel to another process.	**Channel**
Dynamic dispatch	The process that will receive a message is determined by the process at the channel endpoint, not generally known at compile time, which then selects the appropriate branch of a switch statement.	**Switch**
Subtyping	Structural subtyping arises between different forms of external and internal choices. For an external choice, subtyping allows a process accepting more choices to be used wherever a subset of those choices is required. For an internal choice, subtyping allows a process offering less choices to be used wherever a superset of those choices is expected.	**Subtyping**

Table 2. Basic correspondence between object-oriented concepts and CLOO concepts.

```
// current process offering on channel 'lower',
// local variable 'b', uses channel 'higher'
ctr $lower bit(bool b, ctr $higher) {
  loop { // continue until terminated
    switch ($lower) { // wait to receive label
      case Inc:
        if (b == false) {
          b = true;
        } else {
          $higher.Inc; // send carry bit as label 'Inc'
          b = false;
        }
        break;
      case Val:
        $higher.Val;
        int n = recv($higher); // receive value
        send($lower, 2*n+(b?1:0)); // send value
        wait($higher); // wait for 'higher' to terminate
        close($lower); // terminate
}}}

ctr $lower eps() {
  loop {
    switch ($lower) {
      case Inc:
        // create new 'eps' process offering on 'zero'
        ctr $zero = eps();
        // forward requests to newly created 'bit' process
        $lower = bit(true, $zero);
      case Val:
        send($lower, 0);
        close($lower);
}}}
```

Figure 2. CLOO processes implementing session type `ctr` (see Figure 1) as a bit string.

a local variable can be triggered by the receipt of a message, either along the offering channel or any other channels of which the process is a client. For example, process `bit` updates its bit upon receiving the label `Inc` along its offering channel.

A loop statement `loop {...}` repeats the enclosed code block until the process terminates. Loop statements are typically used to implement recursive session types. Alternatively, a recursive session type can be implemented by a recursive process declaration. A `switch ($c) {...}` is used to branch on the cases of an external choice for some channel `$c`. It blocks the process until it receives a label along `$c`. For example, upon receiving the label `Inc`, a non-empty bit string sets its own bit to true, if it is false, or asks its next more significant bit process to increment itself. Each branch must be ended with an explicit terminating statement, such as `break`, `close`, or forwarding, to prevent execution from "falling through". Channel forwarding, such as `$lower = bit(true, $zero)` in the `Inc` branch of process `eps`, consumes the current process and advises the client process to communicate henceforth along the channel to the right of the equal sign.

In addition to process declarations, CLOO supports function declarations (not shown in Figure 2). Function declarations differ from process declarations in that they return a value of a basic type rather than a session type. Therefore, the invocation of a function in the body of a process does not have an effect on the process' externally observable state. Function declarations are helpful to modularize auxiliary code.

The example of a counter illustrates that there is a natural correspondence between processes and objects. Processes fully encapsulate their state because a process' local variables are internal to the process and because a process' protocol state can only be altered by message exchange along the process' channel. Channels thus correspond to references, being the only way to "access" an object. Channels, however, provide stronger guarantees than their object-oriented counterparts because they implement the protocol prescribed by their linear session type (see discussion in Section 4.1). Moreover, channels are bidirectional, which gives rise to the concept of an internal choice, for which there exists no analog in object-oriented languages (see discussion in Section 5.2). The first four entries in Table 2 summarize these correspondences.

3.2 Dynamic Dispatch

To illustrate the fifth correspondence in Table 2, we extend our example with a different implementation of session type `ctr`. Figure 3 shows this alternative implementation. It com-

```
ctr $c nat(int n) {
  loop {
    switch ($c) {
      case Inc: n = n + 1; break;
      case Val: send($c, n); close($c);
}}}
```

Figure 3. CLOO process implementing session type `ctr` (see Figure 1) as an accumulator.

prises the process declaration `nat` and keeps the counter in a local variable n. We would spawn such a process with `ctr $c = nat(0)`.

Given that there can be multiple implementations of the same session type, CLOO naturally supports a form of *dynamic dispatch*, where the actual code to be executed in response to a message is determined by the actual process connected to the channel at run-time. For example, we could have a mixed implementation of a counter where the lower 32 bits are `bit` processes, each holding one bit, while the upper bits are stored as a single number in a `nat` process.

