Analyzing Permission Transfer Channels for Dynamically Typed Languages

Théo Rogliano, Guillermo Polito, Luc Fabresse, Stéphane Ducasse

Analyzing Permission Transfer Channels for Dynamically Typed Languages

Théo Rogliano
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000
Lille, France
theo.rogliano@inria.fr

Guillermo Polito
Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000
Lille, France
guillermo.polito@univ-lille.fr

Luc Fabresse
IMT Lille Douai, Institut Mines-Télécom, Univ. Lille, Centre for Digital Systems, F-59000
Lille, France
luc.fabresse@imt-lille-douai.fr

Stéphane Ducasse
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000
Lille, France
stephane.ducasse@inria.fr

Abstract

Communicating Sequential Process (CSP) is nowadays a popular concurrency model in which threads/processes communicate by exchanging data through channels. Channels help in orchestrating concurrent processes but do not solve perse *data races*. To prevent data races in the channel model, many programming languages rely on type systems to express ownership and behavioural restrictions such as immutability. However, dynamically-typed languages require run-time mechanisms because of the lack of type information at compile-time.

In this paper, we propose to augment channels with four different permission transfer semantics. We explore two mechanisms to implement such permission transfers at run time: write barriers and partial-read barriers. To validate our approach we implemented a channel framework in Pharo, and we extended it with different permission transfer semantics. We report on performance measurements of both (a) the transfer overhead on a single object and on a graph of objects, and (b) the per-object access overhead incurred by ownership checks. This work stands as a cornerstone of future work on adaptive optimizations for permission transfer channels.

CCS Concepts: • **Software and its engineering → Runtime environments**.

Keywords: Concurrency, Channels, Ownership, Permission Transfer, Dynamic Language.

1 Introduction

Communicating Sequential Process (CSP) is nowadays a popular concurrency model in which threads/processes communicate by exchanging data through channels [14]. Channels help in orchestrating concurrent processes but do not solve per-se *data races* [10]. A *data race* is a non-deterministic access by at least two processes[1] to the same memory location or data and at least one process is modifying the content of this data. Those situations lead to incorrect values being processed. To avoid data races, the data needs to be accessible by a unique process during a write operation (See Section 2).

To prevent data races in the channel model, several programming languages implement an *ownership transfer model* where an object has a unique owner at any point in time. In this model, the owner ensures that operations are synchronised to avoid concurrent accesses and ownership-transfer happens when an object is sent through a channel. Most of the existing channel implementations rely on object copies to express ownership on separated memory inspired by the Go language [16, 23, 26, 29, 32]. Channel implementations on shared memory involve type systems to express ownership and behavioural restrictions such as immutability [20, 27, 28, 30], hence they are not suitable for dynamically-typed languages. Instead, dynamically-typed languages require run-time mechanisms because there is not much information available at compile time.

In this paper, we analyze channel-based permission transfers for dynamically-typed languages. Based on our analysis of existing work, we identify and refine four different permission transfer semantics: copy value, full-permissions transfer, exclusive-write permission transfer and read-only permission transfer. We argue that our classification in only four different semantics captures all mechanisms that we encounter in analyzed languages and related work. We leave outside the scope of this paper how such semantics combine.

We evaluate these semantics by implementing a channel framework in Pharo (See Section 3). We extended this framework with our different permission transfer semantics (See Section 4). Then, we report on our experiments using different mechanisms to implement such permission transfers at

[1] In this paper, we use the term process to designate a concurrent execution being it a full process or a lighter one (a.k.a thread).

run time: object copy, write barriers, and partial-read barriers (See Section 5). Our measurements confirm that making copies are linear-time in the number of bytes copied – Using a partial-read barrier is seven times slower for data transfer and using a write barrier slows down data access up to 6%.

The contributions of this paper are:

- An analysis of existing permission transfer semantics in concurrent scenarios.
- A categorization of permission transfer semantics into four families, eliminating redundant and inconsistent semantics.
- A framework to experiment with permission transfer semantics.
- An evaluation of the implementation of each of the identified families.

2 Problem: Efficient and Correct Object Graph Transfer in Dynamically-Typed Languages

2.1 Context: Pharo's Concurrency Model

The Pharo programming language implements concurrency with so-called *processes*: lightweight green-threads scheduled by the virtual machine. The process scheduler schedules processes given their priority. Processes are cooperative amongst the same priority and preemptive amongst different priorities. That is, a process can *yield* to give priority to another process in the same priority, and a process is suspended as soon as a higher priority process is ready [8]. Process switches happen on a timely basis but only at safe execution points: message sends and back jumps.

2.2 Object Graph Transfer by Example

To introduce the problems of object graph transfer, let's consider the example illustrated in Figure 1. The example presents two processes and many objects shared between them. In this example, one process has a reference to the Alice object, and the other process a reference to the Bob object. Alice has a car, which contains a disc, a key and some gas, and Bob has no reference to it: Bob cannot read, write or send messages to any of these objects.

If at some point during execution Bob needs to use the car we need to send a reference to the car from Alice to Bob, for example, by executing bob car: alice car, leading to the situation in Figure 2. As soon as Bob has a reference to the car, he obtains complete access to it *i.e.,* reading, writing, and sending messages to it and all objects reachable from the car.

Handling how objects are shared in a concurrent environment needs special attention. Such a model, in which object sharing relies on just sharing references, *i.e.,* an unrestricted sharing policy, introduces potential data-races. Indeed, if both Alice and Bob have regular references to the car, both may access and modify the entire object graph at the same time producing conflicting side effects.

Even if we take care of revoking Alice's reference to the car (*e.g.,* nilling it), there is still a possibility of data-races when there are shared objects, as it happens in the example with the key object which is directly referenced by Alice and also reachable by Bob from the car. Likewise, if the key has a reference to the car, the car is still reachable by Alice through the key.

