The following paper was originally published in the
Proceedings of the Conference on Domain-Specific Languages
Santa Barbara, California, October 1997

Design and Semantics of Quantum: a Language to Control Resource Consumption in Distributed Computing

Luc Moreau
University of Southampton
Christian Queinnec
Université de Paris

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Design and Semantics of Quantum: a Language to Control Resource Consumption in Distributed Computing.

Luc Moreau
Department of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, UK
L.Moreau@ecs.soton.ac.uk

Christian Queinnec
LIP 6 & INRIA-Rocquencourt
4 place Jussieu, 75252 Paris Cedex
Christian.Queinnec@inria.fr

Abstract

This paper describes the semantics of Quantum, a language that was specifically designed to control resource consumption of distributed computations, such as mobile agent style applications. In Quantum, computations can be driven by mastering their resource consumption. Resources can be understood as processors cycles, geographical expansion, bandwidth or duration of communications, etc. We adopt a generic view by saying that computations need energy to be performed. Quantum relies on three new primitives that deal with energy. The first primitive creates a tank of energy associated with a computation. Asynchronous notifications inform the user of energy exhaustion and computation termination. The other two primitives allow us to implement suspension and resumption of computations by emptying a tank and by supplying more energy to a tank. The semantics takes the form of an abstract machine with explicit parallelism and energy-related primitives.

1 Introduction

Millions of computers are now connected by the Internet. At a fast pace, applications are taking advantage of these new capabilities, and are becoming parallel and distributed. For instance, mobile agents [Mag96a] and multi-agent systems are technologies used in a wide range of activities, such as information discovery on the WWW [DD97] or negotiation on behalf of the user [WJ95]; they exploit parallelism to improve efficiency and autonomy, and they rely on distribution or mobility to increase locality.

A major challenge is to be able to control and monitor computations in such a distributed context. On the one hand, users are ready to delegate negotiation power and responsibility to their agents, but they wish to bound their activities by geographical, temporal, physical (such as memory, ...), or monetary constraints; furthermore, during execution, users might wish to dynamically monitor and control their agent's behaviour by reducing or adding constraints. On the other hand, service providers offer platforms where mobile agents can migrate to in order to use available facilities; they are anxious to ensure that visiting agents act according to a previously negotiated agreement, that they do not exceed temporal or physical limits, and that they are charged according to their usage of facilities.

Our goal is to provide the means by which everybody, users and service providers, can control and monitor resources used by parallel and distributed computations. We believe that parallel computations can be driven by mastering their *resource consumption*. Resources can be understood as processor cycles, bandwidth and duration of communications, or even printer paper. We adopt a more generic view by saying that computations need *energy* to be performed[1].

In this paper, we present a new language, called Quantum, that is specifically designed to monitor and control the resource consumption of computations. The essence of Quantum is summarised by three key ideas. *(i)* Quotas of energy can be associated with computations, and energy is being consumed during every evaluation step. *(ii)* Asynchronous notifications inform of energy exhaustion or computation termination. *(iii)* Mechanisms exist to transfer energy to or from computations; sup-

[1] Other names found in the literature for a similar concept are fuel [HF87], computron [Ray91, p. 102–103], teleclick [Mag96a] or metapill [A+97]

plying more energy to a computation gives the right to continue the computation, while removing energy from a computation acts as energy-based preemption. Even though 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.

$\mathcal{Q}$uantum generalises some approaches adopted in agent scripting languages to control resources [Mag96a, PS97]. Besides its resource-oriented foundations, $\mathcal{Q}$uantum is conceived for parallelism and distribution, while being independent of the actual primitives for parallelism and distribution, and of the memory model (central, distributed, with or without coherence).

This paper reports about our experience with designing $\mathcal{Q}$uantum and defining its semantics. Our requirement is to define the primitives that would allow us to control the energy consumption of distributed computations. We derived a solution from two different ideas: Haynes and Friedman's engines [HF87] can be extended to parallelism and distribution, while actor's sponsors [KH81] can be adapted to our energy view. For the sequential subset of the language, we adopt an applied call-by-value lambda-calculus [Plo75]. The semantics have been our driving force in designing $\mathcal{Q}$uantum. Our operational semantics takes the form of an abstract machine for parallel evaluation; it extends the CEK machine [FF86] with parallelism and the energy-related primitives. The choice of the semantic framework is a major help in defining simple primitives: the abstract machine hides some execution details and offers a suitable degree of atomicity, while it still offers a realistic model of parallelism.

