Cooperative Threads and Preemptive Computations

Frédéric Boussinot, Frederic Dabrowski

Cooperative Threads and Preemptive Computations

Frédéric Dabrowski* and Frédéric Boussinot**

INRIA Sophia-Antipolis, B.P. 93, 06902 Sophia-Antipolis Cedex, France

Abstract. A two-level model for reactive systems programming is introduced in which threads linked to the same scheduler are run cooperatively and have the possibility to escape from the scheduler control to run preemptively. We present a type and effect system to enforce a logical separation of the memory which ensures that, when running in preemptive mode, threads do not interfere with those running in cooperative mode. Thus, the atomicity property at the basis of the cooperative model is preserved.

1 Introduction

Intuitively, a reactive program (RP) reacts to activations coming from the environment. In general, a RP is not supposed to globally terminate. However, a RP which, in response to an activation, does not terminate its reaction is erroneous as it becomes unable to process the next activations (it cannot anymore be called reactive in this case). From this point of view, *a RP is a non-terminating program, all reactions of which terminate.*

To be effective, reaction time should not increase forever: a system with a reaction of 1 second in response to the first activation, of 2 seconds in response to the second activation, and so on, is clearly unacceptable from a reactivity point of view.

Among reactive programs are those for embedded and/or safety critical systems. A crucial question concerning these programs is: how to ensure their reactivity? The work presented here originated from a line of research on synchronous languages [12], in the version of reactive programming described in [1]; it is part of a more global work [3], the purpose of which is to provide static analysis tools to guarantee the reactivity of programs in the context of concurrency.

The decomposition in several concurrent activities often simplifies the task of programming complex systems: concurrency is an important tool for modularity. From the modulartity point of view, it is natural to demand that the execution of a sequential code fragment remains atomic from a logical point of view, when put in a concurrent context; we call this property *preservation of the semantics of sequentiality.*

* with support from ACI CRISS

** with support from ACI ALIDECS

In this paper, we consider a thread-based model for reactive programming. This model combines the cooperative and preemptive approaches, and is equipped with a type system which tests for the preservation of the semantics of sequentiality. We do not consider here the issue of termination of reactions, nor the bounds for computations during reactions, which are presently under investigation (they are discussed in the conclusion).

Having given the context and the motivation of our work, let us now describe in more detail the computing model considered.

We consider a two stage model: Several schedulers are run preemptively and are executing threads *linked* to them. Threads linked to the same scheduler are executed cooperatively and communicate using a common memory. Threads can dynamically go from one scheduler to another one (this can be seen as an elementary form of *thread migration*).

We introduce the possibility for a thread to temporarily unlink from the scheduler to which it is linked, in order to perform an asynchronous autonomous computation. This can be seen as a special case of migration to a dedicated scheduler. After having performed the asynchronous task, the unlinked thread re-links to the initial scheduler and returns to the cooperative mode of execution.

Let us justify the two-level structure of the model. First, we advocate cooperative concurrency, because of its clear and precise semantics. Moreover, cooperativity achieves the preservation of the semantics of sequentiality, which is part of our goal. The traditional criticism made to cooperative threads is that absence of cooperation from one thread blocks the whole system. Note that this criticism does not hold in our context, as we mandatorily ask for termination of reactions which entails cooperativity. Second, we know that cooperative concurrency is not appropriate to deal with some constructs (for example, input/output) which are blocking by nature. We also know that some contexts need some kind of preemptive behaviors, for example when some hardware signals (e.g. interrupts) have to be immediately processed. For these reasons, a preemptive level is introduced in the computing model.

In the paper, we statically check that in programs the semantics of the sequential composition is preserved despite the preemptive level. More precisely, we consider a simplified model with a unique scheduler in which threads are allowed to unlink. We claim that the main issues concerning the combination of preemption and cooperation are still captured in this simplified model. We basically show that one can partition the memory in regions in a way that forbids an unlinked thread to interfere with the other threads.

We introduce a *type and effect* system which enforces a separation of the memory into: (1) a public area which may be used by all threads, but only when linked to the scheduler and (2) private areas, one for each thread; the private area of a thread is only accessible by it, either when linked or when unlinked. We rely on *regions* to distinguish among sharable and unsharable memory locations and we introduce constraints on the creation of memory locations which ensure the separation of the distinct private areas. We rely on effects to ensure that an unlinked thread only has access to unsharable memory locations.

Section 2 introduces the language for programming threads. In section 3, we present the type and effect system which ensures the desired separation property of the memory. In section 4, we prove the correction of the system. Finally, after considering related work in section 5, we conclude in section 6 and sketch future developments.

2 Language

We introduce a small language for programming cooperative threads, providing recursive functions based on pattern-matching, dynamic thread creation and computations in preemptive mode. Intuitively, a cooperative thread in preemptive mode escapes from the cooperative scheduling and acts as a preemptive thread. Its evaluation interleaves with the evaluation of other threads.

2.1 Syntax

We assume two countable sets of memory location names: $\mathcal{L}_s$ (sharable memory locations) and $\mathcal{L}_u$ (unsharable memory locations). We note $\mathcal{L} = \mathcal{L}_s \cup \mathcal{L}_u$.

