Design of a concurrent and distributed language[⋆]

Christian Queinnec[⋆⋆] & David De Roure[⋆⋆⋆]
École Polytechnique & INRIA-Rocquencourt
University of Southampton

Abstract. This paper presents a new dialect of Scheme aimed towards concurrency and distribution. It offers a few primitives, including first-class continuations, with very simple semantics. Numerous examples are given showing how to program the classical concurrent control operators such as `future`, `pcall` and `either`. The implementation is sketched and presented along the lines of a metacircular interpreter.

This paper presents the idiom of Icsla[1], a language belonging to the Lisp family and more precisely a descendant of Scheme. This dialect has been designed with respect to the following main objectives:

- It should have a very simple and understandable semantics, with few but powerful and unrestrictively combinable concepts;
- It should offer concurrency, distribution and some other modern features such as sophisticated control features while not sacrificing the variety of styles traditionally offered by Lisp.

These goals are rather general and deserve further comment. Following Scheme, the semantics must be kept small and clean; this in turn favors correct implementation, precise documentation and eases learning or teaching. The concepts must be understandable and simple, i.e. providing only one functionality since composition is already offered at various levels by procedures, macros or modules. Existing concurrent features, proved useful in other languages, must be programmable—this demonstrates the power of our concepts as well as providing implementations that can be taught. Having no current user community to protect, some concepts have been freely revised, such as continuations, objects and equality. Currently the idiom lacks an abundance of features, since any feature that can be delivered by a library has been postponed. Finally we view our resulting idiom as a proto-language on top of which other languages can be built such as Scheme, EuLisp or CLOS.

⋆ *Revision* : 1.22 presented at the Workshop on Parallel Symbolic Computing: Languages, Systems and Applications, October 1992, Boston (Massachussetts US).

⋆⋆ Laboratoire d'Informatique de l'École Polytechnique (URA 1437), 91128 Palaiseau Cedex, France – Email: `queinnec@polytechnique.fr` This work has been partially funded by GDR-PRC de Programmation du CNRS.

⋆⋆⋆ Declarative Systems Laboratory, Department of Electronics and Computer Science, University of Southampton, Southampton SO9 5NH, UK `D.C.DeRoure@southampton.ac.uk`

[1] Icsla (standing for *interpretation, compilation and semantics of applicative languages*) is the name of a joint project between École Polytechnique and INRIA-Rocquencourt, hence the name of the designed language.

Many concurrent or distributed languages propose (or even impose) a unique model for distribution or concurrency: objects in Emerald [RTL+91], shared objects in Orca [BKT92], threads/objects in Clouds [DJAR91] etc. These models cannot be directly applied to Scheme dialects due to their lack of first-class closures or continuations, which introduce specific problems if added carelessly. Moreover to adopt a unique model would seriously restrict the variety of styles which is traditionally the apanage of Lisp. Purely functional languages offer many opportunities for concurrency and distribution, however they usually reject side-effects, continuations and non-determinacy. Side-effects are necessary since modern computing has to deal with UNIX[2], files, NFS, X-windows: new languages cannot simply ignore the myriad of programs with fixed interfaces they have to cooperate with. Continuations are a very useful concept popularized by Scheme which promoted them to first-class objects. They allow us to program escapes, coroutines and multiple returns. Yet they raise non-trivial problems with concurrency, as shown for instance in [KW90, HD90, Que90]. Finally languages tend to avoid non-determinacy since it makes reasoning about programs difficult, computations rarely reproducible and hence debugging hard. We nevertheless offer non-determinacy and side-effects as basic features (not to be less powerful than the UNIX shells) but expect their use to be encapsulated within appropriate abstractions so to obtain *mostly functional programs*.

The target of the idiom is the development of stand-alone applications that can run on network(s) of workstations[3]. We feel a need for a tool which will extend the usual UNIX shells to offer better support for distribution (compared to `rsh`). Many attempts already exist for such tools (see [CM92] for such an attempt as well as an excellent overview) but have some limitations. They, for instance, require homogeneous computers or ubiquitous file systems. Considering that Lisp dialects are renowned for providing extension languages, the idiom will be an ideal glue to connect, run, and synchronize programs on a network scale. It does not require that everything be written within the idiom but provides basic services such as global network mutable variables, fork-join synchronization etc. in an easily and interactively programmable way.

Speed has not been considered a major objective but cleanliness of the semantics as well as understandable canonical implementation were preponderant. We believe the result to fit a natural mental model of how programmers conceive their applications. Most of the features that compose the idiom of Icsla are not new and might be found in the abundant literature; they nevertheless constitute further improvements upon CD-Scheme [Que92a].

1 Terminology

Applications (i.e. programs) run on a network of interconnected *processors*. Any processor has its own data space. A processor can be identified with a computer or

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

[3] Networks of workstations usually represent the most powerful multicomputers configuration people have access to.

with a UNIX process running on a computer. The evaluation of a program passes through a series of atomic *computation steps*. Computations are performed by *tasks* gathered into evolving tree-structured *groups* of tasks. Each processor has an evolving subset of *tasks* that it executes *tour à tour*. A task can be represented, at any time, by an expression to evaluate, a *lexical context* and a *dynamic context* that contains a *continuation*.

The implementation manages *entities* that can be first-class *values* or behind-the-scene objects (like tasks). Compound entities have *fields*, the content of which might refer to other entities. Fields can be *mutable* or *immutable*.

2 Philosophy

The idiom of Icsla adopts the Distributed Shared Memory concept which is the rather natural distributed extension of the single data space of Lisp. Distribution of data is transparent to the computation, but for speed: no explicit user-invoked copying operation is required in order to pass a value from one processor to another. The burden of managing replication and coherence is performed by the runtime library. To ease the runtime as well as to protect the user (from a software engineering point of view) the mutability of all fields of all entities is known. Logically a mutable entity (an entity that has at least one mutable field) is never copied from the processor where it was created. Conversely immutable entities can be freely replicated. This logical view does not prevent the runtime system from migrating entities rather than replicating them, to avoid copying huge immutable values, or replicating mutable values with an associated coherence protocol.

Concurrency and distribution are explicit: specialized functions exist that can be invoked where concurrency or migration are required. To offer these features as functions allows us to choose dynamically the computations that will benefit from them. Avoiding the use of these functions confers a sequential semantics on our language as in regular Scheme.

