Inferring Ownership Transfer for Efficient Message Passing

Stas Negara Rajesh K. Karmani Gul Agha
University of Illinois at Urbana-Champaign
{snegara2,rkumar8,agha}@illinois.edu

Abstract

One of the more popular paradigms for concurrent programming is the Actor model of message passing; it has been adopted in one form or another by a number of languages and frameworks. By avoiding a shared local state and instead relying on message passing, the Actor model facilitates modular programming. An important challenge for message passing languages is to transmit messages efficiently. This requires retaining the pass-by-value semantics of messages while avoiding making a deep copy on sequential or shared memory multicore processors. A key observation is that many messages have an *ownership transfer* semantics; such messages can be sent efficiently using pointers without introducing shared state between concurrent objects. We propose a conservative static analysis algorithm which infers if the content of a message is compatible with an ownership transfer semantics. Our tool, called SOTER (for *Safe Ownership Transfer enablER*[1]) transforms the program to avoid the cost of copying the contents of a message whenever it can infer the content obeys the ownership transfer semantics. Experiments using a range of programs suggest that our conservative static analysis method is usually able to infer ownership transfer. Performance results demonstrate that the transformed programs execute up to an order of magnitude faster than the original programs.

Categories and Subject Descriptors D.2.0 [*Software*]: SOFTWARE ENGINEERING—General

General Terms Languages, Performance

Keywords Actors, Message Passing, Ownership Transfer, Static Analysis

1. Introduction

The arrival of computing platforms such as multicores and clouds have brought a disruption in the field of computer programming: mainstream developers need to write and maintain parallel programs. It has been argued that the thread based model makes the task of parallel programming unnecessarily more difficult (e.g., see [14]). The alternative, message passing as in the Actor programming model, is gaining increased interest among researchers as well as practitioners. Actors are concurrent, autonomous entities that encapsulate local state and communicate using messages. There are two important advantages of using actors. First, the code is more modular: because the state of different concurrent objects is isolated, compositional reasoning is facilitated. Second, an actor program is portable without modification across sequential processors, shared memory multicore architectures, message passing multicore architectures, as well as clusters.

Some of the well known languages and frameworks that incorporate Actor-oriented programming include Erlang, E, Scala Actors library, SALSA, ActorFoundry, Microsoft's Asynchronous Agents Library and Axum. (For a brief survey, see [10, 12]). A number of significant applications have been written using the Actors model: these include open source applications such as Twitter's message queuing system and Lift Web Framework, and commercial applications such as the image processing application in MS Visual Studio 2010 and Vendetta's game engine. In fact, MPI processes are also a special case of Actors where some of the dynamic aspects of actors are restricted. MPI has been widely used in the High-Performance Computing (HPC) community to write programs.

When actors reside in different nodes of a message passing multicore architecture, or of a networked computers, message passing provides a natural implementation. However, even on a single core, programmers want to divide their code into logical units or objects. Actors add concurrency to objects. The *grain size* of an actor corresponds to the work load generated by an actor and its memory footprint. The most intuitive decomposition of an application often leads to actors that are much smaller than the processing and memory capacity of a node. This means that many actors may time share on a single node. Sending messages between actors on the same node corresponds to a *pass-by-value* semantics [3].

Pass-by-value requires making a deep copy of message contents before the message is sent out with a fresh copy. Such copying can severely degrade performance: deep nested object oriented structures need to be serialized and deserialized to make a fresh copy. The performance penalty is the major source of inefficiency in implementations of Actor programs [10] as well as MPI programs [1]. The issue gains particular urgency with the adoption of message passing paradigm for multicore chips (with shared memory).

An important observation is that many message passing programs tend to have simple structure where messages have an *ownership transfer* semantics, i.e. an actor hands off an object or data stream to another actor by passing it in a message [9]. If such cases can be identified in Actor programs, performance may be significantly enhanced [10]. This problem is closely related to the problem of ensuring *safe yet efficient message passing* [24].

One approach to efficient messaging is to send pointers for all message contents by default, and ask the programmer to explicitly make a copy if required [8]. If the programmer is not careful, this approach breaks Actor *encapsulation* by introducing shared state between actors, thereby reducing it to shared state programming. A

[1] Soter is also the name of the Greek god of safety, deliverance, and preservation from harm.

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0119-0/11/02. . . $10.00

complimentary approach is to require the programmer to annotate message contents which can be sent by reference (pointer) safely, thus explicitly declaring ownership transfer of such objects. A deep copy for remaining message contents may be provided automatically at runtime [10]. Note that both these approaches require the programmer to reason about messages in terms of a dual semantics – pass-by-value and ownership transfer. We believe this creates confusion: for example, the best practices section [15] of the Asynchronous Agents Library documentation suggests sending pointers for large data structures in messages; however, the walk through examples [16, 17] create copies even though it would be safe in these cases to send pointers.

A recent proposal is to ask programmers to use a novel type system to annotate message content. Specifically, variants of *ownership types* have been proposed for Kilim and Scala [7, 24]. The programming model for Singularity operating system [5] also proposes a simple type system based on object capabilities for exchange of objects between different concurrent processes. We believe such type systems may be rather restrictive and cumbersome for ordinary programmers. More importantly, the question is whether requiring such type declarations can be justified for the improved performance they offer: if we could get comparable performance through automatic inference, the answer would be no.

In this paper, we explore the effectiveness of a simpler alternative: we develop and apply a static analysis method to determine the cases where messages contents can be safely passed by reference. Such an analysis relieves the programmer of the burden of using a complicated type system, or of reasoning in terms of a dual semantics for message passing. Although a sound static analysis is necessarily conservative, if it works for a large number of cases, it may be justified by a sufficient performance improvement. We evaluate the effectiveness of our static analysis method by comparing it with explicit annotations done by advanced undergraduate and graduate student programmers.

We describe the design and implementation of a custom static analysis algorithm for Actor programs written using the JVM based frameworks. The algorithm employs a novel *live variable analysis* along with *context-sensitive call graph creation* and *field-sensitive points-to analysis*. Note that the fastest known interprocedural dataflow algorithm [21] does not scale for a small Actor program, given the library calls and the size of the library code. However, we are able to exploit the *sparsity* of message sending sites (relative to the rest of computation) to design a custom interprocedural live variable analysis. Our analysis splits the expensive interprocedural analysis into two phases of intraprocedural analysis: a standard *local live variable analysis* followed by a *forward data-flow analysis*. This enables the algorithm to scale to large Actor programs, albeit by trading some precision. We show that our algorithm is sound, i.e. whenever our algorithm selects an object to be sent by reference, it does not introduce a race between two actors. We also show that the algorithm terminates.