Definition 1. *A program P is a triple $(\mathcal{C}, \mathcal{F}, defs)$ where $\mathcal{C}$ (the constructors) and $\mathcal{F}$ (the function symbols) are disjoint finite sets of symbols and $defs$ is a finite set of equations. An equation has the shape $f(p_1, \ldots, p_n) = e$ where $f \in \mathcal{F}$, $p_1, \ldots, p_n$ are patterns and e is a static expression (i.e, not built with memory location names) such that $fv(e) \subseteq var(p_1, \ldots, p_n)$. For $c \in \mathcal{C}$, $f \in \mathcal{F}$ $\rho \in \{s, u\}$ and $l \in \mathcal{L}$, patterns, values and expressions are defined by the following grammars:*

$$
\begin{array}{lll}
p ::= x \mid c(p, \ldots, p) & & (patterns) \\
v ::= c(v, \ldots, v) \mid l & & (values) \\
e ::= x \mid c(e, \ldots, e) \mid f(e, \ldots, e) & & \\
\quad \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{ref}_\rho(e) \mid\ \texttt{get}(e) \mid \texttt{set}(e, e) & & \\
\quad \mid \texttt{thread}\{f(e, \ldots, e)\} \mid \texttt{unlink}\{e\} \mid l & & (expressions)
\end{array}
$$

Here, $fv(e)$ (resp. $var(p_1, \ldots, p_n)$) is the set of free variables (resp. variables) occurring in e (resp. $p_1, \ldots, p_n$). We always assume that for all function symbol f, patterns do not overlap, i.e at most one equation can be used for a reduction. We also assume a symbol $main \in \mathcal{F}$ of arity $n \geq 0$ and a symbol $()$ of arity 0 $\in \mathcal{C}$. From now on, for the sake of simplicity, we will identify main and P.

Intuitively, () is the single value of a type *unit* denoting side effects and *main* is the first function evaluated when the program P runs. The basic policy for the scheduling of threads is cooperative: one of the threads is elected and reduced until it returns the control to the scheduler. This happens either because it terminates, or because it explicitly requires to escape the cooperative scheduling policy by evaluating $\texttt{unlink}\{e\}$. A preemptive scheduling policy is used for the evaluation of e. Each global reduction consists then in a non-deterministic choice

between a reduction of the elected cooperative thread or a reduction of one of the escaped threads. By evaluating $\texttt{thread}\{f(e_1,\ldots,e_n)\}$, a new thread is created which evaluates $f(e_1,\ldots,e_n)$. The evaluation of $\texttt{ref}_\rho(v)$ creates a new memory location with a name $l \in \mathcal{L}_\rho$ and the initial value v. Here $\rho \in \{s,u\}$ should simply be seen as a typing annotation. The intuitive meaning of s (resp. u) is that the newly created memory location should be sharable (resp. unsharable). The evaluation of $\texttt{get}(l)$ gives the value currently held at l. The evaluation of $\texttt{set}(l,v)$ updates the content of l with v. As usual, the evaluation $\texttt{let}\ x = e\ \texttt{in}\ e'$ is the evaluation of e' where x is replaced by the value denoted by e.

2.2 Dynamic Semantics

We formalize the semantics of the language introduced above. We assume a countable set $\mathcal{T}$ of thread names.

Definition 2. *An assignment is a mapping $O : \mathcal{L}_u \to \mathcal{T}$ such that for all $t \in \mathcal{T}$, the set $O^{-1}(t) = \{l \mid O(l) = t\}$ is infinite and $O^{-1}(t) \cap O^{-1}(t') = \emptyset$ as soon as $t \neq t'$ [1]. A thread (of a program P) is a couple (e,t) where e is a closed expression (without free variables) built with constructors and function symbols of P and $t \in \mathcal{T}$; we will use the notation e^t instead of (e,t).*

We define *actions* which, for each read or write access, capture the name and the mode (*co* for cooperative and *pr* for preemptive) of the thread performing it.

Definition 3. *An action has the shape $access(t,\alpha,l)$ where $t \in \mathcal{T}$, $\alpha \in \{co,pr\}$ and $l \in \mathcal{L}$. A store is a partial function from memory locations to values. We note $S[l := v]$ the store such that $S[l := v](l') = S(l')$ if $l' \neq l$ and $S[l := v](l) = v$. We note $Dom(S)$ the domain of S. We note $S \cup \{l \mapsto v\}$ for the store which maps l to v and which is equal to S on the domain of S.*

A (read or write) access performed by a thread t on a memory location l produces an action $access(t,co,l)$ in cooperative mode and an action $access(t,pr,l)$ in preemptive mode.

In order to define the election of the thread to be executed by the scheduler, we introduce the notion of *marking* of a set of threads; intuitively, a marking of T is T in which one of the threads has been choosen.

Definition 4. *A marking of a set of threads T is either the empty marking ϵ if $T = \emptyset$, or a pair (T,t) where $e^t \in T$. We define a choice operator $\uparrow$ which, given a set of threads, non deterministically chooses one of its markings.*

If (T,t) is a marking of T, we use the notation $e^t \cdot T'$, where $T' = T \backslash e^t$, instead of (T,t) to make the marked thread explicit. Given a set of threads T, we note $|T|$ a possible marking of T. We note $|T|_0, |T|_1, \ldots$ when several markings of T are to be considered.

[1] Given a countable set A, it is always possible to find a countable sequence $A_1, A_2, \ldots$ such $A_i \subseteq A$, A_i is a countable set, and $A_i \cap A_j = \emptyset$ for all $i \neq j$.