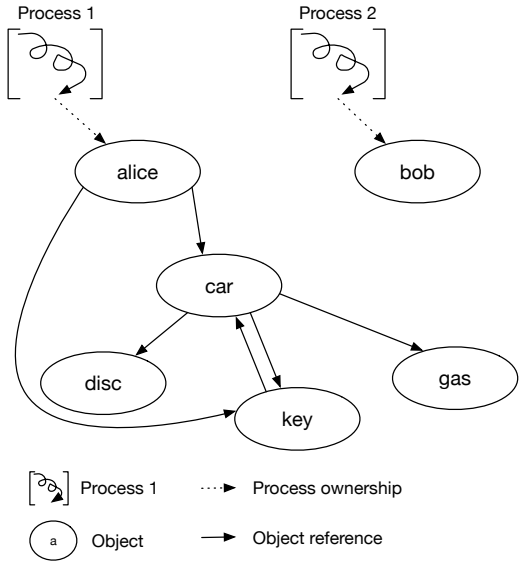

Figure 1. Alice communicates the car object to Bob.

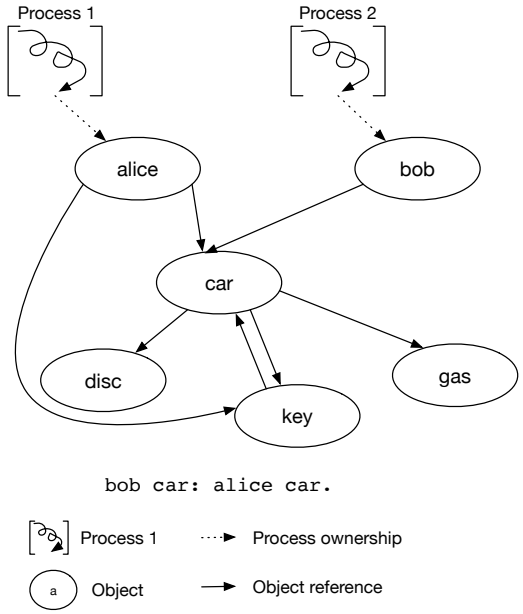

Figure 2. Alice communicates the car object to Bob but Alice also has a reference on the key.

2.3 Challenges of Object Graph Transfer

From the example above, we observe that sharing objects in a concurrent environment presents the following challenges:

Permission Transfer. Unrestricted object reference transfers provide full permissions to the referee on the referred object. To solve this problem we need to control the permissions on shared objects and how those permissions are granted and revoked. As shown in the example above, references give different types of permissions such as read, write, and execution (in the form of message sends). In addition, we need to define a permission model that allows a proper scoping of the sharing.

Object Graph Delimitation. Objects do not exist in isolation but in complex object graphs. When sharing an object, implicit access to its reachable object graph is granted too. We need to control how objects shared between the different graphs behave and how permissions are granted and revoked on an entire object graph. In our example, it would be desirable to grant Bob access permissions to the car without access permissions to the key.

In other words, we need a sharing model preventing shared objects by construction or a model in which we can delimit within an object graph how access is transferred. These models may be left as pure developer responsibility or provide (semi-)automatic ways to do such a delimitation.

Permission Check. Transferring object (and graph) permissions may incur serious performance overheads either when the permissions are transferred or when the objects are accessed. For example, solutions that copy the object graph pay the cost of allocating and copy memory at the moment of the transfer. Solutions using instrumentation to check object access will have an impact on overall performance. An optimal solution will minimize both data transfer overhead and data access overheads.

3 Canal: An Extensible Channel Framework

In this section, we present an overview of our channel framework to experiment with different permission transfer semantics. We decided to use channels as a permission transfer mechanism because they allow a clear delimitation between the sender and the receiver processes while making object sharing explicit. Processes that receive objects from a channel gain some permissions on those objects and processes that sent them may lose some permissions on them. We also describe our per-process ownership model to control write permission on shared objects and thus prevent data races.

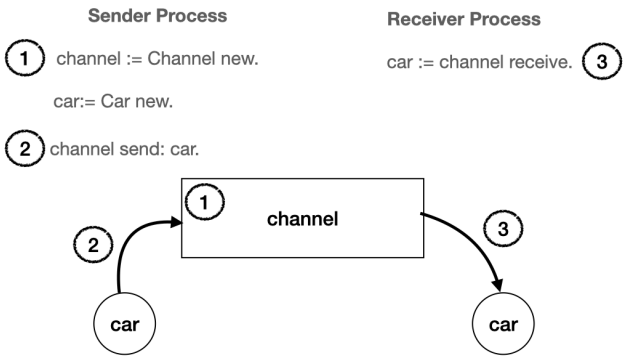

Figure 3. Overview of Object Transfer Through a Channel.

3.1 Extensible Channels and Hooks

Figure 3 depicts the general view of using a channel to transfer an object (the car object on the figure) between two processes. We distinguish two kind of roles a process can take regarding a channel: the sender process and the receiver process. A sender process sends object references into a channel when it does not use this object anymore or wants to share it with other processes. A receiver process acquires references to objects to process them by receiving a reference from a channel.

Channel overview. A channel is a shared data structure between processes that allows one to exchange references. Our channels are first class objects. Channels are unidirectional and can be shared between multiple senders and multiple receivers. Channels are in shared memory, any process having a reference of a channel is able to use it. Channel is the base class of the Channel hierarchy. To guarantee atomicity they are implemented using thread-safe atomic FIFO queues that allow one to transfer any type of objects. The public API is minimalistic with only new, send: and receive: messages. The send: and receive: messages are the ones responsible for the permission transfer and are the hooks to define tailored channel subclasses. The general API is composed of three main messages:

Channel creation. Creating a channel consists only in sending the new message to a specific Channel subclass. It is an extension point for specific initialisations.

Channel send. Sending an object consists in sending the message send: with the desired object as argument. To define channels with specific semantics, the send: message is redefined. The send operation is non-blocking. First specific policies are applied to the object such as revoking write permission then the object is enqueued in the channel.

Channel receive. Receiving an object from a channel consists in sending the receive message to a channel. This message is blocking for the receiver process in the case the channel queue is empty. We chose to make them blocking in this case because when sending a receive message a process expects an object to be returned. Returning nil or an unexpected object only defers possible cause of bugs. When an object is dequeued, its permission is updated according to the receiver process.

3.2 Channel Transfer By Example

The Channel abstract class is the base class of our framework. It implements the exchange of object references using a unique atomic thread-safe queue. Listing 1 shows the Pharo code of the send: and receive methods of this class. Those methods need to be redefined to add the permission transfer.

```
Channel>>send: anObject
  queue nextPut: anObject

Channel>>receive
  | result |
  [ | keepWaiting |
    keepWaiting := false.
    self isClosed
      ifTrue: [ChannelClosedException signal].
    result := queue
      nextIfNone: [ keepWaiting := true ].
    keepWaiting ] whileTrue: [ queue waitForNewItems ].
  ↑ result
```

Listing 1. Definition of send: and receive methods of the Channel base class.