Dynamic extent is a key feature of our language. Dynamic extent already exists in regular Scheme with functions like `with-output-port` or `call-with-input-file` but is not recognized as a basic concept *per se*. In a traditional sequential setting as in COMMON LISP, dynamic extent usually refers to the duration of a computation. In our language, the dynamic extent of an expression encompasses all the tasks that are evaluating (sub)parts of the original expression. When a task makes an expression return a value, it exits the dynamic extent of this expression. The dynamic extent of an expression is not directly related to global time since scheduling interleaves computations.

In our language, many tasks may be running simultaneously. It is then possible that an expression, the evaluation of which is initiated by a single task, returns more than one value if its continuation is used by newly created tasks during the dynamic extent of the original expression. Tasks are thus pervasive but orthogonal to dynamic extent. Tasks are not first class objects in our language, they cannot be suspended, sent signals or have an identity. Computations instead can be controlled. Dynamic extents are well nested or disjoint: they have a tree shape.

Following the usage in Scheme, special functions are preferred over special forms;

this makes some examples rather ugly with numerous thunks, but of course more attractive syntax can be devised.

3 Features

The idiom of Icsla is strongly based on Scheme but adds a few features partially explored in [Que92a, QS91]:

- Assignment atomically exchanges old and new values (it is the only feature that allows data mutation);
- Equality predicates stress instantaneous or eternal substitutability;
- The `breed` function controls concurrency;
- Distribution (data and task migration) is achieved by the `remote` function or `placed-remote` function;
- Groups of tasks can be created and imperatively paused or awakened;
- Dynamic extent is emphasized and `call/cc` is no longer primitive.

We will first introduce these features informally, give some examples in §4 and finally a metacircular interpreter in §5.

3.1 Concurrency

Concurrency is introduced by means of the `breed` function. The `breed` function takes some thunks as arguments and *replaces* the current task by new tasks evaluating these thunks in the same dynamic context.

`(breed` *[thunk...]*`)`

The effect of `breed` is to create, detach and kill tasks: `breed` is the only function that manages tasks and its sole effect is to change the number of tasks. The different tasks that `breed` creates are independent: they have no synchronization constraint. When a task finishes the evaluation of the thunk it was spawned on, it commits suicide and yields no result at all. Tasks can only return information through continuations (as discussed in §3.3), which explicitly represent causality. The `suicide` utility function which allows the current task to vanish obviously returns no value at all, and can be defined simply in terms of `breed` as:

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

Tasks are not first-class objects but groups of cooperating tasks can be managed as explained in §3.2. Had tasks been first-class objects, many questions would arise about their identity with respect to continuations.

No particular scheduling strategy is actually stressed but a fair one is required. The initial continuation of stand-alone applications (resp. the toplevel loop) waits for all spawned tasks to finish before returning to the operating system (resp. starting a new evaluation cycle). The `breed` function introduces indeterminacy which is already present (but to a lesser extent) in Scheme since the evaluation order of the terms of a functional application is unspecified.[4]

[4] The idiom of Icsla has an explicit left to right order of evaluation. For people who want application to be sequential with an unknown order, we offer macros based on `random(3)`.

The `breed` function is equivalent to the existing low-level primitives of some implementations, for example `fork` of MultiLisp [Hal89]. We just unveil it to become the sole concurrency primitive the user can see.

3.2 Groups

Tasks can be created easily by means of `breed` but insufficient control is offered to program the classical parallel-or operator, also known as `amb` [McC63], `either` [Pri80]. The form `(either` *expression*$_0$ *expression*$_1$`)` concurrently spawns two computations to evaluate the two expressions, returns the value of the first one which returns true and kills the other computation since it is no longer useful. We introduced the concept of *group*, inspired from the *sponsor* of [Osb90] and the *controller* of [HDB90] but also related to UNIX's concept of groups of tasks. Groups of tasks can be imperatively controlled, allowing us to program `either` (see section 4).

The `call/de` (for *call with dynamic extent*) function starts a computation under the responsibility of a fresh group.

`(call/de (lambda (`*group*`)` *expression*`))`

This form creates a first-class object called a group, binds it to the variable *group* and evaluates *expression* under its control. Any value yielded by *expression* is a value returned by the `call/de` form.

When a task creates another task, the new task belongs to the same groups as its progenitor task: groups are inherited. The group created by `call/de` initially contains only the current task, i.e. the task that invoked `call/de` then belongs to one more group during the computation of the associated *expression*. All subsequent tasks that will be created by the computation of *expression* will then belong to *group*. When a task makes `call/de` return a value (normally or by premature exit—see `abort` below), the task ceases to belong to the associated group.

A group can be viewed as the set of tasks that participate in the computation of an expression. Actually, `call/de` takes an optional argument, a unary function, that will be called on the group object whenever the continuation of the group i.e. the continuation of the originating `call/de` form, is to be reclaimed. At that time, no new task can be created inside this group and no new value can be produced as a result of the associated expression. This allows us to specify some (finalization) code that will be run (for its effect) when the computation associated to the group finishes. This effect corresponds to a built-in distributed termination facility.

A group can pause or awaken the tasks it controls by means of the associated imperative functions `pause!` and `awake!`.

`(pause!` *group*`)`   `(awake!` *group*`)`

Pausing a group means that all the tasks that belong to this group will be suspended wherever they are geographically and however many there are. Conversely all the tasks of a group can be resumed when wakening the group. The `pause!` and `awake!` functions return unspecified values. A task can determine if it currently participates in a group by using the `within/de?` predicate.

`(within/de?` *group*`)`

A group can also be viewed as the reification of the dynamic extent of the associated expression. A canonical implementation for `call/de`, see §5, is to push a so called group-frame onto the stack of the current task to indicate the creation of the group. It is therefore simple for `within/de?` to check if a particular group appears in the stack of the current task. `breed` pushes a suicide frame onto the stack of the current task then creates new tasks, each of which shares that new stack.

The `awake!` function can simply be implemented as a broadcast that indicates that all tasks of this group must be made runnable. Some protocol must be used to avoid simultaneous overlapping contradicting broadcasts. A naive solution is to dedicate a machine for that task. The `awake!` function can return as soon as the broadcast is emitted. On the other hand, the `pause!` function is more subtle. The naive implementation is to broadcast to all processors a message indicating that all tasks belonging to a particular group must be paused, i.e. removed from the list of runnable tasks. These tasks must not be reclaimed since they can be awakened later: they are stored, associated with their group, in the list of paused tasks of the processor. The `pause!` function returns when all processors have removed these tasks and sent back an acknowledgment. The precise semantics of `pause!` makes it return as soon as no task which is to be paused but is still running can causally affect the behavior of the continuation of `pause!`. This respects causality and leaves room for some optimizations.

