LIF

Laboratoire d'Informatique Fondamentale
de Marseille

Unité Mixte de Recherche 6166
CNRS – Université de Provence – Université de la Méditerranée

Resource Control for Synchronous Cooperative Threads

Roberto M. Amadio and Silvano Dal Zilio

Rapport/Report 22-2004

4 mai 2004

Resource Control for Synchronous Cooperative Threads

Roberto M. Amadio and Silvano Dal Zilio

LIF – Laboratoire d'Informatique Fondamentale de Marseille
UMR 6166
CNRS – Université de Provence – Université de la Méditerranée

{amadio,dalzilio}@lif.univ-mrs.fr

Abstract/Résumé

We develop new methods to statically bound the resources needed for the execution of systems of concurrent, interactive threads.

Our study is concerned with a *synchronous* model of interaction based on cooperative threads whose execution proceeds in synchronous rounds called instants. Our contribution is a system of compositional static analyses to guarantee that each instant terminates and to bound the size of the values computed by the system as a function of the size of its parameters at the beginning of the instant.

Our method generalises an approach designed for first-order functional languages that relies on a combination of standard termination techniques for term rewriting systems and an analysis of the size of the computed values based on the notion of quasi-interpretation.

We show that these two methods can be combined to obtain an explicit polynomial bound on the space needed for the execution of the system during an instant. We also provide evidence for the expressivity of our synchronous programming model and describe a bytecode for a related virtual machine.

Keywords: resource control, synchronous programming, quasi-interpretation, termination, term rewriting systems.

Nous présentons une nouvelle approche permettant de borner statiquement les ressources nécessaires à l'exécution de systèmes de threads interactifs et concurrents.

Notre étude porte sur un modèle d'interaction *synchrone* basé sur des threads coopératifs dont l'exécution procède par "segments synchrones" successifs appelés instants. Notre contribution est un système d'analyses statiques (toutes compositionnelles) qui garantissent que chaque instant se termine et qui permettent de borner la taille des valeurs calculées par le système en fonction de la taille de ses paramètres au début de l'instant.

Notre méthode généralise une approche conçue pour des langages de programmation fonctionnels de premier ordre qui se fonde sur la combinaison de techniques standards pour la preuve de terminaison de systèmes de réécriture et d'une analyse de la taille des valeurs

calculées basée sur la notion de quasi-interprétation.

Nous prouvons que ces méthodes peuvent être combinées pour extraire de chaque programme un polynôme qui borne l'espace requis pour l'exécution du système pendant un instant. Nous montrons également, à l'aide d'exemples, l'expressivité de notre modèle de programmation synchrone et nous décrivons un possible langage de bytecode pour une machine virtuelle associée.

Mots-clés : contrôle de ressources, programmation synchrone, quasi-interprétations, terminaison, systèmes de réécriture.

Relecteurs/Reviewers: Frédéric Dabrowski and Gérard Boudol.

Notes: The authors are partly supported by ACI CRISS.

1 Introduction

The problem of bounding the usage made by programs of their resources has already attracted considerable attention. Automatic extraction of resource bounds has mainly focused on (first-order) functional languages starting from Cobham's characterisation [13] of polynomial time functions by bounded recursion on notation. Following work, see, *e.g.*, [5, 19, 17, 15], has developed various inference techniques that allow for efficient analyses while capturing a sufficiently large range of practical algorithms.

Previous work [20, 8] has shown that polynomial time or space bounds can be obtained by combining traditional termination techniques for term rewriting systems with an analysis of the size of computed values based on the notion of quasi-interpretation. In [2], we have considered the problem of automatically inferring quasi-interpretations in the space of multi-variate max-plus polynomials. In [1], we have presented a virtual machine and a corresponding bytecode for a first-order functional language and shown how size and termination annotations can be formulated and verified at the level of the bytecode. In particular, we can derive from the verification an explicit polynomial bound on the space required to execute a given bytecode.

In this work, we aim at extending and adapting these results to a concurrent framework. Our starting point, is a quite basic and popular model of parallel threads interacting on shared variables. The kind of concurrency we consider is a *cooperative* one. This means that by default a running thread cannot be preempted unless it explicitly decides to return the control to the scheduler. In *preemptive* threads, the opposite hypothesis is made: by default a running thread can be preempted at any point unless it explicitly requires that a series of actions is atomic. We refer to, *e.g.*, [24] for an extended comparison of the cooperative and preemptive models. Our viewpoint is pragmatic: the cooperative model is closer to the sequential one and many applications are easier to program in the cooperative model than in the preemptive one. Thus, as a first step, it makes sense to develop a resource control analysis for the cooperative model.

In models based on preemptive threads, it is difficult to foresee the behaviour of the scheduler which might depend on timing information not available in the model. For this reason and in spite of the fact that most schedulers are deterministic, the scheduler is often modelled as a non-deterministic process. In cooperative threads, the interrupt points are explicit in the program and it is possible to think of the scheduler as a deterministic process. Then the resulting model is deterministic and this fact considerably simplifies its programming, debugging, and analysis.

The second major design choice is to assume that the computation is regulated by a notion of *instant*. An instant lasts as long as a thread can make some progress in the current instant. In other terms, an instant ends when the scheduler realizes that all threads are either stopped, or waiting for the next instant, or waiting for a value that no thread can produce in the current instant. Because of this notion of instant, we regard our model as *synchronous*. Because the model includes a logical notion of time, it is possible for a thread *to react to the absence of an event*.

The reaction to the absence of an event, is typical of synchronous languages such as Esterel [7]. Boussinot *et al.* have proposed a weaker version of this feature where the reaction to the absence happens in the following instant [6] and they have implemented it in various programming environments based on C, Java, and Scheme [26]. They have also advocated the relevance of this concept for the programming of mobile code and demonstrated

that the possibility for a 'synchronous' mobile agent to react to the absence of an event is an added factor of flexibility for programs designed for open distributed systems, whose behaviours are inherently difficult to predict.

Recently, Boudol [4] has proposed a formalisation of this programming model. Our analysis will essentially focus on a small fragment of this model where higher-order functions, dynamic thread creation, and dynamic memory allocation are ruled out. We believe that what is left is still expressive and challenging enough as far as resource control is concerned. Our analysis goes in three main steps. A first step is to guarantee that each instant terminates (Section 4). A second step, is to bound the size of the computed values as a function of the size of the parameters at the beginning of the instant (Section 5). A third step, is to combine the termination and size analyses in order to obtain polynomial bounds on the space needed for the execution of the system during an instant as a function of the size of the parameters at the beginning of the instant (Section 6).

A characteristic of our static analyses is that to a great extent they make abstraction of the memory and the scheduler. This means that each thread can be analysed separately, that the complexity of the analyses grows linearly in the number of threads, and that an incremental analysis of a dynamically changing system of threads is possible. Preliminary to these analyses, is a control flow analysis (Section 3) that guarantees that each thread reads each register at most once in an instant. We will see that without this condition, it is very easy to achieve an exponential growth of the space needed for the execution. From a technical point of view, the benefit of this *read once* condition is that it allows to regard behaviours as *functions* of their initial parameters and the registers they may read in the instant. Taking this functional viewpoint, we are able to adapt the main techniques developed for proving termination and size bounds in the first-order functional setting.

We point out that our static size analyses are not intended to predict the size of the system after arbitrary many instants. This is a harder problem which in general seems to require an understanding of the *global* behaviour of the system: typically one has to find an invariant that shows that the parameters of the system stay within certain bounds. For this reason, we believe that in practice our static analyses should be combined with a dynamic controller that at the end of each instant checks the size of the parameters of the system.

Along the way and in appendix A, we provide a number of programming examples illustrating how certain synchronous and/or concurrent programming paradigms can be represented in our model. These examples suggest that the constraints imposed by the static analyses are not too severe and that their verification can be automated.

Finally, we describe a bytecode for a simple virtual machine implementing our programming model (Section 7). This provides a more precise description of the resources needed for the execution of the systems we consider and opens the way to the verification of resource bounds at the bytecode level, following the 'typed assembly language' approach adopted in [1] for the purely functional fragment of the language. Proofs are available in appendix B.

2 A Model of Synchronous Cooperative Threads

A *system* of synchronous cooperative threads is described by: (1) a list of mutually recursive type definitions, (2) a list of shared registers (or global variables) with a type and a default value, and (3) a list of mutually recursive functions and behaviours definitions relying on pattern matching. In this respect, the resulting programming language is reminiscent of

ERLANG [3], which is a *practical* language to develop concurrent applications.

The set of instructions a behaviour can execute is rather minimal. Indeed, our language is already in a *pre-compiled* form where registers are assigned constant values and behaviours definitions are tail recursive. However, it is quite possible to extend the language and our analyses to have registers' names as first-class values and general recursive behaviours.

Expressions. We rely on standard notation. If α, β are formal terms then $Var(\alpha)$ is the set of free variables in α (variables in patterns are not free) and $[\alpha/x]\beta$ denotes the substitution of α for x in β. If h is a function, $h[u/i]$ denotes a function update.

Expressions and values are built from a finite number of constructors, ranged over by $\mathsf{c}, \mathsf{c}', \ldots$ We use $f, f', \ldots$ to range over function identifiers and $x, x', \ldots$ for variables, and distinguish the following three syntactic categories:

$$\begin{array}{ll} v ::= \mathsf{c}(v, \ldots, v) & \text{(values)} \\ p ::= x \mid \mathsf{c}(p, \ldots, p) & \text{(patterns)} \\ e ::= x \mid \mathsf{c}(e, \ldots, e) \mid f(e, \ldots, e) & \text{(expressions)} \end{array}$$

The *size* of an expression $|e|$ is defined as 0 if e is a constant or a variable and $1 + \Sigma_{i \in 1..n}|e_i|$ if e is of the form $\mathsf{c}(e_1, \ldots, e_n)$ or $f(e_1, \ldots, e_n)$.

A function of arity n is defined by a sequence of pattern-matching *rules* of the form $f(\vec{p_1}) = be_1, \ldots, f(\vec{p_k}) = be_k$, where be_i is either an expression or a thread behaviour (see below), and $\vec{p_1}, \ldots, \vec{p_k}$ are sequences of length n of patterns. We follow the usual hypothesis that the patterns in $\vec{p_1}, \ldots, \vec{p_n}$ are linear (a variable appears at most once). For the sake of simplicity, we will also assume that in a function definition a sequence of values $\vec{v}$ matches exactly a sequence of patterns $\vec{p_i}$ in a function definition. This hypothesis can be relaxed.

Inductive types are defined by equations of the shape $t = \cdots \mid \mathsf{c}\ \mathit{of}\ (t_1 * \cdots * t_n) \mid \cdots$. For instance, the type of natural numbers in unary format can be defined as follows: $\mathit{nat} = \mathsf{z} \mid \mathsf{s}\ \mathit{of}\ \mathit{nat}$. Functions, values, and expressions are assigned first order types of the shape $(t_1 * \cdots * t_n) \to t$ where $t, t_1, \ldots, t_n$ are inductive types.

Behaviours. Some function symbols may return a thread behaviour $b, b', \ldots$ rather than a value. In contrast to 'pure' expressions, a behaviour does not return a result but produces *side-effects* by reading and writing a set of global registers, ranged over by $\mathsf{r}, \mathsf{r}', \ldots$ A behaviour may also affect the scheduling status of the thread executing it (see below).

$$\begin{array}{lcl} be, \ldots & ::= & e \mid b \\ b, b', \ldots & ::= & \mathsf{stop} \mid \mathsf{yield}.b \mid f(\vec{e}) \mid \mathsf{next}.f(\vec{e}) \mid \mathsf{r} := e.b \mid \\ & & \mathsf{match}\ \mathsf{r}\ \mathsf{with}\ p_1 \Rightarrow b_1 \mid \cdots \mid p_k \Rightarrow b_k \mid [x] \Rightarrow f(\vec{e}) \end{array}$$

The *effect* of the various instructions is informally described as follows: stop, terminates the executing thread for ever; $\mathsf{yield}.b$, halts the execution and hands over the control to the scheduler — the control should return to the thread later in the same instant and execution resumes with b; $f(\vec{e})$ and $\mathsf{next}.f(\vec{e})$ switch to another behaviour immediately or at the beginning of the following instant; $\mathsf{r} := e.b$, evaluates the expression e, assigns its value to r and proceeds with the evaluation of b; $\mathsf{match}\ \mathsf{r}\ \mathsf{with}\ p_1 \Rightarrow b_1 \mid \cdots \mid p_k \Rightarrow b_k \mid [x] \Rightarrow f(\vec{e})$, waits until the value of r matches one of the patterns $p_1, \ldots, p_k$ (there could be no delay) and yields the control otherwise. At the end of the instant, if the value of r is v and no rules filter v then start the next instant with the behaviour $[v/x]f(\vec{e})$. By convention, when the $[x] \Rightarrow \ldots$ branch is omitted, it is intended that if the match conditions are not satisfied in the current instant, then they are checked again in the following one.