Listing 2 shows a Ping Pong example where two processes exchange a ping and a pong object through two channels. The sender process first creates a new channel (line 1) to send a Ping object and another channel (line 2) to receive a Pong object. A receiver process is created using the fork message (line 6) sent to a block (lexical closure syntactically delimited by square brackets). This receiver process waits until it receives an object from the channel (line 7) and then sends a Pong object into the channel (line 6). The sender process sends a Ping object (line 8) and then waits until it receives Pong object (line 10).

```
1   pingChannel := ExampleChannel new.
2   pongChannel := ExampleChannel new.
3   "receiver process"
4
5   [ objectReceived := pingChannel receive.
6   pongChannel send: Pong new ] fork.
7
8   pingChannel send: Ping new.
9
10  pongChannel receive.
```

Listing 2. Usage Example of a Channel.

This example uses a Channel subclass that does not redefine send: and receive methods but it would be mostly unchanged using more specialized channels. In the following section, we will extend this minimal model to build specific channels by subclassing the Channel class. By carefully choosing specialized channels, the developer prevents data races on the transferred objects.

3.3 Per-Process Ownership Model

To avoid data races, concurrency models typically impose a unique writer process at any time for a single object [10]. Ownership models, using message passing, achieve this by attaching a unique owner to all objects. These models may be too restrictive because they prevent non-owner processes to access an object.

In our framework, each object has a unique owner process stored in its instance variable named owner. An object's owner is the only process that has the write permission on this object. Initially, the process that creates an object is its owner. Changing the ownership of an object only requires assigning another process in its owner instance variable. An attempt to write to an object from a process that doesn't own an object results in an error. Our ownership model allows multiple read-only references on an object while still guaranteeing the uniqueness of the writer. The process scheduler of Pharo ensures that a read operation does not happen during a write operation.

In the following section we will show how our framework models permission transfer at the level of channels.

4 Permission Transfer Channels

In this section, we first report on our identification of four relevant permission transfer semantics. Then, each following subsection describes a Pharo implementation of each of these semantics by extending our Channel framework presented in Section 3.

4.1 Identifying Permission Transfer Semantics

A Canal channel transfers references to objects along with permissions to those objects. We distinguish three kinds of permissions: write, read, and execute (sending a message). As we explain in what follows, not all combinations of permissions are meaningful, hence it is not necessary to implement them. For example, a channel where both the receiver and the sender processes lose all permissions would result in the object being unusable. To constrain the field of what is possible, we followed two rules:

Write Implies Read Rule. Write permissions imply read permissions, read permissions imply execution permissions. The first part of this rule means that to write the fields of an object we require the permission to read the fields of that object. The second part of this rule implies that to read the field of an object, we need

Permissions Transfer	Sender Process (SP)		Receiver Process (RP)
	Pre-send:	**Post**-send:	**Post**-receive
(1) Copy value (from process with ownership)	$A_{w,r}$	$A_{w,r}$	$A'_{w,r}$
(2) Copy value	$A_{-,r}$	$A_{-,r}$	$A'_{w,r}$
(3) Full transfer (from process with ownership)	$A_{w,r}$	$\emptyset$	$A_{w,r}$
(4) Full transfer	$A_{-,r}$	$A_{-,r}$	$\emptyset$
(5) Exclusive Write (from process with ownership)	$A_{w,r}$	$A_{-,r}$	$A_{w,r}$
(6) Exclusive Write	$A_{-,r}$	$A_{-,r}$	$\emptyset$
(7) Read-only (from process with ownership)	$A_{w,r}$	$A_{w,r}$	$A_{-,r}$
(8) Read-only	$A_{-,r}$	$A_{-,r}$	$A_{-,r}$

A=Object, A'=Object A copy, W = write, R = read, $\emptyset$ = no references, - = not permitted.

Table 1. Four permission transfer semantics based on the evolution of the sender and receiver processes' permissions on the transferred object A. The letters W and R represent respectively the write and read permissions of a process on A. Having a '_' instead of a permission means that a process does not have this permission on the object. $\emptyset$ means that a process does not hold a reference on the object because it never had it or lost it. A' is a copy of object A.

to be able to send it a message. This last part arises from the fact that object fields (instance variables) are encapsulated in Pharo and can only be accessed by the object itself.

Conservation of Permissions Rule. The set of permissions owned by the sender before the transfer must be equals to the set of permissions owned together by the sender and the receiver after the transfer. A first corollary of this rule is that a process cannot grant a permission that it did not have beforehand thus permissions cannot be forged on an object. A second corollary of this rule is that overall permissions over an object cannot be lost, preventing strange situations where an object reference exists but cannot be accessed by any other object.

Given these two rules we identified four permission transfer semantics (See Table 1) in languages based on the evolution of permissions of the sender and receiver processes before and after the transfer. Since message sending to an object is never restricted in our semantics, we omit the execution permission from the rest of the paper. Note that writing to an object is sending a message but we do not prevent from sending the message and instead throw an error.

Table 1 reads as follow. A group of two rows represent a permission transfer semantics. The first row of the group represents a permission transfer when the sender process has the ownership of the transferred object. The second row of the group represents a transfer when the sender process does not have ownership of the transferred object. The first column is the name of the semantics. The second column shows the permissions the sender has before sending an object. The third column shows the permissions the sender has after sending an object. The last column shows the permissions the receiver has after receiving an object.

Taking as example the full transfer semantics represented by the third and forth row. The first column confirms that we are looking at the full transfer semantics.

Reading the third row. In the second column, $A_{w,r}$ means that the sender process will send an object A and has write (ownership) and read permissions on this object. In the third column, $\emptyset$ means that the sender process, after sending A, lost all references on A and all permissions on A. In the last column, $A_{w,r}$ means that the receiver process received a reference on object A and have all permissions on A.

Reading the forth row. In the second column, $A_{-,r}$ means that the sender process will send an object A and has only read permission on this object (no ownership). In the third column, $A_{-,r}$ means that the sender process, after sending A, kept a read-only reference on A. In the last column, $\emptyset$ means that the receiver process never received a reference on object A (in this case because we aborted the transfer).