3.3 Continuations

The continuation of a task defines at any time what that task has yet to do. Any computation step the task performs either extends its continuation, adding new pending evaluations, or yields a value to its continuation.

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 upon them numerous usages from escape constructs such as `catch/throw` *à la* Common Lisp, to coroutines [HFW84]. They are very powerful but difficult to compile well [CHO88, HD90, JG89] and they pose complex implementation problems.

We decided to emphasize the concept of dynamic extent and use the primitives of [QS91] known as `splitter`, `abort` and `call/pc`. Furthermore we identify `splitter` with `call/de`. `call/de` marks the current continuation so that it can be referenced by `abort` and `call/pc`, but these can only be used while in the dynamic extent of the intended group. This can be checked with the `within/de?` predicate. The `abort` function takes a group and a thunk as arguments.

(`abort` *group thunk*)

When a task invokes `abort` it replaces its continuation up to *group* by the evaluation of *thunk*. This thunk is still evaluated in the dynamic extent of the group. The aborting of one task does not affect the others, in particular multiple tasks can invoke `abort` concurrently. Consider for instance the following generator in the spirit of [MH90]:

```
(define (//iota start stop)
  (call/de
   (lambda (return)
     (define (iota start stop)
```

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

The task that invokes `(//iota 0 4)` dies while four new tasks are sequentially created and each of them appears 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 2, 0, 1 and 3. Although these four new tasks were created under the sponsorship of the `return` group, they loose this sponsorship shortly after they call `abort` when leaving the group, i.e. when leaving the dynamic extent of `//iota`. The enclosing group does not gain four new tasks since they already belong to it (by set inclusion).

Premature exits *à la* `catch/throw` can be easily programmed with the `abort` function, which is the sole function (with `protect`) to have a control effect[5]. The `call/pc` function deals with partial continuations which are already described in [QS91] and will not be discussed further here.

`(call/pc` *group* `(lambda (`*partial-continuation*`) ...))`

A useful feature to detect the end of a computation is the `unwind-protect` feature of COMMON LISP. It is simple to know when a computation starts but not so simple to know when it ends. In fact we do not provide any means to know when a computation ends; this would break modularity where the implementor of a package wants to hide the fact that s/he uses (groups of) tasks. It is nevertheless useful to provide a way to detect when a computation returns a value, i.e. to detect when a value exits a dynamic extent. Since premature exits restart a task with a thunk, the `protect` feature we present uses thunks instead of values.

`(protect` *thunk thunk-transformer*`)`

If *thunk* normally returns a value *v* then this value is wrapped into `(lambda ()` *v*`)` which is transformed into a new thunk, the result of `(`*thunk-transformer* `(lambda ()` *v*`))`; this latest thunk is then applied and its value becomes the final value of the `protect` expression. When a premature exit leaves the dynamic extent of *thunk*, then the second argument of `abort` is transformed by *thunk-transformer* and the abort is restarted with its transformed result. For instance:

```
(call/de (lambda (exit)
           (protect (lambda () (abort exit (lambda ()
                                             (display 'meep)
                                             3 )))
             (lambda (thunk) (display 'mip)
                             (lambda () (* 2 (thunk))) ) )
```

displays `mipmeep` *and returns* `6`

This `protect` mechanism not only provides a control effect triggered by a value returning but also enables the user to intercept and transform the passing thunk.

[5] Ignoring errors such as `(car 42)` which invokes the error handler and therefore might induce control effects.

Let us sum up our continuation model. It stresses dynamic extent (checked by `within/de?`) and offers simple escape forms with `abort`. Sharing continuations with `breed` allows multiple returns that can be detected with `protect`. Finally groups of tasks can be managed. The emphasis on dynamic extent would suffice, in regular Scheme, to prevent expressions from returning multiply since continuations would only be used at most once. This is not the case in our language because of the concurrency: expressions can fork into multiple tasks multiply returning. Alternatively we can develop a new programming style based on multiple returns and, for instance, view a function of the X windows library such as `XNextEvent` as an event generator.

Partial continuations, i.e. `call/pc`, allow us to define `call/cc` as in [QS91]. More precisely they allow us to define multiple versions of `call/cc` depending on the required relation with respect to `protect`. We nevertheless do not retain `call/cc` as a primitive function since it can also be defined solely with `call/de`, `breed`, `abort`, `pause!` and `awake!`, see §4.

3.4 Migration

Computations can migrate using the `remote` function which implements a kind of remote procedure call (RPC) [SG90, GL90].

`(remote` *function arguments...*`)`

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 onto a neighboring processor (the precise choice is left unspecified using `remote` but can be imposed with `placed-remote` inspired by `placed par` in Occam [Bur88]) where the migrated f is applied on the migrated a and b. This differs from a regular RPC in 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 (if any).

The `remote` function involves some kind of concurrency which is implicit if the computation is really migrated. On a single processor `(remote` *function arguments*`)` is equivalent to `(call/de (lambda (`g`) (breed (lambda () (abort `g` (lambda () (`*function arguments*`)))))))` which explicitly reveals the underlying `breed` effect.

We have adopted a very simple "migration" model where entities never move, always residing 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 semantics is respected, nor does it exclude multiple duplication of immutable entities such as small integers, booleans, characters, immutable cons cells, function code or universally known primitives like `+` or `cons`. To achieve this, the mutability of all user-created data can be taken into account and a specialized migration policy devised. Fortunately, the percentage of mutable data is usually small.

Some primitives are *geographically strict*: to be applied, their arguments must be local. 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 etc—almost all primitive computations follow this rule. Observe that computations carry their continuation and are therefore not bound to the sites through which they pass. Migration of computations towards data is not unreasonable, especially when the data consists of files or other site-bound resources, but does not exclude moving some data towards computations to lessen communication cost. Adopting the parallel shell view mentioned before, it is sensible to offer another function which achieves migration but towards a precise processor where, for instance, specific resources may be accessed.

Migration can create a subtle problem: a computation can overflow a processor with multiple copies of a single immutable entity. We are studying hash-consing for these values but cannot report more on this topic.