Definition 5. *A state (of a program P) is a tuple $|T_{co}|, T_{pr}, S, A$ where T_{co} and T_{pr} are sets of threads, S is a store, and A is a set of actions. The initial state of a program $P = (\mathcal{C}, \mathcal{F}, defs)$ fed with inputs $v_1, \ldots, v_n$ is $P(v_1, \ldots, v_n)^t \cdot \emptyset, \emptyset, \emptyset, \emptyset$ where $t \in \mathcal{T}$.*

We define the operational semantics of the language as a state transformation. It is a call-by-value semantics and the evaluation order is, as usual, specified by means of evaluation contexts.

Definition 6. *An evaluation context $\mathbf{E}$ is a one-hole context. Evaluation contexts are defined by the following grammar*

$$\begin{aligned}\mathbf{E} ::= [] \mid c(\overline{v}, \mathbf{E}, \overline{e}) \mid f(\overline{v}, \mathbf{E}, \overline{e}) \mid \texttt{let}\ x = \mathbf{E}\ \texttt{in}\ e \\ \mid \texttt{ref}_\rho(\mathbf{E}) \mid \texttt{get}(\mathbf{E}) \mid \texttt{set}(\mathbf{E}, e) \mid \texttt{set}(v, \mathbf{E}) \\ \mid \texttt{thread}\{f(\overline{v}, \mathbf{E}, \overline{e})\}\end{aligned}$$

where $\overline{v}$ (resp. $\overline{e}$) stands for a list of values (resp. expressions). A redex [2] *is an expression defined by the following grammar:*

$$\begin{aligned}r ::= \ & f(v_1, \ldots, v_n) \mid \texttt{let}\ x = v\ \texttt{in}\ e \mid \texttt{ref}_\rho(v) \mid \texttt{get}(l) \mid \texttt{set}(l, v) \\ & \mid \texttt{thread}\{f(v_1, \ldots, v_n)\} \mid \texttt{unlink}\{e\}\end{aligned}$$

Lemma 1. *For all closed expression e which is not a value, there exists an evaluation context $\mathbf{E}$ and a redex r such that $e = \mathbf{E}[r]$. Moreover, this decomposition is unique.*

Definition 7. *The reduction relation $\rightarrow$ on states is defined by rules of Fig. 1 and Fig. 2. We note $\rightarrow^*$ the reflexive and transitive closure of $\rightarrow$.*

A reduction may either be performed by a thread in cooperative mode (rules (co_1), (co_2), (co_3) and (co_4), or by a thread in preemptive mode (rules (pr_1), (pr_2) and (pr_3)). The semantics of thread creation is described by the rule (co_2). The evaluation of $\texttt{thread}\{f(e_1, \ldots, e_n)\}$ terminates immediately, producing a new thread which evaluates $f(e_1, \ldots, e_n)$. A request for a preemptive computation simply switches the mode of the thread as expressed by the rule (co_3) and elects, when possible, a new cooperative thread. Once terminated, a thread is simply dropped and a new thread, when possible, is elected (rule (co_4)). Once the evaluation of a preemptive computation terminates, the thread returns to the cooperative mode, as expressed in rules (pr_2) and (pr_3). Finally, in both modes, a thread may perform actions which do not depend on other threads (rules (co_1) and (pr_1)). We use the auxiliary rules of Figure 2 to enhance readability. The meaning of $\vdash_{t,\alpha} e_0, S \rightarrow e_1, S'$ is that a reduction of the thread e_0^t, in mode α and with the store S, leads to the thread e_1^t and to the store S'. A function call simply returns the body in which parameters have been substituted (rule (r_1)). In the rule (r_1), σ denotes a substitution from variables to values.

[2] Actually, we should use the term pre-redex because such an expression may not be reducible for an untyped program.

$$\frac{\vdash_{t,co} e, S, A \rightarrow_{\varepsilon} e', S', A'}{\mathbf{E}[e]^t \cdot T_{co}, T_{pr}, S, A \rightarrow \mathbf{E}[e']^t \cdot T_{co}, T_{pr}, S', A'} \quad (co_1)$$

$$\frac{t' \in \mathcal{T},\ t'\ fresh}{\mathbf{E}[\texttt{thread}\{f(v_1, \ldots, v_n)\}]^t \cdot T_{co}, T_{pr}, S, A \rightarrow \mathbf{E}[()]^t \cdot T_{co} \cup \{f(v_1, \ldots, v_n)^{t'}\}, T_{pr}, S, A} \quad (co_2)$$

$$\overline{\mathbf{E}[\texttt{unlink}\{e\}]^t \cdot T_{co}, T_{pr}, S, A \rightarrow\uparrow T_{co}, T_{pr} \cup \{\mathbf{E}[\texttt{unlink}\{e\}]^t\}, S, A} \quad (co_3)$$

$$\overline{()^t \cdot T_{co}, T_{pr}, S, A \rightarrow\uparrow T_{co}, T_{pr}, S, A} \quad (co_4)$$

$$\frac{e_0^t \in T_{pr} \quad e_0 = \mathbf{E}_0[\texttt{unlink}\{\mathbf{E}_1[e_1]\}] \quad \vdash_{t,pr} e_1, S, A \rightarrow_{\varepsilon} e_1', S', A'}{|T_{co}|, T_{pr}, S, A \rightarrow |T_{co}|, T_{pr} \backslash \{e_0^t\} \cup \{\mathbf{E}_0[\texttt{unlink}\{\mathbf{E}_1[e_1']\}]^t\}, S', A'} \quad (pr_1)$$

$$\frac{e^t \in T_{pr} \quad e = \mathbf{E}[\texttt{unlink}\{v\}]}{\epsilon, T_{pr}, S, A \rightarrow\uparrow \{\mathbf{E}[v]^t\}, T_{pr} \backslash \{e^t\}, S, A} \quad (pr_2)$$

$$\frac{e^t \in T_{pr} \quad e = \mathbf{E}[\texttt{unlink}\{v\}]}{e_0^{t_0} \cdot T_{co}, T_{pr}, S, A \rightarrow e_0^{t_0} \cdot (T_{co} \cup \{\mathbf{E}[v]^t\}), T_{pr} \backslash \{e^t\}, S, A} \quad (pr_3)$$

