A Concurrent and Distributed Extension of Scheme

Christian Queinnec*
École Polytechnique & INRIA–Rocquencourt

Parallel Architectures and Languages Europe, PARLE '92, jun 92, Paris (France), LNCS 605, Springer-Verlag, pp 431–446.

Abstract

The Lisp family of languages has traditionally been a privileged domain where linguistic experiments were done, this paper presents a new dialect offering concurrency and distribution. This dialect, nicknamed CD-Scheme, has been designed above Scheme with as few as possible features to allow a great expressiveness but still to retain the original consistency and simplicity of Scheme. We explicitly show how common parallel constructs can be written in regular CD-Scheme.

A denotational semantics is also presented that expresses the detailed meaning of assignment, data mutation, continuations in presence of concurrency and distribution. This semantics offers a basis to understand new proposals of concurrent or distributed features and may be used to justify compiler optimizations or implementation techniques. The proposed set of features of CD-Scheme can be also used to extend languages other than Scheme.

CD-Scheme is a concurrent, distributed and conservative extension of Scheme, a dialect of Lisp [CR91] which is a strict sequential untyped language enriching λ-calculus with assignment, mutable data structures and first-class continuations. The design of CD-Scheme has been driven by two rules: *(i)* the computational model must be simple *(ii)* the language must be expressive.

CD-Scheme offers the illusion of a global data-space. This globally accessible but physically distributed data-space respects the single (and simple) workspace tradition (and semantics) of Lisp. CD-Scheme does not favor particular architectures such as shared memory but has been designed to provide an easy language to write programs that run on loosely coupled machines.

According to the philosophy of Scheme, we only introduced the minimal number of new features to deal with concurrency and distribution. To add a small amount of features allows to understand more thoroughly their impact on the base language, their interaction, as well as it simplifies the learning effort of future users. We also think that a few essential features unrestrictively combinable are a more challenging implementation goal. The following features were added to Scheme:

1. assignment (the `set!` special form) atomically exchanges old and new values
2. the `breed` function controls concurrency
3. distribution (process and data migration) is achieved by the `remote` function
4. groups of tasks can be created and imperatively paused or awaked.

Scheme is an undeterministic language and so is its extension: CD-Scheme. To implement such a language, or simply, to master its undeterminacy, a precise semantics must be given. The paper presents a denotational semantics that specify both concurrency and distribution.

Numerous features such as futures [Hal84], queue-based functions [GM84], active objects [AR89], M-structures [BNA91] etc. have been proposed to ease concurrent programming. CD-Scheme allows to program them all as well as variants [KW90]. More traditional constructs such as busy waiting loops or semaphores

*Postal Address: Laboratoire d'Informatique de l'École Polytechnique (URA 1432), 91128 Palaiseau Cedex, France. Email Address: `queinnec@poly.polytechnique.fr`. This work has been partially funded by Greco de Programmation.

can also be programmed thus conferring to CD-Scheme surprising capabilities with respect to its very small number of features.

The main contributions of the paper are: *(i)* a definition of a mostly functional, very simple but expressive language, *(ii)* an analysis of continuations in a concurrent setting, *(iii)* a denotational semantics where concurrency and distribution are explicitly expressed.

The rest of the paper is structured as follows: section 1 describes the features added to Scheme to define CD-Scheme. Section 2 discusses some implementation points showing that CD-Scheme is not an unreasonable language. The denotational semantics is presented in section 3. Examples of programs in CD-Scheme appear in section 4. A comparison with related works and a conclusion follow.

1 CD-Scheme definition

CD-Scheme is grounded on Scheme [CR91]. Scheme is a strict sequential untyped language with assignment, mutable data and first-class continuations. Its very clean and simple semantics makes Scheme easy to extend and allows to appreciate the impact of the extensions. This section focuses only on the new features of CD-Scheme, significative examples appear in section 4.

We decide to keep the Scheme basis intact and to not forbid assignment nor data mutation as in pure functional programming. Existing hardware, libraries and operating systems require side-effects to be operated. CD-Scheme is therefore a "mostly functional programming language" where side-effects exist to be appropriately encapsulated, offering undeterminacy where needed. Following the tradition of the Lisp family, side-effects are also useful to express the details of the implementation of CD-Scheme in CD-Scheme itself and thus confers to CD-Scheme reflective capabilities that make it easy to extend.

1.1 Concurrency

Concurrency is introduced by means of the `breed` function. Programs that do not use `breed` keep their regular semantics: CD-Scheme is therefore a conservative extension to Scheme. Some analyses have been reported, [Hal89], that automatically find places where concurrency might be usefully introduced. As in regular Lisp systems, macros can be designed that annotate programs with appropriate calls to `breed`.

The `breed` function is a low-level but powerful facility that can be used (see §4) to define more elaborate constructs such as `pcall` [Hal84] or `qlambda` [GM84]. The `breed` function takes some thunks as arguments, creates independent processes to invoke them with a suicide continuation. When these processes are created, the process invoking `breed` suicides itself. The `suicide` function is therefore definable in term of `breed` as:

```
(define (suicide . arguments) (breed))
```

The effect of `breed` is to create and detach processes. The different processes are independent, they have no synchronization constraint. They can return information through continuation as discussed in the next section.

CD-Scheme does not enforce a particular scheduling strategy. Therefore `breed` is not even compelled to create new processes and may be implemented, in regular Scheme, with a continuation-based scheduler as in [Wan80]. The `breed` function introduces undeterminacy which is already present (but to a lesser extent) in Scheme since the evaluation order of the terms of a functional application is unspecified.

1.2 Continuations

Continuations are first-class objects in Scheme. They are created (captured) by means of the `call/cc` function. Scheme continuations have an indefinite extent which confers them numerous usages from escape constructs such as `catch/throw` *à la* COMMON LISP, to coroutines [HFW84].

A continuation can be multiply invoked in Scheme. A continuation can also be concurrently invoked since CD-Scheme is a concurrent language. When a process invokes a continuation, this has no effect on the other processes that may share this continuation. Consider for instance the following generator in the spirit of [MH90]:

```
(define (//iota start stop)
```

```
(call/cc (lambda (return)
           (define (iota start stop)
             (if (< start stop)
                 ;;concurrently return a number and continue to generate the others
                 (breed (lambda () (return start))
                        (lambda () (iota (+ 1 start) stop)) )
                 (suicide) ) )
           (iota start stop) )) )
```

A process that invokes (//iota 0 4) is killed while 4 new processes are sequentially created, each of them appear to return 0, 1, 2 or 3 as the value of the (//iota 0 4) form. For instance (display (//iota 0 4)) prints in some undetermined order the integers 0, 1, 2 and 3.

1.3 Atomic exchange

In regular Scheme, the value returned by an assignment is unspecified, we choose to return the former value of the binding. The form (set! *name expression*) first computes the value of *expression* then atomically exchanges this value with the content of the location bound to *name* and returns the latter. The value returned by the first assignment on a variable is still unspecified. This exchange effect can be used as a test-and-set instruction to program a busy-waiting loop as in §4.