Systems. Every thread has a *status*, denoted $X, X', \ldots$, ranging over $\{N, R, S, W\}$ — where N stands for next, R for run, S for stop, and W for wait. A *system* of synchronous threads $B, B', \ldots$ is a finite mapping from thread indexes to pairs (behaviour, status). Each register has a type and a default value — its value at the beginning of an instant — and we use $s, s', \ldots$ to denote a *store*, an association between registers and their values. We suppose the thread indexes $i, k, \ldots$ range over $\mathbf{Z}_n = \{0, 1, \ldots, n-1\}$ and that at the beginning of each instant the store is s_o, such that each registers is assigned its default value. If B is a system and $i \in \mathbf{Z}_n$ a valid thread index then we denote with $B_1(i)$ the behaviour executed in the thread i and with $B_2(i)$ its current status. Initially, all threads have status R, the current thread index is 0, and $B_1(i)$ is a behaviour expression of the shape $f(\vec{v})$. It is a standard exercise to formalise a type system of simple first-order functional types for such a language and, in the following, we assume that all systems we consider are well typed.

Operational semantics. The *operational semantics* is described by three relations of growing complexity, presented in Table 1:

- $e \Downarrow v$, the closed expression e evaluates to the value v,
- $(b, s) \xrightarrow{X} (b', s')$, the behaviour b with store s runs an atomic sequence of actions till b', producing a store s', and returning the control to the scheduler with status X.
- $(B, s, i) \rightarrow (B', s', i')$ the system B with store s and current thread (index) i runs an atomic sequence of actions (read from $B_1(i)$) and becomes (B', s', i').

Behaviour reduction $(b, s) \xrightarrow{X} (b', s')$ is described by 7 rules, (b_1) to (b_7). We note that during the instant, a thread may undergo the following status transitions: $R \rightarrow S, W, N$ and $W \rightarrow R$. Also, the status of a thread may change from R to W if and only if the executed behaviour is of the form $\mathsf{match}\ \mathsf{r}\ \mathsf{with} \ldots$ and no filters match the value of r. System reduction $(B, s, i) \rightarrow (B', s', i')$ is described by 2 rules, (s_1) and (s_2), and relies on: (1) a partial function $\mathcal{N}$ that computes the index of the next thread that will run in the current instant; (2) A function $\mathcal{U}$ that updates the status of the threads at the end of an instant. In particular, it activates the branches $[x] \Rightarrow f(\vec{e})$ of threads in status W.

Scheduler. The scheduler is determined by the functions $\mathcal{N}$ and $\mathcal{U}$. To ensure progress of the scheduling, we assume that if $\mathcal{N}$ returns an index then it must be possible to run the corresponding thread in the current instant and that if $\mathcal{N}$ is undefined (denoted $\mathcal{N}(\ldots) \uparrow$) then no thread can be run in the current instant. In addition, one could arbitrarily enrich the functional behaviour of the scheduler by considering extensions such that $\mathcal{N}$ depends on the history, the store, and/or is defined by means of probabilities. When no more thread can run, the instant ends and the function $\mathcal{U}$ performs the following status transitions: $N \rightarrow R$, $W \rightarrow R$. For simplicity, we assume here that every thread in status W takes the $[x] \Rightarrow \ldots$ branch. Note that the function $\mathcal{N}$ is undefined on the updated system if and only if all threads are stopped.

The cooperative fragment. The 'cooperative' fragment of the model with no synchrony is obtained by removing the next instruction and assuming that for all match instructions the branch $[x] \Rightarrow f(\vec{e})$ is such that $f(\ldots) = \mathsf{stop}$. Then all the interesting computation happens in the first instant, and in the second instant all the threads terminate. This fragment is already powerful enough to simulate, *e.g.*, Kahn networks (see appendix A.1).

Example 1 (channels and signals) *As shown in our informal presentation of behaviours, the* match *instruction allows to read a register subject to certain filter conditions. This is a powerful mechanism which recalls,* e.g.*, Linda communication [11], and that allows to encode various forms of channel and signal communication.*

(1) *We want to represent a* one place channel c *carrying values of type* t*. We introduce a new type* $ch(t) = \mathsf{empty} \mid \mathsf{full}$ *of* t *and a register* c *of type* $ch(t)$ *with default value* empty*. A thread should send a message on* c *only if* c *is empty and it should receive a message only if* c *is* not *empty (a received message is discarded). These operations can be modelled using the following two derived operators:*

$$\begin{array}{ll} \mathsf{send}(\mathsf{c}, e).b & =_{def} \mathsf{match\ c\ with\ empty} \Rightarrow \mathsf{c} := \mathsf{full}(e).b \\ \mathsf{receive}(\mathsf{c}, x).b & =_{def} \mathsf{match\ c\ with\ full}(x) \Rightarrow \mathsf{c} := \mathsf{empty}.b \end{array}$$

(2) *We want to represent a* fifo channel c *carrying values of type* t *such that a thread can always emit a value on* c *but may receive only if there is at least one message in the channel. We introduce a new type* $fch(t) = \mathsf{nil} \mid \mathsf{cons}$ *of* $t * fch(t)$ *and a register* c *of type* $fch(t)$ *with default value* nil*. Hence a fifo channel is modelled by a register holding a list of values. We consider two read operations —* freceive *to fetch the first message on the channel and* freceiveall *to fetch the whole queue of messages — and we use the auxiliary function* insert *to queue messages at the end of the list:*

$$\begin{array}{ll} \mathsf{fsend}(\mathsf{c}, \mathsf{e}).\mathsf{b} & =_{def} \mathsf{match\ c\ with}\ l \Rightarrow \mathsf{c} := \mathit{insert}(e, l).b \\ \mathsf{freceive}(\mathsf{c}, x).b & =_{def} \mathsf{match\ c\ with\ cons}(x, l) \Rightarrow \mathsf{c} := l.b \\ \mathsf{freceiveall}(\mathsf{c}, x).b & =_{def} \mathsf{match\ c\ with\ cons}(y, l) \Rightarrow \mathsf{c} := \mathsf{nil}.[\mathsf{cons}(y, l)/x]b \end{array}$$

$$\mathit{insert}(x, \mathsf{nil}) = \mathsf{cons}(x, \mathsf{nil})\ , \qquad \mathit{insert}(x, \mathsf{cons}(y, l)) = \mathsf{cons}(y, \mathit{insert}(x, l))$$

(3) *We want to represent a signal* s *with the typical associated primitives: emitting a signal and blocking until a signal is present. We define a type sig* $= \mathsf{abst} \mid \mathsf{prst}$ *and a register* s *of type* sig *with default value* abst*, meaning that a signal is originally absent:*

$$\mathsf{emit}(\mathsf{s}).b\ =_{def}\ \mathsf{s} := \mathsf{prst}.b \qquad\qquad \mathsf{wait}(\mathsf{s}).b\ =_{def}\ \mathsf{match\ s\ with\ prst} \Rightarrow b$$

3 Control Flow Analysis

To make possible a compositional analysis for resource control, we propose to restrict the admissible behaviours and we define a simple preliminary control flow analysis that guarantee that this restriction is met. We conclude this section with the definition of a symbolic representation of the states reachable by a behaviour that is a stepping stone in our analyses for termination and value size limitation of the instant.

3.1 Read Once Condition

In order to enable a compositional analysis of the size of the values computed by threads, we constrain the thread behaviours that may be executed. More precisely, we require and statically check on the control flow, that threads can read any given register at most once in an instant. The following simple example shows that *without* the read once restriction, a thread can use a register as an accumulator and produce an exponential growth of the size of the data within an instant.

EXPRESSION EVALUATION:

$$(\mathsf{e}_1)\ \frac{\vec{e} \Downarrow \vec{v}}{\mathsf{c}(\vec{e}) \Downarrow \mathsf{c}(\vec{v})} \qquad (\mathsf{e}_2)\ \frac{\vec{e} \Downarrow \vec{v}, \quad f(\vec{p}) = e, \quad \sigma\vec{p} = \vec{v}, \quad \sigma(e) \Downarrow v}{f(\vec{e}) \Downarrow v}$$

BEHAVIOUR REDUCTION:

$$(\mathsf{b}_1)\ \frac{}{(\mathsf{stop}, s) \xrightarrow{S} (\mathsf{stop}, s)} \qquad (\mathsf{b}_2)\ \frac{}{(\mathsf{yield}.b, s) \xrightarrow{R} (b, s)} \qquad (\mathsf{b}_3)\ \frac{}{(\mathsf{next}.f(\vec{e}), s) \xrightarrow{N} (f(\vec{e}), s)}$$

$$(\mathsf{b}_4)\ \frac{\text{no pattern matches } s(\mathsf{r})}{(\mathsf{match}\ \mathsf{r}\ \mathsf{with} \ldots, s) \xrightarrow{W} (\mathsf{match}\ \mathsf{r}\ \mathsf{with} \ldots, s)} \qquad (\mathsf{b}_5)\ \frac{\sigma p = s(\mathsf{r}), \quad (\sigma b, s) \xrightarrow{X} (b', s')}{(\mathsf{match}\ \mathsf{r}\ \mathsf{with} \cdots \mid p \Rightarrow b \mid \ldots, s) \xrightarrow{X} (b', s')}$$

$$(\mathsf{b}_6)\ \frac{\vec{e} \Downarrow \vec{v}, \quad f(\vec{p}) = b, \quad \sigma\vec{p} = \vec{v}, \quad (\sigma b, s) \xrightarrow{X} (b', s')}{(f(\vec{e}), s) \xrightarrow{X} (b', s')} \qquad (\mathsf{b}_7)\ \frac{e \Downarrow v, \quad (b, s[v/\mathsf{r}]) \xrightarrow{X} (b', s')}{(\mathsf{r} := e.b, s) \xrightarrow{X} (b', s')}$$

SYSTEM REDUCTION:

$$(\mathsf{s}_1)\ \frac{(B_1(i), s) \xrightarrow{X} (b', s'), \quad B_2(i) = R, \quad B' = B[(b', X)/i], \quad \mathcal{N}(B', s', i) = k}{(B, s, i) \to (B'[(B'_1(k), R)/k], s', k)}$$

$$(\mathsf{s}_2)\ \frac{\begin{array}{c}(B_1(i), s) \xrightarrow{X} (b', s'), \quad B_2(i) = R, \quad B' = B[(b', X)/i], \quad \mathcal{N}(B', s', i) \uparrow, \\ B'' = \mathcal{U}(B', s'), \quad \mathcal{N}(B'', s_o, 0) = k\end{array}}{(B, s, i) \to (B'', s_o, k)}$$

CONDITIONS ON THE SCHEDULER:

If $\mathcal{N}(B, s, i) = j$ then $\quad B_2(k) = R$ or ($B_2(k) = W$ and
$B_1(k) = \mathsf{match}\ \mathsf{r}\ \mathsf{with} \cdots \mid p \Rightarrow b \mid \ldots, \sigma p = s(\mathsf{r})$)

If $\mathcal{N}(B, s, i) \uparrow$ then $\quad \forall k \in \mathbf{Z}_n,\ B_2(k) \in \{N, S\}$ or ($B_2(k) = W$,
$B_1(k) = \mathsf{match}\ \mathsf{r}\ \mathsf{with} \ldots$ and no pattern matches $s(\mathsf{r})$)

$$\mathcal{U}(B, s)(i) = \begin{cases} (b, S) & \text{if } B(i) = (b, S) \\ (b, R) & \text{if } B(i) = (b, N) \\ ([s(\mathsf{r})/x](f(\vec{e})), R) & \text{if } B(i) = (\mathsf{match}\ \mathsf{r}\ \mathsf{with} \cdots \mid [x] \Rightarrow f(\vec{e}), W) \end{cases}$$

Table 1: Operational semantics

Example 2 *Let* $nat = \mathsf{z} \mid \mathsf{s}\,of\ nat$ *be the type of tally natural numbers. The function* dble*, defined by the two rules* $dble(\mathsf{z}) = \mathsf{z}$ *and* $dble(\mathsf{s}(n)) = \mathsf{s}(\mathsf{s}(d(n)))$ *doubles a number so that* $|dble(n)| = 2|n|$*. We assume* r *is a register of type nat with initial value* $\mathsf{s}(\mathsf{z})$*. Now consider the following recursive behaviour:*

$$exp(\mathsf{z}) = \mathsf{stop}\ , \qquad exp(\mathsf{s}(n)) = \mathsf{match\ r\ with}\ m \Rightarrow \mathsf{r} := dble(m).exp(n)$$

The evaluation of $exp(n)$ *involves* $|n|$ *reads to the register* r *and, after each read operation, the size of the value stored in* r *doubles. Hence, at end of the instant, the register contains a value of size* $2^{|n|}$*.*

The read once condition is comparable to the restriction on the absence of immediate cyclic definitions in Lustre and does not appear to be a severe limitation on the expressiveness of the language. An important consequence of the *read once* condition is that a behaviour can be described as a *function* of its parameters and the registers it may read during an instant. We stress that we retain the *read once* condition for its simplicity, however it is clear that one could weaken the condition and adapt the analysis in the following section 3.2 to allow the reading of a register at most a constant number of times.