Evaluating the effectiveness of our algorithm poses some difficulty. While there are many JVM-based Actor frameworks [10], we do not have a large base of open source code in these frameworks. However, we were able to use a variety of programs written in *ActorFoundry* [2], a JVM-based Actor framework developed at Illinois. These programs include examples from the ActorFoundry distribution, benchmarks used in [12, 13], "hard" synthetic programs written to test our analysis, and programs written by advanced Computer Science students in the project-oriented Software Engineering course at Illinois. While the programs we use are all relatively small, each of the analyses includes the ActorFoundry framework which is approximately 38.7K lines of code.

Experimental results suggest that our analysis is fast and successfully identifies majority of the places where message contents can be sent by reference. Interestingly, these include many places which were missed by programmers who were providing manual annotations. Therefore, it is a reasonable conjecture that the precision trade off of our proposed analysis is not a significant liability. Experimental results also suggest that the transformation resulting from our algorithm can improve performance of actor programs by up to an order of magnitude in some cases.

We also adapted our implementation to analyze *Scala* actor programs [8]. In Scala, message contents are sent by reference; hence we adapted SOTER to check the safety of sending messages, instead of using it to improve execution efficiency. Our tests result in two important observations. First, SOTER is able to prove the correctness for most message passing sites, and report warnings for other sites. Second, we note that the bytecode generated for Scala programs as well as for the Scala library itself tends to be much larger, as much as $10x$ larger than that for ActorFoundry. The results suggest that our algorithm scales linearly in the size of the bytecode.

We believe that our results suggest that compiler writers should focus on developing more effective inference mechanisms to optimize message passing, and this may be sufficiently effective so that programming language designers need not burden the programmer with added complexity of type systems for the purpose of improving efficiency of message passing.

2. Illustrative Actor Language

We now describe the syntax and semantics of *ActorFoundry*, which will be used to present the motivating examples for this study. ActorFoundry is syntactically similar to Java and in fact is implemented as a Java-based Actor library [2]. In the ActorFoundry, an Actor behavior is defined using a class definition which extends `Actor`. An Actor definition can include zero or more public methods annotated with `@message`, which specify the messages that an actor instance can receive. The method body is executed when an actor instance receives a corresponding message.

Upon receiving a message, an actor can create new actors (using the `create()` method and passing the name of an Actor definition), send messages to other actors (using the `send()` method) and update its local state. Messages are sent asynchronously, i.e. the sending actor does not wait for the message to arrive or be processed at the destination. A send can take an arbitrary number of arguments, which correspond to the arguments of the corresponding method in the destination actor class. Message parameters and return types should be serializable. Blocking messages can also be sent using the `call()` method, which is syntactic sugar [3, 10].

2.1 Semantics

Consider an ActorFoundry program *P*. *P* includes a set of actor definitions used to create actors. Note that there is no shared state among these actors. An actor has a local state comprising of primitives and objects. Each actor also has an unbounded queue in which it receives its messages. We assume that at the beginning of execution the message queue of every actor is empty. An actor and one of its messages is selected as the program's entry point (by the programmer). The ActorFoundry runtime first creates an instance of the initial actor, say *a_init*, and then sends the initial message *msg_init* to it. This action appends the contents of the message *msg_init* to the message queue of actor *a_init*.

Every actor initially tries to dequeue a message from its message queue; if the message queue is empty, the actor blocks and waits for the next message to arrive. If the queue is non-empty, the actor *non-deterministically* removes a message from its own queue. The non-determinism in picking a message models the asynchrony associated with message passing in actors. Each actor executes the following steps in a loop: remove a message from the queue, 'decode' the message, execute the corresponding method (which may update the actor's local state, create new actors and send new messages).

An actor executing a `create` method produces a new instance of an actor. We assume that the new actor is assigned a fresh identifier. An actor communicates with other actors it knows about by sending asynchronous messages using the `send` method (and blocking messages using `call`). An actor may also throw an exception during the processing of a message. A program terminates when there are no pending messages in the system.

3. Illustrative Example

We present a couple of examples to illustrate the problem of identifying messages that transfer ownership, as well as to motivate the different techniques we use to solving the problem. Figure 1 presents a code fragment of a `RefMessenger` actor in ActorFoundry: a `RefMessenger` actor can receive two types of messages, `store` and `transfer` (lines 3 and 6); in response to message `transfer`, it sends two messages `compute` to the actor that is bound to `relayActor` by calling the method `relayPrint` (lines 8-9). We would like to check whether a `RefMessenger` actor transfers the ownership of `item` to the `relayActor` at line 13, which would make it safe to pass by reference the object `data` to `relayActor` (through the call at line 8). In order to answer this question, we perform a static points-to analysis; the analysis detects all objects that may "escape" to `relayActor` through the message at line 13. We also perform a static live variable analysis in order to check whether, after sending the message at line 13, an "escaped" object may be accessed by the `RefMessenger` actor. Both these analyses are interprocedural and are performed on a call graph, a directed graph that represents calling relationships between procedures (subroutines) in a program. We address the particularities of call graph construction in Section 3.3.

```
1  public class RefMessenger extends Actor {
     ...
2    StringBuffer localName = null;
3    @message public void store(String name) {
4       localName = new StringBuffer(name);
5    }
6    @message public void transfer() {
7       StringBuffer data = new StringBuffer("Hi ");
8       relayPrint(data);
9       relayPrint(localName);
10      data.append(localName);
11   }
12   void relayPrint(StringBuffer item) {
13     send(relayActor, "compute", item);
14   }
15 }
```

Figure 1. A simplified version of `RefMessenger` example in ActorFoundry.

3.1 Points-to Analysis

Points-to analysis establishes which pointers may point to which memory locations. The *points-to graph* can tell that the two variables, `data` and `item`, point to the same `StringBuffer` object allocated at line 7. This graph is a result of interprocedural points-to analysis, where allocated objects are represented as nodes denoted with s_i:`T`, where i shows the line number where the object is allocated, and `T` represents its type.

Any object that may be pointed by variable `item` may escape to `relayActor` through the message at line 13 (Figure 1). Thus, performing an interprocedural points-to analysis enables detection of the fact that the object s_7:`StringBuffer` has escaped. But this analysis is not sufficient to answer the question whether this object is accessed by the `RefMessenger` actor after having escaped (sent) to `relayActor`. We employ a live variable analysis in order to answer the latter question.

3.2 Live Variable Analysis

A *live variable analysis* of a program is a dataflow analysis that calculates the set of variables that may be read before being written to in the program. An interprocedural live variable analysis performed on the code fragment in Figure 1[2] detects that variable `data` is live after the call to method `relayPrint` at line 8 completes, because its value is read and modified at line 10. Although the scope of the variable `data` is limited to the method `transfer`, and is not visible inside the method `relayPrint`, the object it points to is live throughout method `relayPrint`, including the program point right after the message to actor `relayActor` is sent (line 13). We require an interprocedural analysis to detect the escaping of the object pointed to by variable `data`, because its intraprocedural counterpart treats every method in isolation and misses the fact that the object is live in method `relayPrint`.