1.4 Migration

Computations can migrate by means of the remote function which implements a kind of remote procedure call (RPC) [SG90]. When a form such as (remote F A B) is to be evaluated, F, A and B are sequentially evaluated yielding values f, a and b. These values are then possibly migrated on a processor of the neighborhood (the precise choice is left unspecified) where the migrated f is applied on the migrated a and b. The difference with a regular RPC is that the continuation k of the original (remote ...) form is also migrated and will be directly resumed, from the neighbor, with the result of the application.

We decide to stick to a very simple "migration" model where entities never move and always reside on the site where they were created. When an entity o on processor μ is to be migrated to the processor μ', a *remote pointer* $<\mu, o>$ is created on the remote processor μ'. This remote pointer can be read as "go to μ and find o". Extra rules can be added to simplify remote remote pointers, for example on machine μ'', $<\mu', <\mu, o>>$ is no more than $<\mu, o>$ if processor μ is accessible from μ''.

An effective implementation of this theoretical migration model does not exclude the "real" migration of entities provided the respect of the semantics, nor it excludes to multiply duplicate immutable entities such as small integers, booleans, characters, immutable cons cells, functions codes or universally known primitives like + or cons. To achieve this, the mutability of all user-created data can be taken into account and specialized migration policy can be devised. Fortunately, the percentage of mutable data is usually small.

Some primitives are *geographically strict*. They impose, to be applied, that their argument are local and not remote. Such strict computations are handled through the following rule:

geographical commutation rule: Applying the geographically strict primitive p on $<\mu', o>$ with continuation k on processor μ is just applying p on o with continuation $<\mu, k>$ on processor μ'.

Examples of strict computations include looking up an identifier in a remote environment (activation record), sending a value to a remote continuation, storing a value in a remote mutable object ... Almost all CD-Scheme primitive computations follow this rule except binary comparison primitives such as eq?, eqv? ... which need a special implementation when comparing two remote entities coming from different directions. Observe that computations carry their continuation and therefore are not bound to the sites through which they pass. To migrate computations towards data is not unreasonable, especially when data are files or other site-bound resources, but does not exclude to move some objects to lessen communication cost.

1.5 Controlling groups of tasks

Tasks can be easily created by means of `breed` but not enough control is offered to program the classical parallel-or operator, also known as `either` [Pri80]. The form `(either` $expression_0$ $expression_1$`)` concurrently creates two processes to evaluate the two expressions, returns the value of the first one which returns true and kills the other process since it is no more useful. We introduced the concept of *group*, inspired from the *sponsor* of [Osb90] and *controller* of [HDB90] but also related to the UNIX[1]'s concept of groups of tasks.

The `sponsor` function starts a computation under the responsibility of a group:

`(sponsor (lambda (`*group*`)` *expression*`))`

This form creates an object called a group, binds it to the variable *group* and evaluates *expression* under its control. All tasks that will be created when evaluating *expression* will belong to this group. Tasks may belong to more than one group since groups can be embedded.

A group can pause or awake the tasks it controls by means of the associated functions `pause!` and `awake!`. Pausing a group means that all the tasks that belong to this group will be suspended wherever they geographically are and how many they are. Conversely all the tasks of a group can be resumed when wakening the group. The group object itself can be inspected, through the predicate `paused?`, to determine if it is paused or awaked. Another predicate, `in-group?` tells if the current computation participates to a particular group or not.

A key point is that groups are first-class objects while processes are not. A computation can be controlled as a whole. Processes are behind-the-scene entities that cannot be captured. Continuations are not processes: a continuation is a return point in a dynamic chain of callers. A process can be suspended by taking its continuation and not invoking it; it can be resumed as soon as its associated continuation is invoked. Moreover a same continuation can be invoked by two different processes.

1.6 Linguistic conclusions

To sum up, CD-Scheme only adds to Scheme the refined behavior of continuations and assignment, as well as a bunch of primitive functions: `breed`, `remote`, `sponsor`, `pause!`, `awake!`, `paused?` and `in-group?`. CD-Scheme is a conservative extension of Scheme which concepts can be unrestrictively combined. It is therefore an easy to learn and powerful language.

2 Implementation hints

CD-Scheme does not require sophisticated techniques to be naïvely implemented. Of course, this does not achieve speed for which more clever compile-time and run-time techniques are required. This section only presents naïve techniques and some hints on more elaborate techniques which we are currently investigating.

Continuations are heap-based rather than stack-based i.e.they aare represented by linked frames allocated in the heap. Such explicit continuations are easy to migrate since their representation is explicit. Migration of immutable objects is a simple copy. Mutable objects are not copied and never move, only remote pointers are created. All accessor (reader or writer) functions are generic: when the object is local, the appropriate action is performed otherwise a geographical commutation is initiated. Assignments can be turned into mutation of activation records via the box transformation of [KKR+86].

A more aggressive technique we plan to use is to copy objects whether mutable or not in order to improve data reading. For each mutable object exists a master copy which is the sole instance to be mutated, its site ensures the serialization of mutations. Whenever a master copy is mutated, a message is broadcasted from the master site to invalidate all copies [Piq91].

Multiprocessing is controlled by a non preemptive scheduler. Periodically on each processor and after a fixed number of invocations, the local scheduler is invoked and a new process is selected. A process object contains all the informations needed to resume the process and, in particular, the list of groups controlling it. When a group of tasks is paused (resp. awaked), a pausing (resp. awakening) order is broadcasted, from the site where the group was created, to suspend (resp. resume) all processes belonging to that group. When the broadcast is fully acknowledged, the `pause!` or `awake!` function returns.

[1] UNIX is a registered trademark of Novell.

$\mathcal{E} : \mathbf{Prog} \rightarrow \mathbf{Env} \times \mathbf{Cont} \rightarrow \mathbf{Step}$	$\pi \in \mathbf{Prog} = \{\textit{The set of programs}\}$
$\rho \in \mathbf{Env} = \mathbf{Id} \rightarrow \mathbf{Addr}$	$\nu \in \mathbf{Id} = \{\textit{The set of identifiers}\}$
$\alpha \in \mathbf{Addr} = \mathbf{Loc} \times \mathbf{Mach}$	$l \in \mathbf{Loc} = \{\textit{The set of locations}\}$
$\mu \in \mathbf{Mach} = \{\textit{The set of machines}\}$	$\kappa \in \mathbf{Cont} = \mathbf{Val} \rightarrow \mathbf{Step}$
$\varepsilon \in \mathbf{Val} = \mathbf{Group} + \mathbf{Fun} + \mathbf{Int} + \ldots$	$\gamma \in \mathbf{Group} = \mathbf{Addr}$
$\phi \in \mathbf{Fun} = \mathbf{Val}^* \times \mathbf{Cont} \rightarrow \mathbf{Step}$	$\psi \in \mathbf{Step} = \mathbf{Group}^* \rightarrow \mathbf{Mach} \times \mathbf{State} \rightarrow \mathbf{Ans}$
$\zeta \in \mathbf{State} = \mathbf{Mach} \rightarrow \mathbf{Local}$	$\chi \in \mathbf{Local} = \mathbf{Thread}^* \times \mathbf{Store}$
$\theta \in \mathbf{Thread} = \mathbf{Group}^* \times \mathbf{Step}$	$\sigma \in \mathbf{Store} = \mathbf{Loc} \rightarrow \mathbf{Val}$

Figure 1: Domains of CD-Scheme

For Garbage Collection, a solution inspired from [LQP92] can be adapted. This distributed, concurrent and fault-tolerant GC allows to dynamically add new processors to the machine.