3.2 Enforcing the Read Once Condition

We now describe a simple analysis that guarantees the read once condition. Consider the set $Reg = \{\mathsf{r}_1, \ldots, \mathsf{r}_m\}$ of the registers as an alphabet. To every function symbol f whose result is a behaviour, we associate the least language $R(f)$ of words over Reg such that ϵ, the empty word, is in $R(f)$ and the following conditions are satisfied:

$$\text{if } f(\vec{p}_1) = b_1, \ldots, f(\vec{p}_n) = b_n \text{ are the rules of } f \text{ then } R(f) =_{\text{def}} R(f) \cdot \textstyle\bigcup_{i\in 1..n} R(b_i)\ ,$$

$$R(\mathsf{match\ r\ with}\ p_1 \Rightarrow b_1 \mid \cdots \mid p_n \Rightarrow b_n \mid [x] \Rightarrow g(\vec{e})) =_{\text{def}} \{\mathsf{r}\} \cdot \textstyle\bigcup_{i\in 1..n} R(b_i)\ ,$$

$$R(\mathsf{stop}) = \{\epsilon\}\ , \qquad R(g(\vec{e})) = R(g)\ , \qquad R(\mathsf{r} := e.b) = R(b)\ ,$$

$$R(\mathsf{yield}.b) = R(b), \qquad R(\mathsf{next}.g(\vec{e})) = \{\epsilon\}\ .$$

Looking at the words in $R(f)$, we get an over-approximation of the sequences of registers that a thread can read in an instant starting from the control point f with arbitrary parameters and store. Note that an expression can never read or write a register.

To determine the sets $R(f)$, we perform an iterative computation according to the equations above. The iteration stops when either (1) we reach a fixpoint — and we are sure that the property holds — or (2) we notice that a word in the current approximation of $R(f)$ contains the same register twice — thus we never need to consider words whose length is greater than the number of registers. Since there is only a finite number of registers, it is easy to prove that the computation of $E(f)$ always terminates. If situation (1) occurs, then for every function symbol f that returns a behaviour we can obtain a list of registers $\vec{\mathsf{r}}_f$ that a thread starting from control point f may read. We are going to consider these registers as *hidden parameters* (variables) of the function f. If situation (2) occurs, we cannot guarantee the read once property and we stop analysing the code.

Example 3 *This will be the running example for this section. We consider the representation of signals as in Example 1(3). We assume two signals* sig *and* ring*. The behaviour* $alarm(n, m)$

will emit a signal on ring *if it detects that no signal is emitted on* sig *for* m *consecutive instants. The alarm delay is reset to* n *if the signal* sig *is present.*

$$\begin{array}{lcl} alarm(x,\, \mathsf{z}) & = & \mathsf{ring} := \mathsf{prst.stop}\ , \\ alarm(x,\, \mathsf{s}(y)) & = & \mathsf{match\ sig\ with\ prst} \Rightarrow \mathsf{next}.alarm(x,x)\ \mid\ [\_] \Rightarrow alarm(x,y) \end{array}$$

By computing R *on this example, we obtain:* $R(alarm) = \{\epsilon\} \cdot (R(\mathsf{ring} := \mathsf{prst.stop}) \cup R(\mathsf{match\ sig\ with}\ \ldots)) = \{\epsilon\} \cdot (\{\epsilon\} \cup (\{\mathsf{sig}\} \cdot \{\epsilon\})) = \{\epsilon, \mathsf{sig}\}$.

3.3 Control Points

We associate with a system satisfying the read once condition a *finite* number of control points. Control points are a symbolic representation of the set of states reachable by a thread based on the control flow graph of its behaviours. A *control point* is a triple $(f(\vec{p}), be, i)$ where, intuitively, f is the currently called function, $\vec{p}$ represents the patterns crossed so far in the function definition plus possibly the registers that still have to be read, be is the continuation, and i is an integer flag in $\{0, 1, 2\}$ that will be used to associate with the control point various kinds of conditions. If the function f returns a value and is defined by the rules $f(\vec{p}_1) = e_1, \ldots, f(\vec{p}_n) = e_n$, then we associate with f the set $\{(f(\vec{p}_1), e_1, 0), \ldots, (f(\vec{p}_n), e_n, 0)\}$.

On the other hand, if the function f is a behaviour defined by the rules $f(\vec{p}_1) = b_1, \ldots, f(\vec{p}_n) = b_n$ then the computation of the control points proceeds as follows. We assume that the registers have been ordered and that for every behaviour definition f, we have an ordered vector $\vec{\mathsf{r}}_f$ of registers that may be read within an instant starting from f. (The vector $\vec{\mathsf{r}}_f$ is obtained from $R(f)$). With every such f we associate a fresh function symbol f^+ whose arity is that of f plus the length of $\vec{\mathsf{r}}_f$ and we regard the registers as part of the formal parameters of f^+. Then from the definition of f we produce the set $\bigcup_{i \in 1..n} \mathcal{C}(f^+, (\vec{p}_i, \vec{\mathsf{r}}_f), b_i)$, where $\mathcal{C}(f, \vec{p}, b)$ is defined inductively on b as follows:

$\mathcal{C}(f^+, \vec{p}, b) =$ case b of

$(\mathcal{C}_1)$ stop : $\{(f^+(\vec{p}), b, 2)\}$

$(\mathcal{C}_2)$ $g(\vec{e})$: $\{(f^+(\vec{p}), b, 0)\}$

$(\mathcal{C}_3)$ $\mathsf{yield}.b'$: $\{(f^+(\vec{p}), b, 2)\} \cup \mathcal{C}(f^+, \vec{p}, b')$

$(\mathcal{C}_4)$ $\mathsf{next}.g(\vec{e})$: $\{(f^+(\vec{p}), b, 2), (f^+(\vec{p}), g(\vec{e}), 2)\}$

$(\mathcal{C}_5)$ $\mathsf{r} := e.b'$: $\{(f^+(\vec{p}), b, 2), (f^+(\vec{p}), e, 1)\} \cup \mathcal{C}(f^+, \vec{p}, b')$

$(\mathcal{C}_6)$ $\mathsf{match\ r\ with}\ p_1 \Rightarrow b_1 \mid \cdots \mid p_n \Rightarrow b_n \mid [x] \Rightarrow g(\vec{e})$:
$\{(f^+(\vec{p}), b, 2), (f^+([x/\mathsf{r}]\vec{p}), g(\vec{e}), 2)\} \cup \mathcal{C}(f^+, ([p_1/\mathsf{r}]\vec{p}), b_1) \cup \ldots \cup \mathcal{C}(f^+, ([p_n/\mathsf{r}]\vec{p}), b_n)$

By inspecting the definitions, we can check that a control point $(f(\vec{p}), be, i)$ has the property that $Var(be) \subseteq Var(\vec{p})$. The read once condition is instrumental to this property. For instance, (i) in case $\mathcal{C}_2$, we know that if g can read some register r then r could not have been already read by f and (ii) in case $\mathcal{C}_6$, we know that the register r has not been already read by f. Hence, in these two cases, the register r must still occur in $\vec{p}$.

Example 4 *With reference to Example 3, we obtain the following control points:*

$(alarm^+(x, \mathsf{z}, \mathsf{sig}), \mathsf{ring} := \mathsf{prst.stop}, 2)$
$(alarm^+(x, \mathsf{z}, \mathsf{sig}), \mathsf{prst}, 1)$
$(alarm^+(x, \mathsf{z}, \mathsf{sig}), \mathsf{stop}, 2)$
$(alarm^+(x, \mathsf{s}(y), \mathsf{sig}), \mathsf{match}\ \ldots, 2)$
$(alarm^+(x, \mathsf{s}(y), \mathsf{prst}), \mathsf{next}.alarm(x,x), 2)$
$(alarm^+(x, \mathsf{s}(y), \mathsf{prst}), alarm(x,x), 2)$
$(alarm^+(x, \mathsf{s}(y), \_), alarm(x,y), 2)$

Definition 5 *An instance of a control point $(f(\vec{p}), b, i)$ is a behaviour $b' = \sigma b$, where σ is a substitution mapping the free variables in b to values.*

The property of being an instance of a control point is preserved by (behaviour and) system reduction. Thus the control points associated with a system do provide a representation of all reachable configurations. Indeed, in Appendix B we show that it is possible to define the reduction on pairs of control points and substitutions.

Proposition 6 *Suppose $(B, s, i) \to (B', s', i')$ and that for all thread indexes $j \in \mathbf{Z}_n$, $B_1(j)$ is an instance of a control point. Then for all $j \in \mathbf{Z}_n$, we have that $B'_1(j)$ is an instance of a control point.*

In order to prove the termination of the instant and to obtain a bound on the size of computed value, we associate order constraints to control points as follows:

Control point:	$(f(\vec{p}), e, 0)$,	$(f^+(\vec{p}), g(\vec{e}), 0)$,	$(f^+(\vec{p}), e, 1)$,	$(f^+(\vec{p}), be, 2)$
Associated constraint:	$f(\vec{p}) \succ_0 e$,	$f^+(\vec{p}) \succ_0 g^+(\vec{e}, \vec{\mathsf{r}}_g)$,	$f^+(\vec{p}) \succ_1 e$,	*no constraints*

A program will be deemed correct if the set of constraints obtained from all the function definitions can be satisfied in suitable structures. We say that a constraint $e \succ_i e'$ has index i. We rely on the constraints of index 0 to enforce termination of the instant and on those of index 0 or 1 to enforce a bound on the size of the computed values. Note that the constraints are on pure first order terms, a property that allows us to reuse techniques developed in the standard term rewriting framework.

Example 7 *With reference to the control points in Example 4, we obtain the constraint $alarm^+(x, \mathsf{z}, \mathsf{sig}) \succ_1 \mathsf{prst}$. We note that no constraints of index 0 are generated and so in this simple case the control flow analysis can already establish the termination of the thread and all is left to do is to check that the size of the data is under control, which will also be easily verified.*

4 Termination of the Instant

We recall that a *reduction order* $>$ over first-order terms is a well-founded order that is closed under context and substitution: $t > s$ implies $C[t] > C[s]$ and $\sigma t > \sigma s$, where C is any one hole context and σ is any substitution (see, *e.g*, [9]).

Definition 8 (termination condition) *We say that a system satisfies the termination condition if there is a reduction order $>$ such that all constraints of index 0 associated with the system hold in the reduction order.*

In this section, we assume that the system satisfies the termination condition. As expected this entails that the evaluation of closed expressions succeeds.

Proposition 9 *Let e be a closed expression. Then there is a value v such that $e \Downarrow v$ and $e \geq v$ with respect to the reduction order.*

Moreover, the following proposition states that a behaviour will always return the control to the scheduler.

Proposition 10 (progress) *Let b be an instance of a control point. Then for all stores s, $(b, s) \xrightarrow{X} (b', s')$.*

Finally, we show that at each instant the system will reach a configuration in which the scheduler detects the end of the instant and proceeds to the reinitialisation of the store and the status (as specified by rule (s_2) in Table 1).