In Section 3.1 we established that the variable `data` points to the object s_7:`StringBuffer`, and that this object escapes to actor `relayActor`. Live variable analysis shows that this object is live in actor `RefMessenger` after escaping to actor `relayActor`. Consequently, we conclude that it is *not* safe to pass the variable `item` by reference, because passing it by reference would result in sharing the object s_7:`StringBuffer` between the `RefMessenger` actor and `relayActor`.

3.3 Call Graph Construction

The construction of a *call graph* has a significant impact on both the precision and the speed of interprocedural analysis. Our approach is based on the *open systems* design, i.e. we do not assume any information about the outside world while analyzing a particular actor. Thus we need to consider all possible messages that an actor can receive from the outside world.

Consider the `RefMessenger` actor in Figure 1. The actor can receive two types of messages, `store` and `transfer`. The execution of `RefMessenger` actor starts when it receive either one of them, and hence the methods `store` and `transfer` serve as two separate entry-points. Our analysis recognizes that they are received by the same instance of `RefMessenger`. This enables us to handle code such as that given in Figure 1 where the instance field `localName` is initialized in one message handler (line 4) but escapes in another one (line 13, through the call at line 9). Our analysis correctly detects that the object s_4:`StringBuffer` escapes to actor `relayActor` at line 13.

Context sensitivity plays an important role in call graph construction. A context-insensitive analysis produces more imprecise and smaller call graph compared to a context-sensitive analysis. In a context-insensitive call graph, every invoked method, distinguished by its signature, is represented with a single node regardless of the context in which this method is invoked.

Consider the code example from Figure 2 and its context-insensitive call graph. Were we use this call graph for our points-to analysis, we would decide that linked list `l1` has to be passed to actor `myActor` by value because there are objects that `l1` transitively points to that are live after the program point where `l1` is passed to actor `myActor` (line 7). Although this decision is safe (i.e., it does not produce a data race), it is too conservative and misses an opportunity for optimization.

[2] Static analysis performed directly on Java source code serves only for the demonstration purposes. SOTER performs both points-to and live variable analises on a low-level intermediate representation (IR) in a static single assignment (SSA) form.

```
1  public class TestActor extends Actor {
     ...
2    @message public void test(){
3      LinkedList<A> l1 = new LinkedList<A>();
4      LinkedList<A> l2 = new LinkedList<A>();
5      l1.add(new A(1));
6      l2.add(new A(2));
7      send(myActor, "process", l1);
8      l2.add(new A(3));
9    }
10 }
```

Figure 2. A code example, whose analysis is highly affected by context-sensitivity of the call graph.

An example of a context-sensitive call graph is a graph that distinguishes invocations of the same method on different receiver instances. Such a call graph would have many more nodes than its context-insensitive counterpart. However it provides much better precision. A *receiver instance context call graph* has two distinct nodes for method `add` of class `LinkedList<A>`: one node represents invocations on the linked list `l1`, and another node represents invocations on the linked list `l2`. The corresponding points-to graph shows that linked lists `l1` and `l2` transitively point to non-intersecting sets of instances of class `A`, and, consequently, an analysis based on such points-to graph would correctly decide to pass linked list `l1` to actor `myActor` by reference (line 7 in Figure 2).

For our static analysis, we construct a receiver instance context call graph in order to exploit the better precision such a call graph offers. Although a context-sensitive call graph is much bigger, it is also considerably sparser than a context-insensitive call graph. As a result, Andersen's points-to analysis performed on a context-sensitive call graph does not take much longer as demonstrated in [22]. For the final step of our analysis, namely the live variable analysis, we describe a custom interprocedural algorithm that scales well for large programs.

4. Static Analysis Algorithm

We describe our static analysis using a simple, illustrative actor program presented in Figure 3. The program consists of four classes, the last two of which specify actor behavior:

1. Class `MutableValue` is a wrapper around an integer value. The value is assigned when an instance of class `MutableValue` is created and may be changed during the lifetime of this object.
2. Class `ValueHolder` holds a `MutableValue` as a field and provides method `getMutableValue` to access the encapsulated object. Instances of class `ValueHolder` are passed between actors. In ActorFoundry, objects between actors are passed by copy, which is implemented using serialization/deserialization of objects. Therefore, both `ValueHolder` and `MutableValue` classes implement the `java.io.Serializable` interface.
3. Class `SumActor` specifies an actor that can receive message `sum` with two arguments of type `ValueHolder`. This message computes the sum of two integer values of `MutableValue` fields of the arguments, stores this sum in the `MutableValue` field of the second argument, and then prints it to the console.
4. Class `ExecutorActor` specifies an actor that can receive message `boot`. The message handler creates an instance of `SumActor` and several instances of `MutableValue` and `ValueHolder`, and then sends two `sum` messages to the created `SumActor`.

4.1 Call Graph Construction

In the first step of our analysis, we construct the program's call graph. It requires identifying all messages that actors of this program can receive, since these serve as entry-points in the constructed call graph. The call graph for the program in Figure 3 has two entry-points: one for message `sum` of `SumActor` and another one for message `boot` of `ExecutorActor`.

```
public class MutableValue implements java.io.Serializable{
  private int value;
  public MutableValue(int value){
    this.value = value;
  }
  public int getValue(){
    return value;
  }
  public void setValue(int value){
    this.value = value;
  }
}

public class ValueHolder implements java.io.Serializable{
  private MutableValue mv;
  public ValueHolder(MutableValue mv){
    this.mv = mv;
  }
  public MutableValue getMutableValue(){
    return mv;
  }
}

public class SumActor extends Actor{
  @message public void sum(ValueHolder vh1, ValueHolder vh2){
    int val = vh1.getMutableValue().getValue();
    MutableValue mv = vh2.getMutableValue();
    mv.setValue(mv.getValue() + val);
    System.out.println("Sum:" + mv.getValue());
  }
}

1  public class ExecutorActor extends Actor{
2    @message
3    public void boot(Integer val)
4                throws RemoteCodeException{
5      MutableValue mv = new MutableValue(val);
6      ValueHolder vh = new ValueHolder(mv);
7      execute(vh);
8      System.out.println("val:" + mv.getValue());
9    }
10   private void execute(ValueHolder vh)
11                 throws RemoteCodeException{
12     ActorName sumActor = create(SumActor.class);
13     add(sumActor, vh);
14   }
15   private void add(ActorName sumActor, ValueHolder vh3){
16     MutableValue mv1 = new MutableValue(1);
17     MutableValue mv2 = new MutableValue(2);
18     ValueHolder vh1 = new ValueHolder(mv1);
19     ValueHolder vh2 = new ValueHolder(mv2);
20     send(sumActor, "sum", vh1, vh2);
21     send(sumActor, "sum", vh2, vh3);
22   }
23 }
```

Figure 3. A running example of an actor program. Import statements are omitted due to space considerations.

