Sharing Mutable Objects and Controlling Groups of Tasks in a Concurrent and Distributed Language

Christian Queinnec*
École Polytechnique & INRIA-Rocquencourt

Abstract. This paper presents: *(i)* an operational semantics, based on a functional framework, for a concurrent and distributed extension of the Scheme programming language, *(ii)* a coherency protocol taking care of shared mutable objects, *(iii)* a new coherency protocol to imperatively control hierarchical groups of cooperating tasks. These two protocols do not require broadcast, nor FIFO communications, nor a centralized machine; they allow to manage an unbound number of shared mutable values and groups of tasks. This paper also advocates for the use of a functional continuation-based presentation for these protocols.

The omnipresence of interconnected networks exacerbates the need for high level languages allowing to express and control widely distributed computations. Much network services such as `news`, `finger`, `archie`, `netfind` etc. [ODL93] basically manage a set of informations which is accessed and enriched on a world-distributed basis. We think that it will become more and more necessary for these services as well as future ones to offer causal coherency. In the case of the `news` system for instance, to receive answers before questions would not be possible.

We have been designing an extension of the Scheme programming language, a Lisp dialect called ICSLAS, allowing to write such applications. We do not expect whole applications to be written in our language but rather wish to offer the ideal glue to tie together sequential sub-applications. It will act as a kind of "shell" offering high level constructs and built-in causal coherency. Therefore the ICSLAS language has not been designed for speed-up but for expressiveness.

Being a member of the Lisp family of dialects, the ICSLAS language is highly dynamic and supported by a garbage collector (GC). This distributed GC is described in [LQP92]. Causal coherency is usually studied independently of any language and often limits the number of shared mutable values or requires broadcast (or multicast) to be achieved. Since ICSLAS allows to create at run-time as many shared mutable values as wanted (within memory constraints) and since communication delays through networks as well as computer failures make broadcast difficult, our coherency protocols circumvent these two problems.

It is often the case that one wants to control the progress of distributed computations and possibly suspends or resumes them. The ICSLAS language does not

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

offer first-class tasks (since they pose semantical problems with respect to first-class continuations) nor Unix[2]-based `signal` machinery but rather offers the concept of groups of tasks. A program (or more generally an expression) can be sponsored by a first-class *group* object which allows to control this evaluation as a whole. Groups can be dynamically created and may be hierarchically embedded. To suspend a group implicitly suspends all its subgroups which can be resumed independently. We propose a coherency protocol to achieve the distributed management of these groups.

Section 1 informally presents the essential features of the Icslas language, already described in more details in [Que92, QD93, Que94]. The basic operational machinery we use to define the language appears in Section 2. We also exercise it to define the functional kernel and the concurrent features of the language. Distribution aspects as well as the protocol to manage shared mutable objects are dealt with in Section 3. Controlling groups of task is handled in Section 4. Related works is considered at the end of each of these sections.

1 An essential concurrent and distributed language

This section informally describes the main features of a concurrent and distributed toy functional language. This language (see figure 1) is based on Scheme [CR91] and is an essential version of the language described with examples in [Que92, QD93, Que94].

ν	(quote ε)	(lambda (ν...) π)	(π π...)
(if π_1 π_2 π_3)	(begin π_1 π_2)	(new-box π)	(box-ref π)
(box-set! π_1 π_2)	(fork π_1 π_2)	(suicide)	(remote π_1 π_2)
(sponsor π)	(pause! π)	(awake! π)	

Fig. 1. Syntax of the essential language

The value of a variable is obtained via its name (ν). Constants values (ε) can be cited with `quote`. Closures are created by abstractions, identified by `lambda`, and can be applied with combinations (π π...). These previous features form the kernel of the language giving it the power of a sugared applied λ-calculus. Alternatives and sequences are identified with `if` or `begin` keywords.

Variables are immutable: there is no assignment. Side-effects can nevertheless be performed through boxes that are similar to ML `ref`erences. Three predefined functions exist to create boxes with an initial content (`new-box`), to read the content of a box (`box-ref`) or to alter the content of a box (`box-set!`). This last function is an atomic exchange i.e., it atomically writes a value in a box while returning as result the former value the box was holding. Mutable variables and other mutable values can easily be simulated by boxes [KKR$^+$86].

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

The (**fork** π_1 π_2) special form kills the current task while creating two new independent tasks, one evaluating π_1 and another one evaluating π_2. If π_1 yields a value, this value is also yielded by the initial **fork** form, and similarly for π_2. Technically speaking, π_1 and π_2 have the same continuation: the very continuation of the **fork** form. Thus, our model allows a form to return multiply as for the generators of Icon-like languages [MH90]. The **fork** special form is the only form that creates tasks. Conversely a task can explicitly terminates itself with **suicide**. An important characteristic of our language is that tasks are not first-class values, they have no identity therefore they cannot be sent signals *à la* UNIX.

Distribution is introduced through the (**remote** π_1 π_2) special form. The first term π_1 is evaluated and must return a *site*. The second term π_2 is then evaluated on that site. A site is a first-class value that represents an independent machine or, an independent process on the same machine with a separate data space. From an implementational point of view, sites can only exchange messages. If π_2 yields a value, this value is also yielded by the original **remote** form on the very site. Our essential language supports the distributed shared memory (DSM) model i.e., **remote** forms are transparent wrt sites.

Tasks can be controlled by the `sponsor` function. When invoked, as in (`sponsor` φ), `sponsor` creates a new object called a *group* and applies its argument (the unary function φ) on it. The body of function φ may perform arbitrary computations which can involve a number of tasks on a number of sites. This computation can be controlled as a whole through the *group* object (received by φ as argument) and two new imperative functions. The `pause!` function when applied on a group, suspends (all the tasks cooperating to perform) the associated computation while the **awake!** function resumes it. When a task makes the `sponsor` call to yield a value, this task exits from the associated group of tasks: it is no longer a member of that group. Technically, tasks belong to a group while they are in the dynamic extent of the `sponsor` function that created that group.

```
(lambda (thunk)
  ((lambda (barrier)
     ((lambda (group+result)
        (if (box-set! barrier #t)
            (suicide)
            (begin (pause! (car group+result))
                   (cdr group+result) ) ) )
      (sponsor (lambda (g)
                 (cons g (fork (remote site1 (thunk))
                               (remote site2 (thunk)) ) ) )) ) )
   (new-box #f) ) )
```

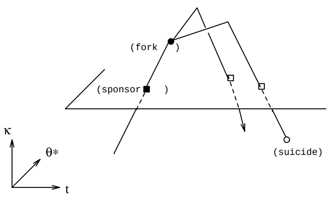

Fig. 2. A program example

The program of figure 2 illustrates nearly all the features of our essential language. We assume the usual `car`, `cdr` and `cons` functions to be predefined and `cons` to only create immutable pairs. The shown function receives a computation encoded as a thunk (a closure based on an abstraction without variables), it then creates a mutable box to hold a boolean flag (initially false `#f`), binds it to the `barrier` (immutable) variable, and starts a `sponsor`-ed computation concurrently invoking the received thunk on two different sites (held in variables `site1` and `site2`). As soon as one of these thunks yields a value, this value is paired with its sponsoring group object and is bound to the `group+result` variable. The boolean held in the `barrier` box is then changed to true (`#t`). The mutation operator `box-set!` returns the former content of the `barrier` box so two cases are possible. If the former content was true then the task commits suicide since it is not the first one that mutates the `barrier` box. Conversely, if the former content was false then the group is paused i.e., any task on any site which is still running under the sponsorship of that group is suspended; the result is then returned as the value of the function.

The right part of figure 2 may help to visualize the main steps. It is a 3D-approximation of the behavior of the above example. The first axis represents time (t), the second axis shows the various tasks (θ^*) and the third axis approximates the depth of continuations (κ) i.e., recursion stack. A plan is associated to the invocation of `sponsor` showing the dynamic extent of the group: tasks belong to that group while above this plan. Entry in the group is marked by a black square, exit with a white square. The invocation to `fork` (marked by a black circle) creates two tasks, one of which commits suicide (marked by a white circle) after exiting the `sponsor` form

Roughly stated, the example spawns two similar computations on two different sites, returns the value of the first one to complete and disposes of the remaining spawned task[3]. A paused group that becomes unreachable is collected by the Garbage Collector (GC), but the paper will not emphasize the connection to the GC. The previous example exhibits a poor-man's implementation of fault tolerance where the burden of making `thunk` insensitive to the fact that it is concurrently invoked is entirely on the shoulders of the programmer.

This concludes the informal description of our essential language. Around a pure functional kernel, it offers orthogonal features such as side-effects, concurrency, distribution and groups of tasks. These new features are added as functions or special forms as felt convenient to express their semantics below. Examples of the fully-fledged version of this essential language as well as a comparison with related work may be found in [Que92, QD93, Que94].