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 examples written in $\mathcal{Q}$uantum. Finally, Section 5 discusses related work and is followed by a conclusion.

2 Intuition of $\mathcal{Q}$uantum

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 primitives of $\mathcal{Q}$uantum that are specific to energy are displayed in the first half of Figure 1. In order to be usable, they need to be glued with other primitives for parallelism and communication, which may vary according to the programmer's taste. A particular selection is presented in the second half of Figure 1; they are briefly described below and they will be used in the examples of Section 4. Afterwards, we describe the three key ideas of $\mathcal{Q}$uantum: *groups*, *asynchronous notifications*, and *energy transfers*.

Energy Specific Primitives

$$
\begin{array}{rcl}
\mathit{primitives} & ::= & \mathsf{call\text{-}with\text{-}group}(F, e, \varphi_e, \varphi_t) \\
 & & \mid \mathsf{pause}(g, \varphi_p) \\
 & & \mid \mathsf{awaken}(g, e) \\
g & \in & \mathit{Group} \\
e & \in & \mathit{Energy} \\
F & : & \mathit{Group} \times \mathit{Energy} \to \alpha \\
\varphi_e, \varphi_t, \varphi_p & : & \mathit{Group} \times \mathit{Energy} \to \mathit{void}
\end{array}
$$

Language Specific Primitives

$$
\begin{array}{rcl}
\mathit{primitives} & ::= & \mathsf{fork}(M) \mid \mathsf{suicide}() \\
 & & \mid \mathsf{channel}() \mid \mathsf{enqueue}(ch, V) \\
 & & \mid \mathsf{dequeue}(ch) \\
ch & \in & \mathit{Channel} \\
V & \in & \mathit{Value}
\end{array}
$$

Figure 1: Abstract Syntax of Primitives

We use the term *task* to denote an evaluation thread created by the construct for parallelism. $\mathcal{Q}$uantum is independent of the primitives for parallelism and distribution. Parallel threads of evaluation may be created using Posix threads [IEEE], or higher-level constructs such as pcall [MR95, QD92] or future [Hal90, FF95, Mor96b]. In this paper, we adopt the construct fork which creates a parallel evaluation thread for its argument. A computation also has the ability to terminate by evaluating the expression suicide().

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 [Mor96a], distributed causally coherent memory [Que94a] are memory models that may be adopted with $\mathcal{Q}$uantum. However, we need primitives to synchronise computations and to exchange information between them. We observed that asynchronous unbounded communication channels [KNY95] offered the appropriate level of abstraction. Figure 1 contains primitives to create channels, to add a value to a channel, and to re-

move a value from a channel.

2.1 Energy and Groups

Our goal is to be able to allocate resources to computations, and to monitor and to control their use as evaluation proceeds. 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 example more resources can be supplied.

In order to be notified of 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 a computation in a Quantum program. So, a group is associated with a computation that may be composed of several tasks 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* [KH81, Osb90b, Hal90] the computation it is associated with. Reciprocally, every computation has a sponsoring group, and so does every task.

At creation time, a group is given an *energy quota*. More specifically, a computation that evaluates the expression call-with-group$(F, e, \varphi_e, \varphi_t)$ 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 and e; hence, the user function F receives a handle on its sponsoring group. As Quantum keeps track of resource consumption, the cost of g_2 creation and the energy e allocated to g_2 are deducted from the energy of g_1. The value of the call-with-group primitive is the value returned by the application of F.

2.2 Asynchronous Notifications

The semantics enforces the following principle: every 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 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 task; 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.

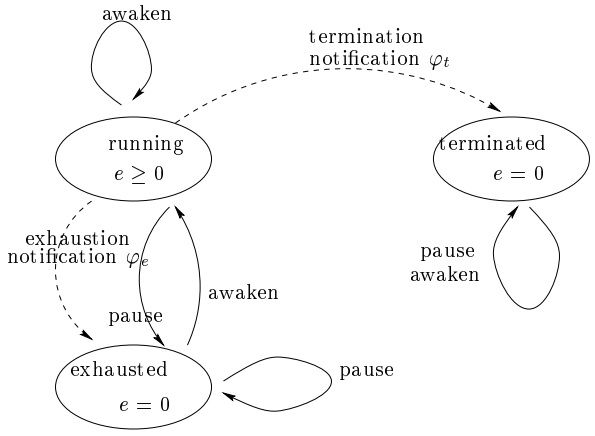

Figure 2: State Transitions

Figure 2 displays the state transition diagram for groups. At creation time, a group is in the running state, which means that the tasks that it sponsors can proceed as long as they do not require more energy than available. Asynchronous notifications are represented by dotted lines. 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 tasks 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). Here we should point out that the primitive fork can only create a thread in the group that sponsors its evaluation.

