INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

On the Finiteness of Resources in Distributed Computing

Luc Moreau (U. of Southampton), Christian Queinnec (Cristal)

N° 3147

Mars 1997

THÈME 2

Rapport de recherche

ISSN 0249-6399

On the Finiteness of Resources in Distributed Computing

Luc Moreau (U. of Southampton), Christian Queinnec (Cristal)

Thème 2 — Génie logiciel
et calcul symbolique
Projet Cristal

Rapport de recherche n° 3147 — Mars 1997 — 28 pages

Abstract: Millions of computers are now connected together by the Internet. At a fast pace, applications are taking profit of these new capabilities, and become parallel and distributed, e.g. applets on the WWW or agent technology. As we live in a world with finite resources, an important challenge is to be able to control computations in such an environment. For instance, a user might like to suspend a computation because another one seems to be more promising. In this paper, we present a paradigm that allows the programmer to monitor and control computations, whether parallel or distributed, by mastering their resource consumption.

Key-words: programming languages, parallelism, distribution, abstract machine

(Résumé : tsvp)

Ces travaux ont été partiellement financés par le contrat EPSRC GR/K30773 et par le projet européen ERB 4050 PL 930186.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33) 01 39 63 55 11 – Télécopie : (33) 01 39 63 53 30

De la finitude des ressources en calcul distribué

Résumé : Internet met en communication des millions d'ordinateurs : les logiciels intègrent donc, de plus en plus, des aspects liés au parallélisme et à la distribution : codons (*applets*) sur la toile (*Web*) ou autres agents. Le monde réel ne possédant que des ressources finies, nous proposons de contrôler les calculs pour en tirer le meilleur parti. Par exemple, un utilisateur peut vouloir suspendre un certain calcul si un autre semble plus prometteur. Dans ce rapport, nous proposons un nouveau paradigme autorisant les programmeurs à contrôler et diriger leurs calculs, qu'ils soient parallèles ou distribués, par la maîtrise de la consommation de ressources informatiques nécessaires pour leur accomplissement.

Mots-clé : Langage de programmation, parallélisme, distribution, machine abstraite

1 Introduction

As we live in a world with finite resources, it is of paramount importance for the user to be able to monitor and control computations. This task is all the more complex since computations may be parallel, distributed, and most probably making use of code written by others. This problem is particularly illustrated by the Internet as the user wishes to search information over the WWW, exploits parallelism to improve efficiency, relies on distribution to increase locality, but also wants to concentrate the computing power in the most promising directions to reduce searching time.

There are two types of applications that we particularly wish to program. First, a user that has initiated a computation over the Internet should be able to suspend the computation in order to analyse the results he has already obtained. If these were unsatisfactory, he must be able to resume the computation where it was suspended in order to collect the next results. In this example, computations should be understood as possibly parallel and distributed over the net. Second, service providers offer computing facilities to customers who subscribe to their service by transferring electronic cash [24]; in return, service providers supply them with a handle to submit jobs, create accounts that they debit according to the usage of the facilities, and inform users of the exhaustion of their account. Again, jobs submitted by customers may generate parallel and distributed computations which must be monitored by the service provider.

Our goal is to provide the means by which everybody, customers and service providers, can get the most one can out of a situation with bounded resources. We believe that parallel computations can be driven by mastering their resource consumption. In this paper, we present a new language, called $\mathcal{Q}$uantum, containing primitives to monitor and control computations. The basic principle of the semantics of $\mathcal{Q}$uantum is to keep track of the resources consumed by computations. Resources can be understood as processors cycles, bandwidth and duration of communications, or even printer paper. We shall adopt more a generic view by saying that computations need *energy* to be performed[1]. Although the notion of energy is part of the semantics of $\mathcal{Q}$uantum, the programmer cannot create energy ex nihilo, but can only transfer it between computations via some primitives of the language. As a result, we were able to ensure a general principle for $\mathcal{Q}$uantum: given a finite amount of energy, any computation is finite.

Besides its resource-oriented semantics, $\mathcal{Q}$uantum is designed to be parallel and multi-user, and to be independent of the memory model (central, distributed, with or without coherence). All these features make it particularly suitable for programming in a distributed environment like the Internet. Furthermore, the language offers the mean to write programs of the form "when this computation ends, do this", which is an essential feature in distributed computing.

This paper is organised of follows. We present the intuition of $\mathcal{Q}$uantum in Section 2 and define its semantics in Section 3. Section 4 contains several applications written in

[1] Other names found in the literature for a similar concept are fuel [13] or computron [29, p. 102–103].

$\mathcal{Q}$uantum. In Section 5, we focus on the implementation of the language. Finally, Section 6 discusses related work and is followed by a conclusion.

2 The Language $\mathcal{Q}$uantum: Rationale

In this section, we introduce the language $\mathcal{Q}$uantum, its constructs and their intuitive semantics, and the considerations that lead to its design. The core of $\mathcal{Q}$uantum is an applied call-by-value lambda-calculus with numbers, booleans, and lists; Figure 1 displays its syntax.

$$
\begin{array}{rcl}
V & ::= & f \mid b \mid x \mid (\lambda x.M) \\
M & ::= & V \mid (M\ M) \mid (\mathsf{parallel}\ M\ M) \mid (\mathsf{suicide}) \\
f & ::= & \mathsf{cons} \mid \mathsf{car} \mid \mathsf{cdr} \mid \mathsf{call\text{-}with\text{-}group} \mid \mathsf{within\text{-}group?} \\
 & & \mid \mathsf{channel} \mid \mathsf{enqueue} \mid \mathsf{dequeue} \mid \mathsf{pause} \mid \mathsf{awake} \\
b & ::= & 0 \mid 1 \mid \ldots \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{nil}
\end{array}
$$

Figure 1: Syntax of $\mathcal{Q}$uantum

In $\mathcal{Q}$uantum, the construct parallel creates parallel evaluations. Intuitively, if an expression (parallel M_1 M_2) is evaluated in an environment ρ with a continuation κ, then the expressions M_1 and M_2 will be evaluated independently and in parallel, in the same lexical environment ρ, and with the same continuation κ. We opted for this model of parallelism because it allows the definition of *generators* [17]. Besides, a computation has the ability to terminate by evaluating the expression (suicide).

Our goal is to be able to allocate resources to computations, and to monitor and to control their use as evaluations proceed. In our view, it is essential to be notified of the *termination* of a computation so that, for instance, unconsumed resources can be transferred to a more suitable computation. Similarly, we want to be informed of the *exhaustion* of the resources allocated to a computation, so that for instance more resources can be supplied.

In order to be notified the termination or energy exhaustion of a computation, we need an entity that represents the computation. A *group* is an object that can be used to refer to an *active* computation in a $\mathcal{Q}$uantum program. As a first-class object, a group has an unlimited lifetime, but its *useful* lifetime is the duration of its associated computation.

So, a group is associated with a computation composed of several evaluations proceeding in parallel; in turn, they can initiate subcomputations by creating subgroups. As a result, our computation model is hierarchical. A group is said to *sponsor* [16, 23, 11] the computation it is associated with. Reciprocally, every computation has a sponsoring group, and so does every evaluation.

At creation time, a group is given an *energy quota*. More specifically, a computation that evaluates the expression (call-with-group F e φ_e φ_t) under the sponsorship of a group g_1, creates a new first-class group g_2 that is allocated an initial quota of energy e and whose parent is g_1. Furthermore, it initiates a computation under the sponsorship of g_2 by applying F to g_2; hence, the user function F receives a handle on its sponsoring group. As Quantum keeps track of resource consumption, the energy e allocated to g_2 is deducted from the energy of g_1.

The semantics enforces the following principle: any computation consumes energy from its sponsoring group. Therefore, not only is a group perceived as a way of naming computations, but also it must be regarded as an *energy tank* for the computation. In addition, two events may be signalled during the useful lifetime of a group: *group termination* and *energy exhaustion* are asynchronously notified by applying the user functions (the *notifiers*) φ_t and φ_e, respectively[2]. A group is said to be terminated, when it has no subgroup and it does not sponsor any evaluation; i.e. no more activity can be performed in the group. When group g_2 is terminated, the function φ_t is asynchronously called on g_2 to notify its termination, and the energy surplus of g_2 is transferred back to g_1. Note that calling φ_t is sponsored by g_1, i.e. the parent of g_2. Similarly, when a computation sponsored by g_2 requires more energy than available in g_2, the function φ_e is asynchronously called on g_2 to notify its energy exhaustion, also under the sponsorship of g_1, with transfer of the remaining energy of g_2 to g_1.

Figure 2 displays the state transition diagram for groups. At creation time, a group is in the running state, which means that the evaluations that it sponsors can proceed as long as they do not require more energy than available. Asynchronous notifications are represented by dotted edges. Once a computation requires more energy than available in its sponsoring group, the state of its group changes to exhausted, and at the same time an asynchronous notification φ_e is run. When all subgroups and all evaluations sponsored by a group terminate, its state becomes terminated, while the asynchronous notifier φ_t is called. Let us observe that the terminated state is a dead end in the state diagram; this guarantees the *stability* of the termination property: once a computation terminates, it is not allowed to restart (as the resource that it did not consume may have been reallocated).