Figure 4 shows a fragment of the constructed call graph that starts from the entry-point for message `boot`. We have omitted the part of the call graph that starts from the entry-point for message `sum`: it does not present any interesting case for our analysis because the functionality of this message does not involve sending messages to other actors. Moreover, we do not show calls to methods that are not defined within the code of class `ExecutorActor` (e.g. framework calls that are inside the body of methods `create` and `send`), and calls that construct the output `String` at line 8. The omitted parts do not affect the analysis, and have been omitted in order to simplify the presentation.

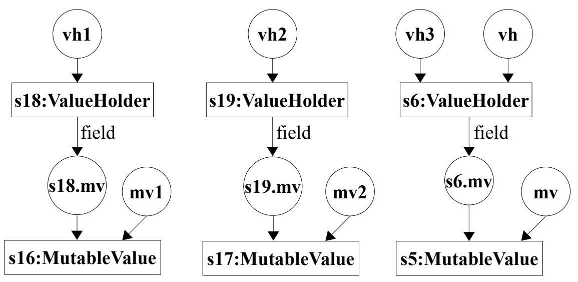

Figure 5. Filtered points-to graph for the code fragment from Figure 3.

4.2 Points-to Analysis

We use a receiver instance context call graph to perform *flow-insensitive* Andersen's points-to analysis [4]. Figure 5 illustrates a fragment of the resulting points-to graph for the code from Figure 3. This fragment is relevant to our analysis: it presents objects that escape to the actor `sumActor` (lines 20 and 21) and pointers that point to them directly or indirectly. Our points-to analysis is *field-sensitive*, i.e. it distinguishes instance fields of different instances of the same class. This allows us to distinguish instance field `mv` of different instances of `ValueHolder` as shown in Figure 5, where every instance field `mv` is represented with a separate pointer, whose name prefix corresponds to the name of the containing `ValueHolder` instance (`s18.mv`, `s19.mv`, `s6.mv`).

4.3 Live Variable Analysis

To detect objects that may be accessed after being passed to other actors, we perform an interprocedural live variable analysis. Even the fastest, polynomial time algorithms for this analysis that are precise, for example the algorithm in [21], can effectively handle only small size programs as shown in [23]. Because any actor program is analyzed together with the ActorFoundry framework, which is a relatively large 38.7KLOCs software, we cannot employ such algorithms (e.g. applying an implementation of the algorithm from [21] we ran out of memory even for the smallest actor programs). In order to be able to handle programs of such a large scale, we elaborate a custom algorithm which conservatively assumes that every instance field is live as long as the containing object is live. The key idea behind our approach is to split an interprocedural analysis into two intraprocedural phases. Our evaluation (Section 5) shows that this algorithm scales well for large programs.

We first present an overview of our algorithm; in Section 4.4, we describe its properties, including soundness and termination. Figure 6 shows an overview of our algorithm. The algorithm takes as input the receiver instance context call graph, *callGraph*, and the results of points-to analysis, *pointstoGraph*, for a given program. The output of the algorithm, *passByValue*, specifies for each argument of every message passing site in the program, whether it needs to be copied. For a particular argument *arg* of a call site *cs* the value of *passByValue[cs,arg]* is *true* when *arg* needs to be copied, and *false* when it is safe to pass *arg* by reference.

The initialization of our algorithm (lines 1-14 in Figure 6) computes the set of all message passing sites in a program, *passingCallSites*, and the set of all call graph nodes, *passingNodes*, that contain at least one message passing site. Also, it initializes all entries of *passByValue* to *false*. The algorithm visits each call site of every call graph node. For every visited call site *cs*, the procedure *isMessagePassingCallSite(cs)* returns *true* if *cs* involves sending a message to another actor (and thus, may escape objects), and *false* in the contrary case. If a call site *cs* may send messages, it is added to *passingCallSites* (line 8) and its containing call graph node is added to *passingNodes* (line 7). In ActorFoundry call sites

```
input: callGraph, pointstoGraph
output: passByValue
1  passingNodes = ∅;
2  passingCallSites = ∅;
3  foreach (Node n: callGraph){
4     callSites = getContainedCallSites(n);
5     foreach (CallSite cs: callSites){
6        if (isMessagePassingCallSite(cs)){
7           passingNodes = passingNodes ∪ n;
8           passingCallSites = passingCallSites ∪ cs;
9           foreach (Argument arg: cs){
10             passByValue[cs,arg] = false;
11          }
12       }
13    }
14 }
15 reachingNodes =
       transitiveClosure(callGraph, passingNodes);
16 callSiteLiveVariables =
       computeLiveVariables(reachingNodes);
17 nodeLiveVariables =
       propagateLiveVariables(reachingNodes,
                              callSiteLiveVariables);
18 foreach (CallSite cs: passingCallSites){
19    Node n = getContainingNode(cs);
20    liveObjects = ∅;
21    liveVariables =
         callSiteLiveVariables[cs] ∪ nodeLiveVariables[n];
22    foreach (LiveVariable var: liveVariables){
23       liveObjects = liveObjects ∪
                   getPointedObjects(pointstoGraph, var);
24    }
25    foreach (Argument arg: cs.getArguments()){
26       escapedObjects =
            getPointedObjects(pointstoGraph, arg);
27       if ((escapedObjects ∩ liveObjects) ≠ ∅){
28          passByValue[cs,arg] = true;
29       }
30    }
31 }
```

Figure 6. Overview of our algorithm for interprocedural live variable analysis.

that may send messages to other actors are calls to methods `send`, `call`, and `create` of class `Actor`. For the program in Figure 3 the set *passingCallSites* contains call sites at lines 12, 20, and 21. And the set *passingNodes* includes call graph nodes that contain these call sites. In Figure 4 these are `executorActor.add` node for call sites at lines 20 and 21, and `executorActor.execute` node for call site at line 12.

The algorithm then computes *reachingNodes* (line 15 in Figure 6) - the set of all call graph nodes that can reach *passingNodes*. The *reachingNodes* is computed by the procedure *transitiveClosure*, which takes the *callGraph* and the *passingNodes* as arguments. Figure 7 shows the procedure *transitiveClosure*, which computes all reaching nodes, *reachingNodes*, that can reach the initial set of nodes, *initNodes*, as a transitive closure of *initNodes* in a particular call graph *callGraph*. The nodes in *reachingNodes* are the only nodes in the call graph, from which the control flow may reach message passing call sites. So, *reachingNodes* contains all call graph nodes that are relevant to our analysis. For our example program *reachingNodes* includes the following nodes from Figure 4: `executorActor.add`, `executorActor.execute`, and `executorActor.boot`.