3 Denotational Semantics

To cope with concurrency is difficult both at the user level and at the implementer level. For instance, what is the semantics of side-effects?, to what group of tasks belongs a particular continuation created under one group and applied under another? An answer to these questions is to exhibit a denotational semantics of CD-Scheme. This semantics has to express both the concurrency and the migration of computations. We choose to model concurrency by means of resumptions. Addresses are represented by a pair of a machine (a processor) and a location in its own store. The domains of the denotation appears in figure 1.

Computations are broken into atomic steps, called resumptions, belonging to **Step**. In order to define `pause!` or `awake!`, computations steps must be inquired for the list of groups controlling them. We thus pair computation steps with lists of groups in domain **Thread**. The state of the computation is represented by **State** which maps machines to their **Local** state: a store and a list of threads that might be run.

The denotation of a reference to a variable takes, as all denotations, an environment ρ and a continuation κ, is invoked under the sponsorship of some groups γ^* and finally runs on a machine μ within the global state ζ. It simply adjoins to the list of eligible threads of machine μ, a new thread $< \gamma^*, \kappa(\mathit{fetch}(\zeta, \mu_1, l)) >$ that will run later, under the same sponsorship, will fetch the content of the address $\alpha =< l, \mu_1 >$ associated to the variable ν in the lexical environment ρ i.e.the content of the location l on machine μ_1, and will return it to the original continuation κ.

$$\begin{array}{l} \mathcal{E}[\![\nu]\!] = \lambda\rho\kappa.\lambda\gamma^*.\lambda\zeta\mu.\ \mathbf{let}\ \alpha = \rho(\nu)\ \mathbf{in}\ \ \mathbf{let}\ l = \alpha \mid \quad \mathbf{and}\ \mu_1 = \alpha \mid \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \mathit{adjoin}(\zeta, \mu, < \gamma^*, \kappa(\mathit{fetch}(\zeta, \mu_1, l)) > \mathit{in}\mathbf{Thread}) \\ \mathit{fetch}(\zeta, \mu, l) = \zeta(\mu) \mid \quad (l) \\ \mathit{adjoin}(\zeta, \mu, \theta) = \mathit{compute}(\zeta[\mu \rightarrow< \theta : \zeta(\mu) \mid \quad {}_*, \zeta(\mu) \mid \quad > \mathit{in}\mathbf{Local}]) \end{array}$$

The alternative has a regular definition similar to its usual denotation [CR91]:

$$\begin{array}{l} \mathcal{E}[\![\texttt{(if}\ \pi\ \pi_1\ \pi_2\texttt{)}]\!] = \\ \lambda\rho\kappa.\lambda\gamma^*.\lambda\zeta\mu.\ \mathbf{let}\ \kappa_1 = \lambda\varepsilon.\lambda\gamma^*{}_1.\lambda\zeta_1\mu_1.\ \mathbf{if}\ \varepsilon\ \mathbf{then}\ \mathcal{E}[\![\pi_1]\!]\ \mathbf{else}\ \mathcal{E}[\![\pi_2]\!]\ \mathbf{endif}\ \ (\rho, \kappa)(\gamma^*{}_1)(\zeta_1, \mu_1) \\ \qquad\qquad\qquad\quad \mathbf{in}\ \mathcal{E}[\![\pi]\!](\rho, \kappa_1)(\gamma^*)(\zeta, \mu) \end{array}$$

The assignment computes a value ε and updates the location associated to the variable ν. Observe that the new ε and the former ε_1 values of the location are atomically swapped and the old value is sent back to the continuation. The *update* function takes care of updating the store of the appropriate machine where resides the location.

$$\begin{array}{l} \mathcal{E}[\![\texttt{(set!}\ \ \nu\ \pi\texttt{)}]\!] = \\ \lambda\rho\kappa.\lambda\gamma^*.\lambda\zeta\mu. \\ \qquad \mathbf{let}\ \kappa_1 = \lambda\varepsilon.\lambda\gamma^*{}_1.\lambda\zeta_1\mu_1.\ \ \mathbf{let}\ \alpha = \rho(\nu) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ \mathbf{let}\ l = \alpha \mid \quad \mathbf{and}\ \mu_2 = \alpha \mid \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ \mathbf{let}\ \varepsilon_1 = \mathit{fetch}(\zeta_1, \mu_2, l)\ \mathbf{and}\ \zeta_2 = \mathit{update}(\zeta_1, \mu_2, l, \varepsilon) \end{array}$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } \mathit{adjoin}(\zeta_2, \mu_1, < {\gamma^*}_1, \kappa(\varepsilon_1) > \mathit{in}\textbf{Thread}) \\
\qquad \textbf{in } \mathcal{E}[\![\pi]\!](\rho, \kappa_1)(\gamma^*)(\zeta, \mu) \\
\mathit{update}(\zeta, \mu, l, \varepsilon) = \ \textbf{let } \chi = \zeta(\mu) \textbf{ in } \zeta[\mu \rightarrow < \chi \mid \qquad {}_*, \chi \mid \qquad [l \rightarrow \varepsilon] > \mathit{in}\textbf{Local}]
\end{array}
$$

Functions are created with `lambda`. We ignore functions with a variable arity. The *new-locations* function is implementation dependent and, given a state and a machine, just allocates some fresh locations on this machine.