This form of dynamic dispatch is analogous to the one resulting from interfaces and classes in Java-like languages. An interface can be implemented by various classes, and the methods defined by the interface are dispatched according to the dynamic type of the receiver object. Similarly, the messages a client process sends along a channel in CLOO will result in the execution of the code of the process that is bound to the channel at run-time.

Process declarations relate to classes in Java-like languages also from a different point of view. Similarly to a class, they serve as a "template" from which processes are instantiated. In the next section we substantiate the correspondence between processes and session types and classes and interfaces, respectively, even further by showing that processes can implement different session types by subtyping.

3.3 Subtyping

Structural subtyping arises in a linear session-typed language between different forms of external and internal choices, respectively. For an external choice, we can substitute a process that accepts additional choices (which will never be selected); for an internal choice we can substitute a process that offers fewer choices (the others will never be sent) [27]. For example, the session type

```
typedef <?choice ctr_inc2> ctr_inc2;
choice ctr_inc2 {
  <ctr_inc2>      Inc;
  <ctr_inc2>      Inc2;
  <!int; >        Val;
};
```

specifies the protocol of a counter that can be incremented in steps of one or two. Because the session type `ctr_inc2` accepts at least the same choices as the session type `ctr` in Figure 1, it is a subtype of `ctr`. As a result, any process implementing `ctr_inc2` can be substituted wherever a process of type `ctr` is expected.

Our current prototype compiler does not support subtyping at this moment, but we expect its implementation to be analogous to [3]. We have already worked out the algorithmic type checking rules to accommodate subtyping between non-recursive session types. Next, we will extend our subtyping relation to a co-inductive subtyping relation between recursive session types, as described in [27]. Our implementation will then have to enforce that recursive session types are contractive [27], guaranteeing type checking of recursive types to be decidable.

4. Language Properties

In this section, we discuss the safety properties provided by an object-oriented programming model that is based on linear session types.

4.1 Session Fidelity

To make sure that the interaction between processes follows the intended protocol, CLOO's type system must guarantee that the session type of a channel along which a client interacts with a process always coincides with the process' protocol state. For example, once a client has sent the label `Val` along a `ctr` channel, the process bound to the channel can only send back an integer value to the client. Any other exchange would be type-incorrect and violate *session fidelity*. The type checker of our prototype compiler implements corresponding checks to ensure session fidelity, a formal proof of session fidelity is in preparation.

The key to guaranteeing session fidelity is *linearity*. Communication always takes place along a channel, and a channel connects exactly two processes, a client process and an offering process. While a process can use any number of processes as clients, linearity of channels guarantees that a process offers a service to *exactly one* other process at any given point in time. A graph of communicating processes thus amounts to a tree, with *ownership transfer* being possible by passing a channel to another process.

Linearity also facilitates reasoning about program correctness. For example, due to the absence of aliasing, it is guaranteed that the counter implementation shown in Figure 2 truthfully reflects the counter's value. If a `bit` process were not to own its `$higher` process (and transitively all more significant `bit` processes) any assumptions it makes on those processes' state could be compromised by changes done to them through aliases.

The benefits of linear session types for program reasoning are indisputable. Linearity, however, also restricts the number of possible ways to implement a problem, a feature that is common to all approaches (e.g., typestate [11, 22], object-oriented sessions [23, 39, 40]) that use alias control or linearity. In CLOO, programmers can use ownership transfer to pass on a channel to another process. In our experience

```
typedef <?choice queue> queue; // external choice
typedef <!choice queue_elem> queue_elem; // internal choice
choice queue {
  <?int; queue>  Enq; // enqueue received value, continue
  <queue_elem>   Deq; // continue as 'queue_elem'
  <!bool; queue> IsEmpty; // check for emptiness
  < >            Dealloc; // terminate
};

choice queue_elem {
  <queue>       None; // send 'None', continue
  <!int; queue> Some; // send 'Some', send value, continue
};

queue $q elem (int x, queue $r) {
  loop {
    switch ($q) {
      case Enq:
        int y = recv($q);
        $r.Enq; send($r, y);
        break;
      case Deq:
        $q.Some; send($q, x);
        $q = $r; // forward requests along 'q' to 'r'
      case IsEmpty:
        send($q, false);
        break;
      case Dealloc:
        $r.Dealloc; wait($r);
        close($q);
}}}

queue $q empty () {
  loop {
    switch ($q) {
      case Enq:
        int y = recv($q);
        queue $e = empty();
        $q = elem(y, $e);
      case Deq:
        $q.None;
        break;
      case IsEmpty:
        send($q, true);
        break;
      case Dealloc:
        close($q);
}}}
```

Figure 4. Queue with constant-time enqueue and dequeue operations from the perspective of the client.

so far, this restriction has never become an obstacle, but we keep it as an issue to think about as part of future work. We expand on linearity further in Section 6.2.