3.5 Assignment

Synchronization is often achieved with a test-and-set operation. Rather than providing additional primitive objects like (some flavors of) semaphores, or offering functions like `replace-if-eq` that only work on pairs, we decided that assignment should play this rôle. The form `(set!` *variable form*`)` first computes *form* then atomically exchanges the obtained value with the former content of the location bound to *variable*; this former value becomes the value of the assignment form. This value is undefined if the location was previously uninitialized.

Following the spirit of [Per87, §3.4], a busy-waiting loop can be written as[6]:

```
(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
     (protect (begin . ,critical-section)
        (lambda (thunk)
          (unless (set! local #t) (set! ,bool #f))  ;frees bool
          thunk )) ) )
```

This syntactic form ensures that only one task will enter[7] the `critical-section` expression. The mutual exclusion is ensured by the atomic control of the `bool` variable[8].

---

[6] To avoid looping until being preempted, the real implementation explicitly calls the task scheduler.

[7] *enter* and not *evaluate* since tasks can be created inside the critical section that might return multiply.

[8] The `(unless (set! local #t) ...)` form protects `bool` from being freed more than

If the value of the assignment is not needed, i.e. if it is used for its effect rather than its value, then the former content of the location need neither be returned nor migrated. Nevertheless for causality reasons the continuation of the assignment is resumed only after the assignment is performed, so the implementation cannot avoid sending back at least an acknowledgment. Continuations explicitly represent causality.

The assignment has been banished by the ML family of languages in favor of first-class mutable cells because they are easier to type-check. On the contrary, we restrict side-effects to be performed only by assignment. No power is lost since it is well known that each can be simulated by the other. For local variables, the advantage of assignment is that all mutation sites can be determined statically. The mutability of global variables can be stated by exportation directives associated with modules [QP91]. All variables thus have a known mutability but only the mutable variables might pose an implementation problem if accessed concurrently, since the assignment must be atomic. A static sharing analysis [Deu90] can be performed to determine those variables which might present such conflicting access. A simple-to-compute superset of these "dangerous" variables is those that are mutated and closed. This allows us to use at no cost common idioms where a local variable is only mutated once.

On the other hand, the simplest implementation for mutable locations is to use boxes as popularized by [KKR+86]. The sole mutable entities of our implementation are boxes which can (logically) never be migrated. An assignment to a box is therefore a request for the processor that hosts this box to update it. It is a simple matter for that processor to serialize these updates. After an update is performed, the hosting processor reports it to the requesting processor which can then resume its computation. More elaborate protocols with partially replicated boxes have also been studied [Piq90, Piq91].

3.6 Equality

The old `eq?` predicate comparing addresses of heap-allocated values bears no meaning in a distributed world: a single entity can be replicated on two different processors where the addresses of the replications have no obvious relation. It seems that only two reasons exist to compare values: they can be compared for substitutability or instantaneous similarity. Two values are substitutable one for the other if there does not exist a program able to distinguish them. Two values are similar if at the time of the comparison they have similar structure and content. Substitutable values are always similar. Similarity is known as `equal?` but requires special precautions for cyclic values. Substitutability is not a computable property but can be safely approximated as follows: two values are substitutable if they are both immutable and the content of their fields are substitutable or if they are the same mutable object [Bak90].

once when the critical-section returns more than one value, but does not prevent these extra computations from taking place. Also observe that the boolean flag is freed whether a value is normally returned or an escape is performed.

Concerning the geographical commutation rule, binary strict predicates such as `substitutable?` or `equal?` require a special implementation. The canonical implementation can be termed as "geographic currying": follow the first argument until reaching the processor where it resides, then send its coordinates to the second argument which will compare it against itself. If data can migrate while being compared, the canonical algorithm has to be refined.

4 Examples

Despite its simplicity, the idiom of Icsla 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-style but hygienic macros[9].

Multiple results for an expression might pose problems. The `once` syntax constrains an expression to return no more than one value. The following definition also ensures that all spawned computations will cease after returning a value. The trick is to pause the group of tasks which computes the expression as soon as it returns a first value. The paused group will be automatically reclaimed once unreferenced. Observe that `call/de` returns at most one value (the first that toggles the `result?` variable). The reclamation of the other tasks is done outside the `g` group.

```
(define-macro (once . expression)
  `(let ((result? #f))                         ;is the result present ?
     (let ((g+v (call/de
                  (lambda (g)
                    (cons g (protect (begin . ,expression)
                              (lambda (thunk)
                                (if (set! result? #t)
                                    (suicide)   ;kills superfluous values
                                    thunk ) ) )) )))) )
        (pause! (car g+v))                     ;kills other spawned tasks
        (cdr g+v) ) ) )
```

4.1 `qlambda`

An effect similar to the `qlambda` feature of Qlisp [GM84, GG88] or the `exlambda` of PaiLisp [IM89], is to define a function whose body is in a critical section. The function is associated with a queue of tasks waiting to enter the body of the function. Tasks are reified into groups that will be awakened one by one (in FIFO order) when the critical section is free[10]. Observe that busy-waiting loops are only used for short mutations of shared variables, thus ensuring inexpensive waits. The first

[9] 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.

[10] This does not ensure that at most one single task evaluates the body of the function since other tasks might be created and concurrently return a value. It ensures that only one task enters when another one exits.

expression of the body, a kind of *barrier*, acts as the 'P' primitive on `bool`, considered as a semaphore, whereas the second argument of `breed` mimics the 'V' primitive. Waiting tasks are resumed in a FIFO order; other strategies can be devised. Observe the second `with-mutex` form, which chooses what to do in a critical section but the resulting choice (a thunk) is not critical and can be evaluated without precautions. The `once` syntax imposed `body` to return at most one value or to exit, the first argument of `breed` also respects escaping.

```
(define-macro (exlambda variables . body)
  `(let ((bool       #f)        ;controls body entrance
         (queue-flag #f)        ;controls the queue shared variable
         (queue      '()) )     ;list of waiting tasks
     (lambda ,variables
       (when (set! bool #t)
         (call/de (lambda (caller)        ;suspends caller
                    (with-mutex queue-flag
                      (set! queue (cons caller queue)) )
                      (pause! caller) )) )
       (protect (once . ,body)            ;computes body (once)
         (lambda (thunk)
           (call/de
            (lambda (return)
              (breed (lambda () (abort return thunk))
                     (with-mutex queue-flag
                       (if (null? queue)
                           (begin (set! bool #f) suicide)
                           (let ((client (car (last-pair queue)))
                             ;;FIFO mode
                             (set! queue (remove client queue))
                             (lambda () (awake! client)) ))))))))))))
```

4.2 `pcall`

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 tasks 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 to all values of E. This requires all values in every `pcall` form to be kept, since to multiply return results is a dynamic (and generally unforeseeable) property. The second meaning is "additive", where 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 is illustrated in figure 1 and detailed in [Que91, Mor92].