In the following subsections, we present more in details these four permission transfer semantics: copy value, full ownership, exclusive write and read-only.

4.2 Copy Value Graph Transfer (CVGT)

A Copy Value Graph Transfer channel corresponds to the first and second rows of Table 1. When sending an object A through this channel, the sender process keeps a reference to A and sends a copy of object A graph to the receiver called A'. The receiver process has all permissions on A'.

While, in a first thought, it seems to break the conservation of permission rule, it does not. The sender process keeps exactly the same permission over object A and the receiver cannot access A (the original object).

During a transfer, the sender process makes the object A' a copy of A Copying object does not keep invariants such as read-only so the sender process has the unique reference on A' with all permissions. After a transfer, the sender process loses this unique reference to A' hence loses all permissions on it. The receiver process gains all permissions A'. This semantics is also found in the solutions of CSP models [15] and inspired Go channels [1].

One way to achieve this behaviour is by implementing a deep-copy of the object graph reachable from the sent object. To implement this, we redefine the method send: to insert in the channel a deep-copy of the object at send-time.

```
CopyValueGraphTransferChannel>>send: anObject
  | copiedObject |
  copiedObject := anObject deepCopy.
  copiedObject graphOwner: nil.
  super send: copiedObject
```

Listing 3. Redefinition of send: for Copy Value Graph Transfer Channel.

We redefine the method receive to set the receiver as the new owner of each object copies inside the object graph, thanks to the method graphOwner:. Thus, the receiver process gains write permission on all objects of the graph.

```
CopyValueGraphTransferChannel>>receive
| receivedObject |
receivedObject := super receive.
receivedObject beWritableObject.
receivedObject graphOwner: Processor activeProcess.
receivedObject beReadOnlyObject.
↑ receivedObject
```

Listing 4. Redefinition of receive for Copy Value Graph Transfer Channel

A CopyValueGraphTransferChannel guarantees that two reads or two writes cannot happen concurrently on the same object because two separate copies of the graph exist at the same time. Moreover, a deep-copy does not produce shared objects but this channel suffers the duplication problem: we can modify the two copies independently.

4.3 Full Ownership Graph Transfer (FOGT)

A Full Ownership Graph Transfer channel corresponds to the third and forth rows of table 1. The thirs row represents the case when the sender process has ownership of object A. After a transfer, the sender process loses all references on A thus all permissions are represented by ∅. The receiver process gains all permissions.

This behaviour respects the conversation of permissions rule since the sender permissions become the receiver permissions. If the sender process does not own object A as in the forth row then the channel throws an error and the transfer does not happen. The receiver has no references on object A. This behaviour also complies with the conservation of permissions rule since the permission did not change. This semantics is also found in the solutions of Rust channel [22] or Kilim [27].

This behaviour is achieved by revoking recursively all references in the object graph. Listing 5 shows the Pharo code of the redefined send: method for this channel. We implemented this channel using Pharo's atomic object reference swapping (*i.e.,* become: is used in the graphBecome: method). Using pointer-swapping, all original references to the sent object are replaced by references to the argument object. After pointer-swapping, the channel object is the only one that has a reference to the object to send.

```
FullOwnershipObjectTransferChannel>>send: anObject
  | objectToSend |
  "Create placeholer object"
  Processor activeProcess = anObjectOwner
    ifTrue:[ anObject graphOwner: nil ]
    ifFalse: [ self error: 'Cannot full transfer
  an object not owned' ].
  objectToSend := Object new.
  "Swap references"
  anObject graphBecome: objectToSend.
  "At this point, objectToSend has
  the sole reference to the sent object"
  queue nextPut: objectToSend
```

Listing 5. Redefinition of send: for Full Ownership Object Transfer Channel.

Later on, when a process calls receive and consumes the reference from the channel, it will get the unique reference to that object. Moreover, the new owner of the object graph is assigned to the receiver process as shown by the redefinition of the receive method in Listing 6.

```
CopyValueGraphTransferChannel>>receive
| receivedObject |
receivedObject owner ifNil:["gain ownership"
  receivedObject := super receive.
  receivedObject beWritableObject.
  receivedObject graphOwner: Processor activeProcess.
  receivedObject beReadOnlyObject.
]
↑ receivedObject
```

Listing 6. Redefinition of receive for Full Ownership Object Transfer Channel.

4.4 Exclusive Write Object Transfer (EWOT)

An Exclusive Write Object Transfer channel corresponds to the fifth and sixth rows of Table 1. In the sixth row the sender process starts with all permissions on object A. During the transfer, the sender process loses the write permission but keeps at least one reference on object A. The receiver process gains all permissions over object A. The receiver process ends up being the only one with write permissions.

This behaviour complies with the conservation of permission rules because the permissions of the sender before the

transfer are the permissions of the receiver after the transfer. If a process does not possess the write permission on object A as in the sisth row then the channel throws an error and the transfer does not happen. It allows us to comply with the conservation of permissions rule. One way to achieve this behaviour is to instrument all writes to object fields and check if the writing is being done from the owner process. This semantics is also found in Haskell or Clojure channel implementation with persistent data [24].

Our current implementation makes use of pre-existing per-object low-overhead write barriers [2] in Pharo.

Listing 7 shows the code of the send: method for the EWOT Channel. Before adding the transferred object into the channel queue, its owner is reset (set to nil) thus preventing any further write access by the sender.

```
ExclusiveWriteObjectTransferChannel>>send: anObject
  Processor activeProcess = anObject owner
    ifTrue: [ anObject owner: nil.
      queue nextPut: objectToSend ]
    ifFalse: [ self error: 'Trying to send
a not owned object' ]
```

Listing 7. Redefinition of send: for Exclusive Write Object Transfer Channel.

In its receive method (See Listing 8), the channel sets the owner of the object to the current process before returning it.

```
ExclusiveWriteObjectTransferChannel>>receive
  | receivedObject |
  receivedObject := super receive.
  receivedObject beWritableObject.
  receivedObject owner: Processor activeProcess.
  receivedObject beReadOnlyObject.
  ↑ receivedObject
```

Listing 8. Redefinition of receive for Exclusive Object Transfer Channel.

