Resource Control for Synchronous Cooperative Threads*

Roberto M. Amadio and Silvano Dal Zilio

Laboratoire d