Energy may be caused to flow between groups, independently of the group hierarchy, under the control of the user program. Two primitives operate on groups: pause and awake. Intuitively, the primitive pause forces a running group *and its subgroups* into the exhausted state, and all the energy that was available in this hierarchy is transferred to the group that sponsored the pause action. The construct (awake g e) "awakens" a group in the exhausted state by transferring it to the running state, with an energy e which is deducted from the group sponsoring the awake action. Let us observe the non-symmetric behaviours of pause and awake: the former operates recursively on a group hierarchy, while the latter acts on a group and not its descendants. However, we might wish to awake a hierarchy recursively, for instance when we wish to resume a paused parallel search. In particular, we might wish to resume the search with the energy distribution that existed when the hierarchy was paused. Unfortunately, such information is no longer available because groups are *memoryless*. It is therefore the programmer's responsibility to leave some information at pausing-time about the way a hierarchy should be awakened. Not only does pause transfer energy, but it does

[2] Subscript t denotes termination, whereas subscript e denotes exhaustion.

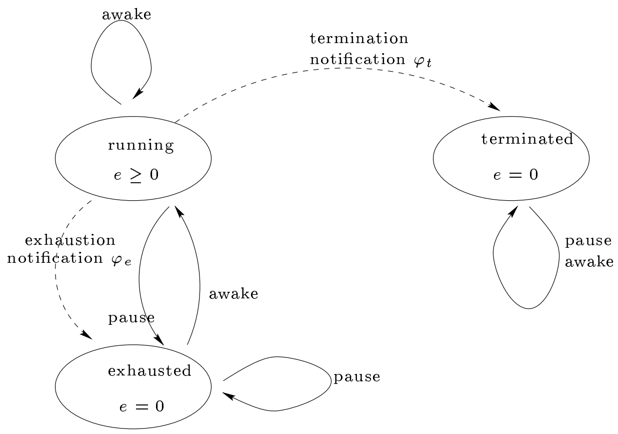

Figure 2: State Transitions

also post a notification for each group in the tree. More precisely, pausing a group g with a notifier φ_p forces into the exhausted state each group g' in the hierarchy rooted by g; moreover, for each g', an evaluation that applies φ_p on g' is created under the sponsorship of the parent of g'. Let us note that notifications are prevented to run as all groups in the hierarchy have been dried out (except the notification on the root g, which is sponsored by the parent of g and then might run). Once the root of the hierarchy is awakened, any notification sponsored by the root will be activated, and may decide to awake the group it is applied on, and step by step, energy may be redistributed among the hierarchy.

Our permanent concern when designing $\mathcal{Q}$uantum was to be able to compute in a distributed framework. Hence, we decided that $\mathcal{Q}$uantum would be independent of the memory model: so, real shared memory, shared memory simulated over a distributed memory [19], distributed causally coherent memory [25] are memory models that may be adopted with $\mathcal{Q}$uantum. However, in the programs that we wrote in $\mathcal{Q}$uantum, we needed primitives to synchronise computations and to exchange information between them. We observed that asynchronous unbounded communication channels [15] offered the appropriate level of abstraction. The expression (channel) returns a new, initially empty channel. Evaluating (enqueue c V) adds a value V to the channel c, maintaining values in a FIFO order; this primitive is asynchronous because it does not require to synchronise with a reading operation on the same channel. The construct (dequeue c) returns and removes the first value contained in the channel c. If the channel is empty, evaluation is blocked until a value is sent on the channel.

$M \in \Lambda_Q$	$::=$	$V_s \mid (M\ M) \mid (\textsf{parallel}\ M\ M) \mid (\textsf{suicide})$	(Term)
$V_s \in SValue$	$::=$	$c_s \mid x \mid (\lambda x.M)$	(Syntactic Value)
$V \in Value$	$::=$	$c \mid \ell \mid f_c \mid (\textsf{cons}\ V\ V) \mid g \mid \langle \textsf{ch}\ \alpha \rangle$	(Semantic Value)
$c_s \in SConst$	$::=$	$f \mid b$	(Syntactic Constant)
$f_c \in PApp$	$::=$	$(\textsf{cons}\ V) \mid (\textsf{enqueue}\ V) \mid (\textsf{within-group?}'\ V)$ $(\textsf{call-with-group}\ V) \mid ((\textsf{call-with-group}\ V)\ V)$ $(((\textsf{call-with-group}\ V)\ V)\ V)$	(Partial Application)
$c \in Const$	$::=$	$c_s \mid d \mid \textsf{within-group?}'$	(Constant)
$b \in BConst$	$=$	$\{\textsf{true}, \textsf{false}, \textsf{nil}, 0, 1, \ldots\}$	(Basic Constant)
$d \in Void$	$=$	$\{\textsf{void}\}$	(Void Constant)
$f \in FConst$	$=$	$\{$cons, car, cdr, call-with-group, within-group?, channel, enqueue, dequeue, pause, awake$\}$	(Functional Constant)
$x \in Vars$	$=$	$\{x, y, z \ldots\}$	(User Variable)
$\varphi \in Notifier$	$\subset$	$Closure$	(Notifier)
$\ell \in Closure$	$::=$	$\langle \textsf{cl}\ \lambda x.M, \rho \rangle$	(Closure)

$\mathcal{M} \in Qconfig$	$::=$	$\langle T, \Gamma, \sigma \rangle$	(Q-Configuration)
$\Gamma \in GMap$	$:$	$Group \to GInfo$	(Group Mapping)
$i \in GInfo$	$::=$	$\langle g, e, s, \varphi_e, \varphi_t, n, g^* \rangle$	(Group Information)
$g \in Group$	$=$	$\{\bot_g\} \cup \{g_0, g_1, \ldots\}$	(Group)
$t \in Task$	$::=$	$\langle C, g \rangle$	(Task)
$C \in CoSt$	$::=$	$\textsf{Ev}\langle M, \rho, \kappa \rangle \mid \textsf{Ret}\langle V, \kappa \rangle$	(Computational State)
$\kappa \in CCode$	$::=$	$(\textbf{init}) \mid (\kappa\ \textbf{fun}\ V) \mid$ $(\kappa\ \textbf{arg}\ M\ \rho) \mid (\kappa\ \textbf{rgroup})$	(Continuation code)
$s \in GState$	$::=$	running $\mid$ exhausted $\mid$ terminated	(Group State)

T	$\subseteq$	$Task$	(Set of Tasks)
$q \in Queue$	$::=$	$\langle\rangle \mid \langle V \rangle \mid q \S q$	(Queue)
$\sigma \in QStore$	$:$	$Loc \to Queue$	(Queue Store)
$\alpha \in Loc$	$=$	$\{\alpha_0, \alpha_1, \ldots\}$	(Location)
$\rho \in Env$	$:$	$Vars \to Value$	(Env)
$n \in \mathbb{N}$			(Number of sponsored tasks)
$e \in Energy$	$\subseteq$	$\mathbb{N}$	(Energy)
$\mathcal{K}$	$:$	$Task \times GMap \to Energy$	(Cost Function)

Figure 3: State Space

3 The Language $\mathcal{Q}$uantum: Semantics

In this section, we present the *parallel* semantics of $\mathcal{Q}$uantum, whereas we focus on its *distributed* implementation in Section 5. The semantics is described by an abstract machine, called the $\mathcal{Q}$-machine. Figure 3 displays its state space, where we can see that the set of terms Λ_Q is the syntax given in Figure 1. Transition rules appear in Figures 4 to 10.

In the sequel, we adopt Barendregt's [2] definitions and conventions on the lambda-calculus; in particular, n-ary functions should be understood as curried functions. We use the notation $f[x \to V]$ to denote the function f' such that $f'(x) = V$ and $f'(y) = f(y)$, $\forall y \neq x$.

$$
\begin{array}{rcll}
\mathsf{Ev}\langle (M_1\ M_2), \rho, \kappa\rangle & \to_{cek} & \mathsf{Ev}\langle M_1, \rho, (\kappa\ \mathbf{arg}\ M_2, \rho)\rangle & (\mathit{rator})\\
\mathsf{Ev}\langle \lambda x.M, \rho, \kappa\rangle & \to_{cek} & \mathsf{Ret}\langle\langle \mathsf{cl}\ \lambda x.M, \rho\rangle, \kappa\rangle & (\mathit{lambda})\\
\mathsf{Ev}\langle c, \rho, \kappa\rangle & \to_{cek} & \mathsf{Ret}\langle c, \kappa\rangle & (\mathit{constant})\\
\mathsf{Ev}\langle x, \rho, \kappa\rangle & \to_{cek} & \mathsf{Ret}\langle \rho(x), \kappa\rangle & (\mathit{variable})\\
\mathsf{Ret}\langle V, (\kappa\ \mathbf{arg}\ M, \rho)\rangle & \to_{cek} & \mathsf{Ev}\langle M, \rho, (\kappa\ \mathbf{fun}\ V)\rangle & (\mathit{rand})\\
\mathsf{Ret}\langle V, (\kappa\ \mathbf{fun}\ \langle \mathsf{cl}\ \lambda x.M, \rho\rangle)\rangle & \to_{cek} & \mathsf{Ev}\langle M, \rho[x \to V], \kappa\rangle & (\mathit{apply})\\
\mathsf{Ret}\langle (\mathsf{cons}\ V_1\ V_2), (\kappa\ \mathbf{fun}\ \mathsf{car})\rangle & \to_{cek} & \mathsf{Ret}\langle V_1, \kappa\rangle & (\mathit{car})\\
\mathsf{Ret}\langle (\mathsf{cons}\ V_1\ V_2), (\kappa\ \mathbf{fun}\ \mathsf{cdr})\rangle & \to_{cek} & \mathsf{Ret}\langle V_2, \kappa\rangle & (\mathit{cdr})\\
\mathsf{Ret}\langle V, (\kappa\ \mathbf{fun}\ f)\rangle & \to_{cek} & \mathsf{Ret}\langle \delta(f, V), \kappa\rangle & (\delta)
\end{array}
$$