4.2 Data Race and Deadlock Freedom

Next, we introduce a more advanced example that demonstrates how easy it is to program concurrently in CLOO and reason about the resulting code. Figure 4 shows the session type and process declarations of a queue. The implementation exploits parallelism for constant-time enqueue and dequeue operations from the perspective of the client.

The queue's protocol sends label messages in both directions, from the client process to the offering queue process and vice versa. Each direction is expressed separately, by an external (`queue`) and internal choice (`queue_element`), respectively. A queue starts out as a `queue`, awaiting to receive any of the labels `Enq`, `Deq`, `IsEmpty`, or `Dealloc`. It transitions to a `queue_elem` upon receiving the label `Deq`, in which case it sends the label `None`, if it is empty, or the label `Some` followed by the queue element, otherwise, and then continues as a `queue`.

The session types `queue` and `queue_element` are implemented by the process declarations `elem` and `empty`. A queue is represented by a sequence of `elem` processes, ended by an `empty` process. While dequeue operations remove the element at the head of the queue, enqueue operations append an element to its end. The latter operation exploits parallelism, guaranteeing that enqueue operations are constant-time too, from the perspective of the client. In its `Enq` branch, process `elem` passes the value to be enqueued on to its neighboring `elem` process and continues to respond to messages received from its client, while the value to be enqueued is passed down in this way until it reaches the `empty` process and is inserted.

Linearity of channels and encapsulation of process-local state guarantee that the queue behaves according to its protocol and rule out the possibility of races on channels or process-local state. Linearity is also key to asserting a global progress property akin to deadlock freedom that guarantees that at least one of all the available processes takes a step, unless the computation is finished. Intuitively, the property holds because processes will always be arranged in the form of a (dynamically changing) tree without sharing or cycles. Since each process adheres to its session type, the only possibility of a communication not occurring is if a process repeats an infinite sequence of internal actions. The language thus satisfies a (weak) form of deadlock freedom, akin to progress in functional languages, where a function may never return due to nontermination. In future work we intend to prove deadlock freedom rigorously. We expect the proof to be analogous to the corresponding proof in [17], modulo the treatment of process-local state and subtyping. To guarantee a stronger form of deadlock freedom, we would need to employ the technique of Toninho et al. [56], which imposes a simple syntactic criterion on processes, analogous to a termination condition in functional languages.

5. New Forms of Expression

In this section, we elaborate on the new forms of expression that arise from a linear session-typed approach to object-oriented programming.

5.1 Type-Directed Reuse

In the development of large software systems, the ability to reuse existing code is a convenient property of a programming language. In languages like Java, code reuse can be achieved through inheritance[2]. Despite its benefits, the approach is also criticized as compromising modularity because inheritance can bypass encapsulation, making a sub-

[2] Java-like languages use inheritance to achieve subtyping and code reuse. Conceptually, the two notions are different, why the term subclassing is also used for the latter.

```
ctr_test $c counter_inc2(ctr $d) {
  loop {
    switch ($c) {
      case Inc2:
        $d.Inc;
        $d.Inc;
        break;
      default:
        $c <=> $d; // type-directed delegation to 'd'
    }
  }
}
```

Figure 5. Type-directed code reuse: in all branches but `Inc2`, requests are satisfied by the process bound to `$d`, and thus process `counter_inc2` "inherits" code from the process implementing its supertype `ctr`.

class dependent on its superclass. This breach gives rise to the fragile base class problem [45] and is also a source of concern for object-oriented program verification [24, 43].

To facilitate program reasoning, CLOO only supports techniques for code reuse that respect encapsulation. A powerful such technique is *type-directed delegation*. Type-directed delegation combines message passing with subtyping such that a process of type σ' delegates any messages to a process of a supertype σ that already implements the requested behavior.

Figure 5 illustrates type-directed delegation on the example of process `counter_inc2`, which implements the session type `ctr_inc2` defined in Section 3.3. The type-directed delegation statement `$c <=> $d` in the default case indicates that in all remaining branches requests to `$c` are delegated correspondingly to `$d`. Because the type of `$c` is a subtype of the type of `$d`, both the remaining cases and the exact delegating code are well-defined and can be inferred.