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(lambda } \nu^* \ \pi\texttt{)}]\!] = \\
\lambda\rho\kappa.\lambda\gamma^*.\lambda\zeta\mu. \ \ \textbf{let } \phi = \lambda\varepsilon^*\kappa_1.\lambda{\gamma^*}_1.\lambda\zeta_1\mu_1. \\
\qquad\qquad\qquad\qquad \textbf{let } \alpha^* = \mathit{new\text{-}locations}(\zeta_1, \mu_1, \#\nu^*) \\
\qquad\qquad\qquad\qquad \textbf{in} \quad \textbf{let } l^* = \mathit{map}(\lambda o.o \mid \quad , \alpha^*) \\
\qquad\qquad\qquad\qquad\qquad \textbf{in } \mathcal{E}[\![\pi]\!](\rho[\nu^* \xrightarrow{*} \alpha^*], \kappa_1)({\gamma^*}_1)(\mathit{update}^*(\zeta_1, \mu_1, l^*, \varepsilon^*), \mu_1) \\
\qquad\qquad\qquad \textbf{in } \mathit{adjoin}(\zeta, \mu, < \gamma^*, \kappa(\phi) > \mathit{in}\textbf{Thread}) \\
\mathit{update}^*(\zeta, \mu, l^*, \varepsilon^*) = \\
\ \ \textbf{if } \#l^* > 0 \textbf{ then} \quad \textbf{if } \#\varepsilon^* > 0 \textbf{ then } \mathit{update}^*(\mathit{update}(\zeta, \mu, l^* \downarrow_1, \varepsilon^* \downarrow_1), \mu, l^* \dagger 1, \varepsilon^* \dagger 1) \\
\qquad\qquad\qquad\quad\ \ \textbf{else } \mathit{wrong}(\text{“}\texttt{Too few images}\text{”}, l^*) \textbf{ endif} \\
\ \ \textbf{else} \quad \textbf{if } \#\varepsilon^* > 0 \textbf{ then } \mathit{wrong}(\text{“}\texttt{Too much points}\text{”}, \varepsilon^*) \textbf{ else } \zeta \textbf{ endif} \quad \textbf{endif} \\
\varphi[x \rightarrow y] = \lambda z. \ \textbf{if } z = x \textbf{ then } y \textbf{ else } \varphi(z) \textbf{ endif} \\
\varphi[x^* \xrightarrow{*} y^*] = \ \textbf{if } \#x^* > 0 \textbf{ then} \quad \textbf{if } \#y^* > 0 \textbf{ then } \varphi[x^* \downarrow_1 \rightarrow y^* \downarrow_1][x^* \dagger 1 \xrightarrow{*} y^* \dagger 1] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \textbf{else } \mathit{wrong}(\text{“}\texttt{Too few images}\text{”}, x^*) \textbf{ endif} \\
\qquad\qquad\qquad \textbf{else} \quad \textbf{if } \#y^* > 0 \textbf{ then } \mathit{wrong}(\text{“}\texttt{Too much points}\text{”}, y^*) \textbf{ else } \varphi \textbf{ endif} \quad \textbf{endif}
\end{array}
$$

The denotation of an application evaluates sequentially the terms (from left to right here) then applies the first value on the others. The auxiliary valuation function $\mathcal{E}^*$ returns the denotations of a list of programs.

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(}\pi \ \pi^*\texttt{)}]\!] = \\
\lambda\rho\kappa.\lambda\gamma^*.\lambda\zeta\mu. \ \ \textbf{let } \kappa_1 = \lambda\phi.\lambda{\gamma^*}_1.\lambda\zeta_1\mu_1. \\
\qquad\qquad\qquad\qquad \textbf{let } \kappa_1 = \lambda\varepsilon^*.\lambda{\gamma^*}_2.\lambda\zeta_2\mu_2.\mathit{adjoin}(\zeta_2, \mu_2, < {\gamma^*}_2, \phi(\varepsilon^*, \kappa) > \mathit{in}\textbf{Thread}) \\
\qquad\qquad\qquad\qquad \textbf{in } \mathcal{E}^*[\![\pi^*]\!](\rho, \kappa_1)({\gamma^*}_1)(\zeta_1, \mu_1) \\
\qquad\qquad\qquad \textbf{in } \mathcal{E}[\![\pi]\!](\rho, \kappa_1)(\gamma^*)(\zeta, \mu) \\
\mathcal{E}^*[\![]\!] = \lambda\rho\kappa.\lambda\gamma^*.\lambda\zeta\mu.\kappa(<>)(\gamma^*)(\zeta, \mu) \\
\mathcal{E}^*[\![\pi \ \pi^*]\!] = \\
\lambda\rho\kappa.\lambda\gamma^*.\lambda\zeta\mu. \ \ \textbf{let } \kappa_1 = \lambda\varepsilon.\lambda{\gamma^*}_1.\lambda\zeta_1\mu_1. \\
\qquad\qquad\qquad\qquad \textbf{let } \kappa_1 = \lambda\varepsilon^*.\lambda{\gamma^*}_2.\lambda\zeta_2\mu_2.\mathit{adjoin}(\zeta_2, \mu_2, < {\gamma^*}_2, \kappa(\varepsilon : \varepsilon^*) > \mathit{in}\textbf{Thread}) \\
\qquad\qquad\qquad\qquad \textbf{in } \mathcal{E}^*[\![\pi^*]\!](\rho, \kappa_1)({\gamma^*}_1)(\zeta_1, \mu_1) \\
\qquad\qquad\qquad \textbf{in } \mathcal{E}[\![\pi]\!](\rho, \kappa_1)(\gamma^*)(\zeta, \mu)
\end{array}
$$

This finishes the special forms of Scheme and CD-Scheme. Among the initial environment are some primitive functions dealing with continuations, concurrency and distribution, `call/cc` is the first one. Observe that a continuation is invoked under the sponsorship of the groups of the invoker rather than the groups of its creator.

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{call/cc}]\!] = \\
\lambda\varepsilon^*\kappa.\lambda\gamma^*.\lambda\zeta\mu. \ \ \textbf{let } \phi = \lambda{\varepsilon^*}_1\kappa_1.\lambda{\gamma^*}_1.\lambda\zeta_1\mu_1.\mathit{adjoin}(\zeta_1, \mu_1, < {\gamma^*}_1, \kappa({\varepsilon^*}_1 \downarrow_1) > \mathit{in}\textbf{Thread}) \\
\qquad\qquad\qquad\quad \textbf{in } \mathit{adjoin}(\zeta, \mu, < \gamma^*, \varepsilon^* \downarrow_1 (< \phi >, \kappa) > \mathit{in}\textbf{Thread})
\end{array}
$$

The `remote` function selects a machine towards which are migrated the terms of the application to remotely perform. Observe that the current machine can be selected, this proves to be useful in case there is only one machine. *All-machines* is the list of all available machines and is considered as constant throughout the denotation.

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{remote}]\!] = \\
\lambda\varepsilon^*\kappa.\lambda\gamma^*.\lambda\zeta\mu.\mathit{oneof}(\ \mathit{All\text{-}machines} \\
\qquad\qquad\qquad\qquad\quad , \lambda\mu_1\mu^*. \ \ \textbf{let } \kappa_1 = \lambda\varepsilon.\lambda{\gamma^*}_1.\lambda\zeta_1\mu_2.\mathit{adjoin}(\zeta_1, \mu, < {\gamma^*}_1, \kappa(\varepsilon) > \mathit{in}\textbf{Thread}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } \mathit{adjoin}(\zeta, \mu_1, < \gamma^*, \varepsilon^* \downarrow_1 (\varepsilon^* \dagger 1, \kappa_1) > \mathit{in}\textbf{Thread}) \ )
\end{array}
$$

The `breed` function adjoins multiple threads to the list of possible threads of the current machine.