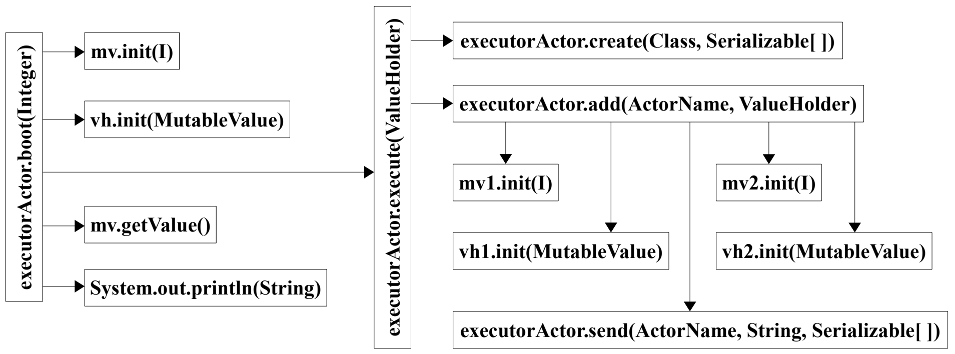
executorActor.boot(Integer)
mv.init(I)
vh.init(MutableValue)
mv.getValue()
System.out.println(String)
executorActor.execute(ValueHolder)
executorActor.create(Class, Serializable[ ])
executorActor.add(ActorName, ValueHolder)
mv1.init(I)
mv2.init(I)
vh1.init(MutableValue)
vh2.init(MutableValue)
executorActor.send(ActorName, String, Serializable[ ])

Figure 4. Filtered call graph for the code fragment from Figure 3.

```
procedure transitiveClosure
input: callGraph, initNodes
output: reachingNodes
reachingNodes = ∅;
foreach (Node n: initNodes){
  workList = {n};
  while (workList ≠ ∅){
    workNode = pop(workList);
    if (workNode ∉ reachingNodes){
      reachingNodes = reachingNodes ∪ workNode
      callers = getCallers(callGraph, workNode);
      append(workList, callers);
    }
  }
}
return reachingNodes;
```

Figure 7. Collecting all call graph nodes that reach *initNodes*.

```
procedure computeLiveVariables
input: reachingNodes
output: callSiteLiveVariables
1  foreach (Node n: reachingNodes){
2    OUT = performLocalLiveVariableAnalysis(n);
3    callSites = getContainedCallSites(n);
4    foreach (CallSite cs: callSites){
5      if (isMessagePassingCallSite(cs) OR
6        (getCalledNode(cs) ∈ reachingNodes)){
7        callSiteLiveVariables[cs] = OUT[cs];
8      }
9    }
10 }
11 return callSiteLiveVariables;
```

Figure 8. Performing local live variable analysis for *reachingNodes* and storing its relevant part in *callSiteLiveVariables*.

Next, our algorithm applies a standard intraprocedural live variable analysis to collect local variables that are live just after the relevant call sites (line 16 in Figure 6). A call site is relevant to our analysis if it is either a message passing call site or is represented as a node in the set *reachingNodes*. Figure 8 presents procedure *computeLiveVariables* that takes as input *reachingNodes* and returns *callSiteLiveVariables* which specifies the set of live variables at the program point just after a relevant call site.

For every node *n* from *reachingNodes*, procedure *computeLiveVariables* performs a standard local live variable analysis (line 2) that calculates the set of live variables for every program point in the analyzed node *n*. In order to reduce the memory consumption, we keep the results only for the program points that are relevant to our analysis, i.e. those program points that are just after relevant call sites (lines 4-9). Relevant call sites and the corresponding sets of live variables for the example program shown in Figure 3 are as follows: line 7 - {`mv`}; line 12 - {`sumActor`, `vh`}; line 13 - {}; line 20 - {`sumActor`, `vh2`, `vh3`}; line 21 - {}.

As we demonstrated in Section 3.2, if a variable *var* is live at the program point just after a call site that represents a call of some call graph node *n*, then it is live in node *n* as well. We call such variable *var* a *node live variable* for node *n*, because it is live at every program point inside node *n*. If node *n* contains other call sites, which represent calls of other call graph nodes, then variable *var* is live in those nodes too and so on. This propagation of variable *var* is a forward data-flow problem defined on the nodes of the underlying call graph. Our algorithm uses procedure *propagateLiveVariables* to compute node live variables for every node from *reachingNodes* (line 17 in Figure 6).

Figure 9 shows procedure *propagateLiveVariables* that propagates live variables forward in the call graph. It takes as input *reachingNodes* and the sets of live variables for all relevant call sites, *callSiteLiveVariables*. The output of this procedure is *nodeLiveVariables*, which specifies for every node from *reachingNodes* the set of node live variables. The initialization part of the procedure (lines 1-8) defines for every node *n* from *reachingNodes* initial values for sets *IN[n]* and *OUT[n]*, which represent correspondingly the set of node live variables at the entry and at the exit of node *n*. Both initial entry and exit sets are a union of all live variables from all call sites that call node *n* (lines 2-7). The computation part of the procedure (lines 9-17) is a fixed-point algorithm for a forward data-flow problem, where the transfer function is identity (line 15), and the meet operator is union (line 13). Procedure *getPredecessors* (line 11) returns a set of call graph nodes that immediately precede the given node. For *reachingNodes* and *callSiteLiveVariables* shown previously for our code example in Figure 3, procedure *propagateLiveVariables* computes the following *nodeLiveVariables*: `executorActor.boot` - {}, `executorActor.execute` - {`mv`}, `executorActor.add` - {`mv`}.

In the end (lines 18-31 in Figure 6), our algorithm computes for every call site *cs* from *passingCallSites* the set of all live variables, *liveVariables*, as a union of local live variables for call site *cs* and node live variables of the call graph node that contains call site *cs* (line 21). Next, we use *pointstoGraph* to compute the set of all live objects (lines 22-24). Then, for every argument *arg* of call site *cs* we compute all objects that *arg* points to, which is the set of objects that may escape to other actors (line 26). Finally, if the intersection of the objects that are live after call site *cs*, *liveObjects*, and the objects that may escape, *escapedObjects*, is not empty, we mark that *arg* should be passed by value (lines 27-29).

For the example program in Figure 3, our algorithm establishes that: call site at line 20 – argument `vh1` can be passed by reference, argument `vh2` should be passed by value; call site at line 21 – argument `vh2` can be passed by reference, argument `vh3` should

```
procedure propagateLiveVariables
input: reachingNodes, callSiteLiveVariables
output: nodeLiveVariables
1  foreach (Node n: reachingNodes){
2      IN[n] = ∅;
3      callSites = getCallingCallSites(n);
4      foreach (CallSite cs: callSites){
5          IN[n] = IN[n] ∪ callSiteLiveVariables[cs];
6      }
7      OUT[n] = IN[n];
8  }
9  do{
10     foreach (Node n: reachingNodes){
11         predecessors = getPredecessors(n);
12         foreach (Node pred: predecessors){
13             IN[n] = IN[n] ∪ OUT[pred];
14         }
15         OUT[n] = IN[n];
16     }
17 } while (changes to any OUT occur);
18 foreach (Node n: reachingNodes){
19     nodeLiveVariables[n] = OUT[n];
20 }
21 return nodeLiveVariables;
```

Figure 9. Propagating live variables through *reachingNodes*.

be passed by value. Argument `vh2` of the call site at line 20 should be passed by value, because variable `vh2` is live at the program point right after the call site at line 20. Argument `vh3` of the call site at line 21 should be passed by value, because according to the points-to graph in Figure 5, it transitively points to the object *s5:MutableValue*, which is live, as it is the same object node live variable *mv* of node `executorActor.add` points to.