Interestingly, the amount of code to be written by the programmer in Figure 5 roughly corresponds to the one of declaring an appropriate subclass that implements the double increment in a class-based object-oriented implementation. Unlike inheritance, however, type-directed reuse is mediated through the session type of the implementing process and thus encapsulation of that process is respected.

The idea to delegate requests to another process that already implements the behavior can also be employed to achieve code reuse in the absence of a subtyping relationship. In this case, inference of the default cases and corresponding delegation code is no longer possible, and programmers are required to provide this information explicitly. On the other hand, this "undirected" form of delegation still saves the actual code of the behavior to be reused and has the advantage exactly that it does not require the two processes to be related, alleviating a process from anticipating possible future reuse scenarios.

Delegation, type-directed and undirected, differs from channel forwarding (see Section 3.1) as it keeps the current process alive and mainly delegates requests to another process of which the current process is a client. Type-directed delegation bears resemblance to SML functors [33, 47] and corresponds to identity expansion in linear logic that eliminates the use of the identity rule at a compound type (`ctr_inc2`) to uses of the identity rule at smaller types (`ctr`). Type-directed delegation is different from the form of delegation found in Self [58] because it does not pass along the original receiver of a message when delegating the message[3]. Moreover, Self is a dynamically-typed language, whereas type-directed delegation uses typing information to infer the delegating code. Our prototype compiler supports undirected delegation, but does not yet support type-directed delegation. We expect its support to be straightforward.

5.2 Internal Choice

Another construct available in CLOO, but missing in existing object-oriented languages, is *internal choice*. In this section we contrast internal choice with external choice and discuss the different characteristics of the two constructs with regard to program extensibility.

Figure 6 shows an alternative implementation of the bit string implementation of a counter shown in Figure 2. The processes `inc` and `zero` implement the session type `bits` that defines the protocol of a bit string in terms of an internal choice. According to this protocol, a process implementing the session type `bits` can either send the label `Eps`, after which it terminates, or the label `Bit`, after which it sends the bit value and then continues as a `bits` process.

Unlike the external choice-based implementation, this implementation does not represent the bit string in terms of `bit` and `eps` processes, but in terms of `Bit` and `Eps` messages sent to a client. The messages are sent starting with the least significant bit and ending with an `Eps` label. For example, the following code spawns a `zero` process and then a sequence of 6 `inc` processes with the `zero` process at its end. Once the loop terminates, channel `$ctr` denotes the most recently spawned `inc` process, which will send the messages `Bit`, `false`, `Bit`, `true`, `Bit`, `true`, `Eps`, and thus the number 6:

```
bits $ctr = zero();
for (int i = 0; i < 6; i++) {
 $ctr = inc($ctr);
}
```

Whereas processes `zero` and `inc` are the processes producing the stream of bit string messages, process `val` consumes such a stream and converts the bit string received in this way to its corresponding decimal number and sends that number along its offering channel. Process `zero` produces the empty bit string. Process `inc`, on the other hand, uses a process that produces a stream of bit string messages representing the number n to offer a stream of bit string messages representing the number $n + 1$.

[3] We have chosen the term "delegation" for type-directed code reuse to avoid confusion with channel forwarding, which CLOO supports as well.

```
typedef <!choice bits> bits; // internal choice
choice bits {
  <>            Eps; // send 'Eps', terminate
  <!bool; bits> Bit; // send 'Bit', send bit, continue
};

bits $zero zero() {
  $zero.Eps;
  close($zero);
}

bits $succ inc(bits $ctr) {
  loop {
    switch($ctr) {
      case Eps:
        wait($ctr);
        $succ.Bit; send($succ, true);
        $succ = zero();
      case Bit:
        bool b = recv($ctr);
        if (b == false) {
          $succ.Bit; send($succ, true);
          $succ = $ctr;
        } else {
          $succ.Bit; send($succ, false);
        }
        break;
}}}

<!int;> $val val(bits $ctr) {
  int n = 0;
  int p = 1; // = 2^0
  loop {
    switch ($ctr) {
      case Eps:
        wait($ctr);
        send($val, n);
        close($val);
      case Bit:
        bool b = recv($ctr);
        n = n + (b?1:0) * p;
        p = 2 * p;
        break;
}}}
```

Figure 6. Alternative implementation of a bit string (see Figure 2) based on internal choice.