$$\mathcal{E}[\![\texttt{breed}]\!] =$$
$$\lambda\varepsilon^*\kappa.\lambda\gamma^*.\lambda\zeta\mu.\ \ \mathbf{let}\ \kappa_1 = \lambda\varepsilon.\lambda{\gamma^*}_1.\lambda\zeta_1\mu_1.compute(\zeta_1)$$
$$\mathbf{in}\ adjoin*(\zeta, \mu, map(\lambda\varepsilon. < \gamma^*, \varepsilon(<>, \kappa_1) > in\mathbf{Thread}, \varepsilon^*))$$
$$adjoin*(\zeta, \mu, \theta^*) = compute(\zeta[\mu \rightarrow < \theta^* \S \zeta(\mu) \mid \quad {}_*, \zeta(\mu) \mid \quad > in\mathbf{Local}])$$

The `sponsor` function creates a new group and runs its argument under an enriched sponsorhsip. The group is associated to a location which will hold the list of suspended tasks in case the group is paused.

$$\mathcal{E}[\![\texttt{sponsor}]\!] =$$
$$\lambda\varepsilon^*\kappa.\lambda\gamma^*.\lambda\zeta\mu.\ \ \mathbf{let}\ \alpha = new\text{-}location(\zeta, \mu)$$
$$\mathbf{in}\ \ \mathbf{let}\ \gamma = < \alpha > in\mathbf{Group}$$
$$\mathbf{in}\ adjoin(update(\zeta, \mu, \alpha \mid \quad, \gamma), \mu, < \gamma : \gamma^*, \varepsilon^* \downarrow_1 (< \gamma >, \kappa) > in\mathbf{Thread})$$

The `awake!` function adjoins to the list of eligible threads, all the threads that are in the location associated to the specified group.

$$\mathcal{E}[\![\texttt{awake!}]\!] = \lambda\varepsilon^*\kappa.\lambda\gamma^*.\lambda\zeta\mu.$$
$$\mathbf{let}\ \alpha = \varepsilon^* \downarrow_1 \mid$$
$$\mathbf{in}\ adjoin(update(reduce(\lambda\zeta_1\Psi.\ \ \mathbf{let}\ \mu_1 = \Psi \downarrow_1\ \ \mathbf{and}\ \theta^* = \Psi \dagger 1$$
$$\mathbf{in}\ \zeta_1[\mu_1 \rightarrow < \theta^* \S \zeta_1(\mu_1) \mid \quad {}_*, \zeta_1(\mu_1) \mid \quad > in\mathbf{Local}]$$
$$, \zeta, fetch(\zeta, \alpha \mid \quad, \alpha \mid \quad)), \alpha \mid \quad, \alpha \mid \quad, <>), \mu, < \gamma^*, \kappa(\varepsilon^* \downarrow_1) > in\mathbf{Thread})$$
$$reduce(\phi, \upsilon, \upsilon^*) =\ \ \mathbf{if}\ \#\upsilon^* > 0\ \mathbf{then}\ \phi(reduce(\phi, \upsilon, \upsilon^* \dagger 1), \upsilon^* \downarrow_1)\ \mathbf{else}\ \upsilon\ \mathbf{endif}$$

The `pause!` function is more complex, it has to scan the entire set of machines to remove all eligible threads belonging to the specified group and store them in the location associated to the group so that they can be later resumed.

$$\mathcal{E}[\![\texttt{pause!}]\!] = \lambda\varepsilon^*\kappa.\lambda\gamma^*.\lambda\zeta\mu.$$
$$\mathbf{let}\ \alpha = \varepsilon^* \downarrow_1 \mid$$
$$\mathbf{in}\ compute(reduce(\lambda\zeta_1\mu_1.sort(\zeta_1(\mu_1) \mid \quad {}_*, \varepsilon^* \downarrow_1$$
$$, \lambda\theta^*{\theta^*}_1.update(\zeta_1[\mu_1 \rightarrow < {\theta^*}_1, \zeta_1(\mu_1) \mid \quad > in\mathbf{Local}], \alpha \mid \quad, \alpha \mid$$
$$, (\mu_1 : \theta^*) : fetch(\zeta_1, \alpha \mid \quad, \alpha \mid \quad))), \zeta[\mu \rightarrow < < \gamma^*, \kappa(\varepsilon^* \downarrow_1) > in\mathbf{Thread} : \zeta(\mu) \mid \quad {}_*$$
$$, \zeta(\mu) \mid \quad > in\mathbf{Local}], All\text{-}machines))$$
$$sort(\theta^*, \gamma, \varphi) =\ \ \mathbf{if}\ \#\theta^* > 0$$
$$\mathbf{then}\ \ \mathbf{if}\ \gamma \in \theta^* \downarrow_1 \mid \quad {}_*\ \ \mathbf{then}\ sort(\theta^* \dagger 1, \gamma, \lambda{\theta^*}_1{\theta^*}_2.\varphi(\theta^* \downarrow_1 : {\theta^*}_1, {\theta^*}_2))$$
$$\mathbf{else}\ sort(\theta^* \dagger 1, \gamma, \lambda{\theta^*}_1{\theta^*}_2.\varphi({\theta^*}_1, \theta^* \downarrow_1 : {\theta^*}_2))\ \mathbf{endif}$$
$$\mathbf{else}\ \varphi(<>, <>)\ \mathbf{endif}$$

The various semantical components have to be initialized with precise values:

$$\zeta_{init}(\mu) = << >, \sigma_{init} > in\mathbf{Local}$$
$$\sigma_{init}(l) = wrong(\text{“}\texttt{No such location}\text{”}, l)$$
$$\rho_{init}(\nu) = wrong(\text{“}\texttt{No such identifier}\text{”}, \nu)$$

It only remains to explain the “scheduling” part of the denotation. Observe that we used the *oneof* operator that had a quite easy interpretation for the implementers. Whenever *oneof* is applied on a list of objects, it selects one, removes this one from the original set of objects and calls the second argument on these two things. The *oneof* operator is for example used to select the thread to evaluate or the machine towards which computation must be migrated. This simple interpretation is not denotational since it introduces undeterminacy in a realm where determinism must prevail. We took care of writing *oneof* in terminal position so that we can define it, not to return one of the possible results, but the list of all possible results. Since results are defined as the list of values that are sequentially returned to the initial continuation, final answers (see the **Ans** domain) are lists of lists of values.

