Locality, Causality and Continuations

Christian Queinnec[*]
École Polytechnique & INRIA-Rocquencourt

Abstract

Concurrency and distribution are topics exacerbated by the omnipresence of Internet. Although many languages address these topics, few offer a real opportunity to control and coordinate widely spread dynamic computations. This paper presents such a language and its prominent features. Besides explaining the software architecture of the implementation (based on objects and generic functions), it also presents an original coherency protocol for shared mutable variables.

We first recall, in Section 1, the main features of our Scheme-based, concurrent, distributed and computation-oriented language already presented in more details and examples in [QD93].

Section 2 explains how to systematically implement a concurrent and distributed interpreter for that language, using a set of program transformations combining Abstract Continuation Passing Style (ACPS) [FWFD88] and Object-Oriented Lifting. The originality of this implementation is that it chiefly uses objects and generic functions in a style that allows to concentrate the problems related to concurrency and migration of computations into the sole discriminating behavior of generic functions. ACPS is not only used to reify continuations but also to enforce locality of computations in presence of distal objects.

In Section 3, we propose a new (to our knowledge) protocol to manage shared mutable variables. This protocol enhances [MSRN92], does not require atomic broadcast, tolerates short communication breakdowns and uses bounded circular clocks. This result comes from the use of a distributed GC [LQP92] (which allows us to maintain an approximation of Global Virtual Time [Jef85]) and from the exploitation of causality as stated by continuations. To give a continuation a value (and a store) clearly expresses that the computations that are present in the continuation causally depend on the invoker of the continuation.

Finally the computation-orientation of our language and mainly the ability to control groups of threads, concurrently running on multiple sites for the completion of the evaluation of a single expression, is shortly sketched in Section 4.

As usual, related works and conclusions end this paper.

[*]Laboratoire d'Informatique de l'École Polytechnique (URA 1439), 91128 Palaiseau Cedex, France Email: Christian.Queinnec@polytechnique.fr This work has been partially funded by GDR-PRC de Programmation du CNRS.

1 Features of Icslas

This Section presents the main features of Icslas, a Scheme-based, concurrent, distributed and computation-oriented language. We shortly recall the main features and the rationale behind some previous design choices detailed in [Que90, Que92, QD93].

1.1 Philosophy

The Icslas language can be roughly described as Scheme minus a predefined `call/cc` but plus the possibility to control birth and death of threads, to migrate computations towards distant *sites* (processes or processors with a private value space) and to control groups of threads cooperating in the evaluation of an expression.

Contrarily to many other concurrent and distributed languages, the goal of Icslas is not to obtain speed-up factors but rather to ease the writing of intrisically distributed systems like the `news` system, the programmation of knowbots to find informations on Internet or, even, the management of Icslas own sources. The typical applications aimed by Icslas involve the management of a distributed but coherent state (or store) over which multiple threads of computations harmoniously cooperate.

We do not intend to program entire applications in Icslas but rather to express in Icslas how to compose computations that are already well handled by other software tools. The Icslas language appears as a glue or as a distributed shell, it must therefore offers at least the possibilities of a shell language i.e., side-effect and indeterminacy. To ease programming, Icslas adopt the distributed shared memory model (DSM).

Although many concurrent and distributed language proposals start from Object-Oriented Languages (OOL), we chose to build upon Scheme since it has a clean and essential semantics and also because Lisp dialects allow more freedom w.r.t. programming styles. We thus reject the prominence of the message-passing paradigm since it is too low-level and too restrictive. Nonetheless we find it useful to implement Icslas itself.

We analyzed the relationship between concurrency and continuations in [Que92] where we gave a denotational semantics for a first sketch of Icslas. First-class threads and first-class continuations cohabit with difficulty: for instance, if a thread θ' invokes a continuation κ reified by a thread θ then, to which thread belongs an invocation of κ ? To avoid these problems, differently addressed in [IM89], we decided to keep first-class continuations which are useful to express control operators, and to relegate threads to backstage. Threads are not first-class values, they cannot be sent signals nor they can support any other UNIX[1]-inspired features.

In fact, it seems to us that writing a distributed system is the art of harmoniously composing multiple simultaneous computations. Thus it must be possible to manage computations as a whole whatever number of threads they comprise and whichever number of involved sites. A computation i.e., the processing power needed for the evaluation of an Sexpression, can be identified to the group of threads running for that evaluation. Contrarily to a thread, a group of threads is a first-class value which incarnates what a computation is. In order to make explicit the relationship between threads and groups of threads, continuations must be tamed i.e., have a behavior compatible with groups i.e., be restricted to their dynamic extent. That is why `call/cc` is ruled out since it concentrates (and blurs) three different effects that are provided by different constructions in ICSLAS: *(i)* dynamic extent escape *à la* `catch/throw`, as in COMMON LISP [Ste90], that we provide as `call/ep` standing for *call with exit procedure* [IM89]; *(ii)* coroutine-like suspension/resumption provided (as well as generalized) by the group control operators `pause!` and `awake!`, see description below and examples in [QD93]; *(iii)* multiple returns where a continuation is multiply invoked. This lattest effect is also possible in ICSLAS because of concurrency which allows threads that share a common continuation to invoke it simultaneously. This has far-reaching effects in Scheme, but also in a concurrent framework, since programs have to protect themselves from that effect, for instance: it is possible to exit from a critical section more than once!

To ease the description of ICSLAS as well as to follow a tradition of Scheme, all the new features are offered as functions rather than special forms. This will make simpler the presentation of our extensions and leaves room for alternate syntaxes using macros.

1.2 New features

This Section briefly describes the main features of ICSLAS that are relevant to concurrency and distribution. Following the Scheme spirit, ICSLAS strives to offer simple and composable basic features rather than sophisticated inflexible features.

Assignment is atomic i.e., it writes the value to be assigned in the appropriate location and returns the former content of this location without interruption. More generally, data structures modifiers behave similarly. This exchange effect allows to write the P and V operations on semaphores, see [QD93].

Threads birth and death is controlled by the `breed` function. It takes thunks as arguments, spawns as many new threads as there are thunks and kills the current thread. The new threads are independent, share the dynamic context of the `breed`-ing thread, and will die upon completion (except if they escape using continuations as can be seen in the example below).

[1]Unix is a registered trademark of UNIX Systems Laboratories, Inc. in the USA and other countries.

The `remote` function migrates its arguments towards a site and remotely applies the first of them to the others. It looks like an annotation that indicates the relevance of a remote procedure application.

Let us give an example of an ICSLAS program. The following function returns to its caller all the integers from an interval. In fact, it creates as many threads as there are integers in the interval; all these threads independently return an integer to the caller of `//iota`. This is an example of a "generator" using multiple returns to a single continuation. Moreover the enumeration itself migrates from site to site.

```
(define (//iota start end)
  (call/ep                              ;reify caller's continuation
   (lambda (return)
     (define (enum start end)  ;enumerate
       (if (< start end)
           (breed                              ;fork
            (lambda () (return start))        ;escape
            (lambda ()                         ;migrate
              (remote enum (+ start 1) end) ) )
           (breed) ) )                         ;suicide
     (remote enum start end) ) ) )
```

The most original feature of ICSLAS is the concept of groups of threads; groups originate from the sponsor model of [Osb90] and the heavyweight future of [GGS89]. A group of threads is a first-class object controlling the evaluation of a form. A group is constructed by the `sponsor/de` primitive, standing for *call with dynamic extent*:

```
(sponsor/de (lambda (group) form)
            (lambda (group) finalization) )
```

Upon invocation, `sponsor/de` builds a group: all the threads that will be created for the evaluation of *form* will belong to this group (independently of the sites on which they run). A thread can determine whether it belongs to some group using the `(within/de? group)` predicate. Whenever *form* yields a value, this value becomes a value yielded by the original `sponsor/de` form. There can be more than one yielded value depending on the number of returning threads that are created during the evaluation of *form*.