It is instructive to compare the two implementations of a bit string with each other. In a certain sense they are "inverse" to each other. Whereas the external-choice-based implementation represents the bit string in terms of `bit` and `eps` *processes*, the internal-choice-based implementation represents the bit string in terms of `Bit` and `Eps` *messages*. As a result, the labels of a choice in one implementation become processes in the other implementation, and vice versa.

The constructs of external and internal choice lead to different modularizations of a program. In choosing one construct over the other, ease of modular extensibility plays an important factor. If the program to be implemented is stable with regard to the operations that it should support, but requires new variants in the future, external choice is preferable. Conversely, if the program is stable with regard to the variants that it should support, but requires new operations in the future, internal choice is preferable.

For example, adding a new operation, such as one to decrease the value of a bit string, can be done modularly in the internal-choice-based implementation because only a new corresponding process has to be defined, but the session type `bits` can be kept unchanged. On the other hand, adding a new variant, such as one representing a minus bit, amounts to a non-modular change because it requires updating the session type `bits` along with all its clients.

Mainstream object-oriented languages lack a corresponding counterpart for CLOO's internal choice. As a result, those programming languages must encode an internal choice indirectly in terms of an external choice. This encoding is achieved by the Visitor pattern [26], for example, where the variants (data structure) to be extended with new operations (visitor) must cater for future extensions by setting up an accept method.

6. Discussion

In this section, we discuss the role and realization of encapsulation in object-oriented languages, elaborate on linearity, and give an outlook on future work.

6.1 Encapsulation

A key idea of the object-oriented paradigm is to encapsulate the operations and the data on which they operate in an object. Languages like Simula [19, 20] pushed this idea and the Actor model [6, 34] distilled it to its purest form by making message exchange the sole mechanism of computation. In CLOO, we follow this tradition and make sure that a process' externally observable state can only be changed by means of message exchange. Since the protocol of message exchange is specified by a linear session type in CLOO, modular reasoning about the resulting program is possible.

The examples used throughout this paper even demonstrate that the notion of state loses its prominent role it takes in mainstream object-oriented programming languages and becomes a mere implementation detail in CLOO. This treatment makes it possible, for example, to have a mixed implementation of a counter where the lower 32 bits are `bit` processes, each storing one bit, whereas the upper bits are stored as a single number in a `nat` process.

In mainstream object-oriented languages, on the other hand, state is an integral part of a program and special provisions must be taken by the programmer to ensure that it is appropriately encapsulated. In Java, for example, fields are exposed to the entire package by default.

A similar breach of encapsulation can also be caused by inheritance. Inheritance allows a subclass to intercept code inherited from its superclass. This coding pattern is typically found in programs that use open recursion for code reuse. Open recursion combines inheritance with dynamically dispatched self-calls, allowing superclasses to invoke customized code in, possibly yet to be written, subclasses (down-call) and subclasses to intercept inherited code (up-call). Whilst open recursion makes code extensible, it aggra-

vates the fragile base class problem and seriously endangers encapsulation and reasoning [7].

As discussed in Section 5.1, CLOO only supports techniques of code reuse that respect encapsulation. Our experience has shown, however, that the technique of delegation is sufficiently powerful to also accommodate the code reuse scenarios targeted by open recursion. By combining undirected delegation with function invocations a similar reuse effect can be achieved in CLOO. Whereas delegation allows "inheriting" unchanged code, the functions encapsulate the code that would otherwise be executed in response to a self-call. Encapsulating that code in separate function declarations is necessary because linearity prevents a process from sending a message to itself.

6.2 Linearity

A question that naturally comes up is whether linearity is a too limiting restriction to implement real-world programs. Using our prototype compiler, we have already implemented numerous programs (including object-oriented design patterns) without linearity becoming an obstacle. We have made similar observations when experimenting with the functional language SILL [3]. Also, we find it very encouraging that the Rust programming language [2] employs an affine [4] type system, which is used as the basis of a recently developed session type library for Rust [41], which has been put to test to define and statically enforce Servo [5] communication protocols. The notion of borrowing [48] implemented in Rust is analogous to ownership transfer in CLOO, where the channel is sent and thus "lent out" to another process to perform an operation and then sent back once the operation is completed.