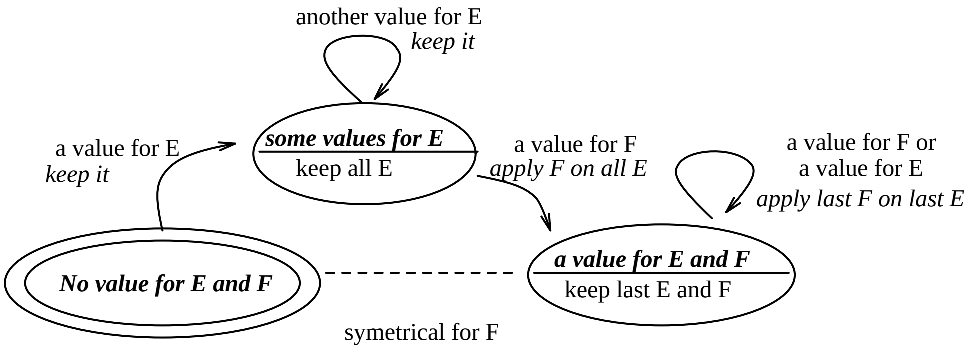

Fig. 1. Continuation of a term of a `pcall` form

The idiom of Icsla allows us to program `pcall` with whatever semantics. The trick appears in the `pcall` syntax: any value produced by a term of the `pcall` form is given with its rank to the "funnel". This funnel can sort, store, or forget the incoming values and thus decide when the application is ready to be performed, with the correct continuation `return`. When the funnel is built, temporary structures can be preallocated on the basis of the number of terms. The code for the additive (or multiplicative) semantics is lengthy and is skipped.

```
(define-macro (pcall . terms)
  `(call/de
    (lambda (return)
      (let ((funnel (make-additive-pcall-handler
                     return ,(length terms) )))
        (breed ,@(map (lambda (i term)
                        `(lambda () (funnel ,i ,term)) )
                      (iota 0 (length terms))
                      terms )) ) ) ) )
```

4.3 `either`

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

```
(define-macro (either e1 e2)
  `(let ((group  'wait)    ;the group of tasks
         (last   #f)       ;is it the last task ?
         (result #f) )     ;the default final result
     (set! result
           (call/de (lambda (g)
                      (set! group g)
                      (define (handle v)
                        (abort g (if v (lambda () #t)
                                       (if (set! last #t) (suicide)
                                           (lambda () #f) ) )) ) )
                      (breed (lambda () (handle (once ,e1)))
                             (lambda () (handle (once ,e2))) ) )) ) )
     (pause! group)
     result ) )
```

4.4 future

Another example is related to the well-known `future` [Hal84, Hal89]. We can define it 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 tasks
         (value      'wait) )   ;future's value
      (call/de
       (lambda (return)
         (breed                ;concurrently (1) and (2)
          (lambda ()
            (abort return      ;(1) returns ...
             (lambda ()        ;← the future
               (call/de
                (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))
                ;Variant: (abort return (lambda () value)) [KW90]
                (suicide)
                (apply breed ;resumes all waiting tasks 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 performed concurrently with its use. This behavior raises subtle issues (detailed in [KW90]) with different behaviors, all of which can be directly programmed in our language.

4.5 `call/cc`

The final example we will give is `call/cc` without using partial continuations. The trick is to encode a continuation into a paused group containing only one task. Awakening the group will achieve the resumption of the continuation once. To allow the continuation to be used more than once, as soon as the group is awakened, it spawns a new task that will pause the group again. Many implementation details concerning access to the shared variable `values` and the race condition between the `abort` and `pause!` effects are not shown here, to avoid encumbering the definition. The interest of this example is to show that `call/cc` bears no relationship with `protect` or `dynamic-wind` as would be the case with partial continuations.

```
(define (call/cc f)
  (let ((values '()))
    ((call/de
      (lambda (g)
        (breed (lambda ()
                 (abort g (lambda ()
                            (lambda ()
                              (f (lambda (v)
                                   (set! values (cons v values))
                                   (awake! g)
                                   (suicide) )) ) )) )
               (lambda ()
                 (let loop ()
                   (if (pair? values)
                       (let ((v (car values)))
                         (set! values (cdr values))
                         (breed (lambda ()
                                  (abort g (lambda ()
                                             (lambda () v) )) )
                                loop ) )
                       (pause! g) ) ) ) ) ))) ) )
```

Other features like M-structures [BNA91], actors or active objects [AR89] can be programmed in the idiom of Icsla 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 More formal semantics

In this section, we provide the definition of a metacircular interpreter for the idiom of Icsla. We will depart from the tradition since the expression to be evaluated is first rewritten using a program transformation based on Abstract Continuation Passing Style (ACPS) [FWFD88, Que92b]. The resulting expression is evaluated using any regular Scheme interpreter, see for instance [Dyb87]. We can thus define our language in a very compressed form with only two key points: a program transformation and a library of functions that are defined in Scheme. This presentation is close to the compilation model: control operators are defined in a module which is compiled without ACPS since they are already written according to this style. The resulting compiled module is linked with the regular modules which are ACPS-rewritten.

Abstract Continuation Passing Style is a program transformation that allows us to remove all known control operators such as `call/cc`, `unwind-protect` as well as others dealing with partial continuations. It can moreover express the dynamic extent concept, dynamic binding etc. ACPS is based on a non-standard representation of the continuation: a list of frames. We extend ACPS by making the set of runnable tasks explicit. A frame now waits for a value, a continuation and the current list of eligible tasks; it selects one of the latter and runs it in the appropriate context. Our metacircular definition does not take into account distribution which is difficult to reveal due to its transparency (see [Que92a] for a formalization of distribution). We concentrate on the control features that were discussed above and hide the description of error handling, dynamic binding and substitutability, which require us to encode types.