Two imperative functions exist to pause or awake a group of threads: `(pause! group)` and `(awake! group)`. When a group is paused, all the threads belonging to it are suspended. They can be resumed with `awake!`. When a group is paused and the GC finds it to be unreachable then the group is finalized. When the GC discovers that a group contains no more threads, the group is also finalized. In those two cases, the second argument of `sponsor/de` is triggered (only once) to finalize the group. To give a second argument to `sponsor/de` provides a built-in distributed termination detection [TM93] making easier to program a wide variety of algorithms.

Numerous examples of the ICSLAS language appear in [QD93] and among them: `pcall`, `qlambda`, `either`, `future`, and even `call/cc` but with a precise behavior w.r.t. groups.

2 Implementation

This Section presents the techniques we used to build a distributed interpreter for ICSLAS. First we present the rules of our implementation language then we expose the path we follow to obtain the interpreter.

2.1 Implementation language

In order to cope with heterogeneous sites, we decided to write an interpreter but, to provide security as well as efficiency, programs to be evaluated are compiled into a form that allows fast interpretation. Compiling towards byte-codes was a possibility but it makes debugging, and particularly stepping, difficult. We therefore adopted another form inspired from the encoding of programs in the Scheme-78 chip [SS80]: a program, an Sexpression, is transformed into a tree (with similar shape) of objects whose class represents their syntactical nature, see figure 1 to get a flavor of that transformation. This is a "real" compilation, even if we name it "objectification", where static properties like lexical offsets computation, arity check of invocation to predefined primitives, removal of duplicated type checks ... can be performed. Taking into consideration the goals of ICSLAS, the choice of an interpreting technology is not restrictive since communication delays are dominant.

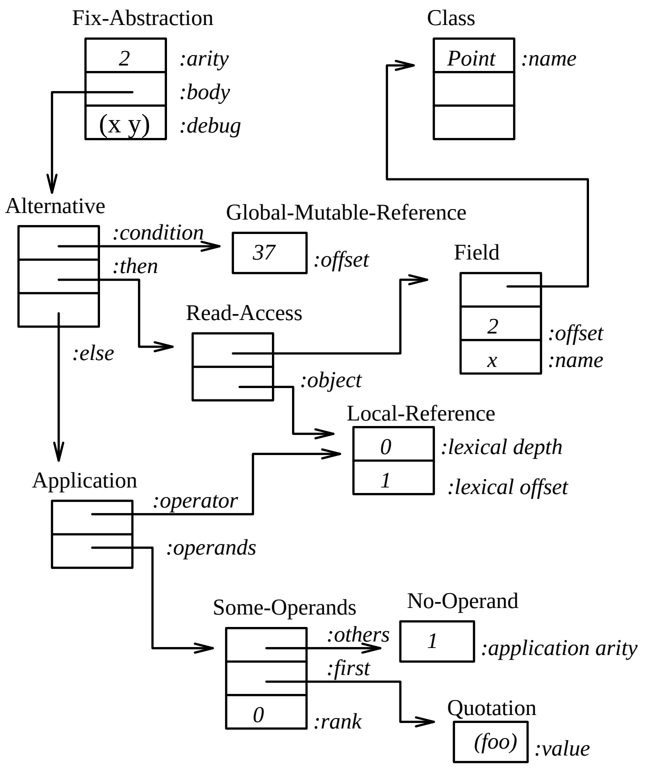

Figure 1: Objectification of `(lambda (x y) (if g (Pointx̄ y) (y '(foo))))`

We decided to implement our interpreter using objects and generic functions. This decision is not contrary to our previous choices, it only reflects the interest of object technology for implementation. Moreover such a OO basis opens wide possibilities for reflection and reification that we will not investigate in this paper.

To implement preemptive concurrency at ICSLAS level, a scheduler must be provided that periodically switches from thread to thread. Since we adopted an interpreter technology, this amounts to implement a non-preemptive scheduler within the interpreter, as in [SS75, Wan80]. To be able to suspend threads, the interpreter maintains reified continuations as done in SML-NJ [App92] but will use the GC technique of [Mat92]. Continuations are reified into linked lists of control frames, this not only facilitates the design of the scheduler but also allows to migrate continuations like any other data objects. To offer the scheduler opportunities to switch between threads, the evaluation is split into myriads of short uninterruptible evaluation steps.

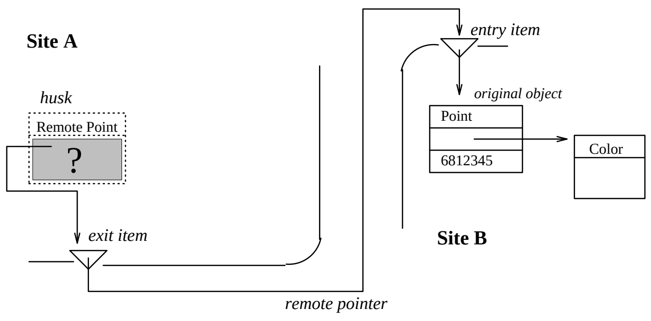

Figure 2: Representation of a distal object

Distribution brings another constraint: an evaluation step must not block because of the unavailability of a distal object. Distal objects, also known as "proxies" [Piq90, Ach93], are represented, see figure 2, by a "husk" having the type and the size of the original object it stands for, as well as a temporary pointer to an "exit item" (terminology of [LQP92]) containing all the necessary informations to access the real object. Whenever needed on a site, an immutable remote object is copied over its husk and its offspring is brought as new husks, see figure 3. Husks are recognized from real objects by their instantiation link. When a husk is overwritten with the content of the object it stands for, its instantiation link becomes normal: this implementation trick (a simple sign change) is indicated on the figures by the `Remote` prefix. To avoid overflowing a site with multiple copies of a same distal object, proxies are hashed to ensure sharing.

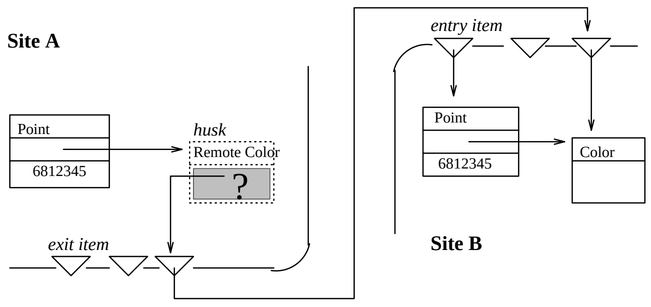

Figure 3: Making local a distal object

There is actually a single class of mutable objects, called `Box`, that is used to implement mutable variables and mutable object slots. Great care is taken in ICSLAS to make the user conscious of the (im)mutability [Que93b] of the manipulated data. Maintaining coherency among the replicated copies of mutable objects is dealt with in section 3.

We finally adopted the grammar of figure 4 for our implementation language. This grammar ensures that evaluation steps are "atomic" i.e., can always be run in a bounded and predictable time and, only requires local resources. The interpreter is made of classes and methods. In the sequel, we will use our own home-made object system, called MEROON. Classes describe the data structures while methods specify the atomic evaluation steps. The methods that intervene in the definition of the interpreter have a single discriminating variable (no multi-methods) and expect the value of their discriminating variable to be local (not a husk). Observe on the grammar that an expression such as `(car (cdr x))` is forbidden since, the value of `(cdr x)` may be a husk in which case, extracting its `car` cannot be done without delay. We must then split such a computation into two steps which may require in between to migrate some datum. On the other hand, an allocation or a type inquiry (even on a husk) is a local operation.

```
(define-class class super (fields...))
(define-method (generic variables) form)
form ::= (begin local... form)
    | (if local form form)
    | (generic local...)
    | local
local ::= variable  | (quote value) | constant
    | (begin local...)
    | (if local local local)
    | (make-class local...)
    | (class? local)
    | (class-field discriminating-variable [index])
    | (set-class-field! discriminating-variable
                        local [index])
index ::= local
```

Figure 4: Grammar of the implementation language

The interest of generic functions is to concentrate, in their behavior, scheduling and migration i.e., the management of distal objects and time quanta. Therefore a generic function has the following behavior:

1. if the value of the discriminating variable is distal, then send a message to its original site to bring it locally. When this migration is finished, reschedule the generic call in the queue of the local scheduler. In some cases rather than migrating the datum to the computation, we can alternatively migrate the computation to the datum for instance, when returning a value to a continuation.