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

2.3 Energy Transfers

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 awaken. 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 awaken(g, e) supplies a group g with some energy e, which is deducted from the group sponsoring the awaken action. In addition, if the group was in the exhausted state, it is changed to the running state; if the group is in a terminated state, awaken acts as a null operation. Let us observe the non-symmetric behaviours of pause and awaken: the former operates recursively on a group hierarchy, while the latter acts on a group and not its descendants. However, we might wish to awaken 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 *memory-less*. 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 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', a task 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 awaken the group it is applied on, and step by step, energy may be redistributed among the hierarchy.

3 The Language Quantum: Semantics

In this section, we present the semantics of Quantum using an abstract machine, called the $\mathcal{Q}$-machine. Figure 3 displays its state space. We can see that the primitives of Figure 1 appear in the set of terms Λ_Q. The core of Λ_Q is an applied call-by-value lambda calculus composed of abstractions, variables, applications, and constants [Plo75]. Let us note that fork and suicide only are essential syntax, whereas the other primitives appear as constants in the set $FConst$. Transition rules appear in Figures 4 to 10.

In the sequel, we adopt Barendregt's [Bar84] 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 $f'(y) = f(y),\ \forall y \neq x$.

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 Quantum, 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 , associates each group g with a parent group, its current energy, its state, the two notifiers 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 $Group \to Energy \to Void$, 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 [FF86] 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 [FF86].

As previously mentioned, the purpose of Quantum is to measure the resources used by computations. In order to be generic, we decided to associate the semantics with two *cost functions* $\mathcal{K}, \mathcal{K}_n$, giving each transition its cost in terms of energy.

> **Warning**. The cost of a transition is a function of the task involved in the transition and of the function , . For the sake of concision, we do not represent this dependency explicitly. We use the symbol

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

$$
\begin{array}{lcll}
\mathcal{M} \in Qconfig & ::= & \langle T, , , \sigma \rangle & \text{(Q-Configuration)} \\
, \in GMap & : & Group \to GInfo & \text{(Group Mapping)} \\
i \in GInfo & ::= & \langle g, e, s, \varphi_e, \varphi_t, n, g^* \rangle & \text{(Group Information)} \\
g \in Group & = & \{\bot_g\} \cup \{g_0, g_1, \ldots\} & \text{(Group)} \\
t \in Task & ::= & \langle C, g \rangle & \text{(Task)} \\
C \in CoSt & ::= & \mathsf{Ev}\langle M, \rho, \kappa \rangle \mid \mathsf{Ret}\langle V, \kappa \rangle & \text{(Computational State)} \\
\kappa \in CCode & ::= & (\mathbf{init}) \mid (\kappa\ \mathbf{fun}\ V) \mid (\kappa\ \mathbf{arg}\ M\ \rho) \mid (\kappa\ \mathbf{rgroup}) & \text{(Continuation code)} \\
s \in GState & ::= & \mathsf{running} \mid \mathsf{exhausted} \mid \mathsf{terminated} & \text{(Group State)} \\
T & \subseteq & Task & \text{(Set of Tasks)} \\
q \in Queue & ::= & \langle\rangle \mid \langle V \rangle \mid q \S q & \text{(Queue)} \\
\sigma \in QStore & : & Loc \to Queue & \text{(Queue Store)} \\
\alpha \in Loc & = & \{\alpha_0, \alpha_1, \ldots\} & \text{(Location)} \\
\rho \in Env & : & Vars \to Value & \text{(Env)} \\
n \in \mathbb{N} & & & \text{(Number of sponsored tasks)} \\
e \in Energy & \subseteq & \mathbb{N} & \text{(Energy)} \\
\mathcal{K}, \mathcal{K}_n & : & Task \times GMap \to Energy & \text{(Cost Function)}
\end{array}
$$

Figure 3: State Space

$\mathcal{K}$ to denote the value of the cost function for the task involved in the transition and a given , . For instance, in rule (*sequential*) of Figure 6, the task involved in the transition is $\langle C, g \rangle$; therefore, the symbol $\mathcal{K}$ stands for $\mathcal{K}(\langle C, g \rangle, ,)$.