Fig. 1. Dynamic Semantics 1

As usual, we consider a substitution modulo α-renaming of variables bound by a `let` construction. The let construction behaves as expected (rule (r_2)). The initialization of a new memory location l, with the value v by a thread t, chooses a name in $\mathcal{L}_u$ if the memory location must be unsharable. A read or write access, at a memory location l by a thread t in mode α, produces an action $access(t, \alpha, l)$ (rules (r_4) and (r_5)).

3 Static Semantics

We introduce a type and effect system [10, 14] with constraints. Intuitively, this system first introduces a distinction between two sets of memory locations, called sharable and unsharable. The sharable memory locations are used by threads to communicate and must not be accessed during preemptive computations. At the opposite, the unsharable memory locations may be accessed in both cooperative and preemptive modes. The system also introduces a distinction between unsharable memory locations owned by different threads. Regions (whose number must be known statically) are not expressive enough to deal with the dynamic nature of the second distinction. We use two regions s and u to distinguish among sharable and unsharable memory locations respectively, and enforce the second condition by means of constraints on communication between threads.

$$\frac{f(p_1,\ldots,p_n) = e \in defs \quad \exists\sigma.p_i\sigma = v_i,\ i = 1,\ldots,n}{\vdash_{t,\alpha} f(v_1,\ldots,v_n), S, A \rightarrow_\varepsilon e\sigma, S, A} \quad (r1)$$

$$\frac{}{\vdash_{t,\alpha} \texttt{let}\ x = v\ \texttt{in}\ e, S, A \rightarrow_\varepsilon e[v/x], S, A} \quad (r2)$$

$$\frac{l \in \mathcal{L}_s,\ l\ fresh}{\vdash_{t,\alpha} \texttt{ref}_s(v), S, A \rightarrow_\varepsilon l, S \cup \{l \mapsto v\}, A} \quad (r3)$$

$$\frac{l \in O^{-1}(t),\ l\ fresh}{\vdash_{t,\alpha} \texttt{ref}_u(v), S, A \rightarrow_\varepsilon l, S \cup \{l \mapsto v\}, A} \quad (r3')$$

$$\frac{S(l) = v}{\vdash_{t,\alpha} \texttt{get}(l), S, A \rightarrow_\varepsilon v, S, A \cup \{access(t,\alpha,l)\}} \quad (r4)$$

$$\frac{}{\vdash_{t,\alpha} \texttt{set}(l,v), S, A \rightarrow_\varepsilon (), S[l := v], A \cup \{access(t,\alpha,l),\}} \quad (r5)$$

Fig. 2. Dynamic Semantics 2

Definition 8. *Let $P = (\mathcal{C}, \mathcal{F}, defs)$. Let N be a set of type names. A N-typing of P associates:*

1. *to each constructor $c \in \mathcal{C}$ of arity n, a type $\tau_1 * \ldots * \tau_n \rightarrow t$ for $t \in N$ (noted $c : \tau_1 * \ldots * \tau_n \rightarrow t$);*
2. *to each function symbol $f \in \mathcal{F}$ of arity n, a type $\tau_1 * \ldots * \tau_n \stackrel{E,F}{\rightarrow} \tau$ (noted $f : \tau_1 * \ldots * \tau_n \stackrel{E,F}{\rightarrow} \tau$).*

where

- *$\tau, \tau_1, \ldots, \tau_n$ are types defined by the grammar $\tau ::= t \mid \tau\ ref_\rho$ for $\rho \in \{s, u\}$.*
- *E and F are effects defined by the grammar $effect ::= access(\rho) \mid thread \mid unlink$, for $\rho \in \{s, u\}$ and τ a type. We always assume that the constructor $()$ is assigned the type unit and that the function main is assigned a type of the shape $\tau_1 * \ldots * \tau_n \stackrel{E,F}{\rightarrow} unit$.*

Each name $t \in N$ defines a (possibly empty) domain V_t which is the set of values $c(v_1, \ldots, v_n)$ for $c : \tau_1 * \ldots * \tau_n \rightarrow t$ and $v_i \in V_{\tau_i}$. The set $V_{\tau\ ref_\rho}$ is the set of memory locations in the region ρ, carrying values of type τ. For a value v of type τ_i, we note $v : \tau_i$.

In a function type $\tau_1 * \ldots * \tau_n \stackrel{E,F}{\rightarrow} \tau$, E and F are sets of effects, denoting the latent effects of f, i.e effects of the body of f. Intuitively, effects in E denote actions performed in cooperative mode while effects in F denote actions performed in preemptive mode.

In order to verify that threads do not transmit their unsharable memory locations to other threads, we define a predicate which guarantees that no un-

sharable memory location is embedded in values of a given type. If the predicate is verified for a type, values of this type may be safely communicated.