Figure 4: CEK Transitions

Unitary Cost Function $\mathcal{K}_u$

$\mathcal{K}_u(\langle \mathsf{Ret}\langle \varphi_t, (\kappa\ \mathbf{fun}\ (((\mathsf{call\text{-}with\text{-}group}\ F)\ e)\ \varphi_e))\rangle, g\rangle, \Gamma) = e + 1$

$\mathcal{K}_u(\langle \mathsf{Ret}\langle e, (\kappa\ \mathbf{fun}\ (\mathsf{awake}\ g_1))\rangle, g\rangle, \Gamma) = e + 1 \quad$ if $g \neq g_1, \Gamma(g_1).s \neq \mathsf{terminated}$

otherwise, $\mathcal{K}_u(\langle C, g\rangle, \Gamma) = 1$

Soundness Constraints on Cost Functions $\mathcal{K}$

$\mathcal{K}(\langle \mathsf{Ret}\langle \varphi_t, (\kappa\ \mathbf{fun}\ (((\mathsf{call\text{-}with\text{-}group}\ F)\ e)\ \varphi_e))\rangle, g\rangle, \Gamma) > e$

$\mathcal{K}(\langle \mathsf{Ret}\langle e, (\kappa\ \mathbf{fun}\ (\mathsf{awake}\ g_1))\rangle, g\rangle, \Gamma) > e \quad$ if $g \neq g_1, \Gamma(g_1).s \neq \mathsf{terminated}$

otherwise, $\mathcal{K}(\langle C, g\rangle, \Gamma) > 0$

Figure 5: Cost Function

$$
\begin{array}{ll}
\langle\{\ \langle C,g\rangle\ \}\ \cup\ T,\Gamma,\sigma\rangle & \\
\quad\rightarrow\quad \langle\{\ \langle C_1,g\rangle\ \}\ \cup\ T,\Gamma[g.e := g.e-\mathcal{K}],\sigma\rangle\quad \text{if } C\rightarrow_{cek} C_1\quad(\star) & (\mathit{sequential})\\
\langle\{\ \langle \mathsf{Ev}\langle(\mathsf{parallel}\ M_1\ M_2,\rho,\kappa\rangle,g\rangle\ \}\ \cup\ T,\Gamma,\sigma\rangle & \\
\quad\rightarrow\quad \langle\{\ \langle \mathsf{Ev}\langle M_1,\rho,\kappa\rangle,g\rangle\ \}\ \cup\ \{\ \langle \mathsf{Ev}\langle M_2,\rho,\kappa\rangle,g\rangle\ \}\ \cup\ T, & (\mathit{sequential})\\
\qquad\quad \Gamma[g.e := g.e-\mathcal{K}][g.n := g.n+1],\sigma\rangle\quad(\star) & (\mathit{parallel})\\
\langle\{\ \langle \mathsf{Ev}\langle(\mathsf{suicide}),\rho,\kappa\rangle,g\rangle\ \}\ \cup\ T,\Gamma,\sigma\rangle & \\
\quad\rightarrow\quad \langle T,\Gamma[g.e := g.e-\mathcal{K}][g.n := g.n-1],\sigma\rangle\quad(\star) & (\mathit{suicide})\\
\langle\{\ \langle \mathsf{Ret}\langle \mathsf{void},(\mathbf{init})\rangle,g\rangle\ \}\ \cup\ T,\Gamma,\sigma\rangle & \\
\quad\rightarrow\quad \langle T,\Gamma[g.e := g.e-\mathcal{K}][g.n := g.n-1],\sigma\rangle\quad(\star) & (\mathit{init})
\end{array}
$$

Convention: $(\star)\equiv\Gamma(g).e\geq\mathcal{K}$

Figure 6: Sequential and Parallel Evaluations

A configuration of the $\mathcal{Q}$-machine, represented as $\mathcal{M}$ in Figure 3, is a triple composed of a set of *tasks*, a set of groups and their associated information, and a set of channels and their contents. A *task*, represented by a pair $\langle C,g\rangle$, is an entity, sponsored by group g, that embodies a *computational state* C. In $\mathcal{Q}$uantum, tasks are anonymous and are not first-class values; instead, groups are reified as first-class objects as a mean to monitor and control computations. The function Γ associates each group g with a parent group, its current energy, its state, the two notifers φ_e and φ_t, and the number of tasks and the set of subgroups that it sponsors. The hierarchy root is the *initial group*, and by convention, the parent of the initial group is represented by $\perp_g$.

Notifiers are closures with a signature $Group\ \rightarrow\ Energy \rightarrow Void$, which receive the group that is notified the event and its remaining energy. As notifications are asynchronous, they are not expected to return values, hence the void value returned. Channels are first-class values represented by $\langle\mathsf{ch}\ \alpha\rangle$, with α a location pointing to a queue in the queue store σ.

The computational state of a task is a CEK-configuration [8] represented as $\mathsf{Ev}\langle M,\rho,\kappa\rangle$ or $\mathsf{Ret}\langle V,\kappa\rangle$, respectively representing the evaluation of a term M in the environment ρ with a continuation κ, and the return of a value V to a continuation κ. The continuation κ is encoded by a data-structure, called *continuation code*.

Figure 4 displays the transition rules for the sequential purely functional subset of the language; for more detail, we refer the reader to [8].

As previously mentioned, the intension of $\mathcal{Q}$uantum is to measure the resources used by computations. In order to be generic, we decided to associate the semantics with a *cost function* $\mathcal{K}$, giving each transition its cost in terms of energy.

Warning. Let us observe that the cost of a transition is a function of the task involved in the transition and of the function Γ. For the sake of concision, we do not represent this dependency explicitly. We use the symbol $\mathcal{K}$ to denote the value of the cost function for the task involved in the transition and a given Γ. For instance, in rule (*sequential*), the task involved in the transition is $\langle C, g\rangle$; therefore, the symbol $\mathcal{K}$ stands for $\mathcal{K}(\langle C, g\rangle, \Gamma)$.

Rule (*sequential*) of Figure 6 states that if there exists a task $\langle C, g\rangle$ sponsored by a group g, such that a CEK-transition reduces C to C_1, then after transition the task becomes $\langle C_1, g\rangle$; the energy of g is decremented by the cost of the transition; the other tasks remain unchanged. Rule (*sequential*), as most other rules, assumes that the energy associated with g is greater than the cost of the transition, which is represented by the side-condition noted $(\star)$.

We use the following notations for accessing and modifying components of the tuple associated with a group g. If $\Gamma(g) = \langle g_p, e, s, \varphi_e, \varphi_t, n, g^*\rangle$, then $\Gamma(g).p = g_p$, $\Gamma(g).e = e$, $\Gamma(g).s = s$, $\Gamma(g).n = n$, $\Gamma(g).g^* = g^*$. Updates are written as follows: $\Gamma[g.e := e_1]$ denotes $\Gamma[g \to \langle g_p, e_1, s, \varphi_e, \varphi_t, n, g^*\rangle]$, $\Gamma[g.s := s_1]$ denotes $\Gamma[g \to \langle g_p, e, s_1, \varphi_e, \varphi_t, n, g^*\rangle]$. Sometimes, we even combine both conventions so that $\Gamma[g.n := g.n - 1]$ should be read as $\Gamma[g \to \langle g_p, e, s, \varphi_e, \varphi_t, n-1, g^*\rangle]$.

As many cost models are conceivable, we decided to parameterise the semantics by the cost model. Figure 5 gives the definition of a cost function $\mathcal{K}_u$, charging a unitary cost for every transition, in addition to the quantity of energy transferred. Other definitions are acceptable as long as they satisfy the soundness constraints of Figure 5, which preserve the following principles: first, transferring some energy costs this amount of energy at least; second, every computation step has a cost.

In Figure 6, the rule for the construct (parallel M_1 M_2) creates two tasks to evaluate M_1 and M_2 with the same environment ρ and continuation κ, resulting in an additional task in the current group. The construct (suicide) removes the current task from its sponsoring group; the $\mathcal{Q}$-machine behaves similarly when a void value is returned to the initial continuation.