We also use the symbol $\mathcal{K}_n$ to denote the cost of a notification, i.e. the cost of rule (*termination*) or (*exhaustion*).

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 sum of the transition cost and the notification cost, which is represented by the side-condition noted ($\star$). This side-condition guarantees that after transition, we still have enough energy to post a notification if required.

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

As many cost models are conceivable, we decided to parameterise the semantics by the cost model. Figure 5 gives the definition of functions $\mathcal{K}, \mathcal{K}_n$, charging a unitary cost for every transition, in addition to

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

Figure 4: CEK Transitions

Unitary Cost Function

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

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

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

$\mathcal{K}_n = 1$ (notification cost)

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

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

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

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

$\mathcal{K}_n \geq 1$

Figure 5: Cost Functions $\mathcal{K}, \mathcal{K}_n$

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, every computation step has a cost; second, transferring some energy costs this amount of energy at least.

In Figure 6, the rule for the construct (fork M) creates a new task to evaluate M with the same environment ρ and an initial continuation, 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 and e. Following the soundness constraints of Figure 5, the sponsoring group g is

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

$\quad \to \quad \langle \{\ \langle C_1, g\rangle\ \} \cup T, \Gamma[g.e := g.e - \mathcal{K}], \sigma\rangle \quad \text{if } C \to_{cek} C_1 \quad (\star)$ *(sequential)*

$\langle \{\ \langle \mathsf{Ev}\langle(\mathsf{fork}\ M, \rho, \kappa\rangle, g\rangle\ \} \cup T, \Gamma, \sigma\rangle$

$\quad \to \quad \langle \{\ \langle \mathsf{Ret}\langle \mathsf{void}, \kappa\rangle, g\rangle, \quad \langle \mathsf{Ev}\langle M, \rho, (\mathbf{init})\rangle, g\rangle\ \} \cup T, \Gamma[g.e := g.e - \mathcal{K}][g.n := g.n + 1], \sigma\rangle \quad (\star)$ *(parallel)*

$\langle \{\ \langle \mathsf{Ev}\langle(\mathsf{suicide}), \rho, \kappa\rangle, g\rangle\ \} \cup T, \Gamma, \sigma\rangle$

$\quad \to \quad \langle T, \Gamma[g.e := g.e - \mathcal{K}][g.n := g.n - 1], \sigma\rangle \quad (\star)$ *(suicide)*

$\langle \{\ \langle \mathsf{Ret}\langle \mathsf{void}, (\mathbf{init})\rangle, g\rangle\ \} \cup T, \Gamma, \sigma\rangle$

$\quad \to \quad \langle T, \Gamma[g.e := g.e - \mathcal{K}][g.n := g.n - 1], \sigma\rangle \quad (\star)$ *(init)*

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

Figure 6: Sequential and Parallel Evaluations

$\langle \{\ \langle \mathsf{Ret}\langle(\mathsf{channel}), \kappa\rangle, g\rangle\ \} \cup T, \Gamma, \sigma\rangle$

$\quad \to \quad \langle \{\ \langle \mathsf{Ret}\langle\langle \mathsf{ch}\ \alpha\rangle, \kappa\rangle, g\rangle\ \} \cup T, \Gamma[g.e := g.e - \mathcal{K}], \sigma[\alpha \to \langle\rangle]\rangle \ \text{ with } \ \alpha \notin DOM(\sigma) \ \ (\star)$ *(channel)*

$\qquad \langle \{\ \langle \mathsf{Ret}\langle V, (\kappa\ \mathbf{fun}\ (\mathsf{enqueue}\ \langle \mathsf{ch}\ \alpha\rangle))\rangle, g\rangle\ \} \cup T, \Gamma, \sigma\rangle$

$\quad \to \quad \langle \{\ \langle \mathsf{Ret}\langle \mathsf{void}, \kappa\rangle, g\rangle\ \} \cup T, \Gamma[g.e := g.e - \mathcal{K}], \sigma[\alpha := \sigma(\alpha)\ \S\ \langle V\rangle]\rangle \quad (\star)$ *(enqueue)*

$\langle \{\ \langle \mathsf{Ret}\langle\langle \mathsf{ch}\ \alpha\rangle, (\kappa\ \mathbf{fun}\ \mathsf{dequeue})\rangle, g\rangle\ \} \cup T, \Gamma, \sigma\rangle$