Definition 9. *The predicate Sharable is defined by:*

$$\frac{\rho = s}{Sharable(\tau\ ref_\rho)} \qquad \frac{\forall c : \tau_1 * \ldots * \tau_n \to t \in \mathcal{D} \quad Sharable(\tau_1) \cdots Sharable(\tau_n)}{Sharable(t)}$$

Definition 10. *A set of actions E is said to safe, noted $Safe(E)$, if and only if all element of E has the shape $access(t, \alpha, l) \in E$ with $l \notin \mathcal{L}_s$.*

We introduce a notion of store model (a mapping from memory locations to types), in order to be able to type expressions containing memory locations. It is now possible to introduce the type and effect system we are going to study in the following subsections.

Definition 11. *A store model is a partial function from memory locations to types, such that for all memory location $l \in \mathcal{L}_s$ we have $Sharable(M(l))$.*

Definition 12. *Let P be a program and $\mathcal{D}$ be a N-typing of P. A typing judgement of P has the shape $\Gamma \vdash_M e : \tau, E, F$ where Γ is a typing environment (a mapping from variables to types), M is a store model and E and F are sets of effects. An expression e is said to be well-typed according to a typing $\mathcal{D}$, an environment Γ and a store model M if there exists a type τ and two sets of effects E and F such that $\Gamma \vdash_M e : \tau, E, F$ is derivable from the rules of Fig. 3.*

As usual, when empty, Γ is simply omitted.

Definition 13. *A store model M is a model of a store S, noted $M \models S$, if for all $l \in Dom(S)$ there exists a type τ such that $M(l) = \tau$ and $\vdash_M S(l) : \tau$. A model M' is an extension of M if $Dom(M) \subseteq Dom(M')$ and if for all $l \in Dom(M), M'(l) = M(l)$.*

We note $E_{i,j}$ for $\bigcup_{k=i,\ldots,j} E_k$.

A variable (resp. a memory location) does not produce effects (*var*) (resp. (*loc*)). The application of a constructor or of a function symbol simply collects the effects of the parameters and the latent effect. A read or write access on a memory location of type $\tau\ ref_\rho$ produces an effect $access(\rho)$. The typing of the *let* construction is standard. The typing rule for expressions of the shape `unlink`$\{e\}$ (*unlink*) requires the effects resulting of the typing of e to be safe, that is to contain only actions performed on unsharable memory locations. It moves cooperative effects to preemptive effects and records the presence of the `unlink` construct by introducing the effect *unlink* in the cooperative effect. The typing rule of expressions of the shape `thread`$\{f(e_1, \ldots, e_n)\}$ requires the types τ_i of each parameter e_i to satisfy $Sharable(\tau_i)$ and introduces the effect *thread* in the cooperative effects.

$$\frac{}{\Gamma, x:\tau \vdash_M x:\tau, \emptyset, \emptyset}\ (var) \qquad \frac{M(l)=\tau,\ l \in \mathcal{L}_\rho}{\Gamma \vdash_M l : \tau\ ref_\rho, \emptyset, \emptyset}\ (loc)$$

$$\frac{c:\tau_1 * \ldots * \tau_n \to \tau \in \mathcal{D} \quad \Gamma \vdash_M e_i : \tau_i, E_i, F_i}{\Gamma \vdash_M c(e_1,\ldots,e_n) : \tau, E_{1,n}, F_{1,n}}\ (cons)$$

$$\frac{f:\tau_1 * \ldots * \tau_n \overset{E,F}{\to} \tau \in \mathcal{D} \quad \Gamma \vdash_M e_i : \tau_i, E_i}{\Gamma \vdash_M f(e_1,\ldots,e_n) : \tau, E_{1,n} \cup E, F_{1,n} \cup F}\ (call)$$

$$\frac{\Gamma \vdash_M e : \tau, E, F}{\Gamma \vdash_M ref_u\ e : \tau\ ref_u, E, F}\ (ref1)$$

$$\frac{\Gamma \vdash_M e : \tau, E, F \quad Sharable(\tau)}{\Gamma \vdash_M ref_s\ e : \tau\ ref_s, E, F}\ (ref2)$$

$$\frac{\Gamma \vdash_M e : \tau\ ref_\rho, E, F}{\Gamma \vdash_M\ !e : \tau, E \cup \{access(\rho)\}, F}\ (read)$$

$$\frac{\Gamma \vdash_M e_1 : \tau\ ref_\rho, E_1, F_1 \quad \mathcal{D}; \Gamma \vdash_M e_2 : \tau, E_2, F_2}{\Gamma \vdash_M e_1 := e_2 : unit, E_{1,2} \cup \{access(\rho)\}, F_{1,2}}\ (write)$$

$$\frac{\Gamma \vdash_M e_1 : \tau_1, E_1, F_1 \quad \Gamma, x:\tau_1 \vdash_M e_2 : \tau_2, E_2, F_2}{\Gamma \vdash_M let\ x = e_1\ in\ e_2 : \tau_2, E_{1,2}, F_{1,2}}\ (let)$$

$$\frac{\Gamma \vdash_M e : \tau, E, F \quad Safe(E)}{\Gamma \vdash_M \texttt{unlink}\{e\} : \tau, \{unlink\}, E \cup F}\ (unlink)$$

$$\frac{f:\tau_1 * \ldots * \tau_n \overset{E,F}{\to} unit \in \mathcal{D} \quad \Gamma \vdash_M e_i : \tau_i, E_i, F_i \quad Sharable(\tau_i)}{\Gamma \vdash_M \texttt{thread}\{f(e_1,\ldots,e_n)\} : unit, E_{1,n} \cup E \cup \{thread\}, F_{1,n} \cup F}\ (thread)$$