2 The functional abstract machine

This section presents an operational semantics describing our language. The protocols that manage coherency of boxes and groups are parts of the definitions. These definitions can also be viewed as the specification of an abstract machine with transitions between states. We assume some familiarity with denotational semantics [Sto77, Sch86] or at least higher order functional programming [Hen80, FWH92]. To improve the legibility of definitions, we adopt a syntax close to that of a programming language. We expect these denotations to be self explanatory.

[3] There can be more than one task if `(thunk)` forks new tasks

The language runs on a network of heterogeneous workstations. A site is represented by a regular UNIX process. A site has a copy of the runtime library of the language, it runs a local scheduler which chooses which task to run in its local queue of tasks. Tasks can only be preempted in well defined places ascribed by the semantics of the language that is why tasks are composed of a succession of uninterruptible but short computation steps. When a box or a group is created, it remains on its birth site. If there are n sites, then any site has n clocks, one of which is its proper clock and is incremented any time a local box (resp. a group) is mutated (resp. paused or awakened). The other clocks represent what the site believes the other proper clocks to be, these clocks are updated any time messages are received [Fid88, Mat88].

The abstract machine has a *state* made of four components: *(i)* a set of threads, *(ii)* a store, *(iii)* a time, *(iv)* and, a list of already produced answers.

$$\mathbf{State} = \mathbb{P}(\mathbf{Thread}) \times \mathbf{Store} \times \mathbf{Time} \times \mathbf{Value}^*$$

A *thread* represents an atomic computation step within a task. When a thread is run, it performs some action and generally yields a new thread (representing the work the task has still to do) that is incorporated back in the set of threads. The inner structure of a thread will be detailed below.

The *store* represents the shared memory seen by all sites. The store associates to any site and any box (resp. group), its content (resp. group status) as perceived by this site. The store is therefore represented by a function mapping a site and a box or group onto a value or a cached-value or a constant saying that no associated information is locally available.

$$\mathbf{Store} = (\mathbf{Site} \times (\mathbf{Box} + \mathbf{Group})) \rightarrow (\mathbf{Value} + \mathbf{CachedValue} + \{\textit{not-here}\})$$

Each site of a distributed shared memory has some clocks. One of these clocks is the proper clock of the site, the others hold what the site thinks the other proper clocks to be. This is represented by the *time* function which associates to a site μ_1 and a site μ_2 the value of the proper clock of μ_2 as perceived by μ_1. Clock values i.e., dates, belong to the **Date** domain and can be seen as natural numbers. For a site μ and a time function τ, the proper time of the site is $\tau(\mu, \mu)$.

$$\mathbf{Time} = (\mathbf{Site} \times \mathbf{Site}) \rightarrow \mathbf{Date}$$

A cached value is a tuple made of a value and a date that indicates the proper time of the site when and where the value was remotely fetched.

$$\mathbf{CachedValue} = \mathbf{Date} \times \mathbf{Value}$$

Due to the `fork` construct, a program may have multiple answers. Therefore the state of the machine accumulates the sequence of already answered values in its fourth component. The **Value** domain contains at least closures, boolean values, boxes, groups and sites which are necessary for our essential language. It may also contain pairs, numbers, etc.

We suppose some sites to exist and to be represented by elements of the **Site** domain. We also suppose the constant *all-sites* to be the finite sequence of all sites:

this constant does not need to be visible from the programmer. The only information attached to a box is its birth site. To any group is not only attached its birth site but also its supergroup i.e., the group that was in control when created. Groups form a tree rooted by γ_{init}. Birth-sites (resp. supergroups) can be known with the *site* (resp. *super*) function.

$$site : (\mathbf{Box} + \mathbf{Group}) \rightarrow \mathbf{Site}$$

$$super : \mathbf{Group} \rightarrow \mathbf{Group}$$

We can now return to the denotation of a thread. Being the first computation step of a task, a thread expresses that some computation is performed on a site and under the control of a group. The various possibilities for computations will be unveiled in the next sections.

$$\mathbf{Thread} = \mathbf{Computation} \times \mathbf{Site} \times \mathbf{Group}$$

Following the great tradition of denotational semantics, all variables are represented by Greek letters making their types explicit. The chosen notations appear on figure 3.

π : **Program**	ν : **Identifier**	ε : **Value**	φ : **Function**
ϵ : **CachedValue**	γ : **Group**	β : **Box**	σ : **Store**
θ : **Thread**	μ : **Site**	χ : **Computation**	κ : **Continuation**
ρ : **Environment**	τ : **Time**	Σ : **State**	

$<$ STATE: $\_, \_, \_, \_ >$:	**State**	$<$ GROUPSTATUS: $\_, \_, \_ >$:	**GroupStatus**
$\boxed{}$:	**Computation**	$\langle\!\langle \_, \_, \_ \rangle\!\rangle$:	**Thread**
$<$ CACHEDVALUE: $\_, \_ >$:	**CachedValue**	$<\ldots>$:	**Continuation**

Fig. 3. Denotational notations

2.1 Denotation of the kernel

This section describes the semantics of the functional kernel of the language in terms of elements of **Computation**. The presentation of the scheduler is delayed to section 4.2 but it informally works as follows: if the current state is $<$ STATE: $\theta^*, \sigma, \tau, \varepsilon^* >$, and if there is a thread among θ^* that can be run, say $\theta = \langle\!\langle \chi, \mu, \gamma \rangle\!\rangle$, then the scheduler invokes:

$$step\ \chi(\mu, \gamma)(\theta^* - \{\theta\}, \sigma, \tau, \varepsilon^*) = <\text{STATE: } {\theta^*}', \sigma', \tau', {\varepsilon^*}' >$$

The *step* function performs the computation χ on site μ and under sponsorship of γ. It also receives the set of the remaining threads $\theta^* - \{\theta\}$, the store σ, the time τ and the sequence of already returned answers ε^*. The result is a new state on which the scheduler is applied again until no task can be run.

For the functional kernel, a thread just yields another thread. To conveniently express that, we introduce the *adjoin* utility function which takes a computation and adjoins it (properly wrapped into a thread on the current site and under the sponsorship of the current group) to the current set of threads.

$$adjoin\ \chi = \lambda\mu\gamma.\lambda\theta^*\sigma\tau\varepsilon^*.\ <\textsc{State}\!:\ \{\langle\!\langle \chi, \mu, \gamma \rangle\!\rangle\} \cup \theta^*, \sigma, \tau, \varepsilon^* >$$

To shorten space, we limit the semantics of functions and combinations to the unary case, we also suppose programs to be correct and error-free. To improve readability, computations are typeset inside (typographic) boxes and start with a curly letter identifying their nature. A computation can be an evaluation (marked as $\boxed{\mathcal{E}[\![\pi]\!]\rho\kappa}$) of a program π within lexical environment ρ and continuation κ. A computation can also be the return of a value ε to a continuation κ (marked as $\boxed{\mathcal{R}\kappa\varepsilon}$). Here follow the denotations of the functional kernel of our language:

$$step\ \boxed{\mathcal{E}[\![\nu]\!]\rho\kappa} = adjoin\ \boxed{\mathcal{R}\kappa\rho(\nu)}$$

$$step\ \boxed{\mathcal{E}[\![\texttt{(lambda (}\nu\texttt{) }\pi\texttt{)}]\!]\rho\kappa} = adjoin\ \boxed{\mathcal{R}\kappa\lambda\kappa'\varepsilon.adjoin\ \boxed{\mathcal{E}[\![\pi]\!]\rho[\nu \rightarrow \varepsilon]\kappa'}}$$

$$step\ \boxed{\mathcal{E}[\![\texttt{(}\pi\ \pi'\texttt{)}]\!]\rho\kappa} = adjoin\ \boxed{\mathcal{E}[\![\pi]\!]\rho\texttt{<arg0 }\kappa\ \rho\ \pi'\texttt{>}}$$

$$step\ \boxed{\mathcal{R}\texttt{<arg0 }\kappa\ \rho\ \pi\texttt{>}\varphi} = adjoin\ \boxed{\mathcal{E}[\![\pi]\!]\rho\texttt{<apply1 }\kappa\ \varphi\texttt{>}}$$

$$step\ \boxed{\mathcal{R}\texttt{<apply1 }\kappa\ \varphi\texttt{>}\varepsilon} = \varphi(\kappa, \varepsilon)$$

The semantics of the unary combination $(\pi\ \pi')$ is the most complex. First, the functional term π (which is in functional position) is evaluated with a new continuation identified by `<arg0...>`. When this continuation receives the value φ of the functional term, it evaluates the second term of the original combination but with a new continuation, identified by `<apply1...>`, memorizing the received function. Finally when this `<apply1...>` continuation receives a value ε, it applies the memorized function φ on ε and the result is given back to the original continuation of the combination.

The semantics of the unary abstraction is to create a closure on this abstraction and to return it immediately to the continuation. When the closure is invoked, for instance with the above `apply1` rule, it will create a new thread that will evaluate the body of the abstraction in the appropriate environment. The notation $\rho[\nu \rightarrow \varepsilon]$ returns a new environment similar to ρ except on point ν where it now yields ε.