Given our experience and the use of linear/affine type systems both in research and practice suggests that linearity is not an actual obstacle, but the question rather becomes how many programs can be elegantly expressed with this restriction in place. In a sense, the situation seems analogous to the developments in pure functional programming languages, where effects were not dismissed a priori, but were accommodated once the right abstractions were found (i.e., monads) to integrate mutable state and I/O without compromising purity.

Shared channels [15, 17, 60], for example, are readily applicable to our work and can be easily integrated into CLOO in terms of the shifting operators defined in [52]. Whilst shared channels result in process replication, they could be combined with traditional locking primitives to share, for example, a database. Achieving this form of sharing operationally seems to be straightforward—more challenging is the question what its logical interpretation might be. We would like to investigate this question as part of future work.

[4] An affine type system only rejects the structural property of contraction, but permits weakening. As a result it allows "dropping" resources, enforcing an *at most once* usage pattern.

6.3 Future Work

As part of future work, we want to complement our current prototype compiler with subtyping and type-directed reuse. We are currently formalizing the CLOO language. This formalization draws from existing work [55] that combines functional and message-passing concurrent computation based on linear logic. After those extensions and formalization, we would like to tackle the introduction of polymorphism in the sense of behavioral polymorphism [16] to facilitate generic data structures and affine types [52] to allow garbage collection of processes.

Existing work on linear session types as a prime notion of computation in functional programming languages is based on the recent discovery that linear logic [29] can be given an operational interpretation as session-typed message-passing concurrent computation [15, 17, 60]. This discovery gives rise to a Curry-Howard correspondence between session-typed processes and the intuitionistic linear sequent calculus, such that session type constructors correspond to linear propositions, processes to proofs, and process interactions to proof reductions. In future work we want to explore this correspondence in our setting. We believe that the distinction between process-local state and externally observable state is beneficial as it separates our language into an imperative sequential and linear concurrent part.

Lastly, we plan to find ways of accommodating full sharing whilst upholding the guarantees of session fidelity and absence of data races and deadlocks. It seems feasible that a combination of existing solutions (see discussion in Section 6.2) can be applied to achieve this goal. A more challenging research goal is to find a solution that sustains the Curry-Howard correspondence established between session-typed processes and the intuitionistic linear sequent calculus.

7. Related Work

In this section, we review related work.

Linear Session Types As discussed in Section 6.3, there exists recent work that exploits the Curry-Howard correspondence [15, 17, 60] discovered between session types and intuitionistic linear logic. Toninho et al. [55] introduce the functional programming language SILL [3] that combines functional and message-passing concurrent computation based on linear logic. Those efforts have given rise to theories of behavioral polymorphism [16] and observational equivalence [51]. Most recently, an elegant integration of synchronous and asynchronous communication has been discovered [52]. Whereas all this research targets a purely functional setting, our work applies linear session types to an imperative, object-oriented setting. However, we expect this existing research to offer guidance in refining our work further because it provides evidence for the robustness of logically justified session types and their application to programming and reasoning.

Session Types for Objects There exists an active line of research that introduces session types to object-oriented programming languages [23] to accommodate safe, concurrent, object-oriented computation, even in a distributed [39] or event-driven [40] setting. Dezani-Ciancaglini et al. [23] introduce the multi-threaded language MOOSE, which extends a Java-like object-oriented language with session types. The authors discuss MOOSE's type system and operational semantics and prove progress and preservation. A prime motivation of the work was to preserve the programming concepts of the host language and only extend it with further functionality: "We wanted MOOSE programming to be as natural as possible to people used to mainstream object-oriented languages." ([23]). As a consequence, sessions are confined to an object's methods. Our work, in contrast, equates processes with objects, which gives processes first-class citizenship and makes session types become the types of objects.

Gay et al. [28] address the limited scope of MOOSE's sessions and allow sessions to extend over several methods. In the distributed, object-oriented programming language the authors describe, a channel can even be stored in an object's field. The main focus of the work, however, is the specification and verification of communication protocols, which is carried out in a sequential setting.

Actors and Message-Passing Concurrency Our work shares with the Actor model [6, 34] the idea that message exchange should be the sole means of externally observable computation. The Actor model has lead to various implementations and has also been integrated into mainstream object-oriented languages [31] in terms of libraries [32, 57]. Actors, however, are traditionally untyped, making it impossible to specify and verify the protocols emerging from message exchange. Actors do not even restrict message arrival order.

Neykova and Yoshida [50] target the expression of the protocols for actor coordination. The authors introduce a Python library that allows the expression of actor protocols based on multiparty session type protocols [38]. Protocols are expressed in the specification language Scribble [4] as annotations of the Python code. Protocol compliance can then be checked at run-time, using the Scribble toolchain.