4.4 Discussion

Our interprocedural live variable analysis consists of two related but distinct phases. In the first phase, we perform a standard intraprocedural live variable analysis for a subset of call graph nodes. Specifically, as shown above, we consider only those nodes of the call graph that are relevant to our analysis. Program statements of a call graph node are translated into an intermediate representation (IR) in a static single assignment (SSA) form, where every variable is assigned exactly once. Such representation significantly reduces both the time and the complexity of intraprocedural live variable analysis. In the second phase, we solve a forward data-flow problem defined on the nodes of the constructed call graph. For this problem we do not consider the internal control flow of the call graph nodes. As a result, this analysis is just like a regular intraprocedural analysis, except that we use call graph nodes instead of basic blocks and call edges instead of control flow edges.

Figure 10 illustrates our two-phase approach. In the first phase (marked with number 1) we perform intraprocedural live variable analysis on the control flow graphs of individual call graph nodes. In the second phase (marked with number 2) we propagate live variables forward in the call graph, disregarding the internal control flow of the call graph nodes. Splitting an interprocedural analysis into two phases, both of which are intraprocedural by nature, makes our algorithm fast and scalable for large programs as demonstrated in Section 5. The trade off is the reduced precision of our analysis. Although our analysis conservatively assumes that every instance field is live as long as the containing object is live, the evaluation results presented in Section 5 show that it is able to detect the majority of optimization opportunities for a variety of actor programs.

Complexity. Our algorithm consists of a standard Andersen's points-to analysis, followed by a standard intraprocedural live variable analysis for a subset of call graph nodes *n*, and a forward data flow problem for live variable propagation. The last two phases are intraprocedural by nature and have a comparable complexity. Hence, the complexity of our algorithm is $O($ complexity of Andersen's points-to analysis $+(n+1)*$ complexity of the intraprocedural live variable analysis$)$.

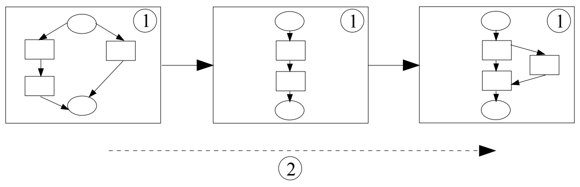

Figure 10. Two phases of our live variable analysis. In the first phase (marked with number 1) we consider internal control flow graphs of individual call graph nodes for the classical intraprocedural live variable analysis. In the second phase (marked with number 2) we propagate live variables forward in the call graph, disregarding the internal control flow of the call graph nodes.

Soundness. An argument *arg* of a message passing call site *cs* in a call graph node *n* is marked by our algorithm to be passed by value, if *arg* transitively points to at least one object *o* that is live after message passing call site *cs* (lines 18-31 in Figure 6). Considering that the employed Andersen's points-to analysis is sound, it is sufficient to demonstrate that our interprocedural live variable algorithm does not miss any objects that are live after some message passing call site. If there is a live object *o* then there should be at least one live variable *var* that transitively points to *o*. There are three kinds of variables that can be live after some message passing call site *cs* in a call graph node *n*:

- Variable *var* is a local variable in the call graph node *n*. Such variable is detected as live at the program point right after the message passing call site *cs* in the first phase of our algorithm, where we perform a standard intraprocedural live variable analysis.
- Variable *var* is a local variable in an immediate or a transitive caller node of the call graph node *n*. The second phase of our algorithm propagates such variable to the call graph node *n*, and variable *var* becomes a node live variable for node *n*.
- Variable *var* is an instance field of the class whose method is represented with the call graph node *n*. Our algorithm conservatively assumes that every instance field is live as long as the containing object is live. In this case, the containing object for variable *var* is object `this` in the method represented with the node *n*. Object `this` is always live in any instance method, and so *var* is live as well.

Thus, if variable *var* is live after some message passing call site, our algorithm detects this regardless of the kind of *var*. Consequently, all objects variable *var* points to are detected as live, including object *o*.

Termination. Observe that the first phase of our algorithm performs a standard intraprocedural live variable analysis for a subset of call graph nodes, *reachingNodes*. An intraprocedural live variable analysis for a call graph node terminates, and the number of nodes in a call graph is finite. Thus, the first phase terminates. In the second phase of our algorithm, we solve a data-flow problem using a fixed-point algorithm (lines 9-17 in Figure 9). For every node *n* from the set of nodes *reachingNodes* the set *OUT[n]* of node live variables never shrinks. Considering that the number of variables in a program is finite and the number of call graph nodes is finite, the fixed-point algorithm eventually reaches a point, when *OUT[n]* does not change for any node $n \in$ *reachingNodes*, and terminates.

Thus, the second phase terminates. Both phases of our algorithm terminate and so, our algorithm terminates.

5. Implementation and Evaluation

SOTER is a Java implementation of the static analysis described in Section 4. SOTER uses IBM T. J. Watson Libraries for Analysis (WALA) framework [25] that provides a flow-insensitive Andersen's points-to analysis and an infrastructure for implementing data-flow analysis. Our analysis algorithm is language-independent. The current implementation takes Java bytecode as input, and thus can be easily extended to handle programs in any language or framework that compiles to Java bytecode (e.g. Kilim, Jetlang, SALSA). We initially implemented support for ActorFoundry [2]. Later, we extended SOTER to support Scala [8] programs, which took only a couple of weeks of part-time effort. SOTER's source code as well as ActorFoundry and Scala subject programs can be found at http://osl.cs.uiuc.edu/soter

We performed all experiments on a 4-core 2.4GHz, 3GB RAM machine. Any ActorFoundry actor program is analyzed together with ActorFoundry framework, a relatively large software, whose bytecode size is 726KB. Any Scala program is analyzed together with Scala library, whose bytecode size is 13MB. In the worst case our analysis took around 78 seconds.

The goal of the evaluation is to assess the effectiveness and usefulness of SOTER. To achieve this goal, we applied SOTER on a variety of ActorFoundry and Scala actor programs.

5.1 ActorFoundry

For ActorFoundry actor programs, we would like to answer two questions:

- **Effectiveness:** How many opportunities to safely pass a message contents by reference are detected by SOTER in comparison to the total number of such opportunities and to what fairly sophisticated programmers can manually achieve?
- **Usefulness:** What is the performance improvement achieved by SOTER?

Table 1 presents results that assess the effectiveness of SOTER. Each row displays data for a particular actor program, whose name appears in the second column. The first column reflects the general category of an actor program.