Fig. 3. Static Semantics

Definition 14. *A program P is well-typed, if there exists a N-typing $\mathcal{D}$ of P such that for all $f : \tau_1, \ldots, \tau_n \overset{E,F}{\to} \tau \in \mathcal{D}$ and for all equation $f(p_1,\ldots,p_n) = e \in defs$, there exists $E' \subseteq E$ and $F' \subseteq F$ such that*

$$p_1 : \tau_1, \ldots, p_n : \tau_n \vdash_\emptyset e : \tau, E', F'$$

where $p_i : \tau_i \equiv x : \tau_i$ if $p_i = x$ and $p_i : \tau_i \equiv q_1 : \tau'_1, \ldots, q_n : \tau'_n$ if $p_i = c(q_1,\ldots,q_n)$ and $c : \tau'_1 * \ldots * \tau'_n \to \tau_i \in \mathcal{D}$. Here, $\emptyset$ denotes the store model which is undefined for all memory locations.

In order to make the properties of well-typed programs more readable, we extend the definition of typing judgments to sets of threads by:

$$\frac{\forall e_i^{t_i} \in T_{co} \cup T_{pr} \quad \vdash_M e_i^{t_i} : unit, E_i, F_i}{\vdash_M T_{co}, T_{pr} : unit, E_{1,n}, F_{1,n}}\ (system)$$

Examples: We consider the type nat defined by the constructors $zero : nat$ and $succ : nat \to nat$. The function g, defined by $g(x) = \texttt{unlink}\{\texttt{set}(x, zero)\}$, admits the type $nat\ ref_u \to unit$. Here the memory location must be private because of an access in preemptive mode. The function $main$ is not typable. Indeed, the function call $g(x)$ appears in a $\texttt{thread}\{\}$ construct; but the type of x denotes a private memory location. This program is rejected by the rule $(thread)$. It is correct to reject such a program. If it was not rejected, the newly created thread could interfere (1) with the other one via this memory location. If the intended meaning of this program is to allow the newly created thread to perform some computations on a copy of the memory location, we can define the function $main'$ which is well-typed. The expression $\texttt{unlink}\{()\}$ can be seen as the `yield` instruction used in thread-based languages.

$$\begin{aligned} main() &= \texttt{let}\ x = \texttt{ref}_u(zero)\ \texttt{in} \\ &\quad \texttt{thread}\{g(x)\}; \texttt{unlink}\{()\}; \texttt{set}(x, succ(zero)); ^{(1)}\ \texttt{get}(x) \end{aligned}$$

$$\begin{aligned} main'() &= \texttt{let}\ x = \texttt{ref}_u(zero)\ \texttt{in} \\ &\quad \texttt{let}\ y = \texttt{ref}_s(\texttt{get}(x))\ \texttt{in} \\ &\quad\ \texttt{thread}\{g(y)\}; \texttt{unlink}\{()\}; \texttt{set}(x, succ(zero)); \texttt{get}(x) \end{aligned}$$

Another example of program which is not typable is given below. Here the thread which evaluates the function f creates a private memory location and transmit it to the thread which evaluates the function g. As in the previous example, the second thread may interfere (2) with the first one.

$$\begin{aligned} f(x) &= \texttt{let}\ y = \texttt{ref}_u(zero)\ \texttt{in} \\ &\quad \texttt{set}(x, y); \texttt{unlink}\{()\}; \texttt{set}(y, succ(zero)); ^{(2)}\ \texttt{get}(y) \end{aligned}$$

$$g(x) = \texttt{let}\ z = \texttt{get}(x)\ \texttt{in}\ \texttt{unlink}\{\texttt{set}(z, zero)\}$$

$$main() = \texttt{let}\ x = \texttt{ref}_s(\texttt{ref}_s(zero))\ \texttt{in}\ \texttt{thread}\{f(x)\}; \texttt{thread}\{g(x)\}$$

4 Confluence

We give a formal definition of the atomicity hypothesis for our language and prove that it is satisfied by well-typed programs. Intuitively, this property guarantees that computations performed by a thread running in cooperative mode do not depend on other threads. Note that a purely cooperative framework always satisfies this hypothesis. More precisely, the non-deterministic choices of the scheduler should not introduce non-determinism (up-to cooperation).

Definition 15. *A program P satisfies the atomicity hypothesis if for all state $St = |T_{co}|, T_{pr}, S, A$ of P, if $T_{co} \neq \emptyset$, $St \to St_1$ and $St \to St_2$ and if none of the reductions is an instance of one of the rules (co_3) or (co_4) then there exist some states St_3 and St_4, equal up to a renaming of threads and memory locations, such that $St_1 \to St_3$ and $St_2 \to St_4$.*

To prove that well-typed programs verify the atomicity hypothesis of the cooperative model, we need some auxiliary results. First, we prove the usual property of subject reduction. Second, we prove that effects are a correct approximation of actions. Third, we prove that any action performed by a thread during a preemptive computation does not involve a sharable memory location. Fourth, we prove that threads do not share their unsharable memory locations. Finally, we we will be able to conclude that the desired property is satisfied.

4.1 Subject Reduction

In the presence of effects, the subject reduction property must also ensure that reductions do not introduce new effects.

Proposition 1. *Let $P : \tau_1 * \ldots * \tau_n \overset{E,F}{\to} unit$ be a well-typed program and let $v_1 : \tau_1, \ldots, v_n : \tau_n$ be some values. If $P(v_1, \ldots, v_n)^t \cdot \emptyset, \emptyset, \emptyset, \emptyset \to^* |T_{co}|, T_{pr}, S, A$ then, for all model M of S, we have $\vdash_M T_{co}, T_{pr} : unit, E', F'$, with $E' \subseteq E$ and $F' \subseteq F$.*

4.2 Correction

The effects introduced by the typing rules are correct approximations of the actions performed during the evaluation. In order to express this property formally, we need a relation between actions and effects.