$\quad \to \quad \langle \{\ \langle \mathsf{Ret}\langle V, \kappa\rangle, g\rangle\ \} \cup T, \Gamma[g.e := g.e - \mathcal{K}], \sigma[\alpha := q]\rangle \quad \text{if } \sigma(\alpha) = \langle V\rangle\ \S\ q \ \ (\star)$ *(dequeue)*

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

Figure 7: Channels Related Operations

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

$\quad \to \quad \langle \{\ \langle \mathsf{Ret}\langle g_1, (((\kappa\ \mathbf{rgroup})\ \mathbf{arg}\ e, \emptyset)\ \mathbf{fun}\ F)\rangle, g_1\rangle\ \} \cup T, \Gamma_1, \sigma\rangle \quad (\star)$ *(make group)*

$\qquad$ with $\Gamma_1 = \Gamma[g.e := g.e - \mathcal{K}][g.n := g.n - 1][g.g^* := g.g^* \cup \{g_1\}][g_1 \to \langle g, e, \mathsf{running}, \varphi_e, \varphi_t, 1, \emptyset\rangle]$,

$\qquad g_1 \notin DOM(\Gamma)$

$\langle \{\ \langle \mathsf{Ret}\langle V, (\kappa\ \mathbf{rgroup})\rangle g,\ \rangle\}\ \cup T, \Gamma, \sigma\rangle$

$\quad \to \quad \langle \{\ \langle \mathsf{Ret}\langle V, \kappa\rangle, g_1\rangle\ \} \cup T, \Gamma[g.e := g.e - \mathcal{K}][g.n := g.n - 1][g_1.n := g_1.n + 1], \sigma\rangle$ *(return group)*

$\qquad$ with $g_1 = \Gamma(g).p \quad (\star)$

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

Figure 8: Groups Related Operations

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

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

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

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

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

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

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

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

Figure 9: Asynchronous Notifications

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 10: Pause and Awaken Operations on Groups

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.

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.

Rules controlling asynchronous notifications appear in Figure 9. If the energy available in a group g is smaller that the sum of the energy required by a task and the energy for a notification, 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. The notification is executed under the sponsorship of the parent group g_1, and 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 $g_1 \neq \bot_g$.

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. Like any other transition, rules (*exhaution*) and (*termination*) are given a cost; for the sake of simplicity, it is defined as $\mathcal{K}_n$. The side-condition ($\star$) used in all rules but (*exhaution*) and (*termination*) ensures that enough energy is left in a group to post a notification.

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 awaken 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 awaken takes the group to be awakened and the energy to be transferred in arguments. Assuming the sponsoring group g has enough energy, rule (*awaken 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.
Definition 1 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.

Definition 1 (Evaluation Relation)
$eval_{\mathcal{K},\mathcal{K}_n}(M,e) = V$ if $\exists \langle T,,,\sigma\rangle$, such that $\langle\{\langle \mathsf{Ev}\langle M,\emptyset,\kappa_0\rangle, g_0\rangle\rangle\},,_0,\sigma_0\rangle \rightarrow^* \langle T,,,\sigma\rangle$, with $V \in \sigma(\alpha_0)$ and $Final(\langle T,,,\sigma\rangle)$, and with:

$$
\begin{array}{rcl}
,_0 & = & \{g_0 \rightarrow \langle \bot_g, e, \varphi_{e0}, \varphi_{t0}, 1, \langle\rangle\rangle\} \\
\sigma_0 & = & \{\alpha_0 \rightarrow \langle\rangle\} \\
\kappa_0 & = & ((\textbf{init})\textbf{fun}\ (\mathsf{enqueue}\ \langle \mathsf{ch}\ \alpha_0\rangle)) \\
Final(\langle T,,,\sigma\rangle) & \equiv & \nexists \langle T',,',\sigma'\rangle, \\
 & & \quad \langle T,,,\sigma\rangle \rightarrow \langle T',,',\sigma'\rangle \\
\varphi_{e0} = \varphi_{t0} & = & \langle \mathsf{cl}\ \lambda g.\lambda e.\mathsf{void}, \emptyset\rangle
\end{array}
$$

□

Let us note that the evaluation relation is parameterised by the cost functions $\mathcal{K}, \mathcal{K}_n$ 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 2 For any cost function satisfying the constraints of Figure 5, total energy decreases as evaluation proceeds. □

Corollary 3 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 adopt Scheme syntax [RC91] and present some examples using our primitives.