These denotations form the λ-calculus-like kernel of our language. The computation is expressed as a succession of transition steps i.e., calls to the *step* function. Some scheduling may occur between these steps i.e., computation steps are interleaved to simulate concurrency.

2.2 Task birth and death

In the initial state, the initial store contains nothing, the initial time function sets all clocks from all sites to zero, no answer is already produced and a single thread exists which has to evaluate the initial program π within the predefined lexical environment and the initial continuation `<result>`.

$$\Sigma_{init} =< \text{STATE: } \{\langle\!\langle \boxed{\mathcal{E}[\![\pi]\!]\rho_{init}\texttt{<result>}}, \mu_{init}, \gamma_{init}\rangle\!\rangle\}, \sigma_{init}, \tau_{init}, <>>$$

When a value is returned to the `<result>` continuation, it is appended ($\varepsilon^* \S < \varepsilon >$) to the sequence of already produced answers and the current task is terminated.

$$step \boxed{\mathcal{R}\texttt{<result>}\varepsilon} = \lambda\mu\gamma.\lambda\theta^*\sigma\tau\varepsilon^*. < \text{STATE: } \theta^*, \sigma, \tau, \varepsilon^* \S < \varepsilon >>$$

A task can be prematurely terminated by the `(suicide)` special form. New tasks can be created by the `fork` special form, they are evaluated with the same lexical environment and continuation. Once created, tasks are independent and can only exchange informations through the store.

$$step \boxed{\mathcal{E}[\![\texttt{(suicide)}]\!]\rho\kappa} = \lambda\mu\gamma.\lambda\theta^*\sigma\tau\varepsilon^*. < \text{STATE: } \theta^*, \sigma, \tau, \varepsilon^* >$$

$$step \boxed{\mathcal{E}[\![\texttt{(fork } \pi \ \pi'\texttt{)}]\!]\rho\kappa} =$$
$$\lambda\mu\gamma.\lambda\theta^*\sigma\tau\varepsilon^*. < \text{STATE: } \{\langle\!\langle \boxed{\mathcal{E}[\![\pi]\!]\rho\kappa}, \mu, \gamma\rangle\!\rangle\} \cup \{\langle\!\langle \boxed{\mathcal{E}[\![\pi']\!]\rho\kappa}, \mu, \gamma\rangle\!\rangle\} \cup \theta^*, \sigma, \tau, \varepsilon^* >$$

Task birth or death do not affect store or time, they are purely functional.

2.3 Discussion

The semantics of the above abstract machine improves on the denotational semantics given in [Que92] by its simplicity. It uses interleaving to express concurrency. Although the scheduler apparently sequentializes all threads, this order can only be perceived by the program through side-effects on boxes. The semantics of our abstract machine is rather similar to the semantics of actors language [AMST92] except that our language is more complex due to the presence of closures that shares bindings i.e. accesses to the store.

Other differences with respect to other formalisms are *(i)* the various perceptions of store and time are now explicit, *(ii)* sites are not sequential processes but have multitasking capacity.

3 Boxes and the `remote` special form

This section describes the distribution aspects and mainly the `remote` special form and the box coherency machinery.

3.1 Migration

A computation can migrate as specified by the `remote` special form. This form evaluates its first term with a continuation (identified by `<migrate...>`) which waits for a site. When receiving a site, this continuation remotely adjoins the evaluation of the second term on that site (instead of the current one if they are different) but with a new `return` continuation that will migrate the resulting value back to the original site. The `return` continuation might be called a "geographic" continuation whose sole rôle is to migrate a value from one site to another.

$$\mathit{step}\ \boxed{\mathcal{E}[\![\texttt{(remote}\ \pi\ \pi'\texttt{)}]\!]\rho\kappa} = \mathit{adjoin}\ \boxed{\mathcal{E}[\![\pi]\!]\rho\texttt{<migrate}\ \kappa\ \rho\ \pi'\texttt{>}}$$

$$\begin{array}{l}
\mathit{step}\ \boxed{\mathcal{R}\texttt{<migrate}\ \kappa\ \rho\ \pi\texttt{>}\mu} = \\
\lambda\mu'\gamma.\ \textbf{if}\ \mu' = \mu \\
\quad\quad \textbf{then}\ \mathit{adjoin}\ \boxed{\mathcal{E}[\![\pi]\!]\rho\kappa}\ (\mu', \gamma) \\
\quad\quad \textbf{else}\ \mathit{remote\text{-}adjoin}\ \mu \boxed{\mathcal{E}[\![\pi]\!]\rho\texttt{<return}\ \kappa\ \mu'\texttt{>}}\ (\mu', \gamma) \\
\quad\quad \textbf{endif}
\end{array}$$

$$\mathit{step}\ \boxed{\mathcal{R}\texttt{<return}\ \kappa\ \mu\texttt{>}\varepsilon} = \mathit{remote\text{-}adjoin}\ \mu \boxed{\mathcal{R}\kappa\varepsilon}$$

Communications use the *remote-adjoin* function that takes a site and a computation, creates a new thread within the current group but on the given site and adjoins this new thread to the set of runnable threads. Although the communication seems instantaneous, the migrated thread has to wait to be scheduled: this is equivalent to safe communications (no message loss and no duplication of messages) with bounded transmission delays provided the scheduler is fair. Observe that messages are not ordered and communications are not FIFO either.

As in many coherency protocols, communications i.e., threads sent by *remote-adjoin*, are accompanied by some time informations to make clock progress. This time update is represented by $\boxed{\mathcal{T}\chi\mu\tau}$: the emitting site μ sends the computation χ, its own identity μ and the current time function τ containing its perception of time. At the other end, when the message is processed, the clocks of the receiving site are updated i.e., every clock of the receiving site is set to the higher value it had on the receiving or the emitting site. This is achieved by the *update-time* function. We note $\tau[\mu_1 \times \mu_2 \longrightarrow t]$, the new time function that is entirely similar to τ except that site μ_1 now believes that the proper clock of site μ_2 is now t. After updating clocks of the receiving site, the computation χ is wrapped into a thread that is added to the set of runnable threads.

$$\mathit{remote\text{-}adjoin}\ \mu\chi = \lambda\mu_1\gamma.\lambda\theta^*\sigma\tau\varepsilon^*.\ <\textsc{State}\text{: } \{\langle\!\langle \boxed{\mathcal{T}\chi\mu_1\tau}\ , \mu, \gamma\rangle\!\rangle\} \cup \theta^*, \sigma, \tau, \varepsilon^* >$$

$$\begin{array}{l}
\mathit{step}\ \boxed{\mathcal{T}\chi\mu\tau} = \\
\lambda\mu_1\gamma.\lambda\theta^*\sigma\tau_1\varepsilon^*.\ <\textsc{State}\text{: } \{\langle\!\langle \chi, \mu_1, \gamma\rangle\!\rangle\} \cup \theta^*, \sigma, \mathit{update\text{-}time}(\mu_1, \tau_1, \mu, \tau_1), \varepsilon^* >
\end{array}$$

$$\begin{array}{l}
\mathit{update\text{-}time}(\mu, \tau, \mu_1, \tau_1) = \\
\ \textbf{let}\ \tau_2 = \mathit{reduce}(\lambda\tau_2\mu_2.\ \textbf{let}\ t = \tau_2(\mu, \mu_2) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{and}\ t_1 = \tau_1(\mu_1, \mu_2) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{in}\ \ \textbf{if}\ t_1 > t
\end{array}$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\quad \textbf{then } \tau_2[\mu \times \mu_2 \rightarrow t_1] \\
\qquad\qquad\qquad\qquad\quad \textbf{else } \tau_2 \\
\qquad\qquad\qquad\qquad\quad \textbf{endif} \quad , \tau, \textit{all sites}) \\
\textbf{in } \tau_2 \\
\textbf{whererec } \ \textit{reduce}(\Phi, \tau_2, \mu^*) = \\
\qquad\qquad \textbf{if } \mu^* = <> \\
\qquad\qquad \textbf{then } \tau_2 \\
\qquad\qquad \textbf{else } \textit{reduce}(\Phi, \Phi(\tau_2, \mu^* \downarrow_1), \mu^* \dagger 1) \\
\qquad\qquad \textbf{endif}
\end{array}
$$

We now introduce the box machinery which requires time consideration for its coherency. All box-related features are implemented as functions. They thus have the interface of unary or binary functions.

3.2 Box creation

The `new-box` function allocates a box and returns it to its continuation along with a modified store recording, on the current site, the initial content of this box. The notation $\sigma_1[\mu \times \beta \rightarrow \varepsilon]$ returns a store function identical to σ_1 except that, on site μ, the box β now contains ε. The creation of a box does not change the time function: a box creation is not a side effect wrt time. The *allocate-a-box* function is implementation dependent (it encapsulates the GC aspects), it allocates, in store σ, a box whose birth site is μ such that $\mathit{site}(\beta) = \mu$. The modified store and the box itself are given to the third argument of *allocate-a-box* as σ_1 and β.

