Proving Copyless Message Passing *

Jules Villard
LSV, ENS Cachan, CNRS

Asynchronous message passing often suffers from two drawbacks: contents of messages have to be copied, and deadlocks can be tricky to avoid. However, if messages to-be live in the same address space, the first issue can be resolved by sending a mere pointer to the memory region where the message is stored instead of issuing a copy. This implementation is sound provided that the emitting thread loses ownership over the message, *i.e.* does not access it for reading or writing after emission.

The Singularity operating system [4] is a prominent application of these ideas. It can safely run processes sharing a unique address space without memory protection. Executable processes are written in the `Sing#` programming language, which supports (copyless) message passing primitives. Ownership violations are detected at compile-time using static analysis techniques. Moreover, communications are ruled by contracts, a form of session types [8], and this feature seems to be essential for the static analysis.

The goal of this abstract is to formalize these ideas and give a proof system that validates ownership transfers and contract obedience for an idealized programming language. The latter allows memory manipulation and asynchronous communications ruled by contracts, following the ideas of `Sing#`. Moreover, we chose to be able to detect memory leaks, whereas `Sing#` is equipped with a garbage collector, and we support complete mobility of channels, similar to the π-calculus, whereas `Sing#` provides internal mobility only. Our proof system is based on separation logic [7], which has already been used to specify and prove various ownership-based paradigms [5, 2].

Programming Language

In the following code snippet, lines 1-2 send the cell pointed to by x over some pre-existing endpoint e of a channel (e, f) using message $cell$, which is received on the other end of the channel (f) on lines 4-5. The two blocks are executed in parallel; if we know that the first one will not access the content of x anymore, then we can guarantee that the disposal of line 5 will not create a memory fault.

```
1 { x = new();
  /* allocate a new cell on the heap */
2   send(e,cell,x);
  /* send a message "cell" containing this cell */
3 } || { /* parallel composition */
4 y = receive(f,cell);
  /* receive a message "cell" containing a cell */
5 dispose(y); }
  /* frees the memory used by the cell */
```

A more elaborate example, which we will use throughout this abstract, is the sending of a linked list (that uses a field tl to access the next element), followed by the closure of the channel: program `putter` sends a list located at x over endpoint e, and then sends e over itself. After each send, it waits for an ack message. `getter` sends this ack after each cell is received, and then proceeds to processing that cell; it eventually receives the endpoint e through its endpoint f, which allows it to close the channel (e, f).[1]

```
1  putter(e,x) {
2    local t;
3    while(x != NULL) {
4      t = x->tl;
5      send(cell,e,x);
6      x = t;
7      receive(ack,e); }
8    send(close_me,e,e); }
9
10 getter(f) {
11   local x, e' = NULL;
12   while(e' == NULL) {
13     switch receive {
14     x = receive(cell,f): {
15       send(ack,e);
16       /* process the cell... */
17       dispose(x); }
18     e' = receive(close_me,f): {}
19   }}
20   close(e',f); }
```

The two programs assume that a channel has been created for communication. In our setting, channels are bidirectional FIFO and always consist of exactly two endpoints, here e and f. Communications are asynchronous. Sending never fails, and receiving may block until the right message

*This is a short and informal version of an article currently under submission that can be found at `http://www.lsv.ens-cachan.fr/~villard/pub/clmp.pdf`.

[1] We have chosen a syntax where both ends of a channel are closed together.

has arrived. The `switch receive` statement indicates that several messages may be received at a given program point.

Let us give the program that, given a linked list starting at x, opens a channel and launches `putter` and `getter` in parallel.

```
21 send_list(x) {
22   local e,f;
23   (e,f) = open();
24   putter(e,x) || getter(f);}
```

Had we omitted the acknowledgment mechanism of our example (by suppressing lines 7 and 15), the contents of the channel would be unbounded, as `putter` might do arbitrarily many sends before `getter` starts receiving. This could be seen as a caveat in a context where memory is scarce, *e.g.* if such code is found in an operating system and is executed in an out-of-memory situation. The ack messages force each send to be processed one by one, thus bounding the channel's size to one.

Contracts. We are interested in checking two properties regarding communications[2]:

Bounded
: The sizes of all channels' queues are bounded.

NoLeak
: No messages are pending when a channel is closed.