Rules dealing with channels, which appear in Figure 7, are straightforward. The construct (channel) returns a new channel $\langle \mathsf{ch}\ \alpha\rangle$ with a newly allocated location α bound to an empty queue in the queue store. The primitive enqueue adds a value V at the end of the queue associated with the channel, while dequeue takes the first element of the queue. Note that the transition (*dequeue*) is allowed to be fired only if the queue is not empty; as a result, a task is not allowed to progress when trying to dequeue an element from an empty channel. For the sake of simplicity, we have decided not to associate a cost with such a "blocked" task. This could be easily overcome by adding a rule for the empty queue case which would charge its cost to the sponsoring group.

Figure 8 displays rules related to groups. Groups are created by evaluating (call-with-group F e φ_e φ_t), which results in the application of the partial application (((call-with-group F) e) φ_e) on φ_t. Then, rule (*make group*) creates a new group g_1 in a running state, whose parent is the sponsoring group g, with an energy e, with one sponsored task applying the closure F on g_1. Following the soundness constraints of Figure 5, the sponsoring group g is deducted of the transition cost, which includes the energy e given to g_1: rule (*make group*) guarantees that no energy is generated during group creation.

$$\begin{array}{ll}
\langle\{\ \langle \mathsf{Ret}\langle(\mathsf{channel}),\kappa\rangle, g\rangle\ \}\ \cup\ T,\Gamma,\sigma\rangle & \\
\quad \to \quad \langle\{\ \langle \mathsf{Ret}\langle\langle \mathsf{ch}\ \alpha\rangle,\kappa\rangle, g\rangle\ \}\ \cup\ T,\Gamma[g.e := g.e-\mathcal{K}],\sigma[\alpha \to \langle\rangle]\rangle & (\mathit{channel}) \\
\qquad \text{with}\ \ \alpha \notin DOM(\sigma)\ \ (\star) & \\
\langle\{\ \langle \mathsf{Ret}\langle V,(\kappa\ \mathbf{fun}\ (\mathsf{enqueue}\ \langle\mathsf{ch}\ \alpha\rangle))\rangle, g\rangle\ \}\ \cup\ T,\Gamma,\sigma\rangle & \\
\quad \to \quad \langle\{\ \langle \mathsf{Ret}\langle\mathsf{void},\kappa\rangle, g\rangle\ \}\ \cup\ T,\Gamma[g.e := g.e-\mathcal{K}],\sigma[\alpha := \sigma(\alpha)\ \S\ \langle V\rangle]\rangle\quad(\star) & (\mathit{enqueue}) \\
\langle\{\ \langle \mathsf{Ret}\langle\langle \mathsf{ch}\ \alpha\rangle,(\kappa\ \mathbf{fun}\ \mathsf{dequeue})\rangle, g\rangle\ \}\ \cup\ T,\Gamma,\sigma\rangle & \\
\quad \to \quad \langle\{\ \langle \mathsf{Ret}\langle V,\kappa\rangle, g\rangle\ \}\ \cup\ T,\Gamma[g.e := g.e-\mathcal{K}],\sigma[\alpha := q]\rangle\quad \text{if}\ \sigma(\alpha)=\langle V\rangle\ \S\ q\ \ (\star) & (\mathit{dequeue})
\end{array}$$

Convention: $(\star) \equiv \Gamma(g).e \geq \mathcal{K}$

Figure 7: Channels Related Operations

$$\begin{array}{ll}
\langle\{\ \langle \mathsf{Ret}\langle\varphi_t,(\kappa\ \mathbf{fun}\ (((\mathsf{call\text{-}with\text{-}group}\ F)\ e)\ \varphi_e))\rangle, g\rangle\ \}\ \cup\ T,\Gamma,\sigma\rangle & \\
\quad \to \quad \langle\{\ \langle \mathsf{Ret}\langle g_1,((\kappa\ \mathbf{rgroup})\ \mathbf{fun}\ F)\rangle, g_1\rangle\ \}\ \cup\ T,\Gamma_1,\sigma\rangle\ \ (\star) & (\mathit{make\ group}) \\
\qquad \text{with}\ \Gamma_1 = \Gamma[g.e := g.e-\mathcal{K}][g.n := g.n-1][g.g^* := g.g^* \cup \{g_1\}][g_1 \to \langle g, e, \mathsf{running}, \varphi_e, \varphi_t, 1, \emptyset\rangle], & \\
\qquad g_1 \notin DOM(\Gamma) & \\
\langle\{\ \langle \mathsf{Ret}\langle V,(\kappa\ \mathbf{rgroup})\rangle, g\rangle\ \}\ \cup\ T,\Gamma,\sigma\rangle & \\
\quad \to \quad \langle\{\ \langle \mathsf{Ret}\langle V,\kappa\rangle, g_1\rangle\ \}\ \cup\ T,\Gamma[g.e := g.e-\mathcal{K}][g.n := g.n-1][g_1.n := g_1.n+1],\sigma\rangle & (\mathit{return\ group}) \\
\qquad \text{with}\ g_1 = \Gamma(g).p\quad (\star) & \\
\langle\{\ \langle \mathsf{Ret}\langle g_1,(\kappa\ \mathbf{fun}\ \mathsf{within\text{-}group?})\rangle, g\rangle\ \}\ \cup\ T,\Gamma,\sigma\rangle & \\
\quad \to \quad \langle\{\ \langle \mathsf{Ret}\langle g_1,(\kappa\ \mathbf{fun}\ (\mathsf{within\text{-}group?'}\ g))\rangle, g\rangle\ \}\ \cup\ T,\Gamma[g.e := g.e-\mathcal{K}],\sigma\rangle\quad(\star) & (\mathit{within\text{-}group\ 1}) \\
\langle\{\ \langle \mathsf{Ret}\langle g_2,(\kappa\ \mathbf{fun}\ (\mathsf{within\text{-}group?'}\ g_1))\rangle, g\rangle\ \}\ \cup\ T,\Gamma,\sigma\rangle & \\
\quad \to \quad \langle\{\ \langle \mathsf{Ret}\langle g_2 = g_1,\kappa\rangle, g\rangle\ \}\ \cup\ T,\Gamma[g.e := g.e-\mathcal{K}],\sigma\rangle & (\mathit{within\text{-}group\ 2}) \\
\qquad \text{if}\ (g_2 = g_1 \vee g_1 = \bot_g)\quad (\star) & \\
\quad \to \quad \langle\{\ \langle \mathsf{Ret}\langle g_2,(\kappa\ \mathbf{fun}\ (\mathsf{within\text{-}group?'}\ \Gamma(g_1).p))\rangle, g\rangle\ \}\ \cup\ T,\Gamma[g.e := g.e-\mathcal{K}],\sigma\rangle & (\mathit{within\text{-}group\ 3}) \\
\qquad \text{if}\ g_2 \neq g_1,\ g_1 \neq \bot_g\quad (\star) &
\end{array}$$

Convention: $(\star) \equiv \Gamma(g).e \geq \mathcal{K}$

Figure 8: Groups Related Operations

$$\langle \{ \ \langle C, g \rangle \ \} \ \cup \ T, \Gamma, \sigma \rangle$$

$$\quad \rightarrow \quad \langle \{ \ \langle \mathsf{Ret}\langle g, (((\mathbf{init}) \ \mathbf{arg} \ e, \emptyset) \ \mathbf{fun} \ \varphi_e)\rangle, g_1 \rangle \ \} \ \cup \ \{ \ \langle C, g \rangle \ \} \ \cup \ T, \Gamma_1, \sigma \rangle \qquad (\textit{exhaustion})$$

$$\qquad \text{if } \Gamma(g) = \langle g_1, e, \mathsf{running}, \varphi_e, \varphi_t, n, g^* \rangle, \ g_1 \neq \bot_g, \ e < \mathcal{K}(\langle C, g \rangle), \ \Gamma(g_1).s = \mathsf{running}$$

$$\qquad \text{with } \Gamma_1 = \Gamma[g.s := \mathsf{exhausted}][g.e := 0][g_1.e := g_1.e + e][g_1.n := g_1.n + 1]$$

$$\langle T, \{ \ (g \rightarrow \langle g_1, e, \mathsf{running}, \varphi_e, \varphi_t, n, g^* \rangle) \ \} \ \cup \ \Gamma, \sigma \rangle$$

$$\quad \rightarrow \quad \langle \{ \ \langle \mathsf{Ret}\langle g, (((\mathbf{init}) \ \mathbf{arg} \ e, \emptyset) \ \mathbf{fun} \ \varphi_t)\rangle, g_1 \rangle \ \} \ \cup \ T, \Gamma_1, \sigma \rangle \qquad (\textit{termination})$$

$$\qquad \text{if } g_1 \neq \bot_g, \ \Gamma(g_1) = \langle g_2, e_1, s, \varphi_{e\,1}, \varphi_{t\,1}, n_1, g_1^* \rangle, \ n = 0, \ g^* = \emptyset, \ \Gamma(g_1).s = \mathsf{running}$$

$$\qquad \text{with } \Gamma_1 = \Gamma[g \rightarrow \langle g_1, 0, \mathsf{terminated}, \varphi_e, \varphi_t, n, g^* \rangle][g_1 := \langle g_2, e_1 + e, s, \varphi_{e\,1}, \varphi_{t\,1}, n_1 + 1, (g_1^* \setminus \{g\}) \rangle]$$