$\rho[\![\texttt{new-box}]\!] =$
$\lambda\kappa\epsilon.\lambda\mu\gamma.\lambda\theta^*\sigma\tau\varepsilon^*.\mathit{allocate\text{-}a\text{-}box}(\sigma, \mu, \lambda\sigma_1\beta.\mathit{adjoin}\ \boxed{\mathcal{R}\kappa\beta}\ (\mu, \gamma)(\theta^*, \sigma_1[\mu \times \beta \rightarrow \epsilon], \tau, \varepsilon^*))$

3.3 Box modification

Writing in a box is simpler than reading it (as far as denotation is concerned). The `box-set!` function is defined by the utility function *perform-box-update* which recognizes two cases. If the box is local i.e., was created on the current site, then the box is modified (the store function is updated and the proper clock is incremented) then the old content of the box is returned to the continuation. Since these actions are done within a single computation step they form an atomic sequence of actions. If the box is not local then a message is sent to the birth site of the box. The message, a <u>b</u>ox <u>w</u>rite request identified by $\boxed{\mathcal{B}_W\texttt{<return }\kappa\ \mu\texttt{>}\beta\varepsilon}$, will perform the mutation and will return the result back to the current site.

$\rho[\![\texttt{box-set!}]\!] = \mathit{perform\text{-}box\text{-}update}$

$\mathit{step}\ \boxed{\mathcal{B}_W\kappa\beta\varepsilon}\ = \mathit{perform\text{-}box\text{-}update}(\kappa, \beta, \varepsilon)$

$\mathit{perform\text{-}box\text{-}update} =$
$\lambda\kappa\beta\varepsilon.\lambda\mu\gamma.\lambda\theta^*\sigma\tau\varepsilon^*.$
$\quad \textbf{if } \mu = \mathit{site}(\beta)$

then **let** $\varepsilon_1 = \sigma(\mu, \beta)$
  in *adjoin* $\boxed{\mathcal{R}\kappa\varepsilon_1}$ $(\mu, \gamma)(\theta^*, \sigma[\mu \times \beta \rightarrow \varepsilon], \tau[\mu \times \mu \rightarrow 1 + \tau(\mu, \mu)], \varepsilon^*)$
else *remote-adjoin site*(β) $\boxed{\mathcal{B}_W \texttt{<return } \kappa\ \mu\texttt{>}\beta\varepsilon}$ $(\mu, \gamma)(\theta^*, \sigma, \tau, \varepsilon^*)$
endif

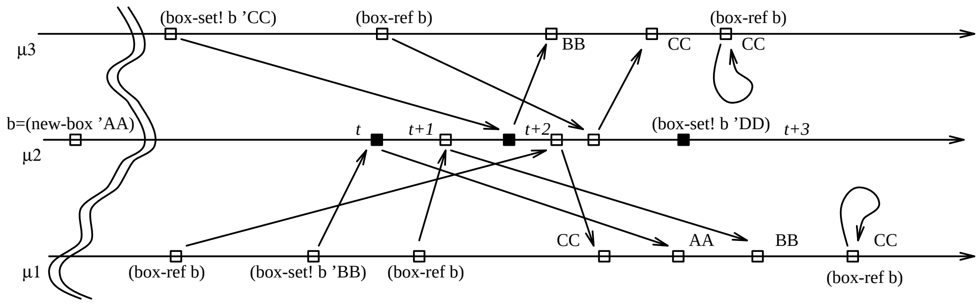

Fig. 4. Box at work

Roughly speaking, all modifications to a box β are migrated towards its birth site i.e.*site*(β) where they are sequentialized. For example in figure 4, the original site of the box is μ_2 and all `box-set!` operations migrate towards μ_2 where they are sequentialized.

3.4 Reading box

As many other protocols, sites locally cache non local boxes to avoid communication delays and speed up multiple readings. But to cache boxes requires to invalidate them when the content of the original boxes change. Instead of eagerly broadcasting an invalidation message, we lazily propagate the invalidation information through messages; the validity of cached boxes is checked when read.

When a box is locally cached, its value is time-stamped with the proper time of the original site when remotely read. For instance on figure 4, the first `(box-ref b)` request from μ_3 returns a cached value $(t + 2, CC)$. The cached value of a box is valid, up to date, if its time-stamp is equal to the time the birth site of the box is believed to be. For instance still on figure 4, while μ_3 believes μ_2 to be at time $t + 2$, the cached value CC is valid so the second `box-ref` on μ_3 is immediately evaluated. When a box is modified on its birth site, the proper clock of the birth site is incremented and this naturally invalidates all the caches that perceives this time incrementation.

As for writing in boxes, two cases are distinguished whether the box is local or not. If it is local, the box is read and its content is returned to the continuation. If the box is not local then two new cases are recognized. If the box is validly cached then the cached value is extracted and returned to the continuation. Otherwise a box reading message (identified as $\boxed{\mathcal{B}_R\kappa\beta}$) is emitted towards the birth site of the box. When this message is processed on the birth site of the box, the content of the box is time-stamped with the proper current time of the birth site to form a cached value that is returned to the continuation. The continuation of a $\mathcal{B}_R$ message is a composed continuation since not only it has to migrate back the cached value towards the requesting site but it also has to locally cache the fetched value to improve future reading on that site, this effect is achieved by the <box-cache...> continuation which is run on the requesting site. Due to transmission or scheduling delay, the <box-cache...> continuation may be activated so late that it is no longer possible to cache the value since it is already obsolete. This is the case in figure 4 on site μ_1 when the second `(box-ref b)` request returns $(t+1, BB)$. This value is obsolete and must not be cached since it is already known by site μ_1 that site μ_2 has a proper clock at least equal to $t+2$. Moreover, there is already an up to date value in the cache, CC, which was brought back by the first `(box-ref b)` on site μ_1. However, due to the semantics of `box-ref`, BB is still returned to the continuation.

$$
\begin{array}{l}
\rho[\![\texttt{box-ref}]\!] = \\
\lambda\kappa\beta.\lambda\mu\gamma.\lambda\theta^*\sigma\tau\varepsilon^*.\ \textbf{if}\ \mu = site(\beta) \\
\qquad\qquad\qquad \textbf{then}\ adjoin\ \boxed{\mathcal{R}\kappa\sigma(\mu,\beta)}\ (\mu,\gamma)(\theta^*,\sigma,\tau,\varepsilon^*) \\
\qquad\qquad\qquad \textbf{else}\ \ \textbf{let}\ \epsilon = \sigma(\mu,\beta) \\
\qquad\qquad\qquad\quad \textbf{in}\ \ \textbf{if}\ \epsilon \in \textbf{CachedValue} \\
\qquad\qquad\qquad\qquad \textbf{then}\ \ \textbf{let}\ \ <\textsc{CachedValue}\!:\ t, \varepsilon >= \epsilon \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ \textbf{if}\ t = \tau(\mu, site(\beta)) \\
\qquad\qquad\qquad\qquad\qquad \textbf{then}\ adjoin\ \boxed{\mathcal{R}\kappa\varepsilon}\ (\mu,\gamma)(\theta^*,\sigma,\tau,\varepsilon^*) \\
\qquad\qquad\qquad\qquad\qquad \textbf{else}\ remote\text{-}adjoin\ site(\beta) \\
\qquad\qquad\qquad\qquad\qquad \boxed{\mathcal{B}_R\texttt{<return <box-cache}\ \kappa\ \beta\texttt{>}\ \mu\texttt{>}\beta}\ (\mu,\gamma)(\theta^*,\sigma,\tau,\varepsilon^*) \\
\qquad\qquad\qquad\qquad\qquad \textbf{endif} \\
\qquad\qquad\qquad\qquad \textbf{else}\ remote\text{-}adjoin\ site(\beta) \\
\qquad\qquad\qquad\qquad\qquad \boxed{\mathcal{B}_R\texttt{<return <box-cache}\ \kappa\ \beta\texttt{>}\ \mu\texttt{>}\beta}\ (\mu,\gamma)(\theta^*,\sigma,\tau,\varepsilon^*) \\
\qquad\qquad\qquad\qquad \textbf{endif} \\
\qquad\qquad\qquad \textbf{endif}
\end{array}
$$

$$
\begin{array}{l}
step\ \boxed{\mathcal{B}_R\kappa\beta}\ = \\
\lambda\mu\gamma.\lambda\theta^*\sigma\tau\varepsilon^*.adjoin\ \boxed{\mathcal{R}\kappa < \textsc{CachedValue}\!:\ \tau(\mu,\mu), \sigma(\mu,\beta) >}\ (\mu,\gamma)(\theta^*,\sigma,\tau,\varepsilon^*)
\end{array}
$$

$$
\begin{array}{l}
step\ \boxed{\mathcal{R}\texttt{<box-cache}\ \kappa\ \beta\texttt{>}\epsilon}\ = \\
\lambda\mu\gamma.\lambda\theta^*\sigma\tau\varepsilon^*.\ \textbf{let}\ \ <\textsc{CachedValue}\!:\ t, \varepsilon >= \epsilon \\
\qquad\qquad \textbf{in}\ adjoin\ \boxed{\mathcal{R}\kappa\varepsilon}\ (\mu,\gamma)(\theta^*,\ \ \textbf{if}\ t = \tau(\mu, site(\beta)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ \sigma[\mu \times \beta \rightarrow \epsilon] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{endif}\ \ ,\tau,\varepsilon^*)
\end{array}
$$

3.5 Discussion

The above coherency protocol is supported by the following principles:

- A side effect perceived by a thread cannot be ignored by the threads in its continuation. This does not depend on the sites where these threads are run.