To be able to check that these properties hold in a tractable way using our proof system, we associate a *contract* to each channel. Contracts are finite state machines that describe the protocol the channel should follow, *i.e.* which sequences of sends and receives are admissible on the channel. A contract C is written from one of the endpoints' point-of-view, the other one following the dual contract $\bar{C}$, where sends ! and receives ? have been swapped. Contracts distinguish an initial state and a set of possible final states. When both endpoints of a channel are in the same final state, the channel may be closed. A possible contract C for our example is:

```
1 contract C {
2   initial state transfer { !cell     -> wait;
3                            !close_me -> end; }
4   state wait { ?ack -> transfer; }
5   final state end {} }
```

The program `send_list` is then modified to specify that the channel follows contract C: line 23 becomes `(e,f) = open(C).`

We give simple syntactical conditions on contracts that suffice to ensure **Bounded** and **NoLeak**:

Deterministic
: From every state of the contract, there should be at most one transition labeled by a given message name and a given direction.

Positional
: Every state of the contract must allow either only sends or only receives.

Synch
: All cycles in the contract must contain at least one send and one receive.

Theorem 1 ***Deterministic*** *&* ***Positional*** *&* ***Synch*** *imply* ***Bounded*** *&* ***NoLeak****.*

Contract C satisfies these three conditions, hence closing the channel on line 20 does not leak memory, as we will see in the next section. If we relax **Synch** to apply only to cycles that contain final states, then **NoLeak** still holds, but **Bounded** does not. This is the case in our example if we suppress the acknowledgment mechanism, as described earlier. In the following, we only consider contracts that satisfy **NoLeak**.

A Separation Logic for Copyless Message Passing

We now turn to defining a proof system for proving that program do not fault (on memory accesses), and, what is new in this work, that they obey ownership transfer of messages. More precisely, we will specify programs p using Hoare triples $\{A\}\ p\ \{B\}$ where A and B are formulas of our logic, an extension of separation logic. Our goal is to define a proof system for Hoare triples and a semantics for programs, and to relate them by a soundness theorem stated informally below.

Theorem 2 (Soundness) *If $\{A\}\ p\ \{B\}$ is derivable in the proof system, then for every initial state satisfying formula A, if p terminates on this state then it does not fault, it is race free, the contracts are respected, and the final states of the program satisfy formula B.*

Two crucial ingredients of separation logic are the separating conjunction, in the assertion language, and the frame rule, in the proof system (which contains also standard Hoare logic rules), that permit local reasoning. The separating conjunction $A * B$ of two formulas A and B is verified by all states that can be split into two *disjoint* parts such that one of them satisfies A and the other satisfies B. The rule for parallel composition uses this to specify that disjoint pieces of the current state should be distributed to each program of the parallel composition at the beginning, and the (disjoint) results should be glued back together at the end:

$$\frac{\{A_1\}\ p_1\ \{B_1\} \qquad \{A_2\}\ p_2\ \{B_2\}}{\{A_1 * A_2\}\ p_1 \parallel p_2\ \{B_1 * B_2\}}\ \textsc{Parallel}$$

[2] We also believe that contracts can be of some help to ensure that communications never fail (*e.g.* when the wrong message is received) nor deadlock, but have not yet investigated this line of work.

The frame rule is based on the idea that to prove a program, one only needs to consider the portion of state that is accessed by it (its *footprint* [6]):

$$\frac{\{A\}\ p\ \{B\}}{\{A * F\}\ p\ \{B * F\}}\ \textsc{Frame}$$

In our model, following standard separation logic, states are given by a stack and a heap. The heap describes cells and endpoints that are currently allocated, and the stack keeps track of the values associated to local variables. Our assertion language then extends the one of separation logic with a points-to predicate of the form $E \overset{ep}{\mapsto} (C\{a\}, F)$ that describes endpoints on the heap. Each expression E and F can be either a value (a memory location or an integer) or a program variable. This predicate means that only the endpoint at address E is allocated on the heap, and this endpoint follows contract C, is in the state a of C, and its *peer* (the other end of the channel) is denoted by expression F. The rest of our logic includes classical first order, $*$ and its adjoint $-\!\!*$ ($A -\!\!* B$ is true of states that satisfy B when any state satisfying A is added to them) and the usual predicates of separation logic, including emp that describes the empty heap. We also treat variables as resources [1]. Our proof system extends separation logic with axioms for `open`, `send`, `receive` and `close`.