Figure 9: Asynchronous Notifications

$$\langle \{ \ \langle \mathsf{Ret}\langle \mathsf{nil}, (\kappa \ \mathbf{fun} \ (\mathsf{pause} \ \varphi_p))\rangle, g \rangle \ \} \ \cup \ T, \Gamma, \sigma \rangle$$

$$\quad \rightarrow \quad \langle \{ \ \langle \mathsf{Ret}\langle \mathsf{void}, \kappa \rangle, g \rangle \ \} \ \cup \ T, \Gamma[g.e := g.e - \mathcal{K}], \sigma \rangle \quad (\star) \qquad (\textit{pause group 1})$$

$$\langle \{ \ \langle \mathsf{Ret}\langle (\mathsf{cons} \ g_1 \ g^*), (\kappa \ \mathbf{fun} \ (\mathsf{pause} \ \varphi_p))\rangle, g \rangle \ \} \ \cup \ T, \Gamma, \sigma \rangle$$

$$\quad \rightarrow \quad \langle \{ \ \langle \mathsf{Ret}\langle (g^* \ \S \ \Gamma(g_1).g^*), (\kappa \ \mathbf{fun} \ (\mathsf{pause} \ \varphi_p))\rangle, g \rangle \ \} \ \cup \ \{ \ t_1 \ \} \ \cup \ T, \Gamma_1, \sigma \rangle$$

$$\qquad \text{if } g \neq g_1, \ \Gamma(g_1).s \neq \mathsf{terminated} \quad (\star) \qquad (\textit{pause group 2})$$

$$\qquad \text{with } t_1 = \langle \mathsf{Ret}\langle g_1, (((\mathbf{init}) \ \mathbf{arg} \ e, \emptyset) \ \mathbf{fun} \ \varphi_p)\rangle, g_2 \rangle$$

$$\qquad \text{with } \Gamma_1 = \Gamma[g.e := g.e + e - \mathcal{K}][g_1.e := 0][g_1.s := \mathsf{exhausted}][g_2.n := g_2.n + 1]$$

$$\qquad \text{with } g_2 = \Gamma(g_1).p, e = \Gamma(g_1).e$$

$$\quad \rightarrow \quad \langle \{ \ \langle \mathsf{Ret}\langle (g^* \ \S \ \Gamma(g_1).g^*), (\kappa \ \mathbf{fun} \ (\mathsf{pause} \ \varphi_p))\rangle, g \rangle \ \} \ \cup \ T, \Gamma[g.e := g.e - \mathcal{K}], \sigma \rangle \qquad (\textit{pause group 3})$$

$$\qquad \text{if } (g = g_1 \vee \Gamma(g_1).s = \mathsf{terminated}) \quad (\star)$$

$$\langle \{ \ \langle \mathsf{Ret}\langle e, (\kappa \ \mathbf{fun} \ (\mathsf{awake} \ g_1))\rangle, g \rangle \ \} \ \cup \ T, \Gamma, \sigma \rangle$$

$$\quad \rightarrow \quad \langle \{ \ \langle \mathsf{Ret}\langle \mathsf{void}, \kappa \rangle, g \rangle \ \} \ \cup \ T, \Gamma[g.e := g.e - \mathcal{K}][g_1.e := g_1.e + e][g_1.s := \mathsf{running}], \sigma \rangle \quad (\textit{awake group 1})$$

$$\qquad \text{if } \ g \neq g_1, \ \Gamma(g_1).s \neq \mathsf{terminated} \quad (\star)$$

$$\quad \rightarrow \quad \langle \{ \ \langle \mathsf{Ret}\langle \mathsf{void}, \kappa \rangle, g \rangle \ \} \ \cup \ T, \Gamma[g.e := g.e - \mathcal{K}], \sigma \rangle \qquad (\textit{awake group 2})$$

$$\qquad \text{if } \ (g = g_1 \vee \Gamma(g_1).s = \mathsf{terminated}) \quad (\star)$$

Convention: $(\star) \equiv \Gamma(g).e \geq \mathcal{K}$

Figure 10: Pause and Awake Operations on Groups

Let us notice that F is applied on g_1, with a continuation (κ **rgroup**) indicating that the evaluation is performed under the sponsorship of a group. When a value is returned to the continuation code **rgroup** as in (*return group*), the task leaves the sponsorship of its group: the task that was sponsored by g now becomes sponsored by its parent g_1; the numbers of tasks sponsored by g and g_1 are updated accordingly.

The predicate within-group? returns true if it is being applied under the sponsorship of the group received in argument, or under the sponsorship of a subgroup of its argument. Such a predicate is implemented by a simple recursion on the parent chain.

Rules controlling asynchronous notifications appear in Figure 9. If the energy required by a task is greater than the energy available in its sponsoring group g, rule (*exhaustion*) changes the state of g to exhausted, and creates a new task applying the notifier φ_e on g and the remaining energy e. Let us observe that the notification is executed under the sponsorship of the parent group g_1, and that the energy e is transferred to g_1.

Asynchronous termination detection follows a similar pattern: if a group g does not sponsor any task and has no subgroup, a notifier φ_t is applied on g and the remaining energy e under the sponsorship of the parent group g_1; the remaining energy is also transferred to g_1.

As notifications are executed under the sponsorship of the parent of the group terminating or being exhausted, care should be taken not to apply these rules to the root of the hierarchy, which is expressed by the condition $g_1 \neq \perp_g$.

A notifier is defined as a user function. Evaluating a call to a notifier is a notification. Posting a notification is creating a task that performs a notification. So far, every computation transition is given a cost. On the contrary, rules (*exhaution*) and (*termination*), which post notifications, do not decrease total energy. At the price of complexity, the semantics could be extended so that a cost can be associated with notification posting. More judiciously, posting notifications could be charged by a new cost model, where posting is prepaid at group-creation or group-awakening time.

Notification rules transfer energy from the notified group to its parent, avoiding energy loss. Notifications allow the user program to *observe* semantically-caused energy transfers between groups. Even though the user code is given access to the amount of energy transferred, energy accounting remains strictly under control of the semantics, which guarantees the safeness of the approach.

The semantics of pause and awake is displayed in Figure 10. The primitive pause requires two arguments: a notification function φ_p and a list of groups to be paused. For each group g_1 of the list, rule (*pause group 2*) sets the state of g_1 to exhausted, transfers its remaining energy e to the group g sponsoring the pause action, creates a task applying the notifier φ_p on g_1 and e under the sponsorship of g_2 the parent of g_1, and adds the subgroups of g_1 to the list of groups remaining to be processed.

Special care is taken in rule (*pause group 3*) to avoid pausing the current group or to avoid setting a terminated group to the exhausted state. In rule (*pause group 1*), we see that the primitive pause returns a void value as it is used for its side-effect on group energies.

The primitive awake takes the group to be awakened and the energy to be transferred in arguments. Assuming the sponsoring group g has enough energy, rule (*awake group 1*) decrements its energy, increments the energy of the awakened group g_1, and sets it to the state running. Again care is taken to avoid awakening a terminated group. Let us observe again that the user specifies the amount of energy to be transferred but accounting is strictly performed at the semantic level.

$eval_{\mathcal{K}}(M,e) = V$ if $\exists\langle T,\Gamma,\sigma\rangle$,
such that $Final(\langle T,\Gamma,\sigma\rangle)$,
with $\langle\{\langle \mathsf{Ev}\langle M,\emptyset,\kappa_0\rangle, g_0\rangle\}, \Gamma_0, \sigma_0\rangle \rightarrow^* \langle T,\Gamma,\sigma\rangle$
and $V \in \sigma(\alpha_0)$.

$$\begin{array}{rcl}
\Gamma_0 & = & \{g_0 \rightarrow \langle \bot_g, e, \varphi_{e0}, \varphi_{t0}, 1, \langle\rangle\}\\
\sigma_0 & = & \{\alpha_0 \rightarrow \langle\rangle\}\\
\kappa_0 & = & ((\mathbf{init})\ \mathbf{fun}\ (\mathsf{enqueue}\ \langle \mathsf{ch}\ \alpha_0\rangle))\\
Final(\langle T,\Gamma,\sigma\rangle) & \equiv & \not\exists\langle T',\Gamma',\sigma'\rangle,\ \langle T,\Gamma,\sigma\rangle \rightarrow \langle T',\Gamma',\sigma'\rangle\\
\varphi_{e0} = \varphi_{t0} & = & \langle \mathsf{cl}\ \lambda g.\lambda e.\mathsf{void}, \emptyset\rangle
\end{array}$$

Figure 11: Evaluation Relation

Figure 11 displays the evaluation *relation* of the language. Evaluation starts with an initial configuration, composed of the initial group g_0, a queue store containing a location α_0 aimed at receiving all values generated by the computation, and an initial task; this task evaluates the program in an empty environment, and with a continuation accumulating the results obtained in location α_0. The evaluation relation associates a program with all the possible final results that can be accumulated in α_0.