– A site can be viewed as an entity within which threads share clocks i.e., a similar perception of time. In other words, a side effect perceived by a thread on a site cannot be ignored by the threads of the same site if they are scheduled after.

The first principle is related to causality i.e., the natural sequentiality present in our language, while the second reflects what sharing clocks is all about.

The above protocol for boxes coherency ensures soundness: modification of boxes are coherently perceived, but this does not ensure liveness. If a site never receives messages then it ignores boxes that may have changed and thus prevents computations to usefully progress if these changes were known. This is the case on the right of figure 4 where box `b` is modified and sites μ_1 and μ_3 did not perceive it yet. This is coherent since it is as if the threads of μ_1 and μ_3 were scheduled on μ_3 before $t + 3$. To ensure liveness, one may program a "worm" that passes regularly trough all sites in order to propagate clocks.

When a box is changed on a site then the proper clock is incremented and this makes obsolete all cached boxes from that site. This may be considered as too strong but can be alleviated by the use of more clocks. Suppose for instance that a site has more than one proper clock, suppose also that it is possible to know to which clock is associated a box, then any time a box is changed only its related clock is incremented, therefore only boxes associated to that same clock will be invalidated. To handle an unbound number of boxes while bounding the size of messages introduces some imprecision. However if a box is of considerable value, it can have its own proper clock (the protocol then resembles that of [MSRN92]). Of course to augment the number of clocks also augments the size of messages that have to carry all these clock values (but see 5.2).

The protocol is rather naïve and not very efficient but it only expresses the essence of the protocol. For instance, a remote box writing can also return the new value to update the local cache. Remote box read requests can be shared to spare bandwidth. Another source of improvement is to only increment clocks if the new value is different from the former value.

Our protocol describes how shared mutable boxes are "migrated". More complex mutable objects can be constructed with our boxes using indirections. if they are to be migrated, immutable values such as numbers, booleans, characters and even closures are simply copied from one site to another. The traditional difference of Lisp between physical equality (`eq?`) and structural equality (`equal?`) vanishes for immutable objects.

To sum up, our protocol (the *lazy invalidation propagation*) innovates on several points:

1. it is specified in a denotational-like style. The description makes use of functions and continuations expressing atomic computation steps and communications. It has a higher level compared to descriptions using messages, automata and side-effects. Related sequential computations are naturally expressed via continuations.
2. it can handle an arbitrary number of boxes,
3. it does not require communications to be FIFO,
4. it does not require a centralized controller,

5. it does not require broadcast technology but favors instead a lazy propagation of invalidation information. This takes naturally into account bounded communication breakdown provided no message is lost.

4 Groups of tasks

This section exposes the coherency protocol used to control hierarchically embedded groups of tasks that can be paused or awakened. Groups of tasks are created via the `sponsor` function. Groups can be arbitrarily embedded, they form a tree. When created, groups are running. They can be suspended (resp. resumed) with the `pause!` (resp. `awake!`) imperative primitive. These functions pose many problems to be defined and implemented.

The first problem comes from their semantics. Very roughly said, to `pause!` a group is similar to set the running status of the given group to false. At that time and on every site, the tasks that belong to the group must be suspended. When this is finished, `pause!` returns to its caller the former running status of the now paused group. Suspending tasks thus seems to impose a sort of global synchronization among all sites. The following protocol avoids this global synchronization and uses instead (once again) a lazy propagation of pausing or awakening information.

If `pause!` (resp. `awake!`) is to return immediately then non local tasks have to be paused (resp. awakened) lazily. It must be ensured that there is no still running task in the group that may alter the behavior of the caller of `pause!`. In other words, if a task pauses a group (and does not belong to this same group otherwise it would be also paused) then it cannot be later affected by the tasks that are supposed to be suspended from its point of view. For example, on figure 5, site μ_2 knows that group `g` is paused, it thus cannot accept to run a thread coming from μ_1 and belonging to that very group: when received on μ_2, this thread is automatically paused.

To each group is associated a box that holds its running status, a boolean stating if the group is running or not i.e., awakened or paused. The box coherency protocol is used to manage these boxes but groups are more complex objects since they form a tree that must be kept coherent. Pausing a group also means pausing all the subgroups it contains etc. The second problem, see figure 6, is that a task can require to pause a group while another task requires to awake one of its subgroup. This contention problem has only two exclusive solutions: or all tasks belonging to `g` are paused even those of `sg` or, all tasks belonging to `g` are paused except those belonging to `sg`. The choice between the two solutions depends on whether pausing `g` is perceived after or before awakening `sg` from the point of view of the birth site of `sg`.

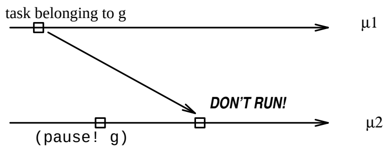

Figure 5: Altering the continuation of `pause!`

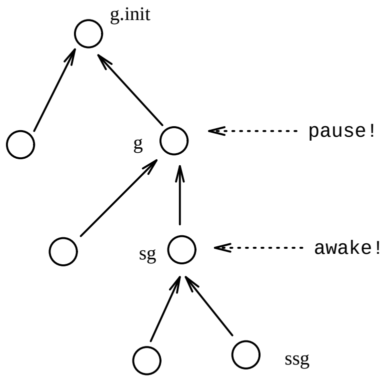

Figure 6: Concurrent `pause!` and `awake!`

The essence of the protocol is to ensure that no decision or action is taken on a group with an uncertain running status. Each site maintains up to date the running status of all the groups to which its local threads belong. Sites know the running status of all their local groups and cache the running status of all their non local groups. If the running status of a group is obsolete or unknown then the site asks for it, meanwhile its local scheduler defers running any threads belonging to that group. When the status of a non local group changes then this new status has to be recursively propagated to the local groups. These mutations of running status of local groups are simple boxes mutations that regularly increment the proper clock of the site. The regular box machinery will propagate these mutations and will force other sites to ask again for the running status of these groups. A group status change is therefore propagated top-down in the tree of groups. An important point is that groups know their supergroups but there is no link from groups to their subgroups since it would be difficult to maintain such a dynamic cyclic structure that will unnecessarily augments the burden of the GC. It is therefore of the duty of groups to ensure that they are aware of the running status of their parentage.

This organization is not sufficient to solve the second problem cited above. For that it must be possible to order all operations on hierarchy of groups. We therefore associate to each group a box containing its group status. This group status not only contains its running status (a boolean) but also the last date when this group status changed as well as the date of the last perceived change in the super-group, see figure 7. These two dates are expressed wrt the proper clocks of the birth site of the group and supergroup. The running status of a group is certain as soon as its group status is known i.e., is local or validly cached, and up to date wrt the group hierarchy i.e., perceived changes on any of its supergroups are not ignored.

4.1 The group protocol

This section details the underlying algorithms of the protocol. The `sponsor` function receives a function φ, allocates a group object (whose supergroup is the current group) on the current site and in the current store, augments the continuation with a `<group...>` continuation then invokes the received function φ under the control of

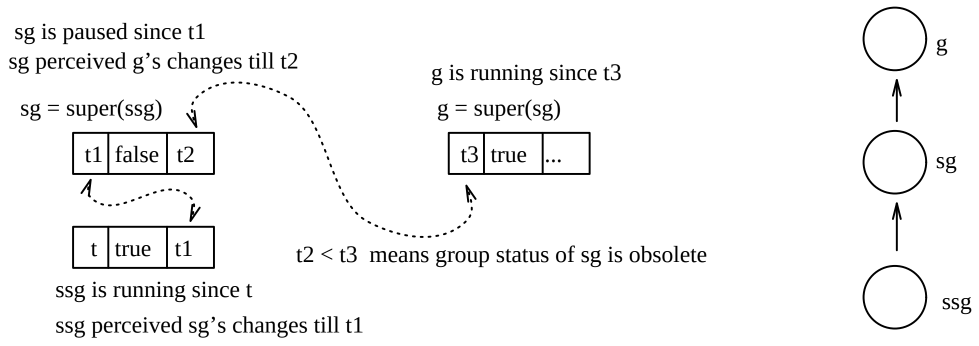

Fig. 7. Group status coherency

this new group. The created group is running i.e., its running status is true. The time function is not changed by this allocation: it is not a side-effect. The `<group...>` continuation is needed to restaure the correct group sponsorship when the function φ returns a value. The group status is a triple. Its first component is the last date it was changed, the second is its running status (a boolean) and its third component is the date of the last perceived supergroup's change. This first component can be extracted from a group by the *group-last-change-date* utility function which recognizes the top-group γ_{init} as a special case to avoid infinite regression.