From its message-passing approach to concurrency CLOO also relates to the Erlang [9] programming language. Erlang is a dynamically-typed functional language that uses asynchronous message passing to achieve concurrency in a possibly distributed setting. CLOO, on the other hand, explores asynchronous message passing in an object-oriented imperative setting and uses linear session types to statically guarantee session fidelity and absence of data races and deadlocks.

Similarly, CLOO is related to occam-pi [1], which combines the concepts of CSP [35] with the π-calculus [46]. Processes in Occam-pi encapsulate their state and communicate via message exchange. Occam-pi further has a "zero-tolerance" of aliasing, which it implements by disallowing (channel) references in the first place. The CLOO type system, on the other hand, uses linearity to control aliasing and also supports session types to define and verify communication protocols.

Typestate for Objects Research on typestate [11, 21, 22, 54] approaches the idea of constraining valid sequences of message exchange between objects from a different angle. By introducing the explicit notion of a typestate and a means to control aliasing (e.g., by fractional permissions [14]), programmers annotate methods with pre- and post-conditions to specify an object's typestate on entry and on return of the method. Adherence to such protocol specifications can then be checked by employing static program analysis techniques [12, 22]. Traditionally, typestate has been used to express protocols in a sequential setting, but extensions to a concurrent setting have been elaborated [53].

8. Conclusions

In this paper, we take a fresh look at object-oriented programming. Starting with the seminal ideas that objects encapsulate the operations along with the data on which they operate, that message exchange is the sole means by which objects alter state, and that objects can simultaneously exchange messages, we derive a new model of concurrent object-oriented programming that internalizes those ideas. We have implemented this model in our prototypical language CLOO (Concurrent Linear Object-Orientation).

Our model of object-oriented computation takes linear session types as its foundation. In the paper we show, based on various programming examples, that object-oriented programming arises naturally from linear session-based communication. According to this correspondence, objects can be viewed as processes, references as channels, and method invocations as label selection of external choices. We show that labels are dynamically dispatched and that subtyping arises between different forms of external and internal choices. Our model also enables new forms of expression such as type-directed reuse and internal choice that are not available in current object-oriented languages.

Thanks to its foundation on linear session types, our language guarantees session fidelity and absence of data races and deadlocks. A formal proof of those guarantees is in progress. An important concern in the development of our language was to accommodate a genuine object-oriented style while at the same time facilitate program reasoning. For this reason our language only supports code reuse techniques that respect encapsulation, a characteristics that does not hold for current mainstream object-oriented languages. In future work, we intend to investigate the logical foundation of our language and evaluate its support in the construction of correct, concurrent programs.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1423168. We would like to thank Henry DeYoung for discussions on type-directed delegation and the anonymous reviewers for their helpful comments.

References

[1] occam-pi: blending the best of CSP and the pi-calculus. University of Kent. `www.cs.kent.ac.uk/projects/ofa/kroc/`.

[2] Rust language. `www.rust-lang.org`.

[3] Sill. `https://github.com/ISANobody/sill`.

[4] Scribble. `http://www.scribble.org`.

[5] Servo project. `https://github.com/servo/servo`.

[6] G. A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press Series in Artificial Intelligence. MIT Press, 1990. ISBN 978-0-262-01092-4.

[7] J. Aldrich and K. Donnelly. Selective open recursion: Modular reasoning about components and inheritance. In *3th International Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, 2004.

[8] D. J. Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128, February 2006.

[9] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG*. Prentice Hall, 2nd edition, 1996.

[10] R. Arnold, F. Pfenning, and R. Simmons. C0: Specification and verification in introductory computer science. `http://c0.typesafety.net`, 2010.

[11] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 301–320. ACM, 2007.

[12] K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *23rd European Conference on Object-Oriented Programming (ECOOP)*, pages 195–219, 2009.

[13] C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *17th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 211–230, New York, NY, USA, 2002. ACM.

[14] J. Boyland. Checking interference with fractional permissions. In *10th International Symposium on Static Analysis (SAS)*, pages 55–72, 2003.

[15] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *21st International Conference on Concurrency Theory (CONCUR)*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010.

[16] L. Caires, J. A. Pérez, F. Pfenning, and B. Toninho. Behavioral polymorphism and parametricity in session-based communication. In *22nd European Symposium on Programming (ESOP)*, volume 7792 of *Lecture Notes in Computer Science*, pages 330–349. Springer, 2013.