The frames of which a continuation is composed are made apparent by ACPS. Each frame has a behavior which specifies what to do when receiving a value, as well as the list of the frames that are below it. The Extended ACPS program transformation appears in table 1. Sending a value to a continuation is expressed with `resume`, while pushing a frame is written `extend`. $\theta^\star$ is the list of eligible tasks, q is the continuation and v some value.

$$
\begin{array}{l}
\mathrm{ACPS}[\![\nu]\!]\theta^\star q \rightarrow \texttt{(resume}\ \ \theta^\star\ q\ \nu\texttt{)} \\
\mathrm{ACPS}[\![\texttt{(quote}\ \varepsilon\texttt{)}]\!]\theta^\star q \rightarrow \texttt{(resume}\ \ \theta^\star\ q\ \texttt{(quote}\ \varepsilon\texttt{))} \\
\mathrm{ACPS}[\![\texttt{(if}\ \pi_0\ \pi_1\ \pi_2\texttt{)}]\!]\theta^\star q \rightarrow \\
\quad \mathrm{ACPS}[\![\pi_0]\!]\theta^\star\texttt{(extend}\ \ q\ \ \texttt{(}\lambda\texttt{(}\theta'^\star\ q'\ v'\texttt{)}\ \texttt{(if}\ v'\ \mathrm{ACPS}[\![\pi_1]\!]\theta'^\star q'\ \mathrm{ACPS}[\![\pi_2]\!]\theta'^\star q'\texttt{)))} \\
\mathrm{ACPS}[\![\texttt{(set!}\ \nu\ \pi\texttt{)}]\!]\theta^\star q \rightarrow \\
\qquad \mathrm{ACPS}[\![\pi]\!]\theta^\star\texttt{(extend}\ \ q\ \ \texttt{(}\lambda\texttt{(}\theta'^\star\ q'\ v'\texttt{)}\ \texttt{(resume}\ \ \theta'^\star\ q'\ \texttt{(set!}\ \nu\ v'\texttt{))))} \\
\mathrm{ACPS}[\![\texttt{(}\lambda\texttt{(}v^*\texttt{)}\ \pi\texttt{)}]\!]\theta^\star q \rightarrow \texttt{(resume}\ \ \theta^\star\ q\ \texttt{(}\lambda\texttt{(}\theta'^\star\ q'\ v^*\texttt{)}\ \mathrm{ACPS}[\![\pi]\!]\theta'^\star q'\texttt{))} \\
\mathrm{ACPS}[\![\texttt{(}\pi_1\ \ldots\pi_n\texttt{)}]\!]\theta_0^\star q_0 \rightarrow \\
\qquad \mathrm{ACPS}[\![\pi_1]\!]\theta_0^\star\texttt{(extend}\ \ q_0\ \ \texttt{(}\lambda\texttt{(}\theta_1^\star\ q_1\ v_1\texttt{)}\ \ldots \\
\qquad\qquad\qquad \mathrm{ACPS}[\![\pi_n]\!]\theta_{n-1}^\star\texttt{(extend}\ \ q_{n-1}\ \ \texttt{(}\lambda\texttt{(}\theta_n^\star\ q_n\ v_n\texttt{)} \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{(}v_1\ \theta_n^\star\ q_n\ v_2\ \ldots v_n\texttt{)))))}
\end{array}
$$

Table 1. Extended ACPS rules

Some utility functions are needed to encapsulate the representation of continuations. The `extend` function pushes a frame onto a continuation. When a value is given to a frame, the frame suspends the current computation into a task and calls the scheduler. The scheduler chooses one task, finds its associated behavior and applies it. The `oneof` operator randomly selects one task among a set and invokes its second argument on this task and the set containing the others[11].

```
  (define (extend q frame)
  (cons frame q) )
(define (resume tasks q value)
  (schedule (cons (make-task q value) tasks)) )
(define (schedule tasks)
  (if (pair? tasks)
      (oneof tasks (lambda (task tasks)
                     (let ((q (task-q task))
                           (value (task-value task)) )
                       ((frame-behavior (car q))
                        tasks (cdr q) value ) ) ))
      end-of-computation ) )
```

To shorten the presentation we will use object technology and define classes for the entities we will manage. The `define-class` defines a class with a particular super-class and a list of proper fields[12].

```
  (define-class Task Object (q value))
(define-class Group Object (running? tasks))
(define-class Frame Object (behavior))
(define-class Group-Frame Frame (group))
(define-class Protect-Frame Frame (cleaner))
```

We can now give the definition of the special functions of the idiom of Icsla. Tasks are created by `breed`, sharing the same continuation with a suicide frame on top of it. The `breed` function calls `schedule` directly, thus committing suicide.

```
  (define (breed tasks q . thunks)
  (let ((q (extend q (make-Frame (lambda (tasks q v)
                                   (schedule tasks) )))))
    (schedule (append (map (lambda (thunk)
                             (make-task
                              (extend q (make-Frame
                                         (lambda (tasks q v)
                                           (thunk tasks q) ) ))
                              'go ) )
                           thunks )
                      tasks )) ) )
```

---

[11] The `oneof` can be given a functional definition with a powerdomain for the final answers, see [Que92a].

[12] We use a personal and small object system called Meroon.

Groups are created by `call/de`, where a group frame is simply pushed onto the continuation to indicate that the dynamic extent is entered.

```
(define (call/de tasks q f)
(let ((new-group (make-Group #t '())))
  (f tasks
     (extend q (make-Group-Frame
                 (lambda (tasks q v) (resume tasks q v))
                 new-group ))
     new-group ) ) )
```

The following function is a utility that, given a continuation and a group, returns `#f` or a pair cutting the continuation into two parts, the above and below parts with respect to the corresponding group frame. Since partial continuations can make a group-frame appear more than once in a continuation, the deepest one is preferred (see also [HD90]).

```
(define (split q group)
(define (search left* right*)
  (if (pair? right*)
      (if (and (Group-Frame? (car right*))
               (eq? (Group-Frame-group (car right*)) group) )
          (or (search (cons (car right*) left*) (cdr right*))
              (cons (reverse left*) right*) )
          (search (cons (car right*) left*) (cdr right*)) )
      #f ) )
(search '() q) )
```

We can now easily express `within/de?` and `call/pc`, which simply use the previous `split`.

```
(define (within/de? tasks q group)
(resume tasks q (not (not (split q group)))) )
(define (call/pc tasks q group f)
  (let ((above+below (split q group)))
    (if above+below
        (f tasks q (lambda (tasks q v)
                     (resume tasks (append (car above+below) q) v) ))
        (wrong tasks q "Not in the dynamic extent of" group) ) ) )
```