$$
\begin{array}{l}
\rho[\![\mathsf{sponsor}]\!] = \\
\lambda\kappa\varphi.\lambda\mu\gamma.\lambda\theta^*\sigma\tau\varepsilon^*.\mathit{allocate\text{-}a\text{-}group}(\sigma,\mu,\gamma, \\
\quad \lambda\sigma_1\gamma_1.\ \mathbf{let}\ \sigma_2 = \sigma_1[\mu\times\gamma_1 \to <\textsc{GroupStatus}\colon\ \tau(\mu,\mu), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{true}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{group\text{-}last\text{-}change\text{-}date}(\gamma,\mu,\sigma_1) >] \\
\qquad\quad \mathbf{in}\ \mathit{adjoin}\ \boxed{\mathcal{R}\texttt{<apply1 <group } \kappa\ \gamma\texttt{>}\ \varphi\texttt{>}\gamma_1}\ (\mu,\gamma_1)(\theta^*,\sigma_2,\tau,\varepsilon^*)\ )
\end{array}
$$

$$
\begin{array}{l}
\mathit{group\text{-}last\text{-}change\text{-}date}(\gamma,\mu,\sigma) = \\
\ \mathbf{if}\ \gamma = \gamma_{init} \\
\ \mathbf{then}\ 0 \\
\ \mathbf{else}\ \ \mathbf{if}\ \mu = \mathit{site}(\gamma) \\
\qquad\quad \mathbf{then}\ \ \mathbf{let}\ \ <\textsc{GroupStatus}\colon\ t,\varepsilon,t_1 >= \sigma(\mu,\gamma) \\
\qquad\qquad\qquad \mathbf{in}\ t \\
\qquad\quad \mathbf{else}\ \ \mathbf{let}\ \ <\textsc{CachedValue}\colon\ t,\varepsilon >= \sigma(\mu,\gamma) \\
\qquad\qquad\quad\ \ \mathbf{in}\ \ \mathbf{let}\ \ <\textsc{GroupStatus}\colon\ t_1,\varepsilon_1,t_2 >= \varepsilon \\
\qquad\qquad\qquad\quad \mathbf{in}\ t_1 \\
\qquad\quad \mathbf{endif} \\
\ \mathbf{endif}
\end{array}
$$

$$
\mathit{step}\ \boxed{\mathcal{R}\texttt{<group } \kappa\ \gamma\texttt{>}\varepsilon} = \lambda\mu\gamma_1.\mathit{adjoin}\ \boxed{\mathcal{R}\kappa\varepsilon}\ (\mu,\gamma)
$$

The `pause!` and `awake!` functions are symmetrical: they just call the *perform-status-change* function to set the running status of the group to **true** or **false**. As usual, the *perform-status-change* function recognizes two cases whether the group object is local or not. If the group is not local then a message (a group write request, identified by $\mathcal{G}_{\mathcal{W}}$) is sent to run *perform-status-change* on the birth site of the group with a `<return>` continuation that will bring back the result.

If the group is local and its group status certain then its running status is changed and the proper clock is incremented.

$$\rho[\![\texttt{pause!}]\!] = \lambda\kappa\gamma.\textit{perform-status-change}(\kappa, \gamma, \mathbf{false})$$

$$\rho[\![\texttt{awake!}]\!] = \lambda\kappa\gamma.\textit{perform-status-change}(\kappa, \gamma, \mathbf{true})$$

$$\textit{step}\ \boxed{\mathcal{G}_W\kappa\gamma\varepsilon} = \textit{perform-status-change}(\kappa, \gamma, \varepsilon)$$

$$\begin{array}{l}
\textit{perform-status-change} = \\
\lambda\kappa\gamma\varepsilon.\lambda\mu\gamma_1.\lambda\theta^*\sigma\tau\varepsilon^*. \\
\quad \mathbf{if}\ \mu = \textit{site}(\gamma) \\
\quad \mathbf{then}\ \ \mathbf{if}\ \textit{group-parentage-known?}(\gamma, \mu, \sigma, \tau) \\
\qquad\qquad \mathbf{then}\ \ \mathbf{let}\ \epsilon = \sigma(\mu, \gamma) \\
\qquad\qquad\qquad\quad \mathbf{and}\ t = 1 + \tau(\mu, \mu) \\
\qquad\qquad\qquad\quad \mathbf{in}\ \ \mathbf{let}\ <\textsc{GroupStatus}\colon t_1, \varepsilon_1, t_2 >= \epsilon \\
\qquad\qquad\qquad\qquad \mathbf{in}\ \ \mathbf{let}\ \varepsilon_2 =< \textsc{GroupStatus}\colon t, \varepsilon, t_2 > \\
\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \textit{adjoin}\ \boxed{\mathcal{R}\kappa\varepsilon_1}\,(\mu, \gamma_1)(\theta^*, \sigma[\mu \times \gamma \rightarrow \varepsilon_2], \tau[\mu \times \mu \rightarrow t], \varepsilon^*) \\
\qquad\qquad \mathbf{else}\ \textit{ensure-parentage}(\gamma, \boxed{\mathcal{G}_W\kappa\gamma\varepsilon}\,)(\mu, \gamma_1)(\theta^*, \sigma, \tau, \varepsilon^*) \\
\qquad\qquad \mathbf{endif} \\
\quad \mathbf{else}\ \textit{remote-adjoin}\ \textit{site}(\gamma)\boxed{\mathcal{G}_W\texttt{<return}\ \kappa\ \mu\texttt{>}\gamma\varepsilon}\,(\mu, \gamma_1)(\theta^*, \sigma, \tau, \varepsilon^*) \\
\quad \mathbf{endif}
\end{array}$$

Two new functions were introduced in the previous definition: *group-parentage-known?* and *ensure-parentage*. The first one checks whether a group status is certain i.e., known for sure on a site. The second is called whenever the first answers false, it then tries to remedy to this situation before restarting the actual computation, more on it below.

The function *group-parentage-known?* checks whether a group has a perfectly known status on a site in a given store and at a precise time. The initial group γ_{init} is always perfectly known to be running since it is a value that the program cannot grab and therefore cannot mutate. A group status is certain if it is local or validly cached, if it is coherent with its supergroup and if its supergroup status is also certain. This is the case if the date of the last change it perceived from its supergroup is not less than the date of the last change as recorded in the group status of the supergroup, see again figure 7.

$$\begin{array}{l}
\textit{group-parentage-known?}(\gamma, \mu, \sigma, \tau) = \\
\textit{check}(\gamma)\,\mathbf{whererec}\ \textit{check} = \\
\quad \lambda\gamma_1.\ \mathbf{if}\ \gamma_1 = \gamma_{init} \\
\qquad\quad \mathbf{then\ true} \\
\qquad\quad \mathbf{else}\ \ \textit{check}(\textit{super}(\gamma_1)) \wedge \\
\qquad\qquad\quad \mathbf{if}\ \mu = \textit{site}(\gamma_1) \\
\qquad\qquad\quad \mathbf{then}\ \ \mathbf{let}\ <\textsc{GroupStatus}\colon t, \varepsilon, t_1 >= \sigma(\mu, \gamma_1) \\
\qquad\qquad\qquad\quad \mathbf{in}\ t_1 \geq \textit{group-last-change-date}(\textit{super}(\gamma_1), \mu, \sigma) \\
\qquad\qquad\quad \mathbf{else}\ \ \mathbf{let}\ \epsilon = \sigma(\mu, \gamma_1)
\end{array}$$

in **if** $\epsilon \in$ **CachedValue**
  then **let** $<$ CACHEDVALUE: $t, \varepsilon >= \epsilon$
    in $t = \tau(\mu, \mathit{site}(\gamma_1)) \wedge$
      let $<$ GROUPSTATUS: $t_1, \varepsilon_1, t_2 >= \varepsilon$
      in $t_2 \geq \mathit{group\text{-}last\text{-}change\text{-}date}(\mathit{super}(\gamma_1), \mu, \sigma)$
  else false
  endif
endif
endif

If some group status is uncertain then the current computation is suspended while this group status and its consequences are established. The *ensure-parentage* is the most complex function of this paper since it has to deal with multiple cases. It first looks for the first uncertain group starting from top to bottom. If the uncertain group is local, two more cases are recognized whether the known supergroup is also local or not. In both cases, the running status of the supergroup has to be propagated to the current group, this is achieved by fetching the status of the supergroup (with a group read request identified as $\mathcal{G}_{\mathcal{R}}$) with a continuation that will propagate this running status to the uncertain group.