It is important to note that thanks to the Pharo's concurrency model (See Section 2.1), writes are atomic thus a read cannot occur while a process is writing on the shared object such as modifying its owner. Also note, the write permission granting is on a per object basis and not directly the whole object graph. It allows one to manually delimit the granting of the write permission on the object graph.

4.5 Read-only Object Transfer (ROOT)

A Read-Only Object Transfer channel corresponds to the penultimate and ultimate rows of Table 1. In both rows, the sender process is keeping the same permissions it had over object A. The receiver process gains a reference on object A and has only the read permission.

This behaviour complies with the conservation of permissions rules because the sender process does not change and the receiver process has only the read permission.

We implemented it with the same write barrier mechanism used in the exclusive write object transfer (EWOT) except that the object ownership remains unmodified. Since the object's owner does not change the receiver process is only able to read the object. Note that the sender may or may not have write permissions on the object.

In Listing 9, a person object is created and its owner is manually set to nil. This removes the write permission of the sender process on this object. Nevertheless, the sender is still able to send the object through a read-only object transfer channel. In this example, the receiver process does not gain the write permission but only the read permission on the object.

```
channel := ReadOnlyObjectTransferChannel new.
objectToTransfer := Person new.

"Change the ownership"
objectToTransfer owner: nil.
objectToTransfer name: 'Alice'. "Raise an exception"

"receiver process"
[ objectReceived := channel receive.
  objectReceived name: 'Bob'. "Raise an exception"
] fork.

channel send: objectToTransfer.
```

Listing 9. Usage Example of a Read-Only Object Transfer Channel.

4.6 Channel limits: transactionality and inconsistent reads

Table 2 summarizes the different semantics and characteristics of all channel semantics.

The EWOT channel granting the read permission to other processes induces inconsistent read. Back to the car example, let's say bob owns the car. Alice reads the title of the disc and process it. Now, Bob changes the disc and Alice reads the number of track of the disc. Alice will read the number of track of the new disc.

Inconsistent reads also occurs with the CVGT channel. Alice gives the car to Bob expecting that Bob does an action with the car. Since Bob has a copy, Alice cannot see the action effect. A new synchronisation is necessary to avoid inconsistent reads. A callback re-transferring the copied object back, or a merging approach is then necessary for the sender process to access the modified object once the receiver is done.

We believe this issue is proper to transactional systems, and is orthogonal to the permission transfer that channels allow.

Property	CVGT	FOGT	EWOT	ROOT
Transfered permissions	Full	Copy	Exclusive Write	Read-only
Granularity	Graph	Graph	Object	Object
Sender Read	Inconsistent Reads	Revoked	Allowed	Allowed
Sender Write	Inconsistent Writes	Revoked	Revoked	Allowed

Table 2. Summary of the Permission Transfer Channel's Properties.

5 Comparing the different Channels

Concurrency mechanisms target first correctness to avoid inconsistencies and then performance. Most of the time, implementations are a trade-off between correctness and performance [12]. In this section, we compare the performance of our different channels. In our case, solutions using copy or pointer-swapping suffer an overhead during the object transfer via a channel meanwhile solutions based on the write barrier do not. In contrast, solutions based on the write barrier suffer from overhead on object access meanwhile the others do not.

In this section, we report on our results benchmarking different scenarios. The Pharo bench message measures the number of times a message is sent per second. The time taken to send a message is inversely proportional to the result of the bench message. In other words, the higher the result of the bench message is, the faster it is. Each channel of each scenario is benched 100 times. A box summarizes 100 benchmarks on a channel. The first and third quartile form the box, the lowest and maximum value form the whiskers. We run all measurements on the same computer with a 2.4 GHz Intel Core i5 quadcore processor and 16 Gio 2133 MHz LPDDR3 ram with all other applications closed.

5.1 Scenario 1: Single Object Transfer Speed

In this scenario, we measure the cost of transferring only one object with our different channels. To achieve this, we reuse a modified version of our Pong example (See Listing 2) with the different channels. The transferred object has 3 instance variables: its owner process, a name and a potential collection of friends not initialized for this scenario.

```
channel := OwnershipGraphTransferPartialReadBarrier
Channel new.
objectToTransfer := OwnedPerson new name: 'Alice'.
[objectReceived := channel receive.
  channel send: objectReceived
  ] fork.
channel send: biggerObjectToTransfer.
channel receive
```

Listing 10. Code example for benchmarks.

Figure 4 shows that Copy Value Graph Transfer channel (red) is on par with the Exclusive Write Object Transfer channel (green). Copying a small object is almost as fast as sending a reference through a channel. Both are around 10%

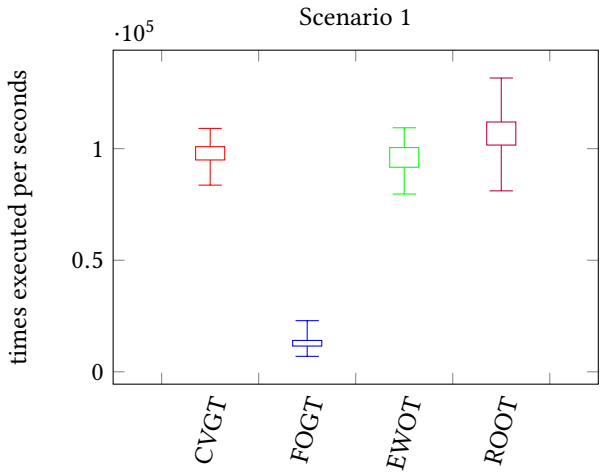

Figure 4. Result of data transfer speed for simple objects. The more times per second it is executed the fastest it is.
CVGT = Copy Value Graph Transfer (red).
FOGT = Full Ownership Graph Transfer (blue).
EWOT = Exclusive Write Object Transfer (green).
ROOT = Read-Only Object Transfer (purple).

slower than the Read-Only Object Transfer channel (purple). The ROOT channel does not transfer ownership so it does not have to update the ownership status and does not need a graph traversal. It explains the better performance of this channel in transfer speed. The Full Ownership Graph Transfer channel is 8 times slower than the other ones. Pointer-swapping is slower than a field update for ownership transfer and also slower than copying small objects. The conclusion is that except for the Full Ownership Graph Transfer channel implementation using pointer-swapping, they are all in the same order.