For instance, the axiom for `open` is the following:

$$\frac{i = \text{init}(C)}{\begin{array}{c}\{e, f \Vdash \mathsf{emp}\} \\ (\mathrm{e}, \mathrm{f})\ =\ \mathrm{open}(\mathrm{C}) \\ \{e, f \Vdash e \overset{ep}{\mapsto} (C\{i\}, f) * f \overset{ep}{\mapsto} (\bar{C}\{i\}, e)\}\end{array}}$$

The pre-condition requires variables e and f to be allocated on the stack and an empty heap, and the post-condition is a two-cell heap made of two coupled endpoints following dual contracts C and $\bar{C}$ and placed in the initial state i of C. Similarly, `close(E,E')` must check that E and E' are two expressions that can be evaluated using available variables listed by O (we will use this notation each time an expression needs to be evaluated in the pre-condition), and whose values are two coupled endpoints in the same final state. Closing such a channel gives an empty heap thanks to Theorem 1.

$$\frac{a \in \text{final}(C)}{\begin{array}{c}\{O \Vdash E \overset{ep}{\mapsto} (C\{a\}, E') * E' \overset{ep}{\mapsto} (\bar{C}\{a\}, E)\} \\ \mathrm{close}\,(\mathrm{E},\mathrm{E'}) \\ \{O \Vdash \mathsf{emp}\}\end{array}}$$

To enforce ownership transfer of messages (required by the copyless paradigm) in a way that permits local reasoning, we must describe the contents associated to messages. For example, *ack* does not carry anything, while *cell* should carry a single cell, pointed to by the third argument (the *parameter*) of `send(cell,e,x)`. Hence, we associate an *invariant* to each message, in the spirit of concurrent separation logic [5]. Invariants are precise formulas where the only variables that may appear free are `val` and `src`, which are both instantiated at run-time, respectively by the value of the parameter and by the sending endpoint. We define the following invariants for our example ($\mapsto$ is used to denote a regular cell, as opposed to endpoints denoted by $\overset{ep}{\mapsto}$):

$$\begin{array}{rcl} I_{ack}(\mathtt{val}, \mathtt{src}) & \triangleq & \mathsf{emp} \\ I_{cell}(\mathtt{val}, \mathtt{src}) & \triangleq & \exists X.\, \mathtt{val} \mapsto X \\ I_{close\_me}(\mathtt{val}, \mathtt{src}) & \triangleq & \exists X.\, \mathtt{val} \overset{ep}{\mapsto} (C\{\mathrm{end}\}, X) \\ & & \quad \wedge\ \mathtt{val} = \mathtt{src} \end{array}$$

Ownership transfer is then materialized by the fact that the invariant of a message is removed from the heap when the message is sent, and tacked on the heap when the message is received:

$$\frac{a \overset{!m}{\longrightarrow} b \in C}{\begin{array}{c}\{O \Vdash E \overset{ep}{\mapsto} (C\{a\}, \varepsilon) * (E \overset{ep}{\mapsto} (C\{b\}, \varepsilon) -\!\!* (I_m(E, F) * A))\} \\ \mathrm{send}(\mathrm{m},\mathrm{E},\mathrm{F}) \\ \{O \Vdash A\}\end{array}}$$

$$\frac{a \overset{?m}{\longrightarrow} b \in C}{\begin{array}{c}\{O, x \Vdash E \overset{ep}{\mapsto} (C\{a\}, \varepsilon)\} \\ \mathrm{x}\ =\ \mathrm{receive}\ (\mathrm{m},\mathrm{E}) \\ \{O, x \Vdash E \overset{ep}{\mapsto} (C\{b\}, \varepsilon) * I_m(\varepsilon, x)\}\end{array}}$$

The rule for `send` first updates the state of the endpoint according to the contract by removing it from the heap and then adding it back in the new state b using $-\!\!*$. The resulting state must contain the invariant and possibly some left-over state described by A. In practice, A will either be $E \overset{ep}{\mapsto} (C\{b\}, \varepsilon)$, meaning that the endpoint is not part of the invariant (like in the case of *cell*), or emp, meaning that the endpoint was sent in the message (like in the case of *close_me*). The latter case explains why we need the intricate construct with $-\!\!*$ in the precondition: the state of the sending endpoint must be updated *before* it is sent.