2. if the time quantum is exhausted, pack the call to the generic function into an object and enqueue it in the queue of the local scheduler. The quantum can be materialized by an integer counting the number of invocations to generic functions or, better, by the exhaustion of the zone where control frames are allocated [Mat92].

3. otherwise apply the appropriate method.

Observe in the grammar that a method may finish by a call to a generic function so the state of a thread can be represented by such an invocation. The queue of the scheduler is made of such reified invocations.

2.2 The kernel

The goal of this Section is to show how to obtain an executable definition of the interpreter with minimal effort i.e., using a systematic method. We do not present (nor build) an entire theory aimed at the writing of a single interpreter, but rather exhibit how to use theories to obtain this interpreter in a comfortable way.

We must reify continuations to be able to migrate them as well as to be able to suspend running threads. Moreover we want to split evaluation into atomic steps. These reasons suggest us to adopt Continuation Passing Style (CPS). CPS is a well known transformation that makes continuations appear but CPS also converts programs into a language with only trivial (in the sense of [Rey72]) forms that are atomic to our sense.

The variant we will use must produce a program respecting the afore-mentioned grammar i.e., it must not convert forms that already respect our grammar. Such a transformation is already discussed in [FWH92, chap 8]. A minor but well known technical problem is that CPS reifies continuation into regular λ-forms which usually must be distinguished from real abstractions, we therefore use Abstract CPS (ACPS) [FWFD88] which puts more emphasis on the construction and use of continuations. The form `(resume` q v`)` will represent the return of the value v to the continuation q, while `(extend` q' `(lambda (`q v`) ...))` will be analogous to pushing a frame onto continuation q'. A frame is represented by a binary abstraction; when triggered, it receives, a value v and all the frames that lay below it in q. The λ-forms created by ACPS only appears in `extend` forms and are thus easy to recognize.

The kernel of ICSLAS is similar to Scheme, they have the same set of special forms. In direct style, a Scheme evaluator is a function that takes an expression and an environment and yields its resulting value. Here is the definition of the alternative, represented by an object of the `Alternative` class, as handled by the, now generic, `evaluate` function:

```
(define-class Alternative Form
   (condition consequent alternative) )
(define-method (evaluate (e Alternative) r)
  (evaluate (if (evaluate (Alternative-condition e) r)
                (Alternative-consequent e)
                (Alternative-alternant e) )
            r ) )
```

This method is ACPS-converted into:

```
(define-method (evaluate q (e Alternative) r)
  (evaluate (extend q
              (lambda (qq bool)
                (evaluate qq
                  (if bool (Alternative-consequent e)
                           (Alternative-alternant e) )
                  r ) ) ) )
            (Alternative-condition e)
            r ) )
```

The abstractions that represent frames do not belong to our target language, so we perform a new transformation analogous to λ-lifting except that we will introduce objects and classes instead of combinators, see figure 5. To push a frame onto a continuation is transformed into an allocation of a specialized frame object containing all the free variables of the abstraction as slots. Since we suppose that all different species of frames inherit from the `Frame` class, the body of the abstraction defines its specialized behavior w.r.t. the `resume` generic function, after suitable substitutions. We name this transformation “OO-lifting” but, despite its sophisticated name, the transformation is similar to what most compilers do when compiling abstractions. Here is the result of the OO-lifting applied on the previous example:

```
(define-method (evaluate q (e Alternative) r)
```

assuming

`(define-class Frame Object (q))`

transform

`(extend` q' `(lambda (`q v`)` π`))`

into

`(make-`X`-Frame` q' ν^*`)`

with

`(define-class` X`-Frame Frame (`ν^*`))`
`(define-method (resume (`q'' X`-Frame)` v`)`
    π`[(Frame-q` q''`)/`q`][(`X`-Frame-`ν q''`)/`ν`]`$_{\forall \nu \in \nu^*}$
where ν^* *are the free variables of* `(lambda (`q v`)` π`))`

Figure 5: OO-lifting

```
  (evaluate (make-Alternative-Frame q e r)
            (Alternative-condition e)
            r ) )
(define-class Alternative-Frame Frame (e r))
(define-method (resume (qq Alternative-Frame) bool)
  (evaluate (Frame-q qq)
            (if bool (Alternative-consequent
                      (Alternative-Frame-e qq) )
                     (Alternative-alternant
                      (Alternative-Frame-e qq) ) )
            (Alternative-Frame-r qq) ) )
```

The result does not belong to our implementation language. It introduced two cascaded accesses, one of which being: `(Alternative-consequent (Alternative-Frame-e qq))`. The trick to avoid such cascades is to systematically precompute the immutable and initialized slots of local objects such as instances of programs. We first rewrite the alternative as:

```
(define-method (evaluate q (e Alternative) r)
  (let ((condition  (Alternative-condition e))
        (consequent (Alternative-consequent e))
        (alternant  (Alternative-alternant e)) )
    (evaluate (extend q
                (lambda (qq bool)
                  (evaluate qq
                   (if bool consequent alternant)
                   r ) ) )
              condition
              r ) ) )
```

Then we OO-lift the resulting expression without any deeper thought. The frame pushed over the continuation is larger but ensures the locality of its behavior when `resumed`.

```
(define-method (evaluate q (e Alternative) r)
  (let ((condition  (Alternative-condition e))
        (consequent (Alternative-consequent e))
        (alternant  (Alternative-alternant e)) )
     (evaluate (make-Alternative-Frame q
                 consequent alternant r )
               condition r ) ) )
(define-class Alternative-Frame Frame
   (consequent alternant r) )
(define-method (resume (qq Alternative-Frame) bool)
  (let ((q    (Frame-q qq))
        (then (Alternative-Frame-consequent qq))
        (else (Alternative-Frame-alternant qq))
        (r    (Alternative-Frame-r qq)) )
    (evaluate q (if bool then else) r) ) )
```

To summarize the rewriting process, we first use ACPS, then open slots of objects then OO-lift the whole. This is a fairly systematic transformation once one gets used to CPS.

The remaining problem of the kernel concerns the application. Usually the arguments of the application are computed by the so-called `evlis` function and gathered in a list. But a list is not a local data structure: it might span several sites thus making difficult to bind values to the variables of the function. We therefore chose to gather the values of the operands of an application into a single, necessarily local object: an activation block. Here is the definition in direct style, it can be straightforwardly rewritten as shown before.

```
(define-method (evaluate (e Application) r)
  (invoke (evaluate (Application-operator e) r)
          (evaluate (Application-operands e) r) ) )
(define-class No-Operand Object (size))
(define-class Some-Operands Object
  (others first rank) )
(define-method (evaluate (e No-operand) r)
  (allocate-Activation-Block
   (No-operands-size e) ) )
(define-method (evaluate (e Some-Operands) r)
  (let* ((first  (Some-Operands-first e))
         (others (Some-Operands-others e))
         (rank   (Some-Operands-rank e))
         (value  (evaluate first r))
         (block  (evaluate others r)) )
   (set-Activation-Block-value! block rank value)
   block ) )
```

The operands of an application form a linked list of objects of class `Some-Operands` terminated by a `No-Operand` object, see figure 1. Values of operands are computed from left to right, the behavior of the terminating `No-Operand` node is to allocate an activation block of the right size (this is precomputed during objectification), then values are stored at their appropriate rank in the activation block which finally is given to the `invoke` generic function which knows how to invoke objects. We will not detail `lambda` in this paper.