$\mathit{ensure\text{-}parentage}(\gamma, \chi) =$
$\lambda\mu\gamma_1.\lambda\theta^*\sigma\tau\varepsilon^*.\mathit{climb}(\gamma, \mathit{super}(\gamma))$ **whererec** $\mathit{climb} =$
$\lambda\gamma_2\gamma_3.$ **if** $\mathit{group\text{-}parentage\text{-}known?}(\gamma_3, \mu, \sigma, \tau)$
  then **if** $\mu = \mathit{site}(\gamma_2)$
    then **if** $\mu = \mathit{site}(\gamma_3)$
      then *adjoin* $\boxed{\mathcal{G}_R\texttt{<propagate } \gamma_2\ \chi\ \gamma_1\texttt{>}\gamma_3}$ $(\mu, \gamma_{init})(\theta^*, \sigma, \tau, \varepsilon^*)$
      else *remote-adjoin* $\mathit{site}(\gamma_3)$
        $\boxed{\mathcal{G}_R\texttt{<return <propagate } \gamma_2\ \chi\ \gamma_1\texttt{> } \mu\texttt{>}\gamma_3}$ $(\mu, \gamma_{init})(\theta^*, \sigma, \tau, \varepsilon^*)$
      endif
    else **let** $\epsilon = \sigma(\mu, \gamma_2)$
      in **if** $\epsilon \in$ **CachedValue**
        then **let** $<$ CACHEDVALUE: $t, \varepsilon >= \epsilon$
          in **if** $t = \tau(\mu, \mathit{site}(\gamma_2))$
            then
    let $\theta = \texttt{<propagate } \gamma_2$ $\boxed{\mathcal{G}_R\texttt{<return <group-cache } \gamma_2\ \chi\ \gamma_1\texttt{> } \mu\texttt{>}\gamma_2}$ $\gamma_{init}\texttt{>}$
    in *remote-adjoin* $\mathit{site}(\gamma_3)$ $\boxed{\mathcal{G}_R\texttt{<return } \theta\ \mathit{site}(\gamma_2)\texttt{>}\gamma_3}$ $(\mu, \gamma_{init})(\theta^*, \sigma, \tau, \varepsilon^*)$
            else *remote-adjoin* $\mathit{site}(\gamma_2)$
        $\boxed{\mathcal{G}_R\texttt{<return <group-cache } \gamma_2\ \chi\ \gamma_1\texttt{> } \mu\texttt{>}\gamma_2}$ $(\mu, \gamma_{init})(\theta^*, \sigma, \tau, \varepsilon^*)$
            endif
        else *remote-adjoin* $\mathit{site}(\gamma_2)$
        $\boxed{\mathcal{G}_R\texttt{<return <group-cache } \gamma_2\ \chi\ \gamma_1\texttt{> } \mu\texttt{>}\gamma_2}$ $(\mu, \gamma_{init})(\theta^*, \sigma, \tau, \varepsilon^*)$
        endif
    endif
  else $\mathit{climb}(\gamma_3, \mathit{super}(\gamma_3))$
  endif

If the uncertain group is not local then its cached group status might be obsolete or unknown. In that case we simply emit towards its birth site a request to read the uncertain group status with a new continuation that will try to cache this group status when coming back (and if still up to date). This behavior is similar to the behavior of boxes as previously seen.

$$
\begin{array}{l}
\mathit{step}\ \boxed{\mathcal{R}\texttt{<group-cache}\ \gamma\ \chi\ \gamma_1\texttt{>}\epsilon}\ = \\
\lambda\mu\gamma_2.\lambda\theta^*\sigma\tau\varepsilon^*.\ \textbf{let}\ <\textsc{CachedValue}:\ t, \varepsilon >= \epsilon \\
\qquad\qquad\quad \textbf{in}\ <\textsc{State}:\ \{\langle\!\langle\chi, \mu, \gamma_1\rangle\!\rangle\} \cup \theta^*,\ \textbf{if}\, t = \tau(\mu, \mathit{site}(\gamma)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ \sigma[\mu \times \gamma \rightarrow \epsilon] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{endif}\ , \tau, \varepsilon^* >
\end{array}
$$

If the uncertain group is not local and if its cached group status is valid then that means that it is incoherent wrt its supergroup i.e., it perceived a modification of the running status of its supergroup it has still not propagated. The action to take there is complex since the current site emits a request to the birth site of the group asking it to update itself with respect to its supergroup. In this case, a remote site perceives that a group is incoherent before the birth site of that group.

When a site receives a $\mathcal{G}_{\mathcal{R}}$ request, it has to return the actual group status of a specified group. It can only do that if the group status is certain otherwise the request is suspended while *ensure-parentage* is run.

$$
\begin{array}{l}
\mathit{step}\ \boxed{\mathcal{G}_R\kappa\gamma}\ = \\
\lambda\mu\gamma_1.\lambda\theta^*\sigma\tau\varepsilon^*.\ \textbf{if}\, \mathit{group\text{-}parentage\text{-}known?}(\gamma, \mu, \sigma, \tau) \\
\qquad\qquad\qquad \textbf{then}\ \mathit{adjoin}\ \boxed{\mathcal{R}\kappa <\textsc{CachedValue}:\ \tau(\mu,\mu), \sigma(\mu,\gamma) >}\ (\mu, \gamma_1)(\theta^*, \sigma, \tau, \varepsilon^*) \\
\qquad\qquad\qquad \textbf{else}\ \mathit{ensure\text{-}parentage}(\gamma, \boxed{\mathcal{G}_R\kappa\gamma}\ )(\mu, \gamma)(\theta^*, \sigma, \tau, \varepsilon^*) \\
\qquad\qquad\qquad \textbf{endif}
\end{array}
$$

Finally the propagation of the running status from a supergroup to a group just accounts to mutate accordingly the running status of the group provided all its parentage is up to date. When this is ensured and the incoherency is still present then the running status is propagated otherwise it is now superseded by already made new group mutations and simply forgotten.

$$
\begin{array}{l}
\mathit{step}\ \boxed{\mathcal{R}\texttt{<propagate}\ \gamma\ \chi\ \gamma_1\texttt{>}\epsilon}\ = \\
\lambda\mu\gamma_2.\lambda\theta^*\sigma\tau\varepsilon^*. \\
\quad \textbf{let}\ <\textsc{CachedValue}:\ t, \varepsilon >= \epsilon \\
\quad \textbf{in}\ \textbf{let}\ <\textsc{GroupStatus}:\ t_1, \varepsilon_1, t_2 >= \varepsilon \\
\qquad \textbf{in}\ \textbf{let}\ \gamma_3 = \mathit{super}(\gamma) \\
\qquad\quad \textbf{in}\ \textbf{let}\ \sigma_1 = \textbf{if}\ \mu \neq \mathit{site}(\gamma_3) \wedge \\
\qquad\qquad\qquad\qquad\qquad t = \tau(\mu, \mathit{site}(\gamma_3)) \\
\qquad\qquad\qquad\qquad \textbf{then}\ \sigma[\mu \times \gamma_3 \rightarrow \epsilon] \\
\qquad\qquad\qquad\qquad \textbf{else}\ \sigma \\
\qquad\qquad\qquad\qquad \textbf{endif} \\
\qquad\qquad \textbf{in}\ \textbf{if}\, \mathit{group\text{-}parentage\text{-}known?}(\gamma_3, \mu, \sigma_1, \tau) \\
\qquad\qquad\quad \textbf{then}\ \textbf{if}\ \textbf{let}\ <\textsc{GroupStatus}:\ t_3, \varepsilon_2, t_4 >= \sigma_1(\mu, \gamma) \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ t_4 \geq \mathit{group\text{-}last\text{-}change\text{-}date}(\gamma_3, \mu, \sigma_1) \\
\qquad\qquad\qquad \textbf{then}\ <\textsc{State}:\ \{\langle\!\langle\chi, \mu, \gamma_1\rangle\!\rangle\} \cup \theta^*, \sigma_1, \tau, \varepsilon^* > \\
\qquad\qquad\qquad \textbf{else}\ \textbf{let}\ t_3 = 1 + \tau(\mu, \mu) \\
\qquad\qquad\qquad\qquad \textbf{in}\ \textbf{let}\ \varepsilon_2 =< \textsc{GroupStatus}:\ t_3, \varepsilon_1, t_1 >
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\mathbf{in}\ <\textsc{State}\text{:}\ \{\langle\!\langle \chi, \mu, \gamma_1 \rangle\!\rangle\} \cup \theta^*, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \sigma_1[\mu \times \gamma \rightarrow \varepsilon_2], \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \tau[\mu \times \mu \rightarrow t_3], \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \varepsilon^* > \\
\qquad\qquad\mathbf{endif} \\
\mathbf{else}\ <\textsc{State}\text{:}\ \{\langle\!\langle \chi, \mu, \gamma_1 \rangle\!\rangle\} \cup \theta^*, \sigma_1, \tau, \varepsilon^* > \\
\mathbf{endif}
\end{array}
$$

4.2 Scheduler

It is now possible to explicit the scheduler. A thread can only be run if all the groups it belongs to are running. If it is not the case then the missing informations are asked for and the thread is temporarily suspended meanwhile i.e., it is wrapped into a new type of computation: the execution, marked as $\boxed{\mathcal{X}\chi\gamma}$, of computation χ under sponsorship of γ.