Theorem 11 (termination of the instant) *All sequences of system reductions involving only rule (s_1) are finite.*

Proposition 10 and Theorem 11 are proven by exhibiting a suitable well-founded measure which is based both on the reduction order and the fact that the number of reads a thread may perform in an instant is finite.

Example 12 *We consider a recursive behaviour monitoring the register* i *(acting as a fifo channel) and parameterised on a number x representing the largest value read so far. At each instant, the behaviour reads the list l of values received on* i *and assigns to* o *the greatest number in x and l.*

$$f(x) \;=\; \mathsf{yield.match\ i\ with}\ l \Rightarrow f_1(maxl(l,x)) \qquad f_1(x) \;=\; \mathsf{o} := x.\mathsf{next}.f(x)$$

$$max(\mathsf{z}, y) = y\ , \qquad max(\mathsf{s}(x), \mathsf{z}) = \mathsf{s}(x)\ , \qquad max(\mathsf{s}(x), \mathsf{s}(y)) = \mathsf{s}(max(x,y))$$
$$maxl(\mathsf{nil}, x) = x\ , \qquad maxl(\mathsf{cons}(y,l), x) = maxl(l, max(y,x))$$

It is easy to prove the termination of the thread by recursive path ordering, where the function symbols are ordered as $f^+ > f_1^+ > maxl > max$, the arguments of $maxl$ are compared lexicographically from left to right, and the constructor symbols are incomparable and smaller than any function symbol.

5 Quasi-Interpretations

Our next task is to control the size of the values computed by the threads. To this end, we propose a suitable notion of quasi-interpretation.

Definition 13 (assignment) *Given a program, an* assignment *q associates with constructors and function symbols, functions over the non-negative reals $\mathbb{R}^+$ such that:*

(1) *If c is a constant then q_c is the constant 0,*

(2) *If c is a constructor with arity $n \geq 1$ then q_c is the function in $(\mathbb{R}^+)^n \to \mathbb{R}^+$ such that $q_\mathsf{c}(x_1, \ldots, x_n) = d + \Sigma_{i \in 1..n} x_i$, for some $d \geq 1$,*

(3) *if f is a function (identifier) with arity n then $q_f : (\mathbb{R}^+)^n \to \mathbb{R}^+$ is monotonic and for all $i \in 1..n$ we have $q_f(x_1, \ldots, x_n) \geq x_i$.*

An assignment q is extended to all expressions e as follows, giving a function expression q_e with variables in $Var(e)$:

$$q_x = x\ , \quad q_{\mathsf{c}(e_1,\ldots,e_n)} = q_\mathsf{c}(q_{e_1}, \ldots, q_{e_n})\ , \quad q_{f(e_1,\ldots,e_n)} = q_f(q_{e_1}, \ldots, q_{e_n})\ .$$

It is easy to check that for all values v, there exists a constant d depending on the quasi-interpretation such that: $|v| \leq q_v \leq d \cdot |v|$.

Definition 14 (quasi-interpretation) *An assignment is a* quasi-interpretation*, if for all constraints associated with the system of the shape $f(\vec{p}) \succ_i e$, with $i \in \{0,1\}$, the inequality $q_{f(\vec{p})} \geq q_e$ holds over the non-negative reals.*

Quasi-interpretations are designed so as to provide a bound on the size of the computed values as a function of the size of the input data. In the following, we assume given a suitable quasi-interpretation, q, for the system under investigation.

Example 15 *With reference to Examples 2 and 12, the following assignment is a quasi-interpretation (we give no quasi-interpretations for the function exp because it fails the read once condition):*

$$q_{\mathsf{nil}} = q_{\mathsf{z}} = 0\ , \qquad q_{\mathsf{s}}(x) = x + 1\ , \qquad q_{\mathsf{cons}}(x,l) = x + l + 1\ , \qquad q_{dble}(x) = 2 \cdot x\ ,$$
$$q_{f^+}(x,\mathsf{i}) = x + \mathsf{i} + 1\ , \qquad q_{f_1^+}(x) = x\ , \qquad q_{maxl}(x,y) = q_{max}(x,y) = max(x,y)\ .$$

One can show [2] that in the purely functional fragment of our language every value v computed during the evaluation of an expression $f(v_1,\ldots,v_n)$ satisfies the following condition:

$$|v| \ \leq\ q_v \ \leq\ q_{f(v_1,\ldots,v_n)} \ =\ q_f(q_{v_1},\ldots,q_{v_n}) \ \leq\ q_f(d|v_1|,\ldots,d|v_n|)\ . \tag{1}$$

We generalise this result to threads as follows.

Theorem 16 *Given a system of synchronous threads B, suppose that at the beginning of the instant $B_1(i) = f(\vec{v})$ for some thread index i. Then the size of the values computed by the thread i during an instant is bound by $q_{f^+(\vec{v},\vec{u})}$ where $\vec{u}$ are the values contained in the registers $\vec{\mathsf{r}}_f$ when they are read by the thread (or some constant value, otherwise).*

Theorem 16 is proven by showing that quasi-interpretations satisfy a suitable invariant. In the following corollary, we note that it is possible to express a bound on the size of the computed values which depends only on the size of the parameters at the beginning of the instant. This is possible because the number of reads a system may perform in an instant is bound by a constant.

Corollary 17 *Let B be a system with m registers and n threads. Suppose $B_1(i) = f_i(\vec{v}_i)$ for $i \in \mathbf{Z}_n$. Let c be a bound of the size of the largest parameter of the functions f_i and the largest default value of the registers. Suppose h is a function bounding all the quasi-interpretations, that is, for all the functions f_i^+ we have $h(x) \geq q_{f_i^+}(x,\ldots,x)$ over the non-negative reals. Then the size of the values computed by the system B during an instant is bound by $h^{n\cdot m+1}(c)$.*

Example 18 *The $n \cdot m$ iterations of the function h predicted by Corollary 17 correspond to a tight bound, as shown by the following example. We assume n threads and m registers (with default value z). The control of each thread is described as follows, where $writeall(e).b$ stands for the behaviour $\mathsf{r}_1 := e.\cdots.\mathsf{r}_m := e.b$:*

$$\begin{aligned} f(x_0) \ =\ & \mathsf{match}\ \mathsf{r}_1\ \mathsf{with}\ x_1 \Rightarrow writeall(dble(max(x_1,x_0))).\\ & \quad \mathsf{match}\ \mathsf{r}_2\ \mathsf{with}\ x_2 \Rightarrow writeall(dble(x_2)).\\ & \qquad \cdots\cdots\\ & \qquad \mathsf{match}\ \mathsf{r}_m\ \mathsf{with}\ x_m \Rightarrow writeall(dble(x_m)).\mathsf{next}.f(dble(x_m))\ . \end{aligned}$$

For this system we have $c \geq |x_0|$ and $h(x) = q_{dble}(x) = 2 \cdot x$. It is easy to show that, at the end of an instant, there have been $m \cdot n$ assignments to each register (m for every thread in the system) and that the value stored in each register is $dble^{m\cdot n}(x_0)$ of size $2^{m\cdot n} \cdot |x_0|$.

6 Combining Termination and Quasi-Interpretations

To bound the space needed for the execution of a system during an instant we also need to bound the number of nested recursive calls, *i.e.*, the number of frames that can be found on the stack (a precise definition of frame is given in the following Section 7). Unfortunately, quasi-interpretations provide a bound on the size of the frames but not on their number (at least not in a direct way). One way to cope with this problem is to combine quasi-interpretations with various families of reduction orders [20, 8]. In the following, we provide an example of this approach based on *recursive path orders* which is a widely used and fully mechanizable technique to prove termination [9].

Definition 19 *We say that a system terminates by LPO, if the reduction order associated with the system is a recursive path order where: (1) function symbols are compared lexicographically; (2) constructor symbols are always smaller than function symbols and two distinct constructor symbols are incomparable; (3) the arguments of constructor symbols are compared componentwise (product order).*

Definition 20 *We say that a system admits a polynomial quasi-interpretation if it has a quasi-interpretation where all functions are bound by a polynomial.*

Theorem 21 *If a system B terminates by LPO and admits a polynomial quasi-interpretation then the computation of the system in an instant runs in space polynomial in the size of the parameters of the threads at the beginning of the instant.*

The proof of Theorem 21 is based on Corollary 17 that provides a polynomial bound on the size of the computed values and on an analysis of nested calls in the LPO order that can be found in [8]. The point is that the depth of such nested calls is polynomial in the size of the values and that this allows to effectively compute a polynomial bounding the space necessary for the execution of the system. We stress that beyond proving that a system 'runs in PSPACE', we can extract a definite polynomial that bounds the size needed to run a system during an instant. For each thread (in a verified system) running the behaviour f with n parameters, we obtain a polynomial $q(x_1, \ldots, x_n)$ such that for all values $v_1, \ldots, v_n$ of the appropriate types, the size needed for the evaluation of the thread $[v_1/x_1, \ldots, v_n/x_n]b$ is bounded by $q(|v_1|, \ldots, |v_n|)$, where $|v|$ stands for the size of the value v.

Example 22 *With reference to Example 12, we can check that the order used there is indeed a LPO. From the quasi-interpretation in Example 15, we can deduce that the function $h(x)$ has the shape $a \cdot x + b$ (it is affine). More precisely, we can choose $h(x) = 2 \cdot x + 1$. In practice, many useful functions admit quasi-interpretations bound by an affine function such as the max-plus polynomials considered in [2]. Note that the parameter of the thread is the largest value received so far. Clearly, bounding the value of this parameter for arbitrary many instants requires a global analysis of the system.*

7 A Virtual Machine

We describe a simple virtual machine for our language thus providing a concrete intuition for the data structures required for the execution of the programs and the scheduler.

Our motivations for introducing a low-level model of execution for synchronous threads are twofold: (i) it offers a simple formal definition for the space needed for the execution of an instant (just take the maximal size of a machine configuration), and (ii) it explains some of the elaborate mechanisms occurring during the execution, like the selection of pattern matching clauses, the synchronisation with the match instruction and the detection of the end of an instant. A further motivation is the possibily to carry on the static analyses at bytecode level. The interest of this approach is now well understood [21, 22]. First, mobile code is shipped around in pre-compiled (or *bytecode*) form and needs to be analysed as such. Second, compilation is an error prone process and it seems safer to perform analyses at the level of the bytecode rather than at source level. In particular, we can reduce the size of the trusted code base and shift from the reliance on the correctness of the whole compilation chain to only the trust on the analyser.

Data structures. We suppose given the code for all the threads running in a system together with a set of types and *constructor names* and a disjoint set of *function names*. A function identifier f will also denote the sequence of instructions of the associated code: $f[i]$ stands for the i^{th} instruction in the (compiled) code of f and $|f|$ stands for the number of instructions.

The configuration of the machine is composed of a *store* s, that maps registers to their current values, a sequence of records describing the state of each threads in the system and three local registers owned by the scheduler: (1) t that stores the identity of the current thread, (2) time for the current time, and (3) wtime for the last time the store was modified. The notion of time here is of a logical nature: time passes whenever the scheduler transfers control to a new thread. Like in the source language, s_o denotes the store at the beginning of each instant.

The state of a thread t is a pair (st_t, M_t) where st_t is a *status* and M_t is the *memory* of the thread. A *memory* M is just a sequence of frames, and a *frame* is a triple (f, pc, ℓ) composed of a function identifier, the value of the program counter (a natural number in $1..|f|$), and a *stack* of values $\ell = v_1 \cdots v_k$. The status of a thread is defined as in the source language, except for the status W which is refined into $W(j, n)$ where: j is the address where to jump at the next instant if the thread does not resume in the current instant, and n is the time at which the thread is suspended.

Instructions. The set of *instructions* of the virtual machine are described in Table 2. All instructions operate on the frame of the current thread t and the memory M_t — the only instructions that depend on or affect the store are `read` and `write`. The *scheduler* executes the current instruction of the current thread t by running $run(t)$ which returns the information X associated with the instruction. According to this information the scheduler may take some action. We denote pc_t the program counter of the top frame (f, pc_t, ℓ) in M_t (if any) and $I_t = f[pc_t]$ the current instruction in the thread. Let us explain the role of the status $W(j, n)$ and of the registers time and wtime. We assume that a thread waiting for a condition to hold can check the condition without modifying the store. Then a thread waiting since time m may pass the condition only if the store has been modified at a time n with $m < n$. Otherwise, there is no point in passing the control to it[1]. With this data structure we also have a simple method to detect the end of an instant, it arises when no thread is in a running status and all waiting threads were interrupted after the last store modification occurred.