To see how the invariant of *close_me* allows us to deduce that both ends of the same channel are present, which is required to close the channel at line 20 of our example, let us show how to derive

$$\begin{array}{l}\{x, e', f \Vdash \exists X.\, f \overset{ep}{\mapsto} (\bar{C}\{\mathrm{transfer}\}, X)\} \\ \mathrm{e'}\ =\ \mathrm{receive}\ (\mathrm{close\_me},\mathrm{f});\ \ \mathrm{close}\ (\mathrm{e'},\mathrm{f}); \\ \{x, e', f \Vdash \mathsf{emp}\}\end{array}$$

After the receive, according to I_{close_me}, we obtain a state satisfying

$$x, e', f \Vdash \exists X.\, f \overset{ep}{\mapsto} (\bar{C}\{\mathrm{end}\}, X) * \exists Y.\, e' \overset{ep}{\mapsto} (C\{\mathrm{end}\}, Y) \wedge e' = X$$

from which we can deduce that $Y = f$. Thus, the invariant of $close\_me$ has helped us to discover information about a distant piece of the heap (the endpoint e' that is received) while keeping the reasoning purely local. This allows us to apply the rule for `close`, which gives the desired post-condition.

We can prove the following specifications for our example, given an inductive predicate $\mathrm{list}(x) \triangleq (x = 0 \wedge \mathsf{emp}) \vee (x \neq 0 \wedge \exists X.\, x \mapsto X * \mathrm{list}(X))$:

$$\{\mathrm{list}(x)\}\ \ \mathsf{send\_list}\ (\mathsf{x})\ \{\mathsf{emp}\}$$

$$\{\exists X.\, e \overset{ep}{\mapsto} (C\{transfer\}, X) * \mathrm{list}(x)\}\ \ \mathsf{putter}\ (\mathsf{e},\mathsf{x})\ \{\mathsf{emp}\}$$

$$\{\exists X.\, f \overset{ep}{\mapsto} (\bar{C}\{transfer\}, X)\}\ \ \mathsf{getter}\ (\mathsf{f})\ \{\mathsf{emp}\}$$

Soundness. To give a semantics to our programming language, we use abstract separation logic [3] that derives a trace-based denotational semantics from the specifications of commands, and automatically ensures soundness of our proof system for this semantics. However, we can already see that the semantics we will obtain based on the axioms of `send` and `receive` will be quite distant from our initial goal of modeling copyless message passing. Indeed, sending will correspond to erasing the contents of the message, and receiving to non-deterministically generating a message satisfying the invariant and adding it to the current state. This is a good description of what happens as far as local reasoning is concerned but, when considering the program as a whole, this is not a satisfying semantics.

Moreover, due to the technical details of how it was conceived, the synchronization mechanism of abstract separation logic is too weak for our setting, which means that the semantics given by abstract separation logic will not prevent a `receive` to happen before the corresponding `send`.

We have thus enriched our model and the semantics of `send` and `receive` to log all communications on every channel and to count the number of exchanged messages on each endpoint (by associating counters to endpoints). The counters allow us to force receives to block until the corresponding send has happened (technically, we rule out states where the receive counter of an endpoint is ahead the send counter of its peer), and logs are used to make sure that the contents of sent and received messages match (again by ruling out states where two corresponding log contents differ).

Finally, we define a third semantics where `send` and `receive` only increment the corresponding counters. Communications are then reduced to mere exchanges of pointers, as was intended. We then show the correspondence of all three semantics for programs that are provable with our proof system in a third theorem:

Theorem 3 *For every provable program p with contracts satisfying **NoLeak**, communications can safely be reduced to pointer-passing.*

Conclusion

In this abstract, we have described informally a new proof system that allows local reasoning for memory-manipulating programs that communicate by message passing, and a semantics for such programs which, while being based on it, goes beyond the limitations of abstract separations logic by exhibiting complex synchronization mechanisms. We also showed how contracts are related to the analysis, thus giving a formal setting to the analysis performed by `Sing#`.

Acknowledgments. The author would like to acknowledge the work of Étienne Lozes on the matter discussed in this abstract, and thank Cristiano Calcagno for his help and initial impulse.

References

[1] R. Bornat, C. Calcagno, and H. Yang. Variables as Resource in Separation Logic. *Electronic Notes in Theoretical Computer Science*, 155:247–276, 2006.

[2] C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular Safety Checking for Fine-Grained Concurrency. *Lecture Notes in Computer Science*, 4634:233, 2007.

[3] Cristiano Calcagno, Peter O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *22nd LICS*, pages 366–378, 2007.

[4] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*, 2006.

[5] P.W. O'Hearn. Resources, concurrency, and local reasoning. *TCS*, 375(1-3):271–307, 2007.

[6] Mohammad Raza and Philippa Gardner. Footprints in local reasoning. In *FoSSaCS*, volume 4962 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2008.

[7] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002*.

[8] K. Takeuchi, K. Honda, and M. Kubo. An Interaction-Based Language and Its Typing System. *Lecture Notes in Computer Science*, pages 398–398, 1994.