The *oneof* operator is therefore defined as:

$$
\begin{array}{rl}
\mathit{oneof}(\upsilon^*, \varphi) = & \textbf{if } \#\upsilon^* > 0 \textbf{ then } \varphi(\upsilon^* \downarrow_1, \upsilon^* \dagger 1) \S \mathit{oneof}(\upsilon^* \dagger 1, \lambda \upsilon {\upsilon^*}_1 . \varphi(\upsilon, \upsilon^* \downarrow_1 : {\upsilon^*}_1)) \\
& \textbf{else } <> \textbf{ endif}
\end{array}
$$

Starting from a state ζ, the answer associated to a program is computed with *compute* and *run*:

$$
\begin{array}{l}
\mathit{compute}(\zeta) = \mathit{run}(\zeta, \mathit{All\text{-}machines}) \\
\mathit{run}(\zeta, \mu^*) = \\
\quad \textbf{if } \#\mu^* > 0 \\
\quad \textbf{then } \mathit{oneof}(\mu^*, \lambda \mu {\mu^*}_1 .\ \textbf{let } \theta^* = \zeta(\mu) \mid \qquad {}_* \ \textbf{ and } \sigma = \zeta(\mu) \mid \\
\qquad\qquad\qquad\qquad\qquad \textbf{in} \quad \textbf{if } \#\theta^* > 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } \mathit{oneof}(\theta^*, \lambda \theta {\theta^*}_1 .\ \textbf{let } \gamma^* = \theta \mid \qquad {}_* \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{and } \psi = \theta \mid \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } \psi(\gamma^*)(\zeta[\mu \rightarrow < {\theta^*}_1, \sigma > \mathit{in}\textbf{Local}], \mu)\ ) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } \mathit{run}(\zeta, {\mu^*}_1) \textbf{ endif} \quad) \\
\quad \textbf{else } <> \textbf{ endif}
\end{array}
$$

4 Examples

Despite its simplicity, CD-Scheme has surprising capabilities and can be used to describe other proposals for concurrent programming. The following examples are mainly syntaxes and are defined with old-styled but hygienic macros[2].

The first example is a busy waiting loop in the spirit of [Per87, §3.4].

```
(define-macro (with-mutex bool . critical-section)
  `(let ((local #t))                              ;occupied
     ;;exchanges local and bool and loops until bool is free
     (while local (set! local (set! ,bool local)))
     ;;evaluates the critical section
     (let ((result (begin . ,critical-section)))
       (unless (set! local #t) (set! ,bool #f))  ;frees bool
       result ) ) )
```

This syntactic form ensures that only one process will evaluate the `critical-section` expression. The mutual exclusion is ensured by the atomic control of the `bool` variable[3].

The `once` syntax imposes that an expression cannot return more than one value. The trick is to pause the group of processes which computes the expression as soon as it returns a first value. The paused group will be automatically reclaimed since unreferenced.

```
(define-macro (once . program)
  `(let ((result? #f))                          ;is the result present ?
     (let ((g+v (sponsor (lambda (g)
                           (let ((value (begin . ,program)))
                             (if (set! result? #t)
                                 (suicide)    ;kills superfluous values
                                 (cons g value) ) ) ) ) )))
        (pause! (car g+v))                      ;kills other spawned processes
        (cdr g+v) ) ) )
```

An effect similar to the `qlambda` feature of Qlisp [GM84, GG88] or the `exlambda` of PaiLisp [IM89], is to define a function which body is in a critical section. The function is associated to a queue of processes waiting to enter the body of the function. Processes are reified into continuations that will be resumed when the critical section is free. Observe that busy-waiting loops are only used for short mutations of shared

[2] We do not use the new macro proposal in the appendix of [CR91] since the `pcall` syntax does not seem easily amenable to this model. We nevertheless suppose our macros to be hygienic [KFFD86] i.e. not to introduce name conflicts.

[3] The `(unless (set! local #t) ...)` form protects `bool` from being freed more than once when `critical-section` returns more than one value but does not prevent these extra computations to take place.

variables thus ensuring unexpensive waits. The expression defining **barrier** acts as the 'P' primitive on **bool**, considered as a semaphore, whereas the second argument of **breed** mimics the 'V' primitive. Waiting processes are resumed in a FIFO order, other strategies can be devised.

```
(define-macro (exlambda variables . body)
  `(let ((bool       #f)       ; controls body entrance
         (queue-flag #f)       ; controls the queue shared variable
         (queue      '()) )    ; list of waiting processes
     (lambda ,variables
       (let* ((barrier (if (set! bool #t)
                           (call/cc (lambda (go) ; suspends caller
                                      (with-mutex queue-flag
                                        (set! queue (cons go queue)) )
                                      (suicide) ))
                           'go ) )
              (result (once . ,body)) )          ; computes body
         (call/cc
          (lambda (return)
            (breed (lambda () (return result))
                   (lambda () ((with-mutex queue-flag
                                 (if (null? queue)
                                     (begin (set! bool #f) suicide)
                                     (let* ((end (last-pair queue))
                                            (proc (car end)) ) ; FIFO mode
                                       (set-cdr! end '()) proc ) ) )
                               'go-now ) ) ) ) ) ) ) ) ) )
```

Many proposals for concurrency [Hal84] include a **pcall** (for parallel call) special form to introduce concurrency. When a form such as (**pcall** F A B) is to be evaluated, the different terms F, A and B are concurrently evaluated by different processes yielding values f, a and b. When all values are computed, f (presumably a function) is applied on a and b. This is an instance of a classical fork-join model.

The continuation of a term of a **pcall** form is problematic due to the concurrency that exists between the terms and the fact that these terms might multiply return results. At least two solutions seem to be natural. Without restraining the problem, let us only consider the two-term combination (**pcall** F E). The first meaning is nicknamed "multiplicative" where all values of F are applied on all values of E. This imposes to keep all values in every **pcall** form since to multiply return results is a dynamic (and generally unforeseeable) property. The second meaning is "additive", the continuation of E is a function that waits for a value e: when invoked the first time and if F already returned some values f_i then all these (presumably) functions f_i are applied on e. If F has not returned any values, then e is memorized as a possible value for E. When invoked later, the last value of F is applied on e and e is memorized as the last value for E. These continuations are symmetrically defined and the number of effective applications is $1 + (n_F - 1) + (n_E - 1)$ where n_F (resp. n_E) is the number of values returned by F (resp. E). This behavior appears on figure 2 and is detailed in [Que91, Mor92].

CD-Scheme allows to program **pcall** with whatever semantics. The following example corresponds to the additive semantics. The definition is lengthy since semantics of continuation of **pcall** terms are complex.

```
(define-macro (pcall . terms)
  (let ((n (length terms)) (i -1))
    `(let ((handler (additive-pcall-handler ,n)))
       (handler (call/cc
                 (lambda (send)
                   (breed ,@(map (lambda (term)
                                   (set! i (+ i 1))
                                   `(lambda () (send (cons ,i ,term))) )
                                 terms )) ) )) ) ) )
```