[1] Of course, this condition can be refined by recording the register on which the thread is waiting, the shape of the expected value,...

INSTRUCTIONS:

$f[pc]$	Current memory		Following memory
`load` i	$M \cdot (f, pc, v_1 \cdots v_i \cdot \ell)$	$\rightarrow$	$M \cdot (f, pc+1, v_1 \cdots v_i \cdot \ell \cdot v_i)$
`pop` i	$M \cdot (f, pc, \ell \cdot v_1 \cdots v_i)$	$\rightarrow$	$M \cdot (f, pc+1, \ell)$
`branch` c j	$M \cdot (f, pc, \ell \cdot \mathsf{c}(v_1, \ldots, v_n))$	$\rightarrow$	$M \cdot (f, pc+1, \ell \cdot v_1 \cdots v_n)$
`branch` c j	$M \cdot (f, pc, \ell \cdot \mathsf{d}(\ldots))$	$\rightarrow$	$M \cdot (f, j, \ell \cdot \mathsf{d}(\ldots))$ $\mathsf{c} \neq \mathsf{d}$
`build` c n	$M \cdot (f, pc, \ell \cdot v_1 \cdots v_n)$	$\rightarrow$	$M \cdot (f, pc+1, \ell \cdot \mathsf{c}(v_1, \ldots, v_n))$
`call` g n	$M \cdot (f, pc, \ell \cdot v_1 \cdots v_n)$	$\rightarrow$	$M \cdot (f, pc, \ell) \cdot (g, 1, v_1 \cdots v_n)$
`tcall` g n	$M \cdot (f, pc, \ell \cdot v_1 \cdots v_n)$	$\rightarrow$	$M \cdot (g, 1, v_1 \cdots v_n)$
`return`	$M \cdot (g, pc', \ell') \cdot (f, pc, \ell \cdot v)$	$\rightarrow$	$M \cdot (g, pc'+1, \ell' \cdot v)$
`read` r	$(M \cdot (f, pc, \ell), s)$	$\rightarrow$	$(M \cdot (f, pc+1, \ell \cdot s(\mathsf{r})), s)$
`write` r	$(M \cdot (f, pc, \ell \cdot v), s)$	$\rightarrow$	$(M \cdot (f, pc+1, \ell \cdot v), s[v/\mathsf{r}])$
`stop`	$M \cdot (f, pc, \ell)$	$\stackrel{S}{\rightarrow}$	ϵ
`yield`	$M \cdot (f, pc, \ell)$	$\stackrel{R}{\rightarrow}$	$M \cdot (f, pc+1, \ell)$
`next`	$M \cdot (f, pc, \ell)$	$\stackrel{N}{\rightarrow}$	$M \cdot (f, pc+1, \ell)$
`wait` j	$M \cdot (f, pc, \ell)$	$\stackrel{W}{\rightarrow}$	$M \cdot (f, pc+1, \ell)$

SCHEDULER:

```
for t in Z_n do { st_t := R; }
s := s_o; t := time := wtime := 0;
repeat forever {
  if I_t = (write _) then wtime := time;      /* retain that the store has been modified */
  X := run(t);                                /* run the current thread */
  case X of N, R, S, W : {
      if I_t = (wait j) then st_t := W(j, time); else st_t := X;
      t' := N(t, st);                         /* compute the index of the next active thread */
      if t' ∈ Z_n                             /* test whether all threads are blocked */
        then { t := t'; st_t := R; time := time + 1; }
      else {  s := s_o; wtime := time;        /* initialisation of the new instant */
              forall i in Z_n do {
                  if st_i = W(j, _) then pc_i := j;
                  if st_i ≠ S then st_i := R; } } }
  }
```

CONDITIONS ON $\mathcal{N}$:

If $\mathcal{N}(t, st) = k \in \mathbf{Z}_n$ then $st_k = R$ or ($st_k = W(j, n)$ and $n <$ wtime)
If $\mathcal{N}(t, st) \notin \mathbf{Z}_n$ then $\forall k \in \mathbf{Z}_n$ ($st_k \neq R$ and ($st_k = W(j, n)$ implies $n \geq$ wtime))

Table 2: Virtual machine

Elements of compilation. The language described in Section 2 admits a direct compilation to bytecode. Variables are replaced by offsets from the base of the stack frame and are accessed with the `load` instructions. Pattern matching is compiled into a nesting of `branch` instructions. The instruction `build` creates new values, the instruction `call` performs a function call, and the instruction `return` returns the control to the calling frame. The instruction `tcall`, for tail recursive calls, allows for a space efficient compilation of tail recursive definitions and the instruction `pop` allows to remove useless values from the top of the stack. The instructions `read`, `write`, `stop`, `yield`, `next` have a rather direct correspondence with the instructions of the source language.

Compilation of match. The instruction `wait` plays an important role in the implementation of pattern matching on registers. We give an intuition for it in the following example and explain how we compile a match instruction by looking at a portion of the bytecode obtained from the function *alarm* (see Example 3) corresponding to the behaviour match sig with prst $\Rightarrow$ next.$alarm(x, x)$ | $[\_] \Rightarrow alarm(x, y)$. We take `goto` i as a macro for `branch` x i where x is some fresh constructor symbol.

6 : `read` sig	read the value stored in sig and push its value on top of the stack,
7 : `branch` prst 12	test the value on top of the stack and jump to instruction 12 if it is different from prst, else go to instruction 8,
8 : `next`	start of the block obtained from the compilation of next.$alarm(x, x)$,
9 : `load` 1	copy the value in first position on the evaluation stack to the top,
10 : `load` 1	idem,
11 : `tcall` *alarm* 2	(tail recursive) call to *alarm*. The parameters of the call are the 2 values on top of the stack,
12 : `wait` 16	pause and wait for the thread to be run. If it is within the current instant skip to instruction 13, if the instant is finished, skip to instruction 16 (start of the block for $alarm(x, y)$)
13 : `pop` 1	discard one value on top of the stack (the old value of sig),
14 : `read` sig	read the current value of sig and push it on top of the evaluation stack,
15 : `goto` 7	jump at the beginning of the block for the match instruction.

At instruction 12 the thread is suspended. It will be waken up in the current instant if the memory has been modified since its suspension time and the computation resumes at instruction 13 — then the execution skips to instruction 7 so that the filter condition is checked again. Otherwise, the thread resumes computation in the following instant at instruction 16. In general, the act of suspending will require the execution of more than one instruction. More precisely, one has to pop all the elements allocated on the stack while checking the filter condition and load on top of the stack the current value of the register. This value is then available to the thread in case it resumes computation only at the next instant.

8 Conclusion

The execution of a thread in a cooperative synchronous model can be regarded as a sequence of instants. One can make each instant simple enough so that it can be described as a function — our experiments with writing sample programs show that the restrictions we impose do not hinder the expressivity of the language. Then well-known static analyses used to bound the resources needed for the execution of first-order functional programs can be extended to handle systems of synchronous cooperative threads. We believe this provides some evidence

for the relevance of these techniques in concurrent/embedded programming. We also expect that our approach can be extended to a richer programming model including, *e.g.*, references as first-class values, transactions-like primitives for error recovery, more elaborate mechanisms for preemption, ...

The static analyses we have considered do not try to analyse the whole system. On the contrary, they focus on each thread separately and can be carried out incrementally. On the basis of our previous work [1] and the virtual machine presented in Section 7, we expect that these analyses can be performed at bytecode level. These characteristics are particularly interesting in the framework of 'mobile code' where threads can enter or leave the system at the end of each instant as described in [4].

References

[1] R. Amadio, S. Coupet-Grimal, S. Dal-Zilio, and L. Jakubiec. A functional scenario for bytecode verification of resource bounds. Research Report LIF 17-2004, 2004.

[2] R. Amadio. Max-plus quasi-interpretations. In *Proc. Typed Lambda Calculi and Applications (TLCA '03)*, Springer LNCS 2701, 2003.

[3] J. Armstrong, R. Virding, C. Wikström, M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall 1996.

[4] G. Boudol, ULM, a core programming model for global computing. In *Proc. of ESOP*, Springer LNCS, 2004.

[5] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

[6] F. Boussinot and R. De Simone, The SL Synchronous Language. *IEEE Trans. on Software Engineering*, 22(4):256–266, 1996.

[7] G. Berry and G. Gonthier, The Esterel synchronous programming language. *Science of computer programming*, 19(2):87–152, 1992.

[8] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On termination methods with space bound certifications. In Proc. *Perspectives of System Informatics*, Springer LNCS, 2001.

[9] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[10] J. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model.* PhD thesis, University of Berkeley, 1993.

[11] N. Carriero and D. Gelernter. Linda in Context. *Commun. ACM*, 32(4): 444-458, 1989.

[12] P. Caspi. Clocks in data flow languages. *Theoretical Computer Science*, 94:125–140, 1992.

[13] A. Cobham. The intrinsic computational difficulty of functions. In *Proc. Logic, Methodology, and Philosophy of Science II, North Holland*, 1965.

[14] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *Proc. ACM Conf. on Functional Programming*, 1996.

[15] M. Hofmann. The strength of non size-increasing computation. In *Proc. POPL*, ACM Press, 2002.

[16] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. IFIP Congress, North-Holland*, 1974.

[17] N. Jones. *Computability and complexity, from a programming perspective.* MIT-Press, 1997.

[18] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, 1:24–35, 1987.

[19] D. Leivant. Predicative recurrence and computational complexity i: word recurrence and poly-time. *Feasible mathematics II, Clote and Remmel (eds.)*, Birkhäuser:320–343, 1994.

[20] J.-Y. Marion. *Complexité implicite des calculs, de la théorie à la pratique.* Université de Nancy, 2000. Habilitation à diriger des recherches.

[21] G. Morriset, D. Walker, K. Crary and N. Glew. From system F to typed assembly language. In *ACM Transactions on Programming Languages and Systems*, 21(3):528-569, 1999.

[22] G. Necula. Proof carrying code. In *Proc. POPL*, ACM Press, 1997.

[23] M. Odersky. Functional nets. In *Proc. ESOP*, Springer Lecture Notes in Comp. Sci. 1782, 2000.

[24] J. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at the 1996 USENIX Technical Conference.

[25] Th. Park. *Bounded scheduling of process networks.* PhD thesis, University of Berkeley, 1995.

[26] Reactive Programming, INRIA, Mimosa Project. `http://www-sop.inria.fr/mimosa/rp`.

A Examples

We illustrate how certain synchronous and/or concurrent programming paradigms can be represented in our model.

A.1 Kahn and Lustre Networks

In Kahn networks [16] sequential threads communicate through unbounded, first-in-first-out, point-to-point channels. A thread can always send a message on a channel but reception will block until there is a message actually present in the channel. Also, it is not possible to test whether a channel is empty.

These *fifo channels* can be simulated as shown in Example 1(2). Since there is no notion of instant in Kahn networks, it is possible to rely just on the cooperative fragment (cf. section 2) of our model and to perform the whole simulation during a single instant.

In the general Kahn model there is no bound on the number of messages that can be written in a fifo channel nor on the size of the messages. Much effort has been put into the *static scheduling* of Kahn networks (see, *e.g.*, [18, 12, 14]). This analysis can be regarded as a form of resource control since it guarantees that the number of messages in fifo channels is bounded (but says nothing about their size). The static scheduling of Kahn network is also motivated by performance issues, since it eliminates the need to schedule threads at run time.

In the following, we look in some detail at the programming language LUSTRE, that can be regarded as a language for programming Kahn networks that can be executed *synchronously*. A LUSTRE network is composed of four types of nodes: the combinatorial node $C(f)$, the delay node $y \rightarrow \_$, the when node *When*, and the merge node *Merge*. Each node c may have several input streams and one output stream. In the following, we use $c(s_1, \ldots, s_n)$ to denote the stream produced by the node c with input streams $s_1, \ldots, s_n$. As usual, one assumes that streams are well-typed — all the elements in a stream have the same type — we use ϵ for the empty stream and $v : s$ to denote a stream with head v and tail s.