These categories include programs from the ActorFoundry distribution, **'Benchmarks'** refers to the programs used in an earlier study [12, 13], **'Synthetic'** category is attributed to actor programs written specifically to test our analysis, and **'Real world'** programs are those written by advanced students in the Software Engineering course in Computer Science at Illinois. All presented actor programs except those from **Synthetic** category were written without the knowledge of a tool such as ours. The third column, **LOC**, shows the number of lines of code in the program (not counting comments and blank lines). The fourth column, **Bytecode size**, displays the size of the analyzed bytecode for every program. Although the size of the programs may seem relatively small, they represent a wide variety of programmers and purpose. Moreover, these programs are written on top of an Actor library. A library encapsulates much of the functionality required to express an actor program, and therefore the actor code itself has a smaller size than it would have without a library-based approach.

The fifth column, **Passed arguments**, represents the total number of message passing call site arguments present in the code of an actor program. The following two columns show correspondingly the number of arguments that could be safely passed by reference, having an ideal understanding of the analyzed program[3], and the number of arguments that SOTER reports as safe to be passed by reference. The next column, **Human misses**, presents the number of arguments that are safe to be passed by reference, which are missed by developers (advanced CS students at Illinois), who manually optimized the program. N/A in this column means that the program is not manually optimized. The following column displays the effectiveness of SOTER, i.e. the ratio of detected opportunities to safely pass arguments by reference to the total number of such opportunities.

SOTER is quite effective: on average it is able to detect around 71% of available optimization opportunities. Also, it detects some opportunities missed by developers. The last column shows how long it takes SOTER to analyze the corresponding actor program. Our analysis is quite fast: for ActorFoundry actor programs it does not exceed 24 seconds.

Table 2 shows the performance improvement achieved by SOTER by comparing the execution time of actor programs before and after application of SOTER. We exclude actor programs whose execution time is too small to base our evaluation on. For the majority of actor programs evaluated, SOTER speeds up the execution more than twice, and for two of them, by more than an order of magnitude. The last column reflects the execution time of actor programs, where all arguments that could be safely passed by reference having an ideal understanding of the actor program are indeed passed by reference. Note that for the majority of actor programs, the ideal execution time and the execution time after applying SOTER are very close. However, for some actor programs, there is still considerable room for improvement even after applying SOTER, which is mainly due to the conservatism of our static analysis. We discuss possible extensions of our analysis in Section 7.

5.2 Scala

For Scala actor programs, we assess effectiveness of SOTER in the same way as for ActorFoundry (Section 5.1), and present our results in Table 3. We divided our Scala subject programs into two categories: those that are manually annotated for Scala message passing safety according to a type system proposed by Haller et al. [7] (annotated), and the rest (unannotated). Column **Total by ref.** reflects both the total number of message passing call site arguments present in the code of an actor program and the number of message arguments passed by reference, because in Scala all messages are sent by reference. Table 3 shows that SOTER automatically proves safety of passing by reference of a significant part of annotated message arguments, as well as completely checks the correctness of 3 unannotated programs. Overall effectiveness of SOTER is around 84%.

Since in Scala all messages are sent by reference, SOTER can not improve performance any further. Instead, the usefulness of SOTER for Scala actor programs is to check automatically whether it is indeed safe to pass message arguments by reference.

According to Table 3, SOTER is able to prove the correctness of passing message arguments by reference in the majority of cases. This strongly suggests that ownership transfer is a common idiom, and SOTER is able to infer it automatically in most cases. In the rest of cases, SOTER generates a warning for the programmer to check the correctness manually. We also note that the size of Scala programs and library is almost an order of magnitude larger than those of ActorFoundry, and the results suggest that SOTER scales almost linearly for Scala programs.

[3] Note that complete knowledge of the semantics of the analyzed program yields far better results than any possible static analysis.

Table 1. The effectiveness of SOTER on different ActorFoundry actor programs. Size of ActorFoundry library, AFL=726KB.

Category	Program	LOC	Bytecode size (KB)	Passed arguments	Ideal by ref.	SOTER by ref.	Human misses	SOTER/Ideal ratio	Analysis time (sec)
ActorFoundry distribution	threadring	43	4.0 + AFL	7	7	7	1	100%	3.4
	concurrent	204	11.3 + AFL	12	12	7	N/A	58%	3.8
	copymessages	80	8.4 + AFL	19	18	10	5	56%	12.5
	performance	126	12.6 + AFL	14	14	12	N/A	86%	3.6
	pingpong	62	6.4 + AFL	9	9	8	N/A	89%	3.5
	refmessages	20	3.3 + AFL	3	3	2	2	67%	3.3
	rpcping	65	7.7 + AFL	9	9	9	N/A	100%	3.4
	sor	320	22.7 + AFL	36	36	18	10	50%	3.8
Benchmarks	chameneos	187	13.6 + AFL	12	12	4	1	33%	3.5
	fibonacci	53	8.8 + AFL	28	28	24	N/A	86%	3.5
	leader	81	7.0 + AFL	12	12	2	N/A	17%	3.4
	philosophers	77	9.5 + AFL	6	6	6	N/A	100%	3.3
	pi	73	8.4 + AFL	6	6	4	N/A	67%	3.4
	shortestpath	126	7.4 + AFL	59	59	52	N/A	88%	3.5
Synthetic	quicksortCopy	76	5.3 + AFL	3	3	3	N/A	100%	12.5
	quicksortCopy2	92	5.3 + AFL	8	8	6	N/A	75%	12.4
Real world	clownfish	700	48.8 + AFL	87	87	59	N/A	68%	24.0
	rainbow_fish	591	47.7 + AFL	68	68	67	N/A	99%	3.6
	swordfish	615	37.1 + AFL	83	83	13	N/A	16%	4.1
	threadfin	471	27.8 + AFL	101	101	98	N/A	97%	12.8

Table 2. The performance improvement achieved by SOTER.

Program	Parameters	Execution time (ms)		Improvement	Speed up	Ideal execution time (ms)
		Before	After			
threadring	504 actors, 1 mil passes	25870	1880	92.7%	13.76	1880
concurrent	601 actors	187510	15990	91.5%	11.73	1390
copymessages	31810 actors, 10000 elements	7730	3710	52.0%	2.08	610
sor	6402 actors, 80 x 80 matrix	76960	61620	19.9%	1.25	5890
chameneos	14 actors, 100000 rendezvous	56890	36640	35.6%	1.55	1620
leader	30001 actors	14050	8190	41.7%	1.72	7380
philosophers	60001 actors, 30000 philosophers	9550	1380	85.5%	6.92	1380
pi	3002 actors, 30000 intervals	4210	3890	7.6%	1.08	3880
quicksortCopy	200002 actors, 100000 elements	24660	4530	81.6%	5.44	4530
quicksortCopy2	200002 actors, 100000 elements	16320	4870	70.2%	3.35	3580

Table 3. The effectiveness of SOTER on different Scala actor programs. Size of Scala library, SL=13MB.