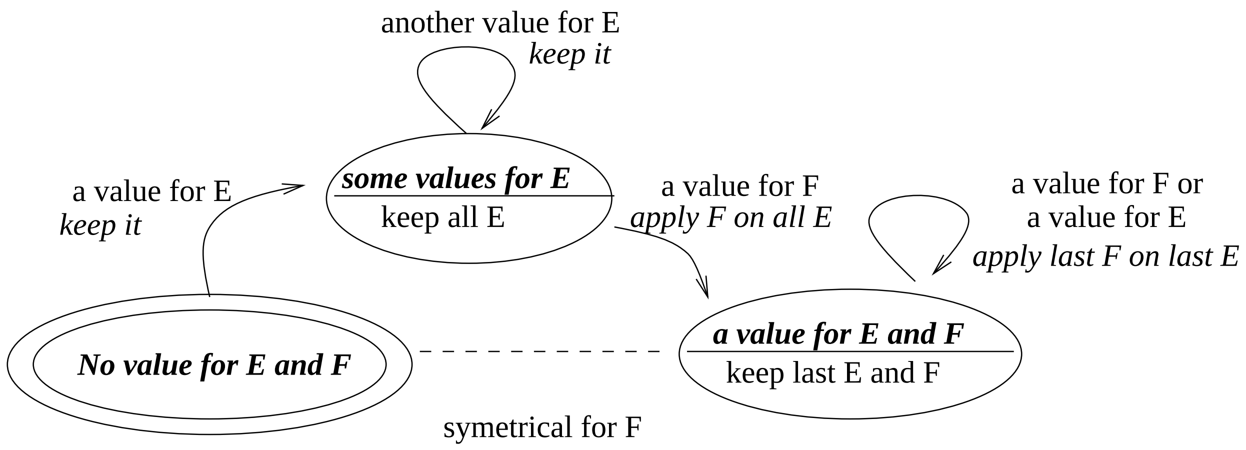

Figure 2: Continuation of a term of a `pcall` form

```
(define additive-pcall-handler
  (let ((not-yet-computed (list 'not-yet-computed)))  ;sentinel
    (lambda (n)
      (letrec
        ((arguments     (make-vector n not-yet-computed))
         (extra-results '())   ;extra results returned until ready?
         (ready?        #f)    ;is arguments complete ?
         (behavior
          (exlambda (index+value return)  ;exclusive body
            (let ((index (car index+value)) (value (cdr index+value)))
              (cond
               ((eq? (vector-ref arguments index) not-yet-computed)
                (vector-set! arguments index value)
                (if (map-and (lambda (content) ;are all arguments there ?
                               (not (eq? content not-yet-computed)) )
                             (vector->list arguments) )
                    (let ((applications (list (vector->list arguments)))
                          (extra extra-results) )
                      (set! ready? #t)
                      (set! extra-results '())
                      (apply breed
                             (map (lambda (application)
                                    (lambda ()
                                      (return (apply (car application)
                                                     (cdr application) )) ) )
                                  (reduce
                                   (lambda (applications iv)
                                     (vector-set! arguments (car iv) (cdr iv))
                                     (cons (vector->list arguments)
                                           applications ) )
                                   applications
                                   extra ) ) ) )
                    (suicide) ) )
               (ready?
```

```
                (vector-set! arguments index value)
                (let ((terms (vector->list arguments)))
                  (breed (lambda ()
                           (return (apply (car terms) (cdr terms))) )) ) )
               (else (set! extra (cons index+value extra))
                     (suicide) ) ) ) ) ) )
         (lambda (index+value)
           (call/cc (lambda (return) (behavior index+value return))) ) ) ) ) )
```

The `either` syntax can also be defined. There again the group of tasks computing the two branches is paused when a true result is produced. When the two expressions both return false, the final value of `either` is false.

```
(define-syntax (either e1 e2)
  `(let ((abort? #f)      ;when true means that result is set
         (group  'wait)   ;the group of tasks
         (last   #f)      ;is it the last process ?
         (result #f) )    ;the default final result
     (sponsor (lambda (g)
                (set! group g)
                (call/cc (lambda (return)
                           (define (handle v)
                             (when abort? (suicide))
                             (if v (unless (set! abort? #t)
                                          (set! result v) (return 'true) )
                                 (when (set! last #t) (return 'false)) )
                             (suicide) )
                           (breed (lambda () (handle (once ,e1)))
                                  (lambda () (handle (once ,e2))) ) )) ) )
     (pause! group)
     result ) )
```

The last example we give is related to the well-known `future` [Hal84, Hal89]. We can define it in regular CD-Scheme without resorting to magic functions such as `make-future` or `determine-future!`. These futures must be explicitly `touch`-ed to deliver their value.

```
(define (touch f) (f))
(define-macro (future e)
  `(let ((computed?  #f)        ;is the future determined ?
         (queue-flag #f)        ;controls queue and computed?
         (queue      '())       ;waiting touching processes
         (value      'wait) )   ;future's value
     (call/cc
      (lambda (return)
        (breed            ;concurrently (1) and (2)
         (lambda ()
           (return        ;(1) returns ...
            (lambda ()    ;← the future
              (call/cc (lambda (caller)
                         (if (with-mutex queue-flag
                               (or computed?
                                   (begin (set! queue (cons caller queue))
                                          #f ) ) )
                             value (suicide) ) )) ) ) )
         (lambda ()     ;(2) computes the value of the future
           (set! value ,e)
           (if (with-mutex queue-flag (set! computed? #t))
```

```
                (suicide)       ; Variant: (return value)
                (apply breed    ; resumes processes in parallel
                       (map (lambda (caller) (lambda () (caller value)))
                            (with-mutex queue-flag
                              (let ((q queue)) (set! queue '()) q)) ) ) ) )
        ) ) ) ) )
```

The difference between a future and a regular Scheme delay is that the computation of the future is done concurrently with its use. This behavior raises subtle issues, detailed in [KW90], with different behaviors all of which can directly be programmed in CD-Scheme.

Other features like M-structures [BNA91] or active objects [AR89] can be programmed in CD-Scheme and are left as exercises to the careful reader. Of course, all the previous syntaxes can be combined into higher-level abstractions since users often do not need to know the details of these macros.

5 Related and future work

The pioneer work of Halstead [Hal84] popularized the `future` concept in the MultiLisp project. Futures were a priori believed to introduce more parallelism than `pcall` since they immediately return a place-holder that can be handled but for its content. The evolution from MultiLisp to MultiScheme, where continuations have an indefinite extent, makes future more problematic since they can be multiply determined [KW90].

CD-Scheme is simpler than MultiLisp, since we showed that futures can be explicitly written in regular CD-Scheme thanks to `breed` and continuations. As futures are written in the language, so can variants of futures [KW90]. On the other hand, CD-Scheme is actually only simulated on a distributed network of workstations whereas MultiLisp is implemented on a shared memory architecture and benefits from very clever compiling techniques [MKH90].

Concurrent Scheme [KS89] is an implementation of Scheme for distributed memories. They introduce a `future`-like mechanism called `make-thread`. Mutual exclusion is handled through *domains*. No two computations can be simultaneously performed in a same domain. Domains define disconnected data subsets: no data sharing is possible between different domains. Data are thus copied before being passed from one domain to another. Domains are therefore good candidates to be associated to processors.

Since domains and threads can be dynamically created, it seems difficult to control if data are shared or copied. This precludes the use of deterministic side-effects and breaks the tradition of the single workspace of Lisp. On another hand, CD-Scheme introduces less concepts than Concurrent Scheme.