The functional behaviour of each type of node is defined by a set of recursive definitions as follows.:

- The node $C(f)$ has n input streams $s_1, \ldots, s_n$, where n is the arity of the function f, and is used to combine the values of each stream. We have $C(f)(s_1, \ldots, s_n) = f(v_1, \ldots, v_n) : C(f)(s'_1, \ldots, s'_n)$ if $s_i = v_i : s'_i$ for all $i \in 1..n$ and $C(f)(s_1, \ldots, s_n) = \epsilon$ otherwise (that is, if one of the streams s_i is ϵ),

- the node $y \rightarrow \_$ has one input stream s and is used to shift the values in s by one position — the value in the first instant is y. We have $y \rightarrow s = y : s$,

- the node *When* has one boolean input stream b — with values of type $bool = \mathsf{false} \mid \mathsf{true}$ — and one input stream of values s. A *When* node is used to output values from s whenever b is true, that is, $When(\mathsf{false} : b, x : s) = When(b, s)$ and $When(\mathsf{true} : b, x : s) = x : When(b, s)$, and $When(b, s) = \epsilon$ otherwise,

- the node *Merge* has one boolean input stream b and two input streams of values s_1, s_2. This node is used to multiplex the values in s_1 and s_2 according to the values in b. We have $Merge(\mathsf{false} : b, s_1, x : s_2) = Merge(\mathsf{true} : b, x : s_1, s_2) = x : Merge(b, s_1, s_2)$, and $Merge(b, s_1, s_2) = \epsilon$ otherwise.

Example 23 *An example of* Lustre *network is given by the following set of equations such that half* $= \mathsf{true} : \mathsf{false} : \mathsf{true} : \ldots$ *is true half of the time,* one *is the constant stream* $1 : 1 : \ldots$, int *is the stream* $1 : 2 : 3 : \ldots$ *of all the natural numbers and* over *is of the form* $2 : 5 : 8 : \ldots$ *— it contains all the integers of the form* $(3 \cdot n + 2)$.

$$half = \mathsf{true} \to C(not)(half) \qquad \text{such that:} \quad not(\mathsf{true}) = \mathsf{false}\ ,\ not(\mathsf{false}) = \mathsf{true}$$

$$one = 1 \to one \qquad int = C(+)(one,\ 0 \to int) \qquad over = C(+)(int, When(half, int))$$

An interesting point about the computation of the stream over *is that we need to store* n *values of* int *in order to compute the* $2 \cdot n^{th}$ *value of* over. *More generally, a* When *node consumes values more often from its input stream than it produces values on its output. On the contrary, a* Merge *node produces values more often on its output than it consumes values on each of its input streams. Therefore it is not possible to run the network within a bounded memory.*

Lustre comes equipped with a (static) *clock analysis* that makes sure that the network can be run in a cyclic fashion with buffers of size 0 — this analysis is mainly concerned with programs manipulating boolean streams, not complex structured values. Next, we describe a possible simulation of such a well-clocked network in our model. In this encoding, nodes communicate on one place channels as described in Example 1(1), *e.g.* a node outputting a boolean signal corresponds to a register of type $ch(bool)$. We use one register c and one thread running the behaviour b_c for each node c in the network. A cycle of the Lustre scheduler corresponds to the execution of an instant in our model.

- If c is the node $C(f)$ connected to the output of the nodes $c_1, \ldots, c_n$ then:

$$\begin{aligned} b_c() = \quad & \mathsf{match}\ \mathsf{c}_1\ \mathsf{with}\ \mathsf{full}(x_1) \Rightarrow \\ & \ldots\ldots \\ & \quad \mathsf{match}\ \mathsf{c}_n\ \mathsf{with}\ \mathsf{full}(x_n) \Rightarrow \mathsf{c} := f(x_1, \ldots, x_n).\mathsf{next}.b_c() \\ & \qquad\qquad \mid\ [\_] \Rightarrow b_c() \ldots\ldots [\_] \Rightarrow b_c() \end{aligned}$$

- if c is the node $y \to c'$ connected to the output of the nodes c' then — we use an auxiliary function $delay(y)$ to store the delayed value read from c':

$$b_c() = delay(y) \qquad \text{where} \quad delay(y) = \mathsf{c} := y.\mathsf{match}\ \mathsf{c}'\ \mathsf{with}\ \mathsf{full}(x) \Rightarrow \mathsf{next}.delay(x)$$

- if c is the node *When* connected to the output of the nodes b, c' then:

$$\begin{aligned} b_c() = \ \mathsf{match}\ \mathsf{b}\ \ \mathsf{with}\ \ & \mathsf{full}(\mathsf{true}) \Rightarrow \\ & \quad \big(\mathsf{match}\ \mathsf{c}'\ \mathsf{with}\ \mathsf{full}(x) \Rightarrow \mathsf{c} := x.\mathsf{next}.b_c() \mid\ [\_] \Rightarrow b_c()\big) \\ \mid\ \ & \mathsf{full}(\mathsf{false}) \Rightarrow \\ & \quad \big(\mathsf{match}\ \mathsf{c}'\ \mathsf{with}\ \mathsf{full}(x) \Rightarrow \mathsf{next}.b_c() \mid\ [\_] \Rightarrow b_c()\big) \end{aligned}$$

- if c is the node *Merge* connected to the output of the nodes b, c_1, c_2 then:

$$\begin{aligned} b_c() = \ \mathsf{match}\ \mathsf{b}\ \ \mathsf{with}\ \ & \mathsf{full}(\mathsf{true}) \Rightarrow \\ & \quad \big(\mathsf{match}\ \mathsf{c}_1\ \mathsf{with}\ \mathsf{full}(x) \Rightarrow \mathsf{c} := x.\mathsf{next}.b_c() \mid\ [\_] \Rightarrow b_c()\big) \\ \mid\ \ & \mathsf{full}(\mathsf{false}) \Rightarrow \\ & \quad \big(\mathsf{match}\ \mathsf{c}_2\ \mathsf{with}\ \mathsf{full}(x) \Rightarrow \mathsf{next}.b_c() \mid\ [\_] \Rightarrow b_c()\big) \end{aligned}$$

Of course, there is a trade-off between static scheduling and programming flexibility. As a rule of thumb, static scheduling techniques work well when the input-output behaviour of the threads is regular and predictable (*e.g.* in signal processing or scientific applications). In a general programming context, some degree of dynamic scheduling seems inevitable [10, 25].

A.2 A Parallel Filter

In our model, a thread cannot filter the contents of two registers in parallel. However, two or more parallel threads can cooperate to achieve a similar effect. An archetypical example of such filter is the *parallel-or* function. Let $\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}$ be one place channels storing boolean values, as defined in Example 1(1), we want to implement a behaviour that reads the signals s_1 and/or s_2 at each instant and assigns the value true to s in the next instant if either s_1 or s_2 are true, and false if both s_1 and s_2 are false.

We can implement this filter using two parallel behaviours t_1, t_2 defined below (we use the notation $\mathsf{next.s} := \mathsf{full}(\mathsf{true}).t()$ to avoid the definition of an unnecessary intermediate function for the continuation of the next instruction):

$$
\begin{array}{lll}
t_1() = & \mathsf{match}\ \mathsf{s}_1\ \mathsf{with} & \\
 & \quad \mathsf{full}(\mathsf{true}) \Rightarrow \mathsf{next.s} := \mathsf{full}(\mathsf{true}).t_1() & \\
 & \mid \quad \mathsf{full}(\mathsf{false}) \Rightarrow \mathsf{match}\ \mathsf{s}_2\ \mathsf{with} & \\
 & \qquad \mathsf{full}(\mathsf{false}) \Rightarrow \mathsf{next.s} := \mathsf{full}(\mathsf{false}).t_1() & \\
 & \qquad \mid \quad [\_] \Rightarrow t_1() &
\end{array}
$$

$$
\begin{array}{lll}
t_2() = & \mathsf{match}\ \mathsf{s}_2\ \mathsf{with} & \\
 & \quad \mathsf{full}(\mathsf{true}) & \Rightarrow \mathsf{next.s} := \mathsf{full}(\mathsf{true}).t_2() \\
 & \mid \quad [\_] & \Rightarrow t_2()
\end{array}
$$

A.3 Readers-Writers and Other Synchronisation Patterns

A simple, maybe the simplest, example of synchronisation and resource protection is the single place buffer. The buffer (initially empty) is implemented by a thread listening to two signals. The first on the register put to fill the buffer with a value if it is empty, the second on the register get to emit the value stored in the buffer by writing it in the special register result and flush the buffer. In this encoding, the register put is a one place channel and get is a signal as in Example 1. Moreover, owing to the read once condition, we are not able to react to several put/get requests during the same instant — only if the buffer is full can we process one get and one put request in the same instant. Note that the value of the buffer is stored on the function call to $\mathit{full}(v)$, hence we use function parameters as a kind of private memory (to compare with registers that model shared memory).

$$
\begin{array}{lll}
\mathit{empty}() = & \mathsf{match}\ \mathsf{put}\ \mathsf{with} & \\
 & \quad \mathsf{full}(x) \Rightarrow \mathsf{next}.\mathit{full}(x) &
\end{array}
$$

$$
\begin{array}{lll}
\mathit{full}(x) = & \mathsf{match}\ \mathsf{get}\ \mathsf{with} & \\
 & \quad \mathsf{prst} \Rightarrow \mathsf{result} := x.\mathsf{yield}.\mathit{empty}() &
\end{array}
$$

Another common example of synchronisation pattern is a situation where we need to protect a resource that may be accessed both by 'readers' (which access the resource without modifying it) and 'writers' (which can access and modify the resource). This form of access control is common in databases and can be implemented using traditional synchronisation mechanisms such as semaphores, but this implementation is far from trivial [23].

In our encoding, a control thread secures the access to the protected resource. The other threads, which may be distinguished by their identity id (a natural number), may initiate a request to access / release the resource by sending a special value on the dedicated register

req. The thread regulating the resource may acknowledge at most one request per instant and allows the sender of a request to proceed by writing its id on the register allow at the next instant. The synchronisation constraints are as follows: there can be multiple concurrent readers, there can be only one writer at any one time, pending write requests have priority over pending read requests (but do not preempt ongoing read operations).

The thread protecting the resource is initialized with the behaviour $onlyreader(\mathsf{z})$, below, meaning the system has no pending requests for reading or writing. We define a new abstract type for assigning requests $request = \mathsf{startRead}(nat) \mid \mathsf{startWrite}(nat) \mid \mathsf{endRead} \mid \mathsf{endWrite} \mid \mathsf{none}$. The value $\mathsf{startRead}(id)$ indicates a read request from the thread id, the other constructors correspond to requests for starting to write, ending to read or ending to write — the value none stands for no requests. A startRead operation requires that there are no pending writes to proceed. In that case we increment the number of ongoing readers and allow the caller proceed. By contrast, a startWrite puts the monitor thread in a state waiting to process the pending write request (function $pwrite$), which waits for the number of readers to be null and then allows the thread that made the pending write request to proceed. A thread that is willing to read / write must send its request at each new instant until it is allowed to proceed. An endRead and endWrite request is always immediately acknowledged.

The behaviour $onlyreader(x)$ encodes the state of the controller when there is no pending write and x readers. In a state with x pending readers, when a startWrite request from the thread id is received, the controller thread switches to the behaviour $pwrite(id, x)$, meaning that the thread id is waiting to write and that we should wait for x endWrite requests before acknowledging the request to write.

$$
\begin{array}{lcl}
onlyreader(\mathsf{z}) & = & \mathsf{match\ req\ with} \\
 & & \quad \mathsf{startWrite}(y) \Rightarrow \mathsf{next.allow} := y.pwrite(y, \mathsf{z}) \\
 & & \mid \quad \mathsf{startRead}(y) \Rightarrow \mathsf{next.allow} := y.onlyreader(\mathsf{s}(\mathsf{z})) \\
onlyreader(\mathsf{s}(x)) & = & \mathsf{match\ req\ with} \\
 & & \quad \mathsf{endRead}(y) \Rightarrow \mathsf{next}.onlyreader(x) \\
 & & \mid \quad \mathsf{startWrite}(y) \Rightarrow \mathsf{next}.pwrite(y, \mathsf{s}(x)) \\
 & & \mid \quad \mathsf{startRead}(y) \Rightarrow \mathsf{next.allow} := y.onlyreader(\mathsf{s}(\mathsf{s}(\mathsf{z}))) \\
pwrite(id, \mathsf{z}) & = & \mathsf{match\ req\ with\ endWrite}(y) \Rightarrow \mathsf{next}.onlyreader(\mathsf{z}) \\
pwrite(id, \mathsf{s}(\mathsf{z})) & = & \mathsf{match\ req\ with\ endRead}(y) \Rightarrow \mathsf{next.allow} := id.pwrite(id, \mathsf{z}) \\
pwrite(id, \mathsf{s}(\mathsf{s}(x))) & = & \mathsf{match\ req\ with\ endRead}(y) \Rightarrow \mathsf{next}.pwrite(id, \mathsf{s}(x))
\end{array}
$$

A thread willing to read on the protected resource should repeatedly try to send its request on the register req then poll the register allow, *e.g.* with the behaviour $\mathsf{askRead}(id).\mathsf{match\ allow\ with}\ id \Rightarrow \cdots$ where $\mathsf{askRead}(id) =_{\mathrm{def}} \mathsf{match\ req\ with\ none} \Rightarrow \mathsf{req} := \mathsf{startRead}(id)$. The code for a thread willing to end a read session is similar. It is simple to change our encoding so that multiple requests are stored in a fifo queue instead of a one place buffer.