Definition 16. *We note $\vdash A : E, F$ if and only if for all $t \in \mathcal{T}$, $l \in \mathcal{L}_\rho$ we have:*

$$\forall access(t, co, l) \in A.\ access(\rho) \in E$$
$$\forall access(t, pr, l) \in A.\ access(\rho) \in F$$

Lemma 2. *Let P be a well-typed program, let $|T_{co}|, T_{pr}, S, A$ be a state of P and let E and F be some effects such that $\vdash A : E, F$. If $\vdash_M T_{co}, T_{pr} : unit, E', F'$ for a model M of S, and if $|T_{co}|, T_{pr}, S, A \to |T_{co}{}'|, T_{pr}{}', S', A'$, then $\vdash A' : E \cup E', F \cup F'$.*

Proposition 2. *Let $P : \tau_1 * \ldots * \tau_n \overset{E,F}{\to} unit$ be a well-typed program and let $v_1 : \tau_1, \ldots, v_n : \tau_n$ be some values. If $P(v_1, \ldots, v_n)^t \cdot \emptyset, \emptyset, \emptyset, \emptyset \to^* |T_{co}|, T_{pr}, S, A$ then we have $\vdash A : E, F$.*

4.3 Privacy

In this subsection, we prove that unsharable memory locations are effectively not shared by threads created by well-typed programs. Our goal is to prove that a thread has only access to the sharable memory locations and to its own unsharable memory locations. Formally, if the memory location concerned with an action is unsharable, then this action has been performed by its owner.

Definition 17. *A set of actions A is coherent if for all $t \in \mathcal{T}$, $l \in O^{-1}(t)$ and $access(t', \alpha, l) \in A$, we have $t = t'$.*

To prove that the consistency of the store is preserved through reductions, we need some formal definitions to watch out the communications between threads. First, we need to be able to extract the set of the memory locations held by a thread or held at a memory location.

Definition 18. *For all expression e, the set $loc(e)$ of memory locations occurring in e is defined by induction on e as follows:*

$$\begin{array}{ll} loc(l) & = \{l\} \\ loc(c(e_1, \ldots, e_n)) = loc(f(e_1, \ldots, e_n)) & = \bigcup_i loc(e_i) \\ loc(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) = loc(\texttt{set}(e_1, e_2)) & = loc(e_1) \cup loc(e_2) \\ loc(\texttt{ref}_\rho(e)) = loc(\texttt{get}(e)) = loc(\texttt{unlink}\{e\}) & = loc(e) \\ loc(\texttt{thread}\{f(e_1, \ldots, e_n\}) & = \bigcup_i loc(e_i) \end{array}$$

Intuitively, the consistency property may only be preserved if a thread cannot have access to the unsharable memory locations of the other threads. More precisely, a thread should not: (1) held unsharable memory locations of other threads, (2) transmit its own unsharable memory locations to other threads and (3) be able to obtain unsharable memory locations of other threads by a reduction. This is expressed more formally in the following definition.

Definition 19. *A state $|T_{co}|, T_{pr}, S, A$ is well-formed if*

1. *For all $e^t \in T_{co} \cup T_{pr}$, we have $loc(e) \subseteq \mathcal{L}_s \cup O^{-1}(t)$.*
2. *For all $l \in Dom(S) \cap \mathcal{L}_s$, we have $loc(S(l)) \subseteq \mathcal{L}_s$*
3. *For all $t \in \mathcal{T}$, $l \in Dom(S) \cap O^{-1}(t)$, we have $loc(S(l)) \subseteq \mathcal{L}_s \cup O^{-1}(t)$.*

A first result is that these conditions are stable through reduction.

Proposition 3. *Let P be a well-typed program. If $|T_{co}|, T_{pr}, S, A$ is a well-formed state of P and if $|T_{co}|, T_{pr}, S, A \rightarrow |T_{co}|', T_{pr}{}', S', A'$ then $|T_{co}|', T_{pr}{}', S', A'$ is a well-formed state.*

Finally, the following theorem states the first property of well-typed programs: an unsharable memory location may only be accessed by its owner.

Theorem 1. *Let $P : \tau_1 * \ldots * \tau_n \stackrel{E,F}{\rightarrow} unit$ be a well-typed program and let $v_1 : \tau_1, \ldots, v_n : \tau_n$ be some values. If $P(v_1, \ldots, v_n)^t \cdot \emptyset, \emptyset, \emptyset, \emptyset \rightarrow^* |T_{co}|, T_{pr}, S, A$ then A is coherent.*

4.4 Safety of preemptive computations

The following theorem states the second property of well-typed programs: threads do not access sharable memory locations while in preemptive mode. A first result is that effects of well-typed program are safe.

Proposition 4. *If $P : \tau_1, \ldots, \tau_n \stackrel{E,F}{\rightarrow} unit$ is a well-typed program then $Safe(F)$.*

The following theorem is a direct consequence of this result and of Proposition 2.