The `abort` control operator is more complex since it must respect the `protect` forms that it has to bypass.

```
(define (abort tasks q group thunk)
(let ((above+below (split q group)))
  (if above+below
      (let ((q-group (cdr above+below)))
        (define (unwind tasks q q-group thunk)
          (cond ((eq? q q-group) (thunk tasks q))
                ((Protect-Frame? (car q))
                 ((Protect-Frame-cleaner (car q))
```

```
                  tasks
                  (extend (cdr q)
                          (make-Frame
                           (lambda (tasks q thunk)
                             (unwind tasks q q-group thunk) ) ) )
                  thunk ) )
                (else (unwind tasks (cdr q) q-group thunk)) ) )
        (unwind tasks q q-group thunk) )
      (wrong tasks q "Not in the dynamic extent of" group) ) ) )
```

The two functions `pause!` and `awake!` have straightforward definitions. We encode them with a side effect on the group object to indicate whether it is running or the set of paused tasks. These are the sole side-effects we use. Due to Extended ACPS, the body of `pause!` and `awake!` are implicitly in a critical section.

```
  (define (awake! tasks q group)
  (if (Group-running? group)
      (resume tasks q an-unspecified-value)
      (let ((paused-tasks (Group-tasks group)))
        (set-Group-tasks! group '())
        (set-Group-running?! group #t)
        (resume (append paused-tasks tasks)
                q an-unspecified-value ) ) ) )
(define (pause! tasks q group)
  (define (sort-tasks tasks k)
    (define (sort tasks paused others)
      (if (pair? tasks)
          (if (split (Task-q (car tasks)) group)
              (sort (cdr tasks) (cons (car tasks) paused) others)
              (sort (cdr tasks) paused (cons (car tasks) others)) )
          (k paused others) ) )
    (sort tasks '() '()) )
  (if (Group-running? group)
      (sort-tasks
       (cons (make-task q an-unspecified-value) tasks)
       (lambda (tasks-to-pause other-tasks)
         (set-Group-running?! group #f)
         (set-Group-tasks! group tasks-to-pause)
         (schedule other-tasks) ) )
      (resume tasks q an-unspecified-value) ) )
```

Finally the `protect` control operator just pushes a special frame that `abort` will consider.

```
  (define (protect tasks q thunk thunk-transformer)
  (thunk tasks
         (extend q (make-Protect-Frame
                    (lambda (tasks q v)
                      (thunk-transformer
```

```
                tasks
                (extend q (make-Frame
                           (lambda (tasks q thunk)
                             (thunk tasks q) ) ))
                (lambda (tasks q) (resume tasks q v)) ) )
             thunk-transformer )) ) )
```

Other features such as dynamic binding and exception handling can be described with the same framework, implemented through special frames in the continuation. Dynamic binding is achieved through two functions `(associate/de` *key value thunk*`)` to associate *key* with *value* in the dynamic extent of *thunk*. The value associated with *key* can be retrieved with `(lookup/de` *key success failure*`)`. One point worth noting is that the dynamic environment is immutable so it can be migrated easily (this does not prevent binding a mutable value to a key). This dynamic binding facility corresponds to a deep binding implementation.

6 Implementation

This section describes the implementation of a distributed interpreter for the Icsla idiom. The interpreter has been developed in order to gain practical experience of the ideas we have presented, and it is intended that much of the technology will be reusable in the runtime system of the proposed compiler. Some of the techniques are based on experience with development and use of the QPL system [DeR90b] [DeR90a].

Preliminary work involved construction of a small interpreter supporting the `breed` and `remote` primitives, embedded in Scheme. This was based on a continuation-passing metacircular evaluator where each task has an associated state consisting of a *(continuation, value)* pair [DeR90c]. A state is selected by the scheduler and the continuation is invoked to advance the state by one computation step, returning a list of states representing the states of tasks to replace the original; in this way a task is replaced by zero or more tasks. To simulate `remote`, 'geography' is introduced by extending the task state to include a *machine* component, and this aspect can be taken further by introducing distinct environments and implementing the geographical commutation rule. This tool enabled us to explore abstractions using `breed` and `remote`.

The current distributed interpreter is closely related to the metacircular interpreter presented in the previous section. There are two versions: a single process version which supports `breed` using resumptions and a scheduler (as above), and a distributed version which additionally supports `remote` by a system of *lazy migration* of objects to other processes. The distributed version employs multiple processes on a network of Unix hosts and enables us to study the migration techniques and to conduct experiments to investigate issues of concurrency, asynchrony and fault tolerance in the Idiom of Icsla.

In the single process version, input expressions are read, expanded, parsed and transformed into an intermediate form consisting of objects. This is then evaluated using a generic `eval` function, under the control of a simple scheduler. The `eval` function is 'pushy' in that it indirectly calls the generic function `resume` which in turn calls `eval`, and ultimately the initial continuation is called whereupon control returns to the top level loop.

The distributed version separates the top level user interaction from the evaluation. A *listener* process performs the read phase, then submits the encoded expression to a separate process called an *evaluation daemon*. The daemon performs the same evaluation as the single process version, supporting multiple tasks (created by `breed`), but it also acts as a client of other evaluation daemons (implementing `remote`) and as a server for incoming remote requests. Eventually the result is returned to the listener and presented to the user. The selection of a neighbor for `remote` can be made by the user (as in the case of a `placed-remote`, which can be used to return a value to the listener and to access specific resources) or by the implementation of `remote`. The latter case must capture the topology of the network, realising the concept of neighbors.

Migration of tasks is achieved by transmitting the various interpreter structures between daemons. Note that this is not a traditional remote evaluation, because the continuation of the `remote` application migrates too. A lazy strategy is essential, and is achieved through remote pointers: when an object is migrated it may not be fully populated, but rather some of its slots contain remote pointers to entities on the original host. When one of these slots is accessed, the pointer is followed. Remote pointers remain valid when they are migrated provided they point to a site that is accessible from their new site. The communications layer supports serialization and deserialization of arbitrary objects, while the migration strategy addresses the details of replication of immutable data, safe update of mutable data, and heuristics for the extent of transfer of specific data structures.

The listener and the evaluation daemons are implemented as UNIX processes which communicate via the RPC mechanism. This layer was chosen as it provides a suitable level of independence from evolution of the particular machines, networks and operating system versions. In fact we do not adhere to the strict RPC client–server model, because the evaluation daemons are both clients and servers, and we wish to avoid blocking operations in these so that other tasks can be scheduled. Essentially, RPC is used as a message-passing layer.