B Proofs

B.1 Preservation of Control Points Instances

Proposition 6 Suppose $(B, s, i) \rightarrow (B', s', i')$ and that for all thread indexes $j \in \mathbf{Z}_n$, $B_1(j)$ is an instance of a control point. Then for all $j \in \mathbf{Z}_n$, we have that $B'_1(j)$ is an instance of a

control point.

We note that $(f^+(\vec{p}), b, i) \in \mathcal{C}(f^+, \vec{p}, b)$, for some $i \in \{0, 1, 2\}$. We start by proving the preservation of instances for behaviour reduction, that is, if b is an instance and $(b, s) \xrightarrow{X} (b', s')$ then b' is an instance.

Proof. We prove this auxiliary property by induction on the derivation of $(b, s) \xrightarrow{X} (b', s')$. We proceed by case analysis on the last reduction rule used.

(b_1) We can use the same control point.

(b_2) Suppose $\mathsf{yield}.b$ is an instance of the control point $(f^+(\vec{p}), \mathsf{yield}.b_0, i)$. This control point must be generated by rule $(\mathcal{C}_3)$ and then $(f^+(\vec{p}), b_0, j)$ is also a symbolic control point and b is one of its instances.

(b_3) We proceed as in the previous case, applying $(\mathcal{C}_4)$.

(b_4) We can use the same control point.

(b_5) Suppose $\mathsf{match\ r\ with} \ldots p \Rightarrow b \ldots$ is an instance of $(f^+(\vec{p}), \mathsf{match\ r\ with} \ldots p \Rightarrow b_0 \ldots, j)$. By $(\mathcal{C}_6)$, $(f^+([p/\mathsf{r}]\vec{p}), b_0, j')$ is a control point and σb is one of its instances if $\sigma p = s(\mathsf{r})$. We conclude by applying the inductive hypothesis on σb.

(b_6) Suppose $f(\vec{e})$ is an instance of $(g^+(\vec{q}), f(\vec{e}_0), i)$. If $\vec{e} \Downarrow \vec{v}$, $f(\vec{p}) = b$ and $\sigma\vec{p} = \vec{v}$, then $(f^+(\vec{p}, \vec{\mathsf{r}}_f), b, i)$ is a control point by definition of $\mathcal{C}$ and σb is one of its instances. We conclude by applying the inductive hypothesis on σb.

(b_7) Suppose $\mathsf{r} := e.b$ is an instance of $(f^+(\vec{p}), \mathsf{r} := e_0.b_0, j)$, then by $(\mathcal{C}_5)$ $(f^+(\vec{p}), b_0, j')$ is a control point and b one of its instances. □

Next we show the main property, namely that being an instance is preserved at the level of system reduction. We proceed by case analysis on the last reduction rule used in the derivation of $(B, s, i) \rightarrow (B', s', i')$.

Proof. (s_1) One of the threads performs one step. The property follows by the analysis carried on above.

(s_2) One of the threads performs one step. Moreover, the threads in waiting status take the $[x] \Rightarrow g(\vec{e})$ branch of the match instructions that were blocking. A thread $\mathsf{match\ r}\ \ldots \mid [x] \Rightarrow g(\vec{e})$ in waiting status is an instance of a control point $(f^+(\vec{p}), \mathsf{match\ r}\ \ldots \mid [x] \Rightarrow g(\vec{e}_0), j)$. By $(\mathcal{C}_6)$, $(f^+([x/\mathsf{r}]\vec{p}), g(\vec{e}_0), 2)$ is a control point, and for any value v, $[v/x]g(\vec{e})$ is one of its instances. □

Our analysis has shown that if b is an instance of a control point and $b \xrightarrow{X} b'$ then we can *effectively* build a control point of which b' is an instance. We can make this remark precise by redefining the behaviour reduction on quadruples $(f^+(\vec{p}), b, \sigma, s)$, where for some i, $(f^+(\vec{p}), b, i)$ is a control point, σ is a substitution mapping the variables in b to values, and s is a store. The revised system for behaviour reduction is described in table 3.

$$(\mathsf{b}_1)\quad \frac{}{(f^+(\vec{p}), \mathtt{stop}, \sigma, s) \xrightarrow{S} (f^+(\vec{p}), \mathtt{stop}, \sigma, s)}$$

$$(\mathsf{b}_2)\quad \frac{}{(f^+(\vec{p}), \mathsf{yield}.b, \sigma, s) \xrightarrow{R} (f^+(\vec{p}), b, \sigma, s)}$$

$$(\mathsf{b}_3)\quad \frac{}{(f^+(\vec{p}), \mathsf{next}.g(\vec{e}), \sigma, s) \xrightarrow{N} (f^+(\vec{p}), g(\vec{e}), \sigma, s)}$$

$$(\mathsf{b}_4)\quad \frac{\text{no pattern matches } s(\mathsf{r})}{(f^+(\vec{p}), \mathsf{match\ r\ with} \ldots, \sigma, s) \xrightarrow{W} (f^+(\vec{p}), \mathsf{match\ r\ with} \ldots, \sigma, s)}$$

$$(\mathsf{b}_5)\quad \frac{\sigma_1 p = s(\mathsf{r}), \quad (f^+([p/\mathsf{r}]\vec{p}), b, \sigma_1 \circ \sigma, s) \xrightarrow{X} (f_1^+(\vec{p'}), b', \sigma', s')}{(f^+(\vec{p}), \mathsf{match\ r\ with} \cdots \mid p \Rightarrow b \mid \ldots, \sigma, s) \xrightarrow{X} (f_1^+(\vec{p'}), b', \sigma', s')}$$

$$(\mathsf{b}_6)\quad \frac{\begin{array}{c}\sigma\vec{e} \Downarrow \vec{v}, \quad g(\vec{q}) = b, \quad \sigma_1\vec{q} = \vec{v}, \\ (g^+(\vec{q}, \vec{\mathsf{r}}_g), b, \sigma_1, s) \xrightarrow{X} (f_1^+(\vec{p'}), b', \sigma', s')\end{array}}{(f^+(\vec{p}), g(\vec{e}), \sigma, s) \xrightarrow{X} (f_1^+(\vec{p'}), b', \sigma', s')}$$

$$(\mathsf{b}_7)\quad \frac{\sigma e \Downarrow v, \quad (f^+(\vec{p}), b, \sigma, s[v/\mathsf{r}]) \xrightarrow{X} (f_1^+(\vec{p'}), b', \sigma', s')}{(f^+(\vec{p}), \mathsf{r} := e.b, \sigma, s) \xrightarrow{X} (f_1^+(\vec{p'}), b', \sigma', s')}$$

Table 3: Behaviour reduction revisited

B.2 Evaluation of Closed Expressions

Proposition 9 Let e be a closed expression. Then there is a value v such that $e \Downarrow v$ and $e \geq v$ with respect to the reduction order.

Proof. By induction on the structure of e and the reduction order $>$.

case $e \equiv \mathsf{c}(e_1, \ldots, e_n)$ If the arity of c is 0 then $\mathsf{c} \Downarrow \mathsf{c}$ and $\mathsf{c} \geq \mathsf{c}$. If $n > 0$ then by inductive hypothesis $e_i \Downarrow v_i$ and $e_i \geq v_i$ for $i = 1, \ldots, n$. Thus $\mathsf{c}(e_1, \ldots, e_n) \Downarrow \mathsf{c}(v_1, \ldots, v_n)$. Moreover, since $>$ is closed under context and transitivity, it follows that $\mathsf{c}(e_1, \ldots, e_n) \geq \mathsf{c}(v_1, \ldots, v_n)$.

case $e \equiv f(e_1, \ldots, e_n)$ By inductive hypothesis, $e_i \Downarrow v_i$ and $e_i \geq v_i$ for $i = 1, \ldots, n$. This implies $f(\vec{e}) \geq f(\vec{v})$. Since the patterns used in function definitions are total, there is a rule $f(\vec{p}) = e$ and a substitution σ such that $\sigma\vec{p} = \vec{v}$. We know that $f(\vec{p}) > e$, thus $f(\sigma\vec{p}) = f(\vec{v}) > \sigma e$. By inductive hypothesis, $\sigma e \Downarrow v$ and $\sigma e \geq v$. Thus $f(\vec{e}) \Downarrow v$ and $f(\vec{e}) \geq f(\vec{v}) > \sigma e \geq v$. □

B.3 Progress

Proposition 10 Let b be an instance of a control point. Then for all stores s, there exists a store s' and a status X such that $(b, s) \xrightarrow{X} (b', s')$.

Proof. We restrict our attention to behaviours which are instances of control point. We have seen in proposition 6 that if b is an instance of a control point and $(b, s) \xrightarrow{X} (b', s')$ then we can effectively determine a control point of which b' is an instance. Thus we can think of the behaviour reduction as acting on both a behaviour and a control point as formalised in Table 3.

We start by defining a suitable well-founded order. If b is a behaviour, then let $nr(b)$ be the maximum number of reads that b may perform in an instant:

$$nr(b) = max\{\, |w| \mid w \in R(b) \,\}$$

Moreover, let $ln(b)$ be the *length* of b inductively defined as follows:

$$ln(\mathtt{stop}) = ln(f(\vec{e})) = 0 \quad ln(\mathsf{yield}.b) = ln(\mathsf{r} := e.b) = 1 + ln(b) \quad ln(\mathsf{next}.f(\vec{e})) = 2$$
$$ln(\mathsf{match\ r\ with} \ldots \mid p_i \Rightarrow b_i \mid \ldots \mid [x] \Rightarrow f(\vec{e})) = 1 + max(\ldots, ln(b_i), \ldots)$$

If the behaviour b is an instance of the control point $\gamma \equiv (f^+(\vec{p}), b_0, i)$ via a substitution σ then we associate to the pair (b, γ) a measure

$$\mu(b, \gamma) = (nr(b), f^+(\sigma\vec{p}), ln(b))$$

We assume that measures are lexicographically ordered from left to right, where the order on the first and third component is the standard one on natural numbers and the order on the second component is the one given by the reduction order considered in the termination condition. This is a well-founded order.

Now we show the assertion by induction on $\mu(b, \gamma)$. We proceed by case analysis on the structure of b.

stop Apply (b_1). Here $X = S$ and the measure stays the same.

$\mathsf{yield}.b'$ Apply (b_2). Here $X = R$ and the measure decreases because $ln(b)$ decreases.

$\mathsf{next}.b'$ Apply (b_3). Here $X = N$ and the measure decreases because $ln(b)$ decreases.

$\mathsf{match}\ \ldots$ If no pattern matches then apply (b_4). The measure is unchanged. If a pattern matches then apply (b_5). The measure decreases because $nr(b)$ decreases and then the induction hypothesis applies.