2.3 The environment

This Section shows how the specific features of ICSLAS are grafted onto the previously described kernel. The problem is that the kernel is now written in CPS while the additional features will be written (for sake of readability) in direct style but with CPS interfaces i.e., with an initial continuation variable. The problem is how to transform these programs into a form that respects the grammar of our implementation language.

Let us take an example and, for instance, a simplified exception mechanism inspired from EULISP [PNB93]. A value can be signalled as an exception using the `signal` function. A handler can be set by `bind-handler` to catch the exceptions that might be signalled during the evaluation of a thunk. This handler will be invoked on the exception in the dynamic context of `signal` but under the protection of the handler that was present at `bind-handler` time.

The `bind-handler` simply pushes a special frame onto the continuation to record the current handler. When normally `resumed`, this frame just passes the value it receives to the next frame (this might be the inherited default behavior of `Frame`). Observe in the following the mix between direct style and CPS interfaces. We also assume from now on that primitives are defined with `define-primitive` and takes a continuation and a local activation block as arguments.

```
(define-primitive (bind-handler q block)
  (let ((handler (Activation-Block-value block 0))
        (thunk   (Activation-Block-value block 1)) )
```

```
    (invoke (make-Handler-Frame q handler)
            thunk
            (allocate-Activation-Block 0) ) ) )
(define-class Handler-Frame Frame (handler))
(define-method (resume (q Handler-Frame) v)
  (resume (Frame-q q) v) )
```

Signalling an exception requires to find the current handler somewhere in the frames forming the continuation, then to invoke it on the exception in the current dynamic context but for the exception handler which must now be the previous one. This is simply achieved by pushing an extra frame that will skip the found handler if looked for again. Direct style and CPS are again mixed together and possibly non-local evaluations may appear in non-terminal positions: this is, for instance, the case of `(find-handler q)` in the following program fragment, which scans the continuation, maybe on other sites, to find the appropriate handler.

```
(define-primitive (signal q block)
  (let* ((exception (Activation-Block-value block 0))
         (hfq       (find-handler q)) )
    (invoke (make-Skip-Frame q (Frame-q hfq))
            (Handler-Frame-handler hfq)
            (make-Activation-Block exception) ) ) )
(define-class Skip-Frame Frame (follow))
(define-method (find-handler (q Frame))
  (find-handler (Frame-q q)) )
(define-method (find-handler (q Handler-Frame))
  q )
(define-method (find-handler (q Skip-Frame))
  (find-handler (Skip-Frame-follow q)) )
```

These programs represent the intent of the exception system, they precisely are what we want to maintain so we look for a way to convert them into a form suitable for the implementation without headaches.

Part of the solution is to apply the same set of transformations we described above i.e., to perform another CPS transformation on the body of that CPS-interfaced function. This might seem strange since continuations generally are the ultimate goal of CPS and, here, continuations are already reified and apparent. But CPS also splits computations into small atomic chunks so it is the tool we look for. To apply CPS again will make apparent new "infra-continuations" corresponding to internal points where computation may be suspended. The raw result is:

```
(define-primitive (signal q block)
  (let ((excn (Activation-Block-value block 0)))
    (find-handler (make-Find-Handler-Frame q excn) q) ) )
(define-method (find-handler qq (q Frame))
  (find-handler qq (Frame-q q)) )
(define-method (find-handler qq (q Handler-Frame))
  (resume qq q) )
(define-method (find-handler qq (q Skip-Frame))
  (find-handler qq (Skip-Frame-follow q)) )
(define-class Find-Handler-Frame Frame
  (exception) )
(define-method (resume (q Find-Handler-Frame) hfq)
  (let ((q    (Frame-q q))
        (excn (Find-Handler-Frame-exception q)) )
    (invoke (make-Skip-Frame q (Frame-q hfq)) ;problem!
            (Handler-Frame-handler hfq)       ;problem!
            (make-Activation-Block excn) ) ) )
```

Unfortunately this is once again a disappointing result since it does not respect our implementation language due to the presence of the two read accesses to slots of `hfq`: since `hfq` is not the discriminating variable of the generic function `resume`, there is no guarantee that the value of `hfq` is local. A systematic way to correct this behavior exists which is to insert a call to a dummy generic function discriminating on `hfq` and, of course, to apply CPS again to place that generic invocation in tail position.

Another more interesting solution is to adopt the "top frame integration" idea of [Que93a]. The generic function `find-handler` is always invoked on top of a `Find-Handler-Frame` so we can specialize `find-handler` for that configuration of the continuation. This amounts to integrate the behavior of `Find-Handler-Frame` in the method for `Handler-Frame` and to add the slots of `Find-Handler-Frame` as additional operands of `find-handler`. The final result is now simpler even if continuations appear twice in the arguments of `find-handler`:

```
(define-primitive (signal q block)
  (let ((excn (Activation-Block-value block 0)))
    (find-handler q excn q) ) )
(define-method (find-handler qq excn (q Frame))
  (find-handler qq excn (Frame-q q)) )
(define-method (find-handler qq excn (hfq Handler-Frame))
  (invoke (make-Skip-Frame qq (Frame-q hfq))
          (Handler-Frame-handler hfq)
          (make-Activation-Block excn) ) )
(define-method (find-handler qq excn (q Skip-Frame))
  (find-handler qq excn (Skip-Frame-follow q)) )
```

The transformations that were used above are not new nor they are automatic: writing an interpreter is still an art! What we gain is that the interpreter is described in small fragments using the appropriate level of details. Transformations justify the final code as a kind of structured implementation documentation. One can even imagine to formally check this equivalence.

The ICSLAS interpreter has been entirely obtained by these techniques and "top frame integration" was used a number of times.

2.4 Concurrency

This Section describes the explicit and implicit concurrency present in ICSLAS. Explicit concurrency is created with the `breed` function while implicit concurrency is mandated by time-slicing or distal objects management. Generally objects do not move out of the site where they were created except if beneficial: immutable objects are often replicated towards the computations that need them while mutable objects may drift closer to these computations. Continuations are stuck to their birth site to distribute computations. Control over the distribution *à la* Trellis-DOWL [Ach93] will be investigated in the future.

Every site runs a copy of the interpreter, the objectified program naturally flows as needed by the cooperating sites i.e., as required by the `evaluate` generic function. Every site has a running scheduler. The scheduler is assumed to be fair. The `local-enqueue!` implementation primitive adds a thread to the local scheduler, while `remote-enqueue!` does a similar job in the queue of a remote scheduler. The `schedule!` function kills the current thread and triggers the local scheduler. These are the sole internal primitives to manage concurrency.

The `breed` function is now simple to express, it just enqueues new threads created on the received thunks with an appropriate `Suicide-Frame` pushed onto the dynamic context.

```
(define-primitive (breed q block)
  (let ((n  (Activation-Block-value-length block))
        (qq (make-Suicide-Frame (Frame-q q))) )
```

```
    (do ((i 0 (+ i 1)))
        ((= i n))
      (local-enqueue!
       (make-invoke-Call qq
        (Activation-Block-value block i)
        (allocate-Activation-Block 0) ) ) )
    (schedule!) ) )
(define-method (resume (q Suicide-Frame) v)
  (schedule!) )
```

The scheduler maintains a queue of suspended calls. With fairness, it extracts one of its elements and runs it. Generic functions such as invoke, resume ... all have an associated class that allows to reify a call to them. These classes provides an appropriate method for run; for instance, here follows the associated class and method for invoke:

```
(define-class invoke-Call Object
  (q function block) )
(define-method (run (thread invoke-Call))
  (invoke (invoke-Call-q thread)
          (invoke-Call-function thread)
          (invoke-Call-block thread) ) )
```

The remote primitive also introduces concurrency but on a different site. It just packs (reifies) the invocation into an object which is enqueued on another site. The remote-enqueue! function is responsible for the migration machinery. It takes an object, encodes it into a stream of bytes using the XDR protocol [xdr], sends it to the designated site, through batch RPC, where it is decoded and run. The encoding function is a generic function so it can be easily customized. It has to migrate the object itself and part of its offspring. On the other hand it is useless to migrate too many objects, a good balance is necessary not to waste computing power. Experimentations have to be performed to acquire a better understanding of these issues.