The object encoding uses the XDR standard [xdr] to facilitate communication with machines on a heterogeneous network and interaction with other software. When sending the same entity more than once in a single message, the algorithm used for serialization generates references to the earlier value, providing an efficient encoding of immutable values (such as strings), preserving substitutability within a transmission and avoiding problems of circularity. Different encoding protocols may be used, indicated by a protocol word which prefixes a message. To preserve equality between entities in different messages it would be necessary to search a large cache during transmission, and to avoid this complexity the protocol includes a mechanism for clearing the caches. Preservation of equality is therefore the responsibility of the remote pointer mechanism built above this lower level of object encoding.

There is a significant 'housekeeping' requirement in the handling of dynamic process creation, multiple daemons per UNIX host, multiple applications per user and multiple users per network. This is achieved through management daemons which run one per host and oversee all these operations. The management daemons perform the combined roles of maintaining the RPC service program numbers (which are transient) and spawning new processes on request (like `inetd`); in addition, they provide diagnostic facilities to aid program development.

7 Related and future work

The pioneer work of Halstead [Hal84] on MultiLisp popularized the `future` concept. As explained in [Hal89], futures are not primitive but defined in terms of lower level functions. The evolution from MultiLisp to MultiScheme, where continuations have an indefinite extent, also makes futures more problematic since they can be multiply determined [KW90].

Our language is simpler than MultiScheme since we promote combinable low-level features and show their constructive power on various examples. We thus do not address the exact nature of futures but provide ways to program variants of them [KW90]. Our language is also cleaner since we propose some solutions to integrate continuations *à la* Scheme compatible with the dynamic extent concept. On the other hand, our language 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. It introduces 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 where data are shared or copied. This precludes the use of deterministic side-effects and breaks the tradition of the single workspace of Lisp. On the other hand, our language introduces fewer concepts than Concurrent Scheme.

Qlisp [GM84, GG88, GGS89] offers a huge number of constructs, such as `qlambda` and `catch`, that can be simulated in our language. We depart from this approach since we based our work on Scheme rather than COMMON LISP but reintroduced dynamic extent. We also separate the control effects of Qlisp `catch` into `abort` and `pause!`.

PaiLisp [IM89] is a parallel extension of Scheme whose kernel only comprises four primitives: `spawn` (to create tasks), `suspend`, `exlambda` (similar to Qlisp's `qlambda`) and `call/cc`. The semantics of continuations is extended to handle the concept of tasks. PaiLisp's `call/cc` has a task-killing or task-resuming effect if the caller and the creator of a continuation are different: this permits imperative control of individual tasks 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 continuations in the presence of concurrency. On the other hand, we chose a different set of primitives offering the simple to learn and useful concept of groups of tasks.

Hieb and Dybvig [HD90] propose a `spawn` primitive which allows control of *task continuations*, i.e.tree-structured continuations. `Spawn` and `call/de` are very close in their effect on groups of tasks. Nevertheless `spawn` associates a partial continuation with the group of tasks, something we think unrelated to the group but only appropriate to a single task. The group object in our language thus has more properties and can be more tightly controlled.

8 Conclusions

We have presented a language which offers simple concepts allied to a great expressiveness. Distribution relies on a neighborhood topology, making it suitable for running programs on a large number of processors. Lisp dialects are often used as extension languages; the idiom of Icsla is such an extension language. We actually sacrifice speed (compared to a non-concurrent, non-distributed implementation of Scheme) in order to easily program widely distributed algorithms in a mostly functional style. Our language is also a pedagogical tool to describe concurrent features.

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

References

[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.

[Bak90] Henry G Baker. Equal rights for functional objects or, the more things change, the more they are the same. Technical report, Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436, USA, October 1990.

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

[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.

[Bur88] Alan Burns. *Programming in Occam 2.* the Instruction Set Series. Addison Wesley, 1988.

[CHO88] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for continuations. In *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, page 124 131, August 1988.

[CM92] Henry Clark and Bruce McMillin. Dawgs – a distributed compute server utilizing idle workstations. *Journal of Parallel and Distributed Computing*, 14(2):175–186, February 1992.

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

[DeR90a] David C. DeRoure. Experiences with Lisp and distributed systems. In *High Performance and Parallel Computing in Lisp*, November 1990. Also appears as Technical Report CSTR90–21, Department of Electronics and Computer Science, University of Southampton.

[DeR90b] David C. DeRoure. *A Lisp Environment for Modelling Distributed Systems.* PhD thesis, Dept. of Electronics and Computer Science, University of Southampton, January 1990.

[DeR90c] David C. DeRoure. QPL3—Continuations, concurrency and communication. Technical Report CSTR 90–20, Department of Electronics and Computer Science, University of Southampton, 1990.

[Deu90] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *POPL '90 – Seventeenth Annual ACM symposium on Principles of Programming Languages*, pages 157–168, San Francisco, January 1990.

[DJAR91] Partha Dasgupta, Richard J LeBlanc Jr, Mustaque Ahamad, and Umakishore Ramachandran. The clouds distributed operating system. *Computer*, 24(11):34–44, November 1991.

[Dyb87] R. Kent Dybvig. *The Scheme Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

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

[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.

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

[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.

[JG89] Pierre Jouvelot and David K Gifford. Reasoning about continuations with control effects. In *ACM SIGPLAN Programming Languages Design and Implementation*, volume 24 of *SIGPLAN Notices*, pages 218–225, Portland (OR), June 1989. SIGPLAN, ACM Press.

[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.

[McC63] John McCarthy. A basis for a mathematical theory of computation. In Braffort and Hirshberg, editors, *Computer Programming and Formal Systems*. North Holland, 1963.

[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.

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

[Piq91] José Piquer. Preserving distributed data coherence using asynchronous broadcasts. In *Eleventh International Conference of the Chilean Computer Science Society*, pages 283–290, Santiago (Chile), October 1991. Plenum Publishing Corporation, New York NY (USA).

[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.

[QP91] Christian Queinnec and Julian Padget. Modules, Macros and Lisp. In *Eleventh International Conference of the Chilean Computer Science Society*, pages 111–123, Santiago (Chile), October 1991. Plenum Publishing Corporation, New York NY (USA).

[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.

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

[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.

[Que92a] 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.

[Que92b] Christian Queinnec. Value transforming style. Research Report LIX RR 92/07, Laboratoire d'Informatique de l'École Polytechnique, 91128 Palaiseau Cedex, France, May 1992.

[RTL+91] Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software — Practice and Experience*, 21(1):91–118, January 1991.

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

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

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