4.1 Energy Critical Section

Even though $\mathcal{Q}$uantum substantially differs from Scheme, it is expressive enough to model first-class mutable boxes. Figure 11 displays the code for mutable boxes in $\mathcal{Q}$uantum, 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 11: 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 12).
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

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

(define (refill-handler g e)
  (awaken g (+ e 1)))
(define (ignore-handler g e)
  (suicide))
```

Figure 12: Energy-critical Section

by the user's program. It is his role to ensure that a paused group does not leave objects such as boxes in an inconsistent state.

4.2 Monitoring Computations

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 13 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*.

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

Figure 13: Probe

4.3 Controlling Computations

Quantum offers the possibility to control computations in a refined way. It is possible to stop a (distributed) computation, to suspend and resume it later, or to adjust it level energy, which introduces a form of energy-based priority. All three operations are implemented using *pause* and *awaken*.

The function *suspend* temporarily pauses a group hierarchy. The hierarchy will be resumed by awakening its root, which will awaken its subgroups step by step using the notifiers left by *pause*. Note that the condition in the notifier prevents to awaken 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 awakens 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 hierarchy before adjustment, while the energy of the hierarchy after adjustment is deducted from the parent of the root. In order to guarantee that *awaken* is executed after *pause*, we introduce an explicit synchronisation by the channel *go*.

```
(define (suspend group)
  (pause (lambda (g e)
           (if (not (eq? g group))
               (awaken 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))
             (awaken g (* e coefficient)))
           (list group))
    (enqueue go #t)))
```

Figure 14: Pause and Awaken of Computations

4.4 A Service Provider

Figure 15 displays the code of a service provider, making use of Wright and Duba's pattern-matching macro [WD95]. 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 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 Discussion and Related Work

For the sake of clarity, we have presented a simple cost model that measures the cost of computing. In practice, we should regard energy as a multidimensional (vector) datastructure of budgetised resources. Such an extended model can take into account time, geographical expansion, file access permission, memory size, number of messages, number of parallel tasks, ... In such an extended model, once a resource is exhausted (or a computation ends), there is an asynchronous notification; the primitives awaken and pause can supply or remove a given resource.
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 [HF84, HF87] introduce the *engine* facility to model timed preemption; variants can also be found in [Dyb87, Eis88, Sit94]. 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 such as pause and awaken. 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 "multi-shot", while groups are "single-shot" [BWD96]. 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 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 [KH81], Osborne's enhanced version of them [Osb90a, Osb90b, Hal90], and subsequently Queinnec's groups [Que94b, QD92], also allow the programmer to control hierarchies of computations in a parallel setting. Osborne's sponsors are entities that give attributes, such as 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 resources are more intuitive. 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 generali-

```
(define subscription-channel (channel))
(define (service-provider subscription-channel)
  (let loop ()
    (let ((subscription (dequeue subscription-channel)))
      (fork (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 e)
                        (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)))
      (awaken group (ecash->energy ecash))))

   (('free-demo name answer)
    (call-with-group (lambda (g e)
                       (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)))