$g(\vec{e})$ Suppose $g(\vec{e})$ is an instance of the control point $(f^+(\vec{p}), g(\vec{e}_0), 0)$ via the substitution σ. Note that $\vec{e} = \sigma\vec{e}_0$. By proposition 9, we know that $\vec{e} \Downarrow \vec{v}$ and $\vec{e} \geq \vec{v}$ in the reduction order. We also know that for some $\vec{q}, b$, $\sigma'\vec{q} = \vec{v}$ and $g(\vec{q}) = b$. Finally, the constraint associated with the control point requires $f^+(\vec{p}) > g^+(\vec{e}_0, \vec{\mathsf{r}}_g)$. Then using the properties of reduction orders we observe:

$$\begin{array}{lll} f^+(\sigma\vec{p}) & > g^+(\sigma\vec{e}_0, \vec{\mathsf{r}}_g) & = g^+(\vec{e}, \vec{\mathsf{r}}_g) \\ & \geq g^+(\vec{v}, \vec{\mathsf{r}}_g) & = g^+(\sigma'\vec{q}, \vec{\mathsf{r}}_g)\ . \end{array}$$

Thus the measure decreases because $f^+(\sigma\vec{p}) > g^+(\sigma'\vec{q}, \vec{\mathsf{r}}_g)$, and then the induction hypothesis applies.

$r := e.b'$ By proposition 9, $e \Downarrow v$. Apply (b_7). The measure decreases because $ln(b)$ decreases, and then the induction hypothesis applies. □

Remark 24 *We point out that in the proof of proposition 10, if $X = R$ then the measure decreases and if $X \in \{N, S, W\}$ then the measure decreases or stays the same. This will be used in the following proof of Theorem 11.*

B.4 Termination of the Instant

Theorem 11 All sequences of system reductions involving only rule (s_1) are finite.

Proof. We order the status of threads as follows: $R > N, S, W$. With a behaviour $B_1(i)$ coming with a control point γ_i, we associate the pair $\mu'(i) = (\mu(B_1(i), \gamma_i), B_2(i))$ where μ is the measure in the proof of Proposition 10. Thus $\mu'(i)$ can be regarded as a quadruple with a lexicographic order from left to right. With a system B of n threads (threads indexes are in $\mathbf{Z}_n$ we associate the tuple $(\mu'(0), \ldots, \mu'(n-1))$. On such tuples, we consider the product order. We prove that every system reduction sequence involving only rule (s_1) terminates by induction on this product order. We recall the rule below:

$$\frac{(B_1(i), s) \xrightarrow{X} (b', s'), \quad B_2(i) = R, \quad B' = B[(b', X)/i], \quad \mathcal{N}(B', s', i) = k}{(B, s, i) \rightarrow (B'[(B'_1(k), R)/k], s', k)}$$

Let $B'' = B'[(B'_1(k), R)/k]$. We proceed by case analysis on X and $B'_2(k)$.

We remark that if $B'_2(k) = W$ then the conditions on the scheduler tell us that $i \neq k$ and that the thread k resuming the computation can read at least one register without being suspended. Then, if the measure on i is stable or decreasing, the induction hypothesis can be applied to the continuation of the read operation, and from this, we can conclude that every reduction sequence from (B'', s', k) terminates. We proceed by case analysis on the status X in the reduction $(B_1(i), s) \xrightarrow{X} (b', s')$.

S Then $\mu'(i)$ decreases by Remark 24. If $B'_2(k) = R$ then $\mu'(k)$ is unchanged. If $B'_2(k) = W$ then the argument above applies.

N Then $\mu'(i)$ decreases by Remark 24. If $B'_2(k) = R$ then $\mu'(k)$ is unchanged. If $B'_2(k) = W$ then the argument above applies.

W Then $\mu'(i)$ decreases by Remark 24. If $B'_2(k) = R$ then $\mu'(k)$ is unchanged. If $B'_2(k) = W$ then the argument above applies.

R Then $\mu'(i)$ decreases by Remark 24. If $B'_2(k) = R$ then $\mu'(k)$ is unchanged. If $B'_2(k) = W$ then the argument above applies. □

B.5 Bounding the Size of Values for Threads

Theorem 16 Given a system of synchronous threads B, suppose that at the beginning of the instant $B_1(i) = f(\vec{v})$ for some thread index i. Then the size of the values computed by the thread i during an instant is bound by $q_{f^+(\vec{v},\vec{u})}$ where $\vec{u}$ are the values contained in the registers $\vec{\mathsf{r}}_f$ when they are read by the thread (or some constant value, otherwise).

In Tables 1 and 3, we define the reduction of behaviours as a *big step* semantics. We reformulate the rules in Table 3, following a *small step* approach:

(b'_2) $(f^+(\vec{p}), \mathsf{yield}.b, \sigma, s) \to (f^+(\vec{p}), b, \sigma, s)$

(b'_5) $(f^+(\vec{p}), \mathsf{match\ r\ with} \cdots \mid p \Rightarrow b \mid \ldots, \sigma, s) \to (f^+([p/\mathsf{r}]\vec{p}), b, \sigma_1 \circ \sigma, s)$ if $\sigma_1 p = s(\mathsf{r})$

(b'_6) $(f^+(\vec{p}), g(\vec{e}), \sigma, s) \to (g^+(\vec{q}, \vec{\mathsf{r}}_g), b, \sigma_1, s)$ if $\sigma\vec{e} \Downarrow \vec{v},\ g(\vec{q}) = b,\ \sigma_1\vec{q} = \vec{v}$

(b'_7) $(f^+(\vec{p}), \mathsf{r} := e.b, \sigma, s) \to (f^+(\vec{p}), b, \sigma, s[v/\mathsf{r}])$ if $\sigma e \Downarrow v$

Note that there are no rules corresponding to (b_1), (b_3), (b_4) since these rules either terminate or suspend the computation of the thread in the instant. If $(f^+(\vec{p}), b, \sigma, s) \to^* (g^+(\vec{q}), b', \sigma', s')$ by the rules above then some register variables $\mathsf{r}_1, \ldots, \mathsf{r}_k$ in $\vec{p}$ can be instantiated. Because of the read once condition, we know that each register can be instantiated at most once. Thus we can uniquely determine a substitution $\delta = [u_1/\mathsf{r}_1, \ldots, u_k/\mathsf{r}_k]$ that associates the values read with the registers.

Next, we prove the following invariant: if $(f^+(\vec{p}), b, \sigma, s) \to (g^+(\vec{q}), b', \sigma', s')$ then we have $q_{\delta(f^+(\sigma(\vec{p})))} \geq q_{g^+((\sigma'(\vec{q})))}$ over the non-negative reals.

Proof. By case analysis on the small step rules.

(b'_2) Then δ is the identity substitution and $q_{f^+(\sigma(\vec{p}))} = q_{f^+(\sigma(\vec{p}))}$.

(b'_5) Then $\delta = [s(\mathsf{r})/\mathsf{r}] = [\sigma_1(p)/\mathsf{r}]$ and recalling that patterns are linear, we note that:

$$\delta(f^+(\sigma(\vec{p}))) = f^+((\delta \circ \sigma)(\vec{p})) = f^+((\sigma_1 \circ \sigma)[p/\mathsf{r}]\vec{p})\ .$$

(b'_6) Then δ is the identity substitution. By the properties of quasi-interpretations, we know that $q_{\sigma\vec{e}} \geq q_{\vec{v}}$. By the constraints generated by the control points, we derive that $q_{f^+(\vec{p})} \geq q_{g^+(\vec{e},\vec{r}_g)}$ over the non-negative reals. By the substitutivity property of quasi-interpretations, this implies that $q_{f^+(\sigma\vec{p})} \geq q_{g^+(\sigma\vec{e},\vec{r}_g)}$. Thus we derive, as required:

$$q_{f^+(\sigma\vec{p})} \geq q_{g^+(\sigma\vec{e},\vec{r}_g)} \geq q_{g^+(\vec{v},\vec{r}_g)} = q_{g^+(\sigma_1\vec{q},\vec{r}_g)}$$

(b'_7) Then δ is the identity substitution and the conclusion follows as in (b'_2). □

It remains to support our claim that all values computed by the thread i during an instant have a size bound by $q_{f(\vec{v},\vec{u})}$ where $\vec{u}$ are either the values read by the thread in the registers or some constant value.

Proof. By inspecting the shape of behaviours we see that a thread *computes values* either when writing into a register or in recursive calls. We consider in turn the two cases.

Writing Suppose $(f^+(\vec{p},\vec{\mathsf{r}}_f),b,\sigma,s) \to^* (g^+(\vec{q}),\mathsf{r} := e.b',\sigma',s')$ by reading a series of registers recorded by the substitution δ and assuming $\sigma\vec{p} = \vec{v}$. Then the invariant we have proved above implies that: $q_{f^+((\delta\circ\sigma)(\vec{p},\vec{r}_f))} \geq q_{g^+(\sigma'\vec{q})}$ over the non-negative reals. If some of the registers $\vec{\mathsf{r}}_f$ are not instantiated by the substitution δ, then we may replace them by some constant. Next, we observe that the constraint of index 0 associated with the control point requires that $q_{g^+(\vec{q})} \geq q_e$ and that if $\sigma e \Downarrow v$ then this implies $q_{g^+(\sigma'\vec{q})} \geq q_{\sigma' e} \geq q_v \geq |v|$.

Recursive call Suppose $(f^+(\vec{p},\vec{\mathsf{r}}_f),b,\sigma,s) \to^* (g^+(\vec{q}),h(\vec{e}),\sigma',s')$ by reading a series of registers recorded by the substitution δ and assuming $\sigma\vec{p} = \vec{v}$. Then the invariant we have proved above implies that: $q_{f^+((\delta\circ\sigma)(\vec{p},\vec{\mathsf{r}}_f))} \geq q_{g^+(\sigma'\vec{q})}$ over the non-negative reals. Again, if some of the registers $\vec{\mathsf{r}}_f$ are not instantiated by the substitution δ, then we may replace them by some constant value. Next we observe that the constraint of index 1 associated with the control point requires that $q_{g^+(\vec{q})} \geq q_{h^+(\vec{e},\vec{\mathsf{r}}_h)}$. Moreover, if $\sigma\vec{e} \Downarrow \vec{v}$ then $q_{g^+(\sigma'\vec{q})} \geq q_{h^+(\sigma'\vec{e},\vec{\mathsf{r}}_h)} \geq q_{h^+(\vec{v},\vec{\mathsf{r}}_h)} \geq q_{v_i} \geq |v_i|$, where v_i is any of the values in $\vec{v}$ (this uses the property $q_f(x_1,\ldots,x_n) \geq x_i$ of assignments, cf. Definition 13). □

B.6 Bounding the Size of Values for Systems

Corollary 17 Let B be a system with m registers and n threads. Suppose $B_1(i) = f_i(\vec{v}_i)$ for $i \in \mathbf{Z}_n$. Let c be a bound of the size of the largest parameter of the functions f_i and the largest default value of the registers. Suppose h is a function bounding all the quasi-interpretations, that is, for all the functions f_i^+ we have $h(x) \geq q_{f_i^+}(x,\ldots,x)$ over the non-negative reals. Then the size of the values computed by the system B during an instant is bound by $h^{n\cdot m+1}(c)$.

Proof. Because of the read once condition, during an instant a system can perform a (successful) read at most $n \cdot m$ times. We proceed by induction on the number k of registers the system has read so far to prove that the size of the values is bound by $h^{k+1}(c)$.

$k = 0$ If no register has been read, then Theorem 16 can be applied to show that all values have size bound by $h(c)$.

$k > 0$ Inductively, the size of the values in the parameters and the registers is bound by $h^k(c)$. Theorem 16 says that all the values that can be computed before reading a new register have a size bound by $h(h^k(c)) = h^{k+1}(c)$. □

B.7 Proof of theorem 21

Theorem 21 If a system B terminates by LPO and admits a polynomial quasi-interpretation then the computation of the system in an instant runs in space polynomial in the size of the parameters of the threads at the beginning of the instant.

Proof. We can always choose a polynomial for the function h in corollary 17. Hence, h^{nm+1} is also a polynomial. This shows that the size of all the values computed by the system is

bound by a polynomial. The number of values in a frame depends on the number of formal parameters and local variables and it can be statically bound. It remains to bound the number of frames on the stack. Note that behaviours are tail recursive. This means that the stack of each thread contains a frame that never returns a value plus possibly a sequence of frames that relate to the evaluation of expressions.

From this point on, one can follow the proof in [8]. The idea is to exploit the characteristics of the LPO order: a nested sequence of recursive calls $f_1(\vec{v}_1), \ldots, f_n(\vec{v}_n)$ must satisfy $f_1(\vec{v}_1) > \cdots > f_n(\vec{v}_n)$, where $>$ is the LPO order on terms. Because of the polynomial bound on the size of the values, one can provide a polynomial bound on the length of such strictly decreasing sequences and therefore a polynomial bound on the size of the stack needed to execute the system. □