5.2 Scenario 2: Object Graph Transfer Speed

In this scenario, we measure the cost of transferring an object but for different size of object graphs. The code is similar to scenario 5.1 but the transferred object now references a list of friends in its variable objectToTransfer. In this scenario, we use three different sizes for the friends list. The EWOT channel and ROOT channel operate the transfer at an object granularity and not on a graph granularity. To be fair in the

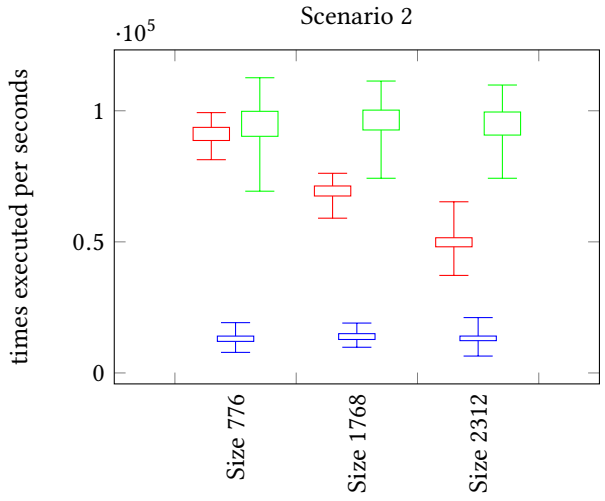

Figure 5. Data transfer speed result for object graphs of different sizes. It is on number of times executed per seconds, the higher the value is, the fastest the implmentation is.
CVGT = Copy Value Graph Transfer (red).
FOGT = Full Ownership Graph Transfer (blue).
EWOT = Exclusive Write Object Transfer (green).

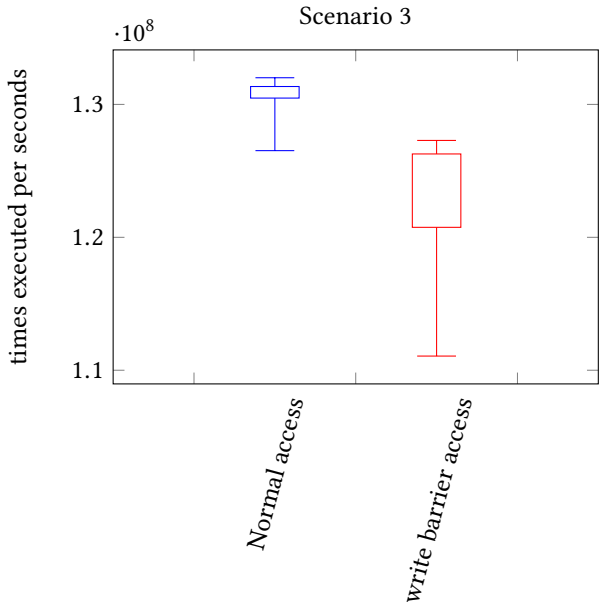

Figure 6. In blue, the non-instrumented accesses. In red, the accesses instrumented with the write barrier

comparison, we adapted the EWOT channel to a graph granularity *i.e.,* we recursively apply a write barrier to all objects in the transferred graph. We omitted the ROOT channel in this comparison because transferring an object reference already gives access to the graph. There is no modification on this channel, hence the result is the one from the previous scenario 5.1.

Figure 5 shows that the Copy Value Graph Transfer channel linear-time in the size of the object graph. It is already 2 times slower by adding two friends in the object graph compared to no friends. It becomes slower than the Full Ownership Graph Transfer after adding five friends. The Full Ownership Object Graph Transfer based on the become: message and the adapted Exclusive Write Transfer channel does not vary much for this sample. Since they all perform the same graph traversal, we conclude that the creation of new objects is expensive. For the FOGT channel, it is quite surprising to repeat a seemingly costly operation and to not degrade performance. An alternative that we did not explore, is pointer-swapping all the objects in the object graph at once. Indeed, the become operation is based on a primitive that performs the pointer-swapping from elements inside an array to elements from another array. In the become's case, both arrays contain one element each, the two objects to swap. The alternative solution is then to collect all the object graphs inside an array and to swap with an array of filler objects by using directly the primitive. An inspection of the implementation of this primitive is necessary to potentially understand our result.

5.3 Scenario 3: Single Object Access Speed

In this scenario we measure the cost of accessing an object with the write barrier compared to accessing without it. Channels using copy or become are not penalized on data access, thus measuring accessing without the barrier is equivalent.

Figure 6 shows that accessing an object field with the write barrier is in average 6% slower than a regular access. For the transfer of an object graph of size 1768, it needs 7000 accesses to the object to have a bigger overhead than doing a copy of the object. For an object graph of size 2312, it requires more than 18000 accesses to the object to have a bigger overhead than doing a copy of the object. In conclusion, channels based on copy lose performance on data transfer depending on the size of the object graph to transfer. Those channels are more suitable for programs heavily accessing objects and doing few object transfers. Channel based on a write barrier lose performance on data access. Those channels are more suitable for programs exchanging a lot of objects and performing few accesses. Finally, channels based on a partial-read barrier mechanism are more appropriate for programs both transferring and accessing a lot of objects.

5.4 Discussion

Mechanism comparison in languages. Other languages having potentially more efficient write barriers do not change the conclusion brought by our results. Those write barriers will still introduce an overhead on object access. It will only change at what point accessing object becomes more expensive than copying or vice versa. In the same way,

others language having potential better copying algorithms will still introduce an overhead on object transfer.

Partial-read barrier in the form of the become message does not relate as much in other languages. The logic behind the become message is hidden in the virtual machine supporting the Pharo language and an inspection of this latter could give us a better understanding.

Memory usage. We did not measure the memory consumption induced by our different channels. Nevertheless, our write barrier is implemented by marking and checking an unused bit in the header of the objects. Therefore, the memory size of the objects is not affected at all. In contrast, the partial-read barrier leaves a placeholder object and copying duplicates the object, which both increase memory usage. Some techniques, such as persistent data, diminish the number of copies but do not completely eliminate them. In constrained memory environments still offering concurrency, a programmer should opt for channels using the write barrier.