(define (request-server channel sponsoring-group)
 (let ((message (dequeue channel)))
  (fork (request-server channel sponsoring-group))
  (match message
   (('submit job answer)
    (call-with-group (lambda (g e)
                       (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
          (awaken group 1)
          (enqueue answer `(restarted ,group)))
        (enqueue answer `(unknown ,group))))
   (else 'discard))
  (suicide)))
```

Figure 15: Service Provider (1)

sation of `unwind-protect` [Ste90]. Hieb and Dybvig [HD90] `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.

The mobile agent community deals with the problem of controlling distributed mobile computations, called *agents*. The most widespread agent systems are Telescript [Mag96a, Mag96b], Tacoma [JvRS95a, JvRS95b], Agent-TCL [Gra96], and Ara agents [Pei97, PS97]. Most of them have a notion of energy: Telescript agents have "permits" in terms of teleclicks [Mag96a, Mag96b], Ara agents [Pei97, PS97] are equipped with resource accounts called *allowances*, Erlang agents [A$^+$97] rely on metapills. However, very few of them are able to control resources in a similar way as Quantum.

Quoting [Mag96b], "if the agent exceeds any of its quantitative limits, the engine destroys the agent unceremoniously. No grace period is extended". Our model is more general than the Telescript approach as it allows us to drive computations using pause and awaken and to monitor them using asynchronous notifications. Also, our model supports agents that perform parallel and distributed computations, what is usually called multi-agent systems. Arthursson *et al.* [A$^+$97] follow an approach similar to Telescript.

Allowances in Ara agents [PS97, Pei97] are similar to our groups [PS97, p. 6]; as indicated by Peine, they are a recent concept, not complete yet. Several agents can share the same allowance, also called a group. Agents can transfer resources explicitly between accounts. In Ara, agents can be suspended or reactivated: these actions however are performed on agents directly and not on groups [PS97, p. 23]; as a result hierarchical computation cannot be controlled as in Quantum. Ara agents are not notified of an exhaustion [PS97, p. 35]; Peine consider that this is not a severe problem as an agent may enquire about

the existence of a resource. We believe that this argument is not valid in parallel/distributed computing because another parallel computation might consume the resource.
Our semantic is sound because it prevents generating energy. Furthermore, our language provides the means to enforce security in different ways: *(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 a handle to their sponsoring group. Security is an important issue in distributed agent-style applications. Using Quantum primitives, there is nothing that prevents users from erroneously transferring resources between groups, or making a group accessible to and then preemptable by another task. However, we believe that some static analysis [VS97] would be able to detect whether a group might become preempted if made accessible in a public data structure for instance.

6 Conclusion

In this paper, we present the language Quantum, whose purpose is to monitor and control resource consumption in a parallel and distributed framework. The semantics uses an abstract notion of energy, but it can be applied to control computation time, memory usage, message sending, file access permission, etc. A companion paper describes a distributed implementation of Quantum built on top of a message-passing library [MQ97]. A refinement of the semantics introduces explicit localities in the spirit of [Ama97, FGLMR96, Mor96a].
Quantum is a language that is aimed at agent application implementers, who are in need of controlling and bounding the resources needed by their agents. It is also useful to implementers of services who need to monitor visiting mobile agents. Also, it provides primitives to build "any-resource" algorithms, applying the idea "any-time" algorithms [DB88] to any form of resource.
Quantum 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.

7 Acknowledgement

Luc Moreau was partially supported by EPSRC GR/K30773 and EC project reference ERB 4050 PL 930186; Christian Queinnec was partially supported by GDR-PRC de Programmation du CNRS and EC project reference ERB 4050 PL 930186. Thanks to Jonathan Dale and the anonymous referees for their useful comments. Andreas Kind and Jean-Pierre Briot mentioned a link between Quantum and anytime functions.

Bibliography

[A+97] J. Arthursson et al. A Platform for Secure Mobile Agents. In *The Second International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 109–120, April 1997.

[Ama97] R. M. Amadio. An Asynchronous Model of Locality, Failure, and Process Mobility. In *COORDINATION 97, LNCS*, 1997.

[Bar84] H. 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.

[BWD96] C. Bruggeman, O. Waddell, and R. K. 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.

[DB88] T. L. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, Minneapolis, Minnesota, 1988.

[DD97] J. Dale and D. DeRoure. Towards a Framework for Developing Mobile Agents for Managing Distributed Information Resources. Technical Report M97/1, University of Southampton, February 1997.

[Dyb87] R. K. Dybvig. *The Scheme Programming Language*. Prentice-Hall, 1987.

[Eis88] M. Eisenberg. *Programming in Scheme*. The Scientific Press, 507 Seaport Court. Redwood City. CA 94063-2731, 1988.