Let us note that the evaluation relation is parameterised by a cost function $\mathcal{K}$ and by the initial energy quota e given to the $\mathcal{Q}$-machine. The initial group has no parent, receives the initial energy quota, sponsors the initial task; the notification functions are arbitrary because they are never called, as seen in rules (*exhaustion*) and (*termination*).

We establish the soundness of the semantics with respect to energy by the next two propositions.

Proposition 1 For any cost function satisfying the constraints of Figure 5, total energy decreases as evaluation (notification posting excluded) proceeds. □

Corollary 2 For any cost function satisfying the constraints of Figure 5, and for a finite positive initial energy, any computation is finite. □

4 Examples

In this Section, we present examples written in Scheme extended with our primitives. Even though $\mathcal{Q}$uantum substantially differs from an idealised Scheme language, it is expressive enough to model first-class mutable boxes and first-class continuations.

Figure 12 displays the code for mutable boxes in *Quantum*, where a mutable box is represented by a channel. Functions *deref* and *setref!* maintain the invariant that the channel contains one and only one value, by first dequeuing the current value, and then enqueuing another one. Simultaneous accesses to a same box are protected by the atomicity of the primitive *dequeue*.

```
(define (makeref V)
  (let ((c (channel)))
    (enqueue c V)
    c))
(define (deref c)
  (with-enough-energy
    (let ((v (dequeue c)))
      (enqueue c v)
      v)))
(define (setref! c v)
  (with-enough-energy
    (let ((old (dequeue c)))
      (enqueue c v)
      old)))
```

Figure 12: First-Class mutable boxes

However, a program could run out of energy after having read a value and before having stored the new one into the channel: this would leave the box in a inconsistent state, unusable by other tasks. Therefore, we must be sure that an exhaustion notification cannot occur between these two operations. This kind of "energy-critical section" is implemented by creating a group which receives the amount of energy `minimal-energy` required to perform both operations (Figure 13).

Let us notice that such a group does not prevent the program to be paused from outside. However, if such a pausing action occurs, it can only be caused by the user's program. It is his role to ensure that a paused group does not leave objects likes boxes in an inconsistent state.

First-class continuations were rejected in the design of *Quantum* because they could be used to resume terminated computations; such a feature would be in contradiction with the stability of the termination property. However, Figure 14 shows that *call/cc* [30] can be programmed in *Quantum*. A first-class continuation is represented by a communication channel; invoking a continuation on a value sends the value on the communication channel. In addition, *call/cc* creates a parallel task which acts as a *server*: it dequeues any value sent on the channel, spawns a task to continue the rest of evaluation, and repeats the process with the next value to be dequeued. The unlimited nature of the extent of the continuation

```
(define-syntax with-enough-energy
  (syntax-rules ()
    ((with-enough-energy form ...)
     (call-with-group (lambda (g)
                         form ...)
                       minimal-energy
                       refill-handler
                       ignore-handler))))
(define (refill-handler g e)
  (awake g (+ e 1)))
(define (ignore-handler g e)
  (suicide))
```

Figure 13: Energy-critical Section

is modelled in Quantum by the presence of such a server, which must be given energy as long as the continuation remains invokable.

Non-local block exits are a restricted form of continuation, which can only be called in the dynamic extent of the construct that created them. In Figure 14, *call/ep*, which stands for *call-with-exit-procedure*, is derived from the code of *call/cc*. Here, we create a group and when we detect its termination, we send a distinguished value in the channel to shutdown the server. Furthermore, the predicate *within-group?* is used to detect that a continuation is applied in the dynamic extent of the *call/ep* construct.

In traditional computing, we have no tool to tell us whether a computation is running or what is the amount of work done by a given task. Such tools usually exist at the operating-system level, but deal with "processes" and not with tasks, and they do not take into account tasks executed on remote hosts. Figure 15 displays the code of a probe, which updates the content of a box with the amount of energy already consumed by a computation generated by a *thunk*. When the computation ends, the box is updated with a pair indicating the total consumption of the *thunk*.

In Figure 16, the primitive *pause* is used with three different intentions. The function *suspend* temporarily pauses a group hierarchy. The hierarchy will be resumed by awakening its root, which will awake its subgroups step by step using the notifiers left by *pause*. Note that the condition in the notifier prevents to awake the root of the hierarchy immediately after the root has received the notification.

On the contrary, the function *kill* pauses a hierarchy without leaving any opportunity to resume it, unless the programmer has explicitly kept handles on the groups that belong to the hierarchy, and explicitly awakes them.

Last, *adjust-energy* pauses and immediately resumes a hierarchy with a quota of energy which is proportional to the one it had before. The energy transfer that occurs during this operation is worth noticing: the group calling *adjust-energy* will be credited of the energy of

```
(define (call/cc f)
  (let ((c (channel)))
    (parallel
     (let loop ((v (dequeue c)))
       (parallel v (loop (dequeue c))))
     (f (lambda (v)
          (enqueue c v)
          (suicide))))))
(define (call/ep f)
  (let ((c   (channel))
        (end (cons 1 2)) )
    (parallel
     (let loop ((v (dequeue c)))
       (if (eq? v end)
           (suicide)
           (parallel v (loop (dequeue c)))))
     (call-with-group (lambda (g)
                        (f (lambda (v)
                             (if (within-group? g)
                                 (begin
                                   (enqueue c v)
                                   (suicide))
                                 (error)))))
                      1
                      refill-handler
                      (lambda (g)
                        (enqueue c end))))))
```

Figure 14: First-Class Continuations

hierarchy before adjustment, while the energy of the hierarchy after adjustment is deducted from the parent of the root. In order to guarantee that *awake* is executed after *pause*, we introduce an explicit synchronisation by the channel *go*.

Figures 17 and 18 display the code of a service provider, making use of Wright and Duba's pattern-matching macro [35]. A communication channel *subscription-channel* is publicly advertised as the entry point to the service provider. A user is allowed to subscribe to the service by giving its name (possibly authenticated by a specialised protocol) some electronic cash, and a channel to which the service provider answers. The service provider creates a new communication channel that is returned to the user and that will be used as an access point to resources offered by the service provider. This access point is served

```
(define (probe box thunk unit)
  (call-with-group
        (lambda (g) (thunk))
        unit
        (lambda (g e)
          (setref! box (+ unit (deref box)))
          (awake g (+ unit e)))
        (lambda (g e)
          (setref! box (cons 'has-consumed
                             (- (deref box) e))))))
```

Figure 15: Probe

```
(define (suspend group)
  (pause (lambda (g e)
           (if (not (eq? g group))
               (awake g e)))
         (list group)))
(define (kill group)
  (pause (lambda (g e)
           (suicide))
         (list group)))
(define (adjust-energy group coefficient)
  (let ((go (channel)))
    (pause (lambda (g e)
             (if (eq? g group)
                 (dequeue go))
             (awake g (* e coefficient)))
           (list group))
    (enqueue go #t)))
```

Figure 16: Pause and Awake of Computations

by a *request-server*, whose operations are sponsored by a group that has received an initial amount of energy corresponding to the electronic cash transmitted at subscription time.

The user can submit jobs to the *request-server*, which in turn creates a new group to sponsor the evaluation of *job* and returns it to the user. This group can be used by the user to pause, kill, or restart a computation. Let us observe that the user is never given a handle either on the group initially created with his electronic cash, or on the group sponsoring the administration program. Thanks to lexical scoping, these groups can be hidden, and security is insured because nobody will be able to pause and steal energy from such groups.

When the account of a user is exhausted, a message is sent to the user, who gets the opportunity to transfer more electronic cash to his account via a message 'pay. This request is sent on the publicly advertised channel, *subscription-channel*, and might need some authentication protocol.

The service providers may also offer a demonstration account which will be usable for a fixed quota of energy energy-quota-for-free-demo and which may offer restricted facilities only. This program can be extended by offering the possibility to close an account and to refund the electronic cash corresponding to the remaining energy.

5 Implementation Hints

This section describes the main lines of a possible implementation. First, it presents a solution in a non-distributed setting then extends that implementation to cover distributed aspects.

5.1 Single Space Model

The parallel model is rather simple. Although tasks are not first-class values, they do exist within the implementation. They are created with **parallel** and are terminated by *suicide*. As usual, a scheduler manages the set of all tasks.

A task contains a reference to its sponsoring group and embodies the continuation of the evaluation it represents; the continuation includes the marks left by groups when created. Therefore, *within-group?* is easily implemented by inspecting the continuation.

Groups are hierarchically organised; a group knows *(i)* its parent group, *(ii)* its exhaustion and termination notifiers, *(iii)* the evolving set of *direct* tasks that is, the tasks that have this very group as sponsoring group, *(iv)* the evolving set of *direct* subgroups, that is, the groups that have this group as parent group.

Each group is also associated with a *tank of energy.* The energy spent by a running task is deducted from the tank of its sponsoring group. When a tank is exhausted, a notification is posted to the parent group. The scheduler prevents the tasks of a group with an empty tank to run.