3 Shared variables coherency

This Section explains how variables are implemented. In a mostly-functional style, variables are often immutable. Also, it is often the case, as can be seen in the examples of [QD93], that mutable variables are local. Non local mutable shared variables are thus the central problem to solve. Mutable variables are implemented via an indirection through a box, an object containing a single value, see top-left part of figure 6. Mutable slots within objects are similarly handled. Boxes are analogous to the reference of ML and were also used in the Orbit compiler [KKR+86]. Although boxes are used to implement mutable variables, we still prefer assignment over reference since all assignments to local variables are statically known (in fact, a local variable is only known to be mutable if it is the target of a set!): this knowledge may improve compilation.

3.1 Representation

This Section describes how mutable variables are represented and usually managed. Once created on a site, boxes are never copied; if a box has to be migrated then a proxy, an instance of Cached-Box, is remotely created whose rôle is to serve, for read and write accesses, as a relay towards the original box.

As previously shown, for instance in [Piq91], replication of data requires some care. Reference counters on entry items seem the most appropriate method to implement a distributed GC [LQP92] but maintaining distributed counters

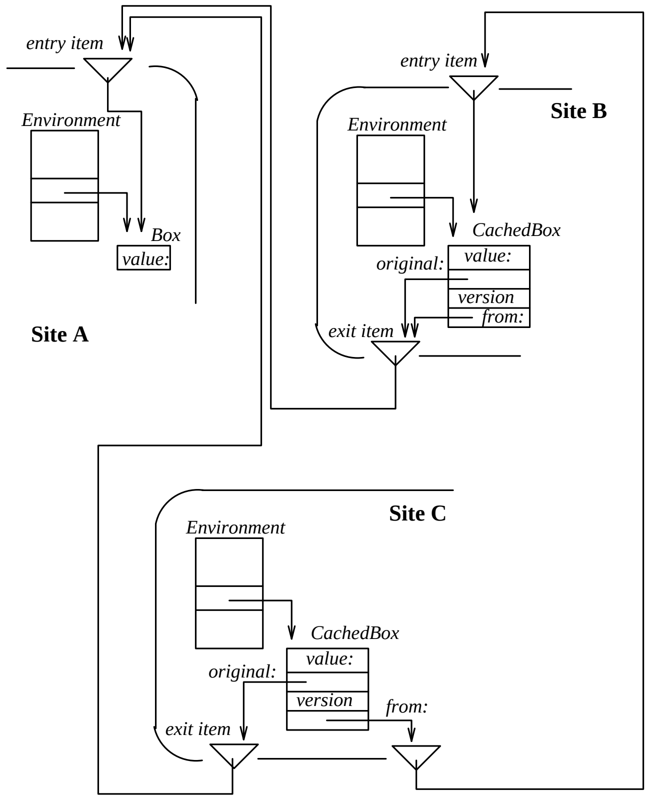

Figure 6: Shared mutable variable

is difficult due to the race condition between the messages that increment or decrement these reference counters. A reference counter may reach zero and leads to the reclamation of the datum it stands for, while increment messages are still in transit towards this same entry item. To avoid this problem, we use the indirect reference counting scheme of [Piq91] where a replicated object maintains two references: the first refers to the original object to relay access while the second refers to the object from which it was copied, see an example in the bottom of figure 6. When a replicated object is reclaimed, it only sends a decrement message to the object it comes from. This technique totally eliminates increment messages (since incrementation is always local) and therefore race condition. More details and related bibliography can be found in [Piq91] but are unessential to what follows.

When a thread on a site wants to read or write a non local box, a message may be sent to the original site where is the original box: this is the naïve implementation but an inefficient one since it requires too many messages. The usual solution is to locally cache values of boxes i.e., to copy boxes into cached boxes, see again figure 6. When a cached box must be read, its cached value is returned if present; otherwise a message is sent to the original box and the fetched value is stored in the cached box to improve future readings. For a write access, the protocol is different: a message is sent to the original box along with the value to be assigned. When this message is received, the original box is overwritten with the new value and the former content is returned back to the continuation. This is a "write-through" cache behavior.

With this protocol, the problem now is to consistently change all the obsolete cached values when the original box is mutated. The usual solution is to atomically broadcast an invalidation message so that all receiving sites can reset their cache. But such an atomic broadcast is an expensive solution that generally does not support non-answering sites. Moreover many protocols require or, the total number of shared variables to be statically known or, the existence of a centralized machine to maintain some matrices (for "who knows what" knowledge) or, the program duration to be bounded so that clocks never overflow. We propose hereafter a new protocol free of these three problems.

3.2 Causality

The basic observation is the following. Due to thread scheduling in Icslas, there is no reason why a mutation should appear to take effect instantaneously and ubiquitously (to anti-paraphrase [HW90, p. 464]). What is essential with mutation is that when a thread modifies a box, this modification must not be ignored by the computations induced by the continuation of this thread. Conversely, unrelated threads do not need to be aware of it immediately but are compelled to perceive it eventually. The basic rule is that *a thread cannot ignore what its invoker knew*! Corollarily, *(i)* since sites run many threads in the same value space, a thread cannot ignore what its site is aware of; *(ii)* whenever a thread is migrated, the receiving site cannot ignore what knew the thread on its emitting site i.e., what knew the emitting site. We may then convert an atomic invalidating broadcast into a lazy propagation of the invalidation. This protocol will be more efficient since it does not require global synchronization, allows invalidation messages to be combined and supports non answering sites for some time.

Consider the following queer Icslas program, where the `schedule!` function is also offered at the user level to allow a thread to voluntarily release control. Forms explicitly using `remote` do not appear since they are intended to be transparent.

```
(let ((x 0)(y 0))
  (breed (lambda ()
           (set! x (+ x 1))
           (display x) )                  ;displays 1 or 2.
         (lambda ()
           (until (> x 0) (schedule!)) ;busy wait
           (set! x (+ x 1))
           (set! y (+ y 1)) )
         (lambda ()
           (until (> y 0)
             (display (list y x))       ;observe
             (schedule!) ) ) ) )
```

The two first spawned threads are synchronized via the two shared variables: `x` and `y`. The third thread observes these shared variables. When scheduled, this thread repeatedly evaluates the `(display (list y x))` form which can only display an arbitrary number (even zero) of times `(0 0)` then `(0 1)` then `(0 2)` and, at most once, `(1 2)` (evaluation is left to right). The point is that the third thread cannot observe the new value of `y` and, in the same time, ignore the mutations of `x` since the mutation of `y` is only triggered by the first mutation on `x` and therefore is causally dependent of this mutation. This is causality and is totally independent of any `remote` function that might be inserted in the previous program. Another point is that the observer will eventually see the mutations since the semantics should not depend on the number of sites and if there were a single site, these mutations will be immediately perceived.