[FF86] M. Felleisen and D. 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).

[FF95] C. Flanagan and M. 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.

[FGLMR96] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings of CONCUR'96*, LNCS 1119, pages 406–421, Pisa, Italy, 1996. Springer-Verlag.

[Gra96] R. S. Gray. Agent Tcl: A Flexible and Secure Mobile-Agent System. In *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96)*, Monterey, California, July 1996. http://www.cs.dartmouth.edu/~agent/papers/index.html.

[Hal90] R. H. Halstead, Jr. New Ideas in Parallel Lisp : Language Design, Implementation. In *Parallel Lisp : Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, LNCS 441, pages 2–57. Springer-Verlag, 1990.

[HD90] R. Hieb and R. K. Dybvig. Continuations and Concurrency. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 128–136, March 1990.

[HF84] C. T. Haynes and D. P. Friedman. Engines Build Process Abstractions. In *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming*, pages 18–24. ACM, 1984.

[HF87] C. T. Haynes and D. P. Friedman. Abstracting Timed Preemption with Engines. *Comput. Lang.*, 12(2):109–121, 1987.

[IEEE] IEEE. *IEEE P1003.1c/D10 Draft Standard for Information Technology – Portable Operating Systems Interface (POSIX)*, September 1994.

[JvRS95a] D. Johansen, R. van Renesse, and F. B. Schneider. An Introduction to the TACOMA Distributed System Ver 1. Technical Report 95–23, Department of Computer Science, University of Troms, Norway, 1995. http://www.cs.uit.no/DOS/Tacoma/Publications.html.

[JvRS95b] D. Johansen, R. van Renesse, and F. B. Schneider. Operating System Support for Mobile Agents. In *5th IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, Wa, USA, 1995. Also available as Technical Report TR94-1468, Department of Computer Science, Cornell University. http://www.cs.uit.no/DOS/Tacoma/Publications.html.

[KH81] W. A. Kornfeld and C. E. Hewitt. The Scientific Community Metaphor. *IEEE Trans. on Systems, Man, and Cybernetics*, pages 24–33, January 1981.

[KNY95] N. Kobayashi, M. Nakade, and A. Yonezawa. Static analysis of communication for asynchronous concurrent programming languages. In *Second International Static Analysis Symposium (SAS'95)*, LNCS 983, pages 225–242. Springer-Verlag, 1995.

[Mag96a] General Magic. Telescript Technology: Mobile Agents. http://www.genmagic.com/Telescript/Whitepapers/wp4/white paper-4.html, 1996.

[Mag96b] General Magic. Telescript technology: The foundation for the electronic marketplace. http://www.genmagic.com/Telescript/Whitepapers/wp1/white paper-1.html, 1996.

[Mor96a] L. Moreau. Correctness of a Distributed-Memory Model for Scheme. In *Second International Europar Conference (EURO-PAR'96)*, LNCS 1123,

pages 615–624, Lyon, France, August 1996. Springer-Verlag.

[Mor96b] L. 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.

[MQ97] L. Moreau and C. Queinnec. Distributed computations driven by resource consumption. Technical report, University of Southampton, 1997.

[MR95] L. Moreau and D. Ribbens. The Semantics of pcall and fork. In *PSLS 95 – Parallel Symbolic Langages and Systems*, LNCS 1068, pages 52–77, Beaune, France, October 1995. Springer-Verlag.

[Osb90a] R. B. Osborne. Speculative Computation in Multilisp. In *Parallel Lisp : Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, LNCS 441, pages 103–137. Springer-Verlag, 1990.

[Osb90b] R. 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.

[Pei97] H. Peine. An Introduction to Mobile Agent Programming and the Ara System. Technical Report ZRI report 1/9, Dept. of Computer Science, University of Kaiserslautern, Germany, 1997. http:// www.uni-kl.de/AG-Nehmer/Ara/.

[Plo75] G. D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, pages 125–159, 1975.

[PS97] H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In *Proc. of the First International Workshop on Mobile Agents MA'97*, LNCS 1219, Berlin, Germany, April 1997. Springer-Verlag. http:// www.uni-kl.de/AG-Nehmer/Ara/.

[QD92] C. Queinnec and D. DeRoure. Design of a Concurrent and Distributed Language. In *Parallel Symbolic Computing: Languages, Systems and Applications*, LNCS 748, pages 234–259, Boston, Massachussetts, October 1992. Springer-Verlag.

[Que94a] C. Queinnec. Locality, Causality and Continuations. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, June 1994.

[Que94b] C. Queinnec. Sharing mutable objects and controlling groups of tasks in a concurrent and distributed language. In *Proceedings of the Workshop on Theory and Practice of Parallel Programming (TPPP'94)*, LNCS 700, pages 70–93, Sendai (Japan), November 1994. Springer-Verlag.

[Ray91] E. Raymond. *The New Hacker's Dictionary*. MIT Press, 1991.

[RC91] J. Rees and W. Clinger. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991.

[Sit94] D. Sitaram. *Models of Control and Their Implications for Programming Language Design*. PhD thesis, Rice University, Houston, Texas, April 1994.

[Ste90] G. L. Steele, Jr. *Common Lisp. The Language*. Digital Press, second edition, 1990.

[VS97] D. Volpano and G. Smith. A type-based approach to program security. In *Theory and Practice of Software Development (TAPSOFT'97)*, LNCS 1214, pages 607–621, Lille, France, April 1997. Springer-Verlag.

[WD95] A. W. Wright and B. F. Duba. Pattern Matching for Scheme. Technical report, Rice University, Hoston, TX 77251-1892, May 1995.

[WJ95] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), June 1995.