[17] L. Caires, F. Pfenning, and B. Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 2013. To appear. Special Issue on Behavioural Types.

[18] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *13th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 48–64. ACM, 1998.

[19] O.-J. Dahl and K. Nygaard. SIMULA - an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[20] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA 67 Common Base Language*. Norwegian Computing Center, 1968.

[21] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 59–69, 2001.

[22] R. DeLine and M. Fähndrich. Typestates for objects. In *18th European Conference on Object-Oriented Programming (ECOOP'04)*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004.

[23] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *20th European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 328–352. Springer, 2006.

[24] S. Drossopoulou, A. Francalanza, P. Müller, and A. Summers. A unified framework for verification techniques for object invariants. In *22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *Lecture Notes in Computer Science*, pages 412–437. Springer, 2008.

[25] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 293–303. ACM, 2008.

[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[27] S. J. Gay and M. Hole. Subtyping for session types in the π-calculus. *Acta Informatica*, 42(2–3):191–225, 2005.

[28] S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-orientedriented programming. In *37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 299–312, 2010.

[29] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50: 1–102, 1987.

[30] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[31] P. Haller. On the integration of the actor model into mainstream technologies: A scala perspective. In *2nd International*

Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!), pages 1–6. ACM, 2012.

[32] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.

[33] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 123–137. ACM, 1994.

[34] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[35] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[36] K. Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory (CONCUR)*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.

[37] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming (ESOP)*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[38] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 273–284. ACM, 2008.

[39] R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in Java. In *22nd European Conference on Object-Oriented Programming (ECOOP)*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer, 2008.

[40] R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, and K. Honda. Type-safe eventful sessions in java. In *24th European Conference on Object-Oriented Programming (ECOOP)*, volume 6183 of *Lecture Notes in Computer Science*, pages 329–353. Springer, 2010.

[41] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen. Session types for rust. In *11th ACM SIGPLAN Workshop on Generic Programming (WGP)*, 2015.

[42] A. Kay. On the meaning of object-oriented programming. `http://www.purl.org/stefan_ram/pub/doc_kay_oop_en`, July 2003. Email exchange.

[43] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.

[44] N. D. Matsakis and T. R. Gross. A time-aware type system for data-race protection and guaranteed initialization. In *25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 634–651. ACM, 2010.

[45] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *12th European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, pages 355–382. Springer, 1998.

[46] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[47] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *Definition of Standard ML (Revised)*. MIT Press, 1997.

[48] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A type system for borrowing permissions. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 557–570. ACM, 2012.

[49] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 308–319. ACM, 2006.

[50] R. Neykova and N. Yoshida. Multiparty session actors. In *16th International Conference on Coordination Models and Languages (COORDINATION)*, volume 8459 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2014.

[51] J. A. Pérez, L. Caires, F. Pfenning, and B. Toninho. Linear logical relations and observational equivalences for session-based concurrency. *Information and Computation*, 239:254–302, 2014.

[52] F. Pfenning and D. Griffith. Polarized substructural session types. In *18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 9034 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2015.

[53] S. Stork, K. Naden, J. Sunshine, M. Mohr, A. Fonseca, P. Marques, and J. Aldrich. Aeminium: A permission-based concurrent-by-default programming language approach. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(1):2:1–2:42, 2014. ISSN 0164-0925.

[54] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering (TSE)*, 12(1):157–171, 1986.

[55] B. Toninho, L. Caires, and F. Pfenning. Higher-order processes, functions, and sessions: a monadic integration. In *22nd European Symposium on Programming (ESOP)*, volume 7792 of *Lecture Notes in Computer Science*, pages 350–369. Springer, 2013.

[56] B. Toninho, L. Caires, and F. Pfenning. Corecursion and non-divergence in session types. In *9th International Symposium on Trustworthy Global Computing (TGC)*, volume 8902 of *Lecture Notes in Computer Science*, pages 159–175. Springer, 2014.

[57] Typesafe. Akka framework. `http://akka.io`.

[58] D. Ungar and R. B. Smith. Self: The power of simplicity. In *2nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'87)*, pages 227–242. ACM, 1987.

[59] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented languages. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, pages 334–345. ACM, 2006.

[60] P. Wadler. Propositions as sessions. In *17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 273–286. ACM, 2012.