When a group has no subgroup and looses its ultimate sponsored task, the group must be terminated: it is removed from the subgroups of its parent group and its tank is poured into the tank of its parent group. The group object may still be reachable from the user program, but nevertheless has entered its non-useful lifetime.

Energy may flow independently of the hierarchy of groups using the *pause* and *awake* operations. In order to *awake* a target group with a given amount of energy, we remove this energy from the tank of the current group, and tranfer it into the tank of the target group.

Pausing a group g is the most complex operation since it recursively dries the tanks of all the groups in the tree rooted at g. When *pause* empties the tank of a group g', it posts

```
(define subscription-channel (channel))
(define (service-provider subscription-channel)
  (let loop ()
    (let ((subscription (dequeue subscription-channel)))
      (parallel (process-subscription subscription)
                (loop)))))
(define (process-subscription subscription)
  (match subscription
    (('subscribe name ecash answer)
     (let ((private-channel (channel))
           (energy (ecash->energy ecash)))
       (call-with-group (lambda (g)
                          (register name g private-channel)
                          (enqueue answer private-channel)
                          (request-server private-channel g))
                        energy
                        (lambda (g e)
                          (enqueue answer
                                   "Account exhausted"))
                        ignore-handler)))

    (('pay name ecash)
     (let ((group (get-group name)))
       (awake group (ecash->energy ecash))))

    (('free-demo name answer)
     (call-with-group (lambda (g)
                        (let ((private-channel (channel)))
                          (enqueue answer private-channel)
                          (request-server private-channel g)))
                      energy-quota-for-free-demo
                      (lambda (g e)
                        (enqueue answer
                                 "Demo account exhausted"))
                      ignore-handler))
    (else 'discard))
  (suicide))
```

Figure 17: Service Provider (1)

```
(define (request-server channel sponsoring-group)
  (let ((message (dequeue channel)))
    (parallel
     (request-server channel sponsoring-group)
     (begin
       (match message
         (('submit job answer)
          (call-with-group (lambda (g)
                             (add-group! sponsoring-group g)
                             (enqueue answer `(created ,g))
                             (job))
                           1
                           refill-handler
                           (lambda (g e)
                             (remove-group! sponsoring-group g)
                             (enqueue answer `(done ,g)))))
         (('pause group answer)
          (if (member group (subgroups sponsoring-group))
              (begin
                (suspend group)
                (enqueue answer `(paused ,group)))
              (enqueue answer `(unknown ,group))))
         (('kill group answer)
          (if (member group (subgroups sponsoring-group))
              (begin
                (remove-group! sponsoring-group group)
                (kill group)
                (enqueue answer `(killed ,group)))
              (enqueue answer `(unknown ,group))))
         (('restart group answer)
          (if (member group (subgroups sponsoring-group))
              (begin
                (awake group 1)
                (enqueue answer `(restarted ,group)))
              (enqueue answer `(unknown ,group))))
         (else 'discard))
       (suicide)))))
```

Figure 18: Service Provider (2)

a notification in the parent group of g' mentioning the amount of stolen energy; the notifier is the function given as first argument to *pause*; the notification is a task whose job is to invoke the notifier with the robbed group and the stolen amount of energy.

The semantics rules are to be considered as atomic. Even though the *awake* function is atomic, the *pause* function is not atomic as it is defined by a set of rules. The *pause* function returns after the complete visit of a hierarchy of groups. This visit is performed by one or more tasks that run concurrently with all other tasks. Note also that multiple notifications for a same group may be simultaneously running if a group was paused or awakened multiply. Notifiers are not run in mutual exclusion.

Let us note that the garbage collector should reclaim unreachable groups that are both exhausted and without subgroups. Indeed, such groups cannot be awakened since they are unreachable, and no task can be created under their sponsorship according to the semantics.

5.2 Distributed Space Model

Distribution introduces multiple disjoint spaces of values linked by remote pointers and communicating by messages. Following the approach of [28], distribution is introduced by the construct (*placed-remote s f args* ...) which creates a remote task on site s, applying the function f on arguments *args*. The remote result(s) are returned to the continuation of the *placed-remote* call, on the site that evaluated this expression. All sites have a scheduler managing local tasks.

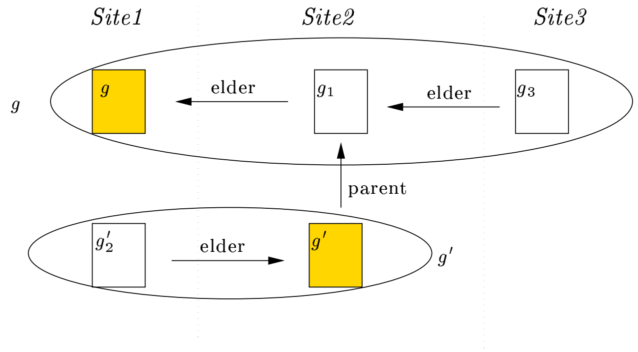

Figure 19: Local Groups and Brotherhood

The previously exposed implementation may be easily extended to cover distribution after introducing the notion of *group brotherhood*. A group stands for several tasks on several sites. To cope with distribution, a group is split into a hierarchical family of *local groups* ordered by a brotherhood relationship, each of them being local to a site. When a group g (see Figure 19) is created by *call-with-group* on *Site1*, it is known as the *eldest* *local group* of a family so far containing one brother only. When a task sponsored by g creates a remote task on *Site2*, a *younger local group* g_1 is created on *Site2* with g as elder brother. Symetrically, g adds g_1 as younger brother. When a subgroup g' is created by *call-with-group* on *Site2* under the sponsorship of g_1, its parent will be g_1.

To sum up, a *local group* belongs to a site and holds the following information: *(i)* its *elder* brother, *(ii)* its *direct younger* brothers, *(iii)* its direct local tasks, i.e. the evolving set of tasks that run under its sponsorship on the local site, *(iv)* its direct local subgroups, i.e. its locally created subgroups.

A group is represented by at most one local group per site; local groups are implementation entities, but only eldest local groups are first-class values that incarnate the group they stand for. Therefore, in addition to the information of a local group, an eldest local group also knows: *(i)* its parent group, *(ii)* its exhaustion and termination notifiers.

The key point of the distributed implementation is that brethren groups interact as groups in parenthood relationship except that they use implementation-defined notifiers instead of user-defined ones. When a younger local group runs out of energy, it asks its elder brother for a refill. When a younger local group terminates, i.e. when it has no subgroups, no younger brothers, and no tasks, it posts a notification (accompanied with its remaining energy) to its elder brother which then removes it from its younger brothers. When an eldest local group terminates, it reacts as previously described with respect to its parent group. The difficult case is when an eldest local group runs out of energy, since it should not post a notification to its parent unless all its younger brothers are also out of energy. The solution is then to dry the tanks of the younger brothers (but not the subgroups) to the benefit of the eldest brother. A group has no centralised tank but a collection of local tanks connected by implementation-defined notifiers. These inner transfers of energy are invisible to the user; they may use sophisticated techniques to ensure *load balancing* of energy inside the connected tanks of a group[3]. It is only when this connected tank is empty that the eldest local group is allowed to notify its parent of its exhaustion.

6 Discussion and Related Work

Our notion of group is at the intersection of two different ideas: Haynes and Friedman's engines and Kornfeld, Hewitt, and Osborne's sponsors, which we develop below.

Haynes and Friedman [12, 13] introduce the *engine* facility to model timed preemption; variants can also be found in [5, 6, 31]. Engines differ from our groups in a number of ways. Engines are defined in a *sequential* framework and are used to simulate multiprogramming. Since engines do not deal with parallelism, they do not offer control facilities like pause and awake. Another major difference is that a given engine can be executed several times, while a group can only be executed once. Using continuation terminology, engines are "multishot", while groups are "single-shot" [4]. A group is a name and an energy tank for a computation, but, unlike an engine, it does not embody its continuation. Our decision to

[3]The same schema may be used to provide each task with its own tank, hereby reducing communications with groups.

design "single-shot" groups is motivated as follows. The ability to restart several times a same computation is an unrealistic feature for a distributed language because the computation may be composed of several tasks distributed over the net. Haynes and Friedman also propose *nested engines*, i.e. engines that can create other engines. In their approach, nested engines have the same temporal vision of the world, because each computation consumes ticks, i.e. energy quanta, from parent engines (direct *and* indirect). On the contrary, groups offer more a distributed vision of the world, because groups are tanks, from which local tasks consume energy.