Theorem 2. *Let* $P : \tau_1 * \ldots * \tau_n \stackrel{E,F}{\rightarrow} unit$ *be a well-typed program and let* $v_1 : \tau_1, \ldots, v_n : \tau_n$ *be some values. If* $P(v_1, \ldots, v_n)^t \cdot \emptyset, \emptyset, \emptyset, \emptyset \rightarrow^* |T_{co}|, T_{pr}, S, A$ *then for all* t'

$$\bigcup_{l \in \mathcal{L}_s} \{access(t', pr, l)\} \cap A = \emptyset$$

4.5 Validity of the atomicity hypothesis

Theorem 3. *For well-typed programs, the atomicity hypothesis is satisfied.*

Remark 1. A first consequence of Theorem 3 is that preemptive computations steps might be delayed till (but before) the cooperation of the running cooperative thread, thus reducing the number of context switches.

Remark 2. In the semantics, a thread may be in cooperative or in preemptive mode. In a real implementation, it would be possible to map preemptive computations to kernel-level threads. Thanks to Theorem 3 This would allow one to benefit from the underlying system (possibly a multi-processor one). In that case, the cooperative thread which asked for a preemptive computation should appear as suspended until the termination of the preemptive computation.

Remark 3. Note that preemptive computations never prevent the cooperation of the other threads. This is another consequence of Theorem 3. The effective cooperation of a thread does not depend on preemptive computations running concurrently. Of course, this supposes that the scheduler maintains some kind of fairness between cooperative and preemptive modes.

4.6 Blocking primitives, non-cooperative tasks and efficiency

Blocking primitives should only be used in preemptive computations. This can be easily obtained by introducing a new effect, say *blocking*, and by rejecting programs which exhibit it as an action performed in cooperative mode. Another interesting feature offered by preemptive computations is the possibility to perform non terminating computations which do not cooperate. This would be also interesting in a language such as the one considered in [3] where the cooperation is enforced by a static analysis: programs not recognized as being cooperative could still be executed in a preemptive way.

5 Related Work

The Gnu-Pth[7] thread library designed for Unix platforms provides cooperative scheduling in the context of POSIX/ANSI-C. Blocking I/O primitives have been rewritten in order to work in a cooperative scheduling. This differs from our proposal which gives users the freedom to safely code preemptive computations.

The Cyclone[9] language proposes a safe variant of C by limiting the use of pointers. Technically, regions are used in order to control that a pointer is not

used outside its definition scope. An extension of Cyclone to multithreading is proposed in [8]. The main difference with our approach is that only preemptive threads are considered, controlled by locks and with possibilities of deadlocks.

The FairThreads model defines a framework to mix cooperative and preemptive threads in C[6], Java[5], or Scheme[13]. A cooperative thread linked to a scheduler can unlink from it to perform preemptive computations. This programming model is close to the one we present here but is not concerned by safety.

The ReactiveML[11] library introduces reactive programming in Objective Caml[2]. Reactive programming is a variation of the cooperative model introducing a notion of logical time. However, preemptive computations are not presently covered by ReactiveML, thus restricting it to purely cooperative systems.

6 Conclusion

In this paper, we have introduced a small language for programming cooperative threads in which one can define preemptive computations to handle tasks that are not suited to a purely cooperative framework. We have introduced a type and effect system which ensures that preemptive computations do not interfere with the threads running in cooperative mode. This work is part of a more global work which focuses on the notion of reactivity. The model we consider here is a subset of the full model but we think that it captures the main difficulties raised by the preemptive scheduling at the upper level (Indeed, in the full model, an unlinked thread can be seen as a thread running alone on a special dedicated scheduler). We are currently working both on the design of a type inference algorithm for the type system presented here and on its extension to the full model. Meanwhile, we are investigating the notion of reactivity for this model. In particular, we have developed static analysis tools to ensure the reactivity of programs [4, 3]. Using techniques borrowed from term rewriting systems, we showed that polynomial-time termination of reactions can be ensured.

References

1. http://www.inria.fr/mimosa/rp.
2. The Objective Caml System. http://www.ocaml.org.
3. R.M. Amadio and F. Dabrowski. Feasible reactivity for synchronous cooperative threads. *presented at the workshop Expressiveness in Concurrency, San Francisco, to appear*, 2005.
4. Roberto M. Amadio and Silvano Dal Zilio. Resource control for synchronous cooperative threads. In *Proc. of CONCUR 2004 – 15th International Conference on Concurrency Theory*, pages 68–82. Lecture Notes in Computer Science, Vol. 3170, Springer-Verlag, 2004.
5. F. Boussinot. *Java Fair Threads.* Inria research report, RR-4139, 2001.
6. F. Boussinot. FairThreads: mixing cooperative and preemptive threads in C. *Concurrency and Computation-Practice and Experience*, in press, 2005.

7. Ralf S. Engelschall. *Portable Multithreading.* Proc. USENIX Annual Technical Conference, San Diego, California, 2000.
8. Dan Grossman. Type-safe multithreading in Cyclone. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international works hop on Types in languages design and implementation*, pages 13–25, New York, NY, USA, 2003. ACM Press.
9. Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.
10. J. M. Lucassen. Types and effects: Towards the integration of functional and imperative programming. In *Ph.D. Thesis MIT/LCS/TR-408.* Massachusetts Institute of Technology, 1987.
11. Louis Mandel and Marc Pouzet. ReactiveML, a reactive extension to ML. In *ACM International conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.
12. N. Halbwachs. *Synchronous programming of reactive systems.* 1993.
13. Manuel Serrano, Frédéric Boussinot, and Bernard Serpette. Scheme Fair Threads. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 203–214, New York, NY, USA, 2004. ACM Press.
14. Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, pages 162–173, Los Alamitos, California, 1992. IEEE Computer Society Press.