Category	Program	LOC	Bytecode size (KB)	Total by ref.	SOTER by ref.	SOTER/Total ratio	Analysis time (sec)
annotated	RayTracer	693	247.1 + SL	3	2	67%	77.9
	RayTracer2	688	253.3 + SL	3	2	67%	75.7
unannotated	Leader	91	35.1 + SL	10	10	100%	21.2
	ShortestPath	99	32.6 + SL	3	2	67%	25.7
	Fibonacci	69	30.2 + SL	4	4	100%	21.1
	QuickSortCopy	83	40.0 + SL	4	2	50%	22.3
	DiningPhilosophers	78	56.4 + SL	5	5	100%	21.7

6. Related Work

The problem of safe yet efficient message passing has attracted much interest among researchers recently. Erlang, which has been around for more than 20 years, uses only immutable types in messages. However, immutable objects can have a severe performance penalty as every update to the object involves copying data values or pointers. This increases space and time complexity of implementing immutable types, specially for large data structures. We discuss more recent proposals based on type systems below.

6.1 Type Systems

Various type systems have been proposed to control aliasing in ways which makes it safe and efficient to share objects. These include variants of Linear types, Ownership types and recently Universe Types [19]. A detailed summary and comparison has been presented in [7]. We briefly mention three systems which are proposed specifically for message passing. The Singularity Operating System is architected essentially using actors [9]. Messages are not allowed to have internal aliasing, but are exchanged using a special exchange heap [5]. Kilim [24] proposes a variant of Linear Types combined with an interprocedural shape analysis. The type system is quite restrictive in terms of shapes of message objects (tree shaped) and may not be easily understandable by programmers. Recently a type system based on uniqueness and capabilities has been proposed for Scala Actors [7]. This system allows richer message structures and the examples in the paper show that it requires fewer annotations.

An inherent difficulty with type systems is that programmers have to annotate their code and this puts varying degree of overhead on them. However, the type systems do provide guidance on what properties to analyze and infer in order to guarantee other properties like ownership.

6.2 Inferring Ownership and other Types

In order to ease this burden, mechanisms have been proposed to infer some of the aforementioned types. Specifically, techniques to infer Universe Types have been proposed in [11, 20]. These techniques are based on a static analysis and a SAT solving step. The techniques are augmented with a dynamic analysis in [6]. In [18], a static analysis based on Andersen's points-to analysis is proposed to infer the ownership of heap objects.

Note that these methods only infer ownership but do not deal with transfer of ownership. Actors on the other hand have the ownership of their state by construction, and we are interested in cases when the ownership is transferred to other actors.

7. Discussion and Future Work

Our preliminary results suggest that an inexpensive but conservative static analysis method can infer ownership transfer so that it may identify most places where messages can be sent efficiently by passing pointers instead of making a deep copy. The results suggest that much of the efficiency of message passing programs executed on shared memory multicores may be regained without requiring programmers to deal with the complexity of type annotations or to reason about a dual message passing semantics. In fact, our experimental results suggest that programmers may often be able to do no better.

SOTER currently does not identify safe read-read sharing. In cases where the entire program is available (closed systems), identifying such sharing would enable further optimizations. Our analysis also does not extend to cases of false sharing where different segments of a large data structure are accessed by different actors, possibly at different times. In this case, our static analysis may improve the efficiency of a dynamic analysis framework which checks the safety of such sharing. However, we believe that as languages and frameworks evolve to be more Actor-oriented, data structures themselves will be designed and implemented as collections of actors, thus avoiding problems created by passing large data structures. Such actor collections will enable concurrency while enforcing the consistency required by the abstract data type defined by the collection.

Acknowledgments

The authors would like to thank the anonymous reviewers for their detailed and useful feedback, and members of the development teams of the Axum and Orleans projects at Microsoft for useful discussions leading to this work. We would like to acknowledge the assistance of Philipp Haller in obtaining Scala programs, and Samira Tasharofi, Steven Lauterburg, and others for providing actor programs.

References

[1] Panel on A single programming model for clusters and multiprocessor nodes: Dream, nightmare, reality, or vision at UIUC, 2009.

[2] ActorFoundry. ActorFoundry homepage. http://osl.cs.uiuc.edu/af, 1998-2010.

[3] G. Agha, I. A. Mason, S. Smith, and C. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7(01):1–72, 1997.

[4] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.

[5] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity os. *SIGOPS Oper. Syst. Rev.*, 40 (4):177–190, 2006.

[6] A. Fuerer. Combining run-time and static universe type inference. Master's thesis, ETH Zurich, 2007.

[7] P. Haller and M. Odersky. Capabilities for Uniqueness and Borrowing. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP)*, 2010.

[8] P. Haller and M. Odersky. Actors that unify threads and events. In *COORDINATION*, 2007.

[9] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007. doi: http://doi.acm.org/10.1145/1243418.1243424.

[10] R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: A comparative analysis. In *Proceedings of the 7th International Conference on the Principles and Practice of Programming in Java*, 2009.

[11] N. Kelleberger. Static universe type inference. Master's thesis, ETH Zurich, 2005.

[12] S. Lauterburg, M. Dotta, D. Marinov, and G. Agha. A framework for state-space exploration of Java-based actor programs. In *24th IEEE/ACM International Conference on Automated Software Engineering, ASE 2009*. IEEE, 2009.

[13] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In *Fundamental Approaches to Software Engineering (FASE) with ETAPS*, 2010.

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

[15] Microsoft Corporation. Best Practices in the Asynchronous Agents Library - Do Not Pass Large Message Payloads by Value. http://msdn.microsoft.com/en-us/library/ff601928.aspx, 2010.

[16] Microsoft Corporation. Asynchronous Agents Library Walkthrough: Creating an Agent-Based Application. http://msdn.microsoft.com/en-us/library/dd504791.aspx, 2010.

[17] Microsoft Corporation. Asynchronous Agents Library Walkthrough: Creating a Dataflow Agent. http://msdn.microsoft.com/en-us/library/dd504791.aspx, 2010.

[18] A. Milanova. Static Inference of Universe Types. In *Intl. Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, 2008.

[19] P. Müller and A. Rudich. Ownership transfer in universe types. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 461–478, 2007.

[20] M. Niklaus. Static universe type inference using a SAT-solver. Master's thesis, ETH Zurich, 2006.

[21] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[22] M. Sridharan and S. J. Fink. The complexity of Andersen's analysis in practice. In *SAS '09: Proceedings of the 16th International Symposium on Static Analysis*, pages 205–221, Berlin, Heidelberg, 2009. Springer-Verlag.

[23] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 112–122, New York, NY, USA, 2007. ACM. doi: http://doi.acm.org/10.1145/1250734.1250748.

[24] S. Srinivasan and A. Mycroft. Kilim: Isolation typed actors for Java. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP)*, 2008.

[25] WALA. WALA Static Analysis Library. http://wala.sourceforge.net/.