Kornfeld and Hewitt's sponsors [16], Osborne's enhanced version of them [22, 23, 11], and subsequently Queinnec's groups [28, 26], also allow the programmer to control hierarchies of computations in a parallel setting. Osborne's sponsors are entities that give attributes, like priority, to tasks, which can inherit attributes from several sponsors. A combining rule yields the effective attributes of a task, and then determines the resources allocated to the task. If the group hierarchy changes, priorities should be recomputed, which can be costly, especially in a distributed environment. With $\mathcal{Q}$uantum groups, scheduling of a task is only decided by examining the energy available in its only sponsoring group, which is local. Furthermore, priority is a difficult notion to grasp in a heterogeneous environment, while total work accomplished is more intuitive. In particular, our cost model allows us to program applications searching the best solution at a given amount of energy, i.e. at a given cost. Queinnec's Icsla language has a notion of group which substantially differs from the one presented here. As Icsla is energy-less, pausing a group does not collect energy and can be performed lazily. Also, Icsla does not have any of the notifications of $\mathcal{Q}$uantum. Let us observe that termination notification is a generalisation of `unwind-protect` [32]. Hieb and Dybvig [14] `spawn` operator returns a controller, which can be invoked to suspend or restart part of a computation tree; their approach relies on a notion of partial continuation.

There is an analogy between Unix `fork` and the construct parallel: `fork` duplicates processes, while they still share the same file system. However, Unix processes are organised hierarchically as opposed to our parallel tasks. Similarly, tasks created by `pcall` [21] or future [1, 7, 10, 11, 20] are ordered according to a "mandatory-speculative" relation. The construct pcall offers a "fork and join" type of parallelism, which could be simulated by using the queue store in $\mathcal{Q}$uantum[18, 28]. The flat model of independent computations provided by parallel is convenient to create independent servers.

Unix-like operating systems offer a different model to control processes: a process is given a unique identifier which can be used to send signals to it, e.g. kill. We believe that this model of control does not provide the appropriate abstraction [23]. Indeed, we want to be able to control computation, but we might not know the processes it is composed of, because the code we execute was not written by us. By associating groups with computations, we abstract from the details of execution. In addition, our model was designed for running in a distributed framework, so that groups can control tasks over different machines.

Our semantic is sound because it prevents generating energy. Furthermore, our language provides security by different means: *(i)* energy cannot be generated, but can only be transferred between computations; all "accounting" operations remain under absolute control of the semantics; *(ii)* groups are the only handle to control computations, and lexical scoping guarantees that groups will be visible only where the programmer wishes them to be, *(iii)* there is no primitive that returns the group in which the user code is running, which ensures that user code cannot control its sponsoring group, and hence it cannot control tasks running in parallel with it, unless explicitly passed handles to their sponsoring groups.

Our semantics measures the resources used by computations according to an independent cost model; our approach generalises the "complexity" monad [9, 33]. As our semantics is independent of the cost model, one might prefer to have more a realistic cost model, for instance, taking into account memory occupancy (duration or size) or blocked dequeue operations. In order to have a finer control on energy consumption, we noticed that the system could be extended to let the user create and manage different budgets from which energy can be consumed. An other interesting question is to decide how garbage collection should be billed in a multi-user environment, like the service provider. At the moment, garbage collection is part of the administration cost of the system and is indirectly billed via the cost of allocators.

7 Conclusion

In this paper, we present the language $\mathcal{Q}$uantum, whose purpose is to monitor and control computations in a parallel and distributed framework. $\mathcal{Q}$uantum provides a model to charge distributed computations, and therefore can be used to program account administration applications. Besides, it offers the end user the possibility to control hierarchies of computations, to do some load-balancing, and to introduce some energy-based priority between parallel computations.

$\mathcal{Q}$uantum is the core of a consumption-oriented language which is particularly suitable to program over the Internet. In the future, we plan to investigate a fault-tolerant version of the language, which would be energy aware.

Bibliography

[1] Henry Baker and C. Hewitt. The Incremental Garbage Collection of Processes. AI Memo 454, MIT AI Lab, March 1978.

[2] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, second edition, 1984.

[3] Robert S. Boyer and J. Struther Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the ACM*, 31(3):441–485, July 1984.

[4] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representig Control in the Presence of One-Shot Continuations. In *ACM SIGPLAN 96 Conference on Programming*

Language Design and Implementation, pages 99–107, Philadelphia, Pennsylvania, May 1996.

[5] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, 1987.

[6] Michael Eisenberg. *Programming in Scheme*. The Scientific Press, 507 Seaport Court. Redwood City. CA 94063-2731, 1988.

[7] Marc Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD thesis, Brandeis University, 1993.

[8] Matthias Felleisen and Daniel P. Friedman. Control Operators, the SECD-Machine and the λ-Calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217, Amsterdam, 1986. Elsevier Science Publishers B.V. (North-Holland).

[9] Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science. Carnegie Mellon University, May 1996.

[10] Cormac Flanagan and Matthias Felleisen. The Semantics of Future and Its Use in Program Optimization. In *Proceedings of the Twenty Second Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 1995. Technical Reports 238, 239, Rice University, 1994.

[11] Robert H. Halstead, Jr. New Ideas in Parallel Lisp : Language Design, Implementation. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp : Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, number 441 in Lecture Notes in Computer Science, pages 2–57. Springer-Verlag, 1990.

[12] Christopher T. Haynes and Daniel P. Friedman. Engines Build Process Abstractions. In *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming*, pages 18–24. ACM, 1984.

[13] Christopher T. Haynes and Daniel P. Friedman. Abstracting Timed Preemption with Engines. *Comput. Lang.*, 12(2):109–121, 1987.

[14] Robert Hieb and R. Kent Dybvig. Continuations and Concurrency. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 128–136, March 1990.

[15] Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. Static analaysis of communication for asynchronous concurrent programming languages. In *Second International Static Analysis Symposium (SAS'95)*, number 983 in Lecture Notes in Computer Science, pages 225–242. Springer-Verlag, 1995.

[16] William A. Kornfeld and Carl E. Hewitt. The Scientific Community Metaphor. *IEEE Trans. on Systems, Man, and Cybernetics*, pages 24–33, January 1981.

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

[18] Luc Moreau. *Sound Evaluation of Parallel Functional Programs with First-Class Continuations*. PhD thesis, University of Liège, Service d'Informatique, Institut Montefiore B28, 4000 Liège, Belgium, June 1994. Also available by anonymous ftp from `ftp.montefiore.ulg.ac.be` in directory `pub/moreau`.

[19] Luc Moreau. Correctness of a Distributed-Memory Model for Scheme. In *Second International Europar Conference (EURO-PAR'96)*, number 1123 in Lecture Notes in Computer Science, pages 615–624, Lyon, France, August 1996. Springer-Verlag.

[20] Luc Moreau. The Semantics of Scheme with Future. In *In ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 146–156, Philadelphia, Pennsylvania, May 1996.

[21] Luc Moreau and Daniel Ribbens. The Semantics of pcall and fork. In R. Halstead, T. Ito, and C. Queinnec, editors, *PSLS 95 – Parallel Symbolic Langages and Systems*, number 1068 in Lecture Notes in Computer Science, pages 52–77, Beaune, France, October 1995. Springer-Verlag.

[22] Randy B. Osborne. Speculative Computation in Multilisp. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp : Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, number 441 in Lecture Notes in Computer Science, pages 103–137. Springer-Verlag, 1990.

[23] Randy B. Osborne. Speculative Computation in Multilisp. An Overview. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 198–208, Nice, France, June 1990.

[24] Patiwat Panurach. Money in electronic commerce. *Communications of the ACM*, 39(6):45–50, June 1996.

[25] Christian Queinnec. Locality, Causality and Continuations. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, June 1994.

[26] Christian Queinnec. Sharing mutable objects and controlling groups of tasks in a concurrent and distributed language. In Takayasu Ito and Akinori Yonezawa, editors, *Proceedings of the Workshop on Theory and Practice of Parallel Programming (TPPP'94)*, number 700 in Lecture Notes in Computer Science, pages 70–93, Sendai (Japan), November 1994. Springer-Verlag.

[27] Christian Queinnec. DMEROON: a Distributed Class-based Causally-coherent Data Model: Preliminary Report. In *Parallel Symbolic Languages and Systems.*, Beaune, France, October 1995.

[28] Christian Queinnec and David De Roure. Design of a Concurrent and Distributed Language. In A. Agarwal, R. H. Halstead, and Takayasu Ito, editors, *Parallel Symbolic Computing: Languages, Systems and Applications*, number 748 in Lecture Notes in Computer Science, pages 234–259, Boston, Massachussetts, October 1992. Springer-Verlag.

[29] Eric Raymond. *The New Hacker's Dictionary.* MIT Press, 1991.

[30] Jonathan Rees and William Clinger, editors. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991.

[31] Dorai Sitaram. *Models of Control and Their Implications for Programming Language Design.* PhD thesis, Rice University, Houston, Texas, April 1994.

[32] Guy Lewis Steele, Jr. *Common Lisp. The Language.* Digital Press, second edition, 1990.

[33] Philip Wadler. The Essence of Functional Programming (invited talk). In *Proceedings of the Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, New Mexico, January 1992.

[34] Mitchell Wand. Continuation-Based Multiprocessing. In *Conference Record of the 1980 Lisp Conference*, pages 19–28, 1980.

[35] Andrew W. Wright and Bruce F. Duba. Pattern Matching for Scheme. Technical report, Rice University, Hoston, TX 77251-1892, May 1995.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS
Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
(France)
http://www.inria.fr
ISSN 0249-6399