A parallel abstract machine was proposed in [GL90] where an implicit `remote` facility was present. Since their language is purely functional they do not handle assignment, nor data mutation, nor first-class continuations that we have shown to be valuable tools. The current version of their machine only handles booleans and small integers while we handle any kind of Lisp objects.

Qlisp offers some constructs, `qlambda` and `catch`, that can be simulated in CD-Scheme. The simulation is not completely accurate since Qlisp is based on COMMON LISP, the difference being that continuations have a dynamic extent in COMMON LISP rather than an indefinite extent as in Scheme. The work of [QS91] reconciles the two worlds. A new implementation of CD-Scheme, currently under progress, merges the `splitter` operator of [QS91] and the above `sponsor` facility.

PaiLisp [IM89] is a parallel extension of Scheme which kernel only comprises four primitives: `spawn` (to create processes), `suspend`, `exlambda` (similar to Qlisp's `qlambda`) and `call/cc`. The definition the authors gave of PaiLisp is through a metacircular interpreter using side-effects. The semantics of continuations is extended to handle the concept of process. PaiLisp's `call/cc` has a process-killing or process-resuming effect if the caller and the creator of a continuation are different: this allows to imperatively control individual processes which are identified with their unique starting continuation. As in PaiLisp, we tried hard to reduce the number of non primitive features; similarly we altered the semantics of continuation in presence of concurrency. On the other hand, we chose a different set of primitives offering the simple to learn and useful concept of group of processes and having a non-metacircular semantics.

Hieb and Dybvig [HD90] propose a `spawn` primitive which allows to control *process continuation* i.e.tree-structured continuations. `Spawn` and `sponsor` are very close in their effect on groups of tasks. Nevertheless `spawn` associates a partial continuation to the group of tasks, something we think unrelated to the group but only appropriate to a single process. The group object in CD-Scheme thus has more properties and can be more tightly controlled, the current implementation of CD-Scheme allows to reify the continuation of a process upto one of the groups to which it belongs in a manner reminiscent of [QS91].

Partial continuations offer the power of continuations created by `call/cc` but are respectful of the dynamic extent concept. We therefore plan to abandon full continuations in favor of partial continuations. The `splitter` operator of [QS91] can be identified to the above `sponsor` and allows to remove `call/cc` from CD-Scheme and its questionable definition where the invocation of a continuation is done within the groups of the caller rather than these of the creator. This weird behaviour does not arise if dynamic extent is respected.

6 Conclusions

We presented a language, CD-Scheme, which offers simple concepts allied to a great expressiveness. Distribution relies on a neighborhood topology and thus makes suitable to run programs on a large number of processors. Lisp dialects are often used as extension languages, CD-Scheme is such an extension language. We actually sacrifice speed (opposed to a non-concurrent, non distributed implementation of Scheme) to easily program widely distributed algorithms in a mostly functional style. CD-Scheme is also a pedagogical tool to describe concurrent features.

We realized an interpreted simulation of CD-Scheme showing that naïvely implemented migration and geographical commutation have tremendous costs but do not preclude to obtain some speedup [Que91]. The simulation also shows that, in the worst case, never moving objects is not unreasonable: some objects like files do not naturally move and shared mutable objects are simpler to implement on a unique site. Objects created on a variety of processors naturally load balance the work.

A real compiler is actually under progress associated to a distributed and concurrent GC along the lines of [LQP92]. Thorough experimentations will be led and reported in the future.

Bibliography

[AR89] Pierre America and Jan Rutten. A parallel object-oriented language: design and semantic foundations. In J W Bakker, editor, *Languages for Parallel Architectures: Design, Semantics, Implementation Models*. Wiley, 1989.

[BNA91] Paul S Barth, Rishiyur S Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In John Hughes, editor, *FPCA '91 – Functional Programming and Computer Architecture*, volume Lecture Notes in Computer Science 523, pages 538–568, Cambridge (Mass. USA), August 1991. Springer-Verlag.

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

[GG88] Ron Goldman and Richard P. Gabriel. Preliminary results with the initial implementation of qlisp. In *LFP '88 – ACM Symposium on Lisp and Functional Programming*, pages 143–152, Snowbird (Utah USA), 1988.

[GL90] J F Giorgi and Daniel Le Métayer. Continuation-based parallel implementation of functional programming languages. In *LFP '90 – ACM Symposium on Lisp and Functional Programming*, pages 209–217, Nice (France), June 1990.

[GM84] Richard P Gabriel and John McCarthy. Queue-based multi-processing lisp. In *LFP '84 – ACM Symposium on Lisp and Functional Programming*, pages 9–17, Austin (Texas USA), 1984.

[Hal84] Robert H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *LFP '84 – ACM Symposium on Lisp and Functional Programming*, pages 9–17, Austin (Texas USA), 1984.

[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.

[HD90] Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In *PPOPP '90 – ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 128–136, Seattle (Washington US), March 1990.

[HDB90] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77, White Plains, New York, June 1990.

[HFW84] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298, Austin, TX., 1984.

[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.

[KFFD86] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. *Symposium on LISP and Functional Programming*, pages 151–161, August 1986.

[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.

[KS89] Robert R. Kessler and Mark R. Swanson. Concurrent scheme. In *US-Japan Workshop on Parallel Lisp*, volume Lecture Notes in Computer Science 441, Sendai (Japan), June 1989. Springer-Verlag.

[KW90] Morry Katz and Daniel Weise. Continuing into the future: On the interaction of futures and first-class continuations. In *LFP '90 – ACM Symposium on Lisp and Functional Programming*, pages 176–184, Nice (France), 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.

[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.

[MKH90] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. In *LFP '90 – ACM Symposium on Lisp and Functional Programming*, pages 185–198, Nice (France), 1990.

[Mor92] Luc Moreau. An operational semantics for a parallel functional language with continuations. In D. Etiemble and J-C. Syre, editors, *PARLE '92 – Parallel Architectures and Languages Europe*, pages 415–430, Paris (France), June 1992. Lecture Notes in Computer Science 605, Springer-Verlag.

[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.

[Per87] R. H. Perrott. *Parallel Programming*. Addison Wesley, 1987.

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

[Pri80] Gianfranco Prini. Explicit parallelism in Lisp-like languages. In *Conference Record of the 1980 Lisp Conference*, pages 13–18, Stanford (California USA), August 1980. The Lisp Conference, P.O. Box 487, Redwood Estates CA 95044.

[QS91] Christian Queinnec and Bernard Serpette. A Dynamic Extent Control Operator for Partial Continuations. In *POPL '91 – Eighteenth Annual ACM symposium on Principles of Programming Languages*, pages 174–184, Orlando (Florida USA), January 1991.

[Que91] Christian Queinnec. Crystal Scheme, A Language for Massively Parallel Machines. In M Durand and F El Dabaghi, editors, *Second Symposium on High Performance Computing*, pages 91–102, Montpellier (France), October 1991. North Holland.

[SG90] James W. Stamos and David K. Gifford. Implementing remote evaluation. *IEEE Trans. on Software Engineering*, 16(7):710–722, July 1990.

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