3.3 Essence

This Section exposes the essence of our protocol. Nothing special must be done to guarantee causality on a single site, this is naturally implied by the uniqueness of the memory. For a distributed world, we adopt the vectorized time of [Fid88, Mat88]. Roughly stated: any site has a "proper clock" counting the number of times its local boxes were mutated. A site also has as many other clocks as there are other sites, these clocks record the time of these other sites as far as the site knows them. The complete set of clocks represents the "view of the world" the site has. This view is only guaranteed to be accurate for the proper clock of the site.

When a thread is migrated then the receiving site must have a view of the world that must be compatible with that of the emitting site. A compatible view of the world means that the receiving site cannot ignore a mutation already perceived by the emitting site. Therefore to ensure causality *(i)* any cached box has a `version` slot that dates the cached value, this date is the time of the site of the original box when the value was copied from it; *(ii)* whenever a thread is migrated, the view of the world of the emitting site is also migrated (via `remote-enqueue!`) so the receiving site can invalidate all the cached boxes with out of order versions, see figure 7. The interest of that scheme is twofold: reading may be local using caches, invalidation is lazily propagated and checked when reading.

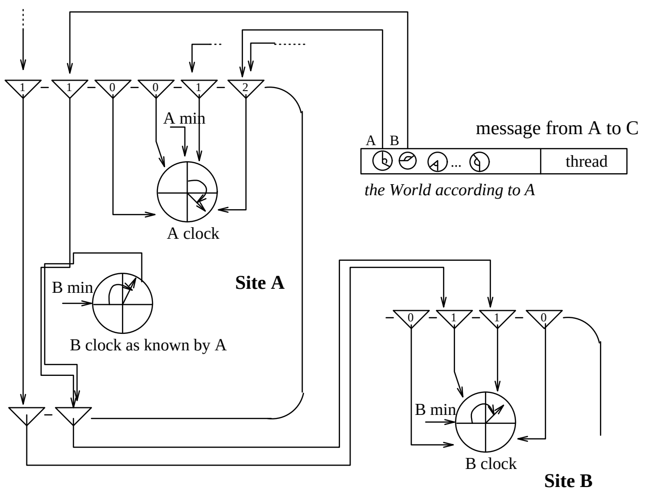

Figure 7: Message shape

When the proper clock of a site is incremented, all the sites that will perceive it (later) will invalidate all the cached values coming from that site. This is rather drastic so we may adopt an intermediate solution with n proper clocks per site. Any remotely accessed box has an associated "entry item" which never changes. We can use its address and the IP number of its site as a basis to compute which one of the n proper clocks is associated to the box. Other policies such as associating clocks on a per-type basis may also be interesting. One can even associate a clock to a single very important box. To use a big n confuses less boxes but augments the size of messages (note that communication delays are quite similar for a few bytes or a single Kbyte).

Clocks are represented by integers. Since we intend to use ICSLAS for wide and long computations (factorizing big numbers often takes months! [Mor91]) we must take care of possible clock overflow. This problem is rarely addressed, except in [LK90]; our solution is based on the use of the underlying distributed GC. A cached value is invalidated as soon as its version differs from the last known value of the clock of its original site. Therefore what we must ensure is: when a clock is incremented, it must not use a number again if there are, somewhere, some cached boxes holding this number in their `version` slot. In other words, we are free to reuse clock numbers if they are unused: this is clearly a typical GC problem.

We therefore decide to split the clock into s different sectors and to associate objects to these sectors. Since these objects must be remotely accessible, associated entry items are created for them and will never be reclaimed. When a message is sent, we not only send the values of the clocks a site is aware of, but we also send the references to the associated sector objects containing these values. On figure 7, the message emitted by site A transmits the value of the proper clock of site A and a reference to the currently used sector of clock A. The message also transmits the value of clock B as known by site A: it passes the reference onto the appropriate sector of the proper clock of B, this reference is already held by site A. Note, on the left of figure 7, a passing-through reference to an older sector of clock B presumably passed earlier and coming from a site which has a not up-to-date view on B.

When incrementing a proper clock and when the current sector is exhausted, to know if the numbers of the next sector are unused, it is sufficient to check whether the reference counter of its associated entry item is zero. This GC-aided technique allows each site to maintain a minimal index to the lowest but still used sector of its proper clock, thus all used numbers for any clock are bound by the minimal and actual value of this clock. This minimal value allows to compare clock values and select the highest one: w.r.t. some minimal value min, n_1 is lower than n_2 if n_1 lies between min and n_2 clockwise. Since all incoming messages on a site impose to reset the local view of the world to the highest possible values combining the local view with the view conveyed in the message, messages not only contain values of clocks but also their minimal values (see again figure 7) i.e., rather than sending the current value of clocks, we transmit the interval of used values.

When all sites are normally running, each clock is used for only a few sectors provided regular communications allow sites to propagate their clocks. When a site does not answer, it keeps references towards sectors of any clock and thus will prevent (in some future) the normal incrementation of these clocks. We basically have two solutions. We can memorize all the messages that the non answering site does not receive so it can resynchronize itself when reconnecting. This is possible if the memorization of these messages is not too expensive in space and if clocks still have a large number of future available ticks before being blocked.

The other solution is to identify and to exclude these faulty sites in order to allow the running ones to continue their job. We will not address the problem of lost data and leave it for the application which has to duplicate computations on different sites to support failure. To exclude some sites can be done according to [LQP92]. Reference counters can be used with Christopher's algorithm [Chr84] to determine which references come from faulty sites. These references can be suppressed by running sites thus freeing clocks; outgoing references towards these sites must also be set in a state where access through them will signal errors. No message coming from (resp. directed to) these excluded sites must be accepted (resp. sent) before their value spaces are correctly reset: they must not use the old annihilated references nor the running sites may expect values that were referenced to be in a coherent state. Excluding sites is a difficult synchronization problem.

3.4 Protocol

This Section details the protocol. Let us assume that the local site can be known using the `(current-site)` form. A site is a compound object with at least two slots containing its index (a number identifying it) and, its view of the world i.e., an indexed sequence of clocks. Clocks are compound objects too, they have a current value (regularly incremented by local mutations) and a minimal value (referring to the last sector object still in use from elsewhere). When the clock is not proper, it also refers to a sector-object associated to the value the site believes the original clock to be.

```
(define-class Site Object (index clock ...))
(define-class Clock Object (value min))
(define-class NonLocalClock Clock (sector))
```

3.4.1 Writing

The `box-set!` function distinguishes two cases whether the box is local or remote. If the box is remote, we just send a message delegating the modification of the box. It is not necessary to invalidate the possibly locally cached value nor to update it, this will be done automatically when the former value of the location comes back towards the continuation. If the box is local and remotely referenced then its associated proper clock must be incremented. An improvement is to increment the clock only if the content of the box seems to change (equality is discussed in [QD93]). The predicate `seems-different?` is an atomically computable (to avoid remote or lengthy computations) approximation of inequality, it only has to err on the safe side: it may always return true!

```
(define-method (box-set! q (box Box) new)
  (let ((old (Box-content box)))
    (when (seems-different? old new)
      (increment-clock!
       (Site-clock (current-site)
        (Site-index (current-site)) ) ) )
    (set-Box-content! box new)
    (resume q old) ) )
(define-method (box-set! q (box Cached-Box) new)
  (let* ((exit-item (Cached-Box-original box))
         (site      (Exit-Item-site exit-item)) )
    (remote-enqueue! site
     (make-box-set!-Call q box new) )
    (schedule!) ) )
```

To increment a proper clock is done as follows. If the current sector is not exhausted, then the current value of the clock is incremented by one. If the sector is exhausted then the next sector in clockwise order is checked to be unused i.e., to have a null reference counter in the associated entry item. If it is the case, this sector can be used and the current value of the clock is set to the first value of that sector. If the sector is still in use then the incrementation cannot be performed and a "panic mode" as in [Hug85] has to be entered. The first action is to force every answering site to be aware of the exact state of the current site then to let the distributed GC reacts so it can raise the minimal value of clocks i.e., discovers that some sectors are now unused. If the next sector is still in use then there are some non answering sites that must be excluded by all running sites in a coordinated way. The references coming from these faulty sites will be identified by Christopher's technique [Chr84] and annihilated. When the panic mode ends, the next sector is free again and the incrementation can proceed. This drastic cut in processing power as well as this data loss will probably have a deep effect on the application.