Permissions Transfer	CVGT	FOGT	EWOT	ROOT
Rust	X	✓	X	X
Kilim	X	✓	X	X
Erlang	X	X	X	✓
Go	✓	X	X	X
C++	✓	X	X	X
Java	✓	X	X	X
Javascript	✓	X	X	X
Kotlin	✓	X	X	X
Lua	✓	X	X	X
Clojure	X	X	✓	X
Haskell	✓	X	✓	X
Pony	✓	✓	✓	✓

✓= the language offers a channel with the semantics, X = the language does not offer a channel with this semantics

Table 3. Channels permission transfer semantics offered in other languages.

6 Related Work

6.1 Permission and ownership

Capabilities as presented by Mark Miller [21] is an association between an object reference and the access permissions on this object. It can be a proxy or a handle on this association directly exchanged by processes. Capabilities evolve only by restricting further the permissions and not necessarily during a capability transfer. In our model, we exchange direct references and permissions evolve during the transfer.

Object ownership was originally introduced to control the effects of object aliasing in the context of Flexible Alias Protection. It was first embodied as a type system with ownership types [4]. Gordon et al. [11] provides ownership for dynamically-typed language for encapsulation. The ownership is by object and forms ownership trees. It encapsulates the object graph but does not handle which process is able to use this object graph. The ownership model proposed is also restrictive, the owner has all permissions while the others have none. Other models exist with more relaxed permission models such as the one proposed by Wernli et all [31]. In our model, permission is also more fine-grained granting also write or read permission.

6.2 Message passing

Message passing is present in languages focusing on distributed computing such as Erlang, Go, Scala. Singularity OS [9] also focuses on message passing between their isolated process. Pipelines, the Communicating Sequential Process (CSP) model [14], and the actor model [13] are message passing models where processes synchronize by passing messages. In Go [1], when one process finishes processing a datum, it signals it. Processes wait their turn to access a datum and consume it. This is achieved with FIFO queues called channels for CSP [15], pipe for pipeline or mailboxes for actors. Programatically, the advantages are that the communication is easy for developers to reason about as a mean of synchronisation. Nevertheless, it usually requires to copy the whole graph of data to be referenced. This is not trivial to handle [19] since the graph may be large and in the worst case, can be the whole application data. The notion of permission is not explicit with those queues but their semantics is comparable to our Copy Value Graph Transfer channel.

6.3 Shared memory

In a shared memory model, processes share some parts or all of their memory among them. Writing concurrent programs with shared memory is difficult and error prone [17] due to data races. Nowadays, each programming language provides its own ownership transfer model to support concurrency. We categorize them in two categories: the run-time and compile-time checking approaches.

Run-time checking approaches. Older languages mostly rely on run-time checking approaches. Many models exist such as the thread/mutex model [7] and the Software Transactional Memory(STM) [25] model. The most known is the mutex model in which data access is controlled by a mutex. A process has the right to access the data only if it was able to lock the mutex hence gaining ownership of the data. While it solves data races issues other problems appear such as dead or live locks [33]. Acquiring the mutex is an implicit ownership transfer and write permission gain. This semantics is similar to our semantics of Exclusive Write Object Transfer channel except the transfer is explicit in our channel.

Some CSP models coupled with shared memory exist but they only allow exchanging immutable or frozen data [18]. At run time those properties are checked when accessing the data. If the property is broken they return a run-time exception [30]. Note that only the data transferred is immutable and data referenced by this one are still freely mutable. For example, in the case of a frozen array only the array is immutable but all the elements inside are mutable, it is the developer responsibility to froze all the elements inside the array when needed. This semantics is similar to our Read-Only Object Transfer channel except in those models the write permission is completly lost whereas in our model a process retains the write permission.

Checking during execution induces an overhead specifically on frequently accessed data. The STM model with the use of persistent data aims to reduce the number of checks. A persistent data structure [24] is a data structure that preserves one or multiple previous versions of itself when it is modified. Here one process writes on the to-be-modified version meanwhile other processes read on a preserved version. The check is delayed when the modified version needs to become the preserved version. Instead of having many little checks during execution, there is only one big check. Processes accessing the data only to read are then not penalized. With all those solutions an overhead still exists at least when writing onto the data but the data transfer cost is close to non-existant. The semantics is the one of our Exclusive Write Object Transfer channel in the fact that only one process has the write permission but has the side-effect of the Copy Value object Graph Transfer channel where inconsistent reads happen.

Compile-time checking approaches. The idea for type annotation in the objective of sharing [3] data has only be demonstrated in some recent languages such as Rust, Pony and Project Midori. To synchronize between processes, Rust offers a CSP with channels but with shared memory [22]. It guarantees the uniqueness of a reference to a datum with a static analysis during compilation with a borrow-checker. The owner of this unique reference is simply the owner of the data. Rust channels are Full Ownership Object Transfer channels. Pony offers an actor model, it is one of the few actor model with shared memory. Pony [5] guarantees the uniqueness of the writer with a static analysis during compilation with type annotations. Contrary to Rust, it is possible to have multiple references to a datum but with different capabilities. If there is already a reference with the write capabilities all further reference will not have this capability for the lifetime of the first reference with the write capability. Pony type annotations allow one to express the same transfer semantics than our channels. While compile-time checking does not suffer from overhead or bigger memory usage at run time, it imposes a discipline on the developer to produce code in accordance with the permission rules [6]. However this technique is possible for statically typed language, it is not an easy feat for dynamically typed language where the control flow graph depends on the type of the receiver. They are a starting point to enhance the performance of our channels.

Table 3 summarizes the semantics tied to object transfer through channels in other languages. This list of languages is not exhaustive. Some languages are not represented because we could not determine exactly in which category they belong such as C# and Ruby. Rust and Kilim both offer FOGT semantics thanks to compile-time checks. Even though Erlang effectively copy messages, they only allow the sharing of immutable data thus having the same semantics as ROOT. Other languages that we did not list (notably functional ones) take this approach. Most of the languages with CVGT semantics follow the Go trend. They deep-copy the data to send. Note that the notion of pointer exists in some of those languages and sending a pointer is not restricted causing data races. Clojure and Haskell propose channels coupled with STM or persistent data that allow one writer and many readers. This is the EWOT semantics. Finally, Pony with its type system allows for a fine grain of permission transfer and offers each of the semantics.

7 Conclusion

We showed that sharing an object between processes is not only sharing an object but sharing all the graph of object reachable from this object as root. In a concurrent environment with shared memory, this object graph will be subject to data races. To avoid this issue, we need to control process permissions on shared object graphs while keeping good performance. Channels set a proper framework to experiment with permission transmission because of the clear delimitation between the processes send objects and the ones that acquire them. We propose an extensible channel-based permission transfer framework for experimentation, and designed four kinds of permission transfer.