Among the whole set of threads, the scheduler just *try* one of them, obtain a new state and reiterates the process until all threads are *not-runnable* thus ending the whole program. In fact, a thread can be run if the first group to which it belongs is certain and running. This is sufficient since *group-parentage-known?* implies that all supergroups are running.

Observe that missing informations on groups are asked for under the sponsorship of γ_{init} which cannot be suspended.

$$
\begin{array}{l}
\mathit{try}(\theta) = \\
\lambda\theta^*\sigma\tau\varepsilon^*.\ \mathbf{let}\ \langle\!\langle \chi, \mu, \gamma \rangle\!\rangle = \theta \\
\qquad\quad \mathbf{in}\ \ \mathbf{if}\ \mathit{group\text{-}parentage\text{-}known?}(\gamma, \mu, \sigma, \tau) \\
\qquad\qquad\ \mathbf{then}\ \ \mathbf{if}\ \mathit{group\text{-}running?}(\gamma, \mu, \sigma, \tau) \\
\qquad\qquad\qquad\quad \mathbf{then}\ \mathit{step}\ \chi(\mu, \gamma)(\theta^*, \sigma, \tau, \varepsilon^*) \\
\qquad\qquad\qquad\quad \mathbf{else}\ \mathit{not\text{-}runnable} \\
\qquad\qquad\qquad\quad \mathbf{endif} \\
\qquad\qquad\ \mathbf{else}\ \mathit{ensure\text{-}parentage}(\gamma, \boxed{\mathcal{X}\chi\gamma})(\mu, \gamma)(\theta^*, \sigma, \tau, \varepsilon^*) \\
\qquad\qquad\ \mathbf{endif}
\end{array}
$$

$$
\begin{array}{l}
\mathit{group\text{-}running?}(\gamma, \mu, \sigma, \tau) = \\
\gamma = \gamma_{init} \vee\ \ \mathbf{if}\ \mu = \mathit{site}(\gamma) \\
\qquad\qquad\quad \mathbf{then}\ \mathbf{let}\ <\textsc{GroupStatus}\text{:}\ t, \varepsilon, t_1 >= \sigma(\mu, \gamma) \\
\qquad\qquad\qquad\qquad \mathbf{in}\ \varepsilon \\
\qquad\qquad\quad \mathbf{else}\ \mathbf{let}\ <\textsc{CachedValue}\text{:}\ t, \varepsilon >= \sigma(\mu, \gamma) \\
\qquad\qquad\qquad\quad \mathbf{in}\ \mathbf{let}\ <\textsc{GroupStatus}\text{:}\ t_1, \varepsilon_1, t_2 >= \varepsilon \\
\qquad\qquad\qquad\qquad\ \mathbf{in}\ \varepsilon_1 \\
\qquad\qquad\quad \mathbf{endif}
\end{array}
$$

$$
\mathit{step}\ \boxed{\mathcal{X}\chi\gamma} = \lambda\mu\gamma_1.\mathit{adjoin}\ \chi(\mu, \gamma)
$$

4.3 Discussion

The above group coherency protocol is supported by the previous box protocol. The information on groups is propagated lazily but with an additional constraint related to the tree shape of groups. Any action on a group (scheduling, reading or mutating its running status) is preceded by a check verifying whether the information on that group is up to date. This check ensures a sequentialization of mutations in case of simultaneous `pause!` or `awake!` on embedded groups.

Due to their importance, groups are examples of objects which can be associated to dedicated clocks. To have on each site a second clock recording mutations on local groups independently of local boxes is more efficient since group mutation are more rare and because a box mutation will not invalidate the local groups and thus does not affect the scheduler.

This protocol is rather naïve and may be improved: time-updates messages as encapsulated within *remote-adjoin* are actually run under the sponsorship of a group. If the group is uncertain, then the time-update is deferred instead of being immediately processed. This can be optimized. As for boxes, information on an uncertain group may be asked more than once and would gain if shared. This has to be done in a real implementation to spare bandwidth.

Our implementation of protocols supposes that caches have an arbitrary size since they are never flushed. This would require to abandon the functional representation of the store since it is not possible to inspect the domain of a function in a denotational framework.

5 Extensions

The ICSLAS language tries to offer a convenient framework to express distributed computations. This section sketches two new techniques to manage increasing numbers of sites.

5.1 Adding sites

Messages carry the set of clocks expressing the perception the emitting site had of all proper clocks. It is rather simple to adjoin a new site to the net of already existing sites. When a site μ decides to involve a new site μ' into the computation it participates in, it only has to mention the new clock of μ' in all the messages it emits. When a third site hears about the existence of μ', it simply adds it to the set of sites it is aware of and it then propagates the existence of μ' (through μ''s clock) in all the messages it now emits.

When a message is emitted, the emitting site pairs all the sites it knows with the clock value it thinks they are. These pairs are appended to the content of the message and transmitted. When a message is received and as pairs (site, clock) are decoded, the existence of new sites is lazily propagated.

If FIFO channels (say TCP connections) were used between two sites then those sites may avoid to exchange any stale clock. A simple trick to achieve this is to time-stamp (with another clock, private to the emitting site and never exchanged) every clock update and to record, associated to the connection, the last time-stamp associated to the clocks sent through this connection. Only clocks incremented after that time-stamp need then to be exchanged. This reduces message size but requires FIFOness.

The problem of removing a site from a computation is much much complex since all the data held by this site have to be saved or migrated towards elsewhere. We do not have actually a good solution for this problem.

5.2 Clustering sites

Message size suffers from the number of clocks that have to be propagated. To limit this number, it is possible to gather sites within clusters. Suppose that there exists a single gateway that intercepts all the messages that are emitted by the sites from the cluster to the sites out of the cluster; the gateway does not have to be aware of the internal messages exchanged by sites inside the cluster. Suppose also, to simplify, that this gateway holds a single clock which represents the perception the gateway has of the sum of all the clocks of all the internal sites. Whenever a message is issued from the cluster to the outside, the gateway updates the perception it has of the internal clocks, increments its proper clock to reflect the sum of all the internal clocks; it then removes these clocks from the message, replaces them with its single proper clock and resumes the transmission of the message. In a way, internal clocks are folded into a single one for the exclusive view of the outside of the cluster: a cluster is viewed as a regular site from the outside.

A single change inside the cluster increments the gateway's clock and therefore invalidates all the cached boxes outside the cluster. It is, of course, possible to let the gateway possess more than one clock to limit the invalidations. There again a good mapping from mutable data to their associated clocks is necessary to achieve good performances.

This idea was developed conjunctly with José Piquer.

6 Conclusion

Besides the lazy invalidation protocol already presented in [Que94], this paper proposes some original results:

- a functional specification of an abstract machine for an essential applicative, concurrent and distributed language,
- a protocol for the imperative control of groups of cooperating tasks.
- some new ideas to manage greater number of sites.

Although the definitions given above have a strong functional flavor, they can help to figure out how they can be implemented. For instance, distribution is entirely encapsulated within *remote-adjoin* while concurrency is private to *adjoin*. The rest of the definitions do not need to bother about these aspects.

Allied to the GC technique described in [LQP92] and the indirect reference count technique of [Piq91], we expect to release soon a first version of the ICSLAS language. Due to the avoidance of broadcast technology, it seems adapted to the niche of loosely coupled computers, the laziness of its protocols allows it to easily cope with short-term communication failures provided no message is lost. Since ICSLAS belong to the Lisp family of dialects, it has all its usual features: macros, high-order functions, dynamic evaluation, objects and classes [Que93], interactive debugging tools written in Lisp etc.

Bibliography

[AMST92] Gul Agha, Ian Mason, Scott Smith, and Carolyn Talcott. Towards a theory of actor computation. In W R Cleaveland, editor, *CONCUR '92 – Proceedings of the Third International Conference on Concurrency Theory*, LNCS630, pages 565–579. Springer-Verlag, 1992.

[CR91] William Clinger and Jonathan A Rees. The revised^4 report on the algorithmic language scheme. *Lisp Pointer*, 4(3), 1991.

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

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

[Hen80] Peter Henderson. *Functional Programming, Application and Implementation*. International Series in Computer Science. Prentice-Hall, 1980.

[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.

[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.

[MH90] Thanasis Mitsolides and Malcolm Harrison. Generators and the replicator control structure in the parallel environment of alloy. In *PLDI '90 –ACM SIGPLAN Programming Languages Design and Implementation*, pages 189–196, White Plains (New-York USA), 1990.

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

[ODL93] Katia Obraczka, Peter B Danzig, and Shih-Hao Li. Internet resource discovery services. *Computer*, 26(9):8–24, September 1993.

[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.

[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.

[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.

[Que93] 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. Locality, causality and continuations. In *LFP '94 – ACM Symposium on Lisp and Functional Programming*, pages 91–102, Orlando (Florida, USA), June 1994. ACM Press.

[Sch86] David A Schmidt. *Denotational Semantics, a Methodology for Language Development.* Allyn and Bacon, 1986.

[Sto77] Joseph E Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge Massachussetts USA, 1977.

This article was processed using the LaTeX macro package with LLNCS style