3.4.2 Reading

To read a box is simple if the box is local or has a locally up-to-date cached value: its content is returned. That a box is up-to-date is easily checked: we just compare its `version` content with the latest value obtained from the clock of its original site. If they are equal then no new mutation was perceived from that site so the cached value is correct. Since clocks never reuse already used numbers, a simple integer comparison is sufficient. Otherwise a remote read access is emitted with a continuation that will try to cache the resulting read value. Some care must be taken since when a value comes back, the site may well be ahead of time with the cache filled with a more up-to-date value. The cache is written only if the value is up-to-date that is why `box-fetch` returns the value of the box and the time that was current when read. To fill the cache is simply done by pushing an appropriate frame onto the continuation and sending a regular `remote-enqueue!` message with that modified continuation.

While a `box-fetch` call is remotely running, other threads may try to read the same non local box. They may be blocked waiting for the box to be filled (this spares bandwidth) or, they may ignore each other and thus detect more recent mutations. We only present this latter variant here:

```
(define-method (box-ref q (box Box))
  (resume q (Box-content box)) )
(define-method (box-fetch q (box Box))
  (resume q (cons (Box-content box)
             (Clock-value
              (Site-clock (current-site)
               (Site-index (current-site)) ) ) )) )
(define-class Cached-Box Box
  (original version from) )
(define-method (box-ref q (box Cached-Box))
  (let* ((exit-item (Cached-Box-original box))
         (site      (Exit-Item-site exit-item))
         (clock     (Site-clock (current-site)
                     (Site-index site) )) )
    (if (= (Clock-value clock)
           (Cached-Box-version box) )
        (resume q (Cached-Box-content box))
        (begin (remote-enqueue! site
                (make-box-fetch-Call
                 (make-Cache-Frame q box)
                 box ) )
               (schedule!) ) ) ) )
(define-class Cache-Frame Frame (box))
(define-method (resume (q Cache-Frame) value+time)
  (let* ((value      (car value+time))
         (time       (cdr value+time))
         (box        (Cache-Frame-box q))
         (exit-item  (Cached-Box-original box))
         (site       (Exit-Item-site exit-item))
         (clock      (Site-clock (current-site)
                      (Site-index site) )) )
    (when (= time (Clock-value clock))
      (set-Cached-Box-content! box value)
      (set-Cached-Box-version! box time) )
    (resume (Cache-Frame-q q) value) ) )
```

3.4.3 Updating clocks

The `remote-enqueue!` primitive is responsible for the management, on the receiving site, of the compatibility between the views of the world of the two communicating sites. When `remote-enqueue!` migrates a thread, it also sends the view of the world of the sending site. We assume `send` to perform the encoding of the values to be sent and do not detail it further (it may also send additional informations such as the identity of the emitting site for security reason, piggyback the actual load average for load balancing etc.)

```
(define (remote-enqueue! site thread)
  (send site (Site-clocks (current-site)) thread) )
```

Upon reception, the incoming view serves to update the local view. For any clocks, its current value is compared to the corresponding incoming value and if the latter is more recent, the receiving clock is updated with it. We assume the `time<?` predicate to clockwise compare its second and third argument w.r.t. the first one. Since a proper clock is always accurate, it cannot be reset from outside.

```
(define (update-clock! local remote)
  (when (time<? (Clock-min local)
                (Clock-value local)
                (Clock-value remote) )
    (set-Clock-value! local (Clock-value remote))
    (set-Clock-sector! local (Clock-sector remote))
    (set-Clock-min! local (Clock-min remote)) ) )
```

The migrated thread is simply enqueued in the local scheduler. In the case of `Cache-Frame` resumptions, it may also be immediately run to take benefit earlier of cache filling.

```
(define (receive view thread)
  (for-each update-clock! (Site-clocks (current-site))
                          view )
  (local-enqueue! thread) )
```

In some sense, this new protocol trades an atomic broadcast for a lazy propagation of cache invalidations. Due to its laziness, it easily supports short communication breakdowns; sites will be resynchronized at their next successful communication (provided no messages are lost). Our protocol is sound i.e., ensures the correct simulation of a distributed shared memory, but it does not enforce liveness. For instance, in the above queer example, it does not ensure that the third thread will eventually displays anything different from `(0 0)`. To force liveness simply requires that, periodically, sites communicate, for instance to exchange their load average or, as described with panic mode, to update the minimal value of clocks. Note that this minimal value corresponds to a lower approximation of the Global Virtual Time (GVT) [Jef85] of the application.

4 Groups

This Section only sketches some of the main points of the implementation of groups of threads, additional details can be found in [Que94]. The status of a group whether it is awaked or paused is recorded as a boolean in a mutable box.

Sites maintain the tree structure of the groups they know so they are able to atomically (i.e., locally) check whether a thread belongs to a group or not. When a message migrates a thread, it also conveys the most specific group to which this thread belongs. When an unknown group is received, sites delay all operations on its associated threads until the full ancestry of that group is locally known. Threads refer to their most specific group (groups do not refer to their associated threads since it would be too difficult to maintain such a data structure and since it would create cycles slowing down the GC and endangering termination detection). Groups refer to their super-groups i.e., the group in which they were created. They also hold the last date when the status of their immediate supergroup was surely known. This is necessary to sequentialize not causally-related concurrent `pause!` or `awake!` operations on a group and one of its (direct or indirect) subgroups. The running status of a group is the content of the (cached) box provided (this cached value is valid and) no mutation was perceived on the running status of any of its supergroups.

5 Related Work

Concurrency was already present in the first implementation of Scheme [SS75] ever since multiple implementations where studied and among them [KHM89, Hal89, GGS89, JP92]. They all offer exchange instructions, `breed` equivalent and some of them made steps towards the group concept. We differ from these on three points: threads are not first-class, computations are assimilated to groups and, we are not interested in speed-up for shared memory computers. We rather tried to address the problems of "programming in the (very) large". We also studied for a long time, the relationship between continuations and threads and presented what we think is a good combination even if it reintroduces the concept of dynamic extent.

Since [SS75] we were not aware of any published implementation of a concurrent interpreter. We hope to have partially filled this gap. The implementation uses concepts borrowed from modern object technology [BDG+88]: combined with CPS, generic functions allow to confine threads and distal objects management, freeing the body of methods from these details. OO-lifting is new (in its form not in its effect). The implementation is written by fragments with the appropriate level of details. These fragments may be written in direct style or in CPS or in direct style with CPS-interfaces. We then convert these fragments towards our implementation language and sometimes require to use CPS twice. The transformations are rather mechanical, they constitute by themselves an interesting documentation. The use of CPS to finally ensure the locality of computations is a new interesting facet of this program transformation that is heavily used by the implementation.

Many aspects of distribution are supported by the underlying distributed GC. This is the case of termination detection, inspired from [TM93], and also of the management of clocks sectors. Details on this distributed GC appear in [LQP92]. The coherency protocol does not require atomic broadcast which is more appropriate for multi-processors caches [Bro90, BKT92]. It improves upon [MSRN92] since it does not require a centralized shared memory manager and only uses a small fixed number of clocks per site independently of the number of mutable variables.

6 Conclusions

This paper presented two main results:

1. a software architecture for the implementation of a concurrent and distributed interpreter.
2. a new coherency protocol for shared variables with interesting properties.

We hope that these explanations will foster the realization of new implementations of these kind of dialects.