We compared the performance of our transfer permission channels. On one hand, permissions transfer using pointer swapping is constantly 7 to 8 times slower than the baseline. Using a deep copy is linearly slower depending on the size of the object graph to transfer. On the other hand, using a write barrier introduces an overhead of up to 6 % on all object field writes but it does not penalize object field reads.

As future work, we want to allow the combination of channels. In another future work, we aim to improve performance. Some optimizations already exists with static analysis such as escape analysis. For dynamically-typed languages such an analysis is only possible after a number of interpretation of the program. With this analysis, clear delimitations of which part of the object graph are really used appear. Then, only for those objects are copied or the write barrier is activated. Furthermore, we would like to explore type annotation similar

to Rust or Pony and their implication for dynamically-typed languages.

References

[1] [n.d.]. The Go Programming Language. http://golang.org.

[2] Clément Béra. 2016. A low Overhead Per Object Write Barrier for the Cog VM. In *International Workshop on Smalltalk Technologies IWST'16*. Prague, Czech Republic. https://doi.org/10.1145/2991041.2991063

[3] John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing, A Generalisation of Uniqueness and Read-Only. In *Proceedings ECOOP 2001 (LNCS)*. Springer, 2–27.

[4] David G. Clarke, John M. Potter, and James Noble. 1998. Ownership types for flexible alias protection. In *Proceedings OOPSLA '98*. ACM Press, 48–64. https://doi.org/10.1145/286936.286947

[5] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 1–12.

[6] Will Crichton. 2020. The Usability of Ownership. arXiv:cs.PL/2011.06171

[7] E. W. Dijkstra. 1965. Solution of a Problem in Concurrent Programming Control. *Commun. ACM* 8, 9 (Sept. 1965), 569. https://doi.org/10.1145/365559.365617

[8] Stéphane Ducasse and Guillermo Polito. 2020. Concurrent Programming in Pharo.

[9] M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. Larus, and S. Levi. 2006. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *1st EuroSys Conference*. ACM.

[10] Daniel Schnetzer Fava and Martin Steffen. 2020. Ready, set, Go!: Data-race detection and the Go language. *Science of Computer Programming* 195 (2020), 102473.

[11] Donald Gordon and James Noble. 2007. Dynamic ownership in a dynamic language. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, Pascal Costanza and Robert Hirschfeld (Eds.). ACM, New York, NY, USA, 41–52.

[12] Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. 2013. Quantitative Relaxation of Concurrent Data Structures. *SIGPLAN Notices* 48, 1 (Jan. 2013), 317 328. https://doi.org/10.1145/2480359.2429109

[13] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI'73)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245.

[14] C.A.R. Hoare. 1978. Communicating Sequential Processes. *CACM* 21, 8 (Aug. 1978), 666–677.

[15] C.A.R. Hoare. 1985. *Communicating Sequential Processes*. Prentice-Hall.

[16] Kavya. 2017. Inner workings of go channels. (2017). https://speakerdeck.com/kavya719/understanding-channels Go doc github.

[17] E. A. Lee. 2006. The problem with threads. *Computer* 39, 5 (2006), 33–42. https://doi.org/10.1109/MC.2006.180

[18] K Rustan M Leino, Peter Müller, and Angela Wallenburg. 2008. Flexible Immutability with Frozen Objects. In *Flexible Immutability with Frozen Objects*, Vol. 5295. 192 208. https://doi.org/10.1007/978-3-540-87873-5_17

[19] Mariano Martinez Peck. 2012. *Application-Level Virtual Memory for Object-Oriented Systems*. Ph.D. Dissertation. Ecole des Mines de Douai - France & Université Lille 1 - France.

[20] Nicholas D Matsakis and Felix S Klock II. 2014. The Rust Language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

[21] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. Johns Hopkins University, Baltimore, Maryland, USA.

[22] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 21–39. https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram

[23] Ciprian Paduraru and Marius-Constantin Melemciuc. 2018. Parallelism in C++ Using Sequential Communicating Processes. *17th International Symposium on Parallel and Distributed Computing* (jun 2018), 157–163. https://doi.org/10.1109/ISPDC2018.2018.00030

[24] Neil Ivor Sarnak. 1986. *Persistent Data Structures*. Ph.D. Dissertation. New York University, USA. AAI8706779.

[25] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proc. of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. 204–213.

[26] Alexandre Skyrme, Noemi de La Rocque Rodriguez, and Roberto Ierusalimschy. 2008. Exploring Lua for Concurrent Programming. *J. Univers. Comput. Sci.* 14, 21 (2008), 3556–3572. https://doi.org/10.3217/jucs-014-21-3556

[27] S. Srinivasan. 2010. *Kilim : a server framework with lightweight actors isolation types zero-copy messaging*. Ph.D. Dissertation. University of Cambridge, King s College.

[28] George Steed and Sophia Drossopoulou. 2016. A principled design of capabilities in Pony. *Master's thesis, Imperial College* (2016).

[29] Stephen Toub. 2019. C# channels. (2019). https://devblogs.microsoft.com/dotnet/an-introduction-to-system-threading-channels/ Intro.

[30] Tom Van Cutsem and Mark S. Miller. 2013. Trustworthy Proxies - Virtualizing Objects with Invariants. In *ECOOP'13*.

[31] Erwann Wernli, Pascal Maerki, and Oscar Nierstrasz. 2013. Ownership, filters and crossing handlers: flexible ownership in dynamic languages. *ACM SIGPLAN Notices* 48 (jan 2013), 83–94. https://doi.org/10.1145/2480360.2384589

[32] Justin Wozniak, Timothy Armstrong, Michael Wilde, Daniel S. Katz, Ewing Lusk, and Ian Foster. 2013. Swift/T: scalable data flow programming for many-task applications. *ACM SIGPLAN Notices* 48 (feb 2013), 309. https://doi.org/10.1145/2517327.2442559

[33] Dieter Zöbel. 1983. The Deadlock Problem: A Classifying Bibliography. *SIGOPS Oper. Syst. Rev.* 17, 4 (Oct. 1983), 6 15. https://doi.org/10.1145/850752.850753