Bibliography

[Ach93] Bruno Achauer. Implementation of distributed trellis. In Oscar M Nierstrasz, editor, *ECOOP '93 — 7th European Conference on Object-Oriented Programming*, volume Lecture Notes in Computer Science 707, pages 103–117, Kaiserslautern (Germany), July 1993. Springer-Verlag.

[App92] Andrew Appel. *Compiling with continuations.* Cambridge Press, 1992.

[BDG+88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. *SIGPLAN Notices*, 23:special issue, September 1988.

[BKT92] H E Bal, M F Kaashoek, and A S Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[Bro90] G Brown. Asynchronous multicaches. *Distributed Computing*, 4:31–36, 11990.

[Chr84] Thomas W Christopher. Reference count garbage collection. *Software–Practice and Experience*, 14(6):503–507, June 1984.

[Fid88] J. Fidge. Timestamps in message passing systems that preserve the partial ordering. In *Proc. 11th. Australian Computer Science Conference*, pages 55–66, 1988.

[FWFD88] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce Duba. Abstract continuations: a mathematical semantics for handling functional jumps. In *Proceedings of the 1988 ACM Symposium on LISP and Functional Programming*, Salt Lake City, Utah., July 1988.

[FWH92] Daniel P Friedman, Mitchell Wand, and Christopher Haynes. *Essentials of Programming Languages.* MIT Press, Cambridge MA and McGraw-Hill, 1992.

[GGS89] Ron Goldman, Richard P. Gabriel, and Carol Sexton. Qlisp: An interim report. In Takayasu Ito and Robert H Halstead Jr., editors, *Parallel Lisp: Languages and Systems, US/Japan Workshop*, volume Lecture Notes in Computer Science 441, Sendai (Japan), June 1989. Springer-Verlag.

[Hal89] Robert H. Halstead, Jr. New ideas in parallel lisp: Language design, implementation, and programming tools. In Robert H Halstead, Jr. and Takayasu Ito, editors, *US-Japan Workshop on Parallel Lisp*, volume Lecture Notes in Computer Science 441, Sendai (Japan), June 1989. Springer-Verlag.

[Hug85] John Hughes. A distributed garbage collection. In *Functional Programming and Computer Architecture*, pages 256–272. Lecture Notes in Computer Science 201, Springer-Verlag, September 1985.

[HW90] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transaction on Programming Languages and Systems*, 12(3):463–492, July 1990.

[IM89] Takayasu Ito and Manabu Matsui. A parallel lisp language PaiLisp and its kernel specification. In Takayasu Ito and Robert H Halstead, Jr., editors, *Proceedings of the US/Japan Workshop on Parallel Lisp*, volume Lecture Notes in Computer Science 441, pages 58–100, Sendai (Japan), June 1989. Springer-Verlag.

[Jef85] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[JP92] Suresh Jagannathan and Jim Philbin. A foundation for an efficient multi-threaded scheme system. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 345–357, San Francisco, USA, June 1992.

[KHM89] David A. Kranz, Robert H. Halstead, and Eric Mohr. Mul-T: A high-performance parallel Lisp. In *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, pages 81–90, Portland, Oregon, June 1989. ACM Press. Published as *SIGPLAN Notices 24*(7), July 1989.

[KKR$^+$86] David Kranz, Richard Kelsey, Jonathan A. Rees, Paul Hudak, James Philbin, and Norman I. Adams. Orbit: an optimizing compiler for scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233. ACM, June 1986.

[LK90] William S Lloyd and Phil Kearns. Bounding sequence numbers in distributed systems: A general approach. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 312–319, Paris (France), May-June 1990.

[LQP92] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *POPL '92 – Nineteenth Annual ACM symposium on Principles of Programming Languages*, pages 39–50, Albuquerque (New Mexico, USA), January 1992.

[Mat88] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1988.

[Mat92] Luis Mateu. Efficient implementation of coroutines. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 230–247, Saint-Malo (France), September 1992. Springer-Verlag.

[Mor91] François Morain. Distributed primality proving and the primality of $(2^{3539}+1)/3$. In I. B. Damgård, editor, *Advances in Cryptology – EUROCRYPT '90*, volume 473 of *Lecture Notes in Computer Science*, pages 110–123. Springer–Verlag, 1991. Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques, Aarhus, Denmark, May 21–24, 1990.

[MSRN92] Masaaki Mizuno, Gurdip Singh, Michel Raynal, and Mitchell L Neilsen. Communication efficient distributed shared memories. Research Report 1817, INRIA, December 1992.

[Osb90] Randy B. Osborne. Speculative computation in MultiLisp, an overview. In *LFP '90 – ACM Symposium on Lisp and Functional Programming*, pages 198–208, Nice (France), 1990.

[Piq90] José Piquer. Sharing data structures in a distributed lisp. In *High Performance and Parallel Computing in Lisp*, Twickenham, London (UK), November 1990. a EUROPAL workshop.

[Piq91] José Miguel Piquer. Indirect reference counting: A distributed garbage collection algorithm. In *PARLE '91 – Parallel Architectures and Languages Europe*, pages 150–165. Lecture Notes in Computer Science 505, Springer-Verlag, June 1991.

[PNB93] Padget, J.A., Nuyens, G., and Bretthauer, H. An overview of EuLisp. *Lisp and Symbolic Computation*, 6(1/2):9–98, 1993.

[QD93] Christian Queinnec and David De Roure. Design of a concurrent and distributed language. In Robert H Halstead Jr and Takayasu Ito, editors, *Parallel Symbolic Computing: Languages, Systems, and Applications, (US/Japan Workshop Proceedings)*, volume Lecture Notes in Computer Science 748, pages 234–259, Boston (Massachussetts USA), October 1993.

[Que90] Christian Queinnec. PolyScheme : A Semantics for a Concurrent Scheme. In *Workshop on High Performance and Parallel Computing in Lisp*, Twickenham (UK), November 1990. European Conference on Lisp and its Practical Applications.

[Que92] Christian Queinnec. A concurrent and distributed extension to scheme. In D. Etiemble and J-C. Syre, editors, *PARLE '92 – Parallel Architectures and Languages Europe*, pages 431–446, Paris (France), June 1992. Lecture Notes in Computer Science 605, Springer-Verlag.

[Que93a] Christian Queinnec. Continuation conscious compilation. *Lisp Pointers*, 6(1):2–14, January 1993.

[Que93b] Christian Queinnec. Designing MEROON v3. In Christian Rathke, Jürgen Kopp, Hubertus Hohl, and Harry Bretthauer, editors, *Object-Oriented Programming in Lisp: Languages and Applications. A Report on the ECOOP'93 Workshop*, number 788, Sankt Augustin (Germany), September 1993.

[Que94] Christian Queinnec. Sharing mutable objects and controlling groups of tasks in a concurrent and distributed language. submitted, 1994.

[Rey72] John Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, pages 717–740. ACM, 1972.

[SS75] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: an interpreter for extended lambda calculus. MIT AI Memo 349, Massachusetts Institute of Technology, Cambridge, Mass., December 1975.

[SS80] Guy L. Steele, Jr. and Gerald Jay Sussman. Design of a lisp-based processor. *CACM*, 23(11):628–645, November 1980.

[Ste90] Guy L. Steele, Jr. *Common Lisp, the Language*. Digital Press, Burlington MA (USA), 2nd edition edition, 1990.

[TM93] Gerard Tel and Friedmann Mattern. The derivation of distributed termination detection algorithms from garbage collection schemes. *ACM Transaction on Programming Languages and Systems*, 15(1):1–35, January 1993.

[Wan80] Mitchell Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 Lisp Conference*, pages 19–28. The Lisp Conference, 1980.

[xdr] Rfc 1014: external data representation standard: Protocol specification. Technical report, ARPA Network Information Center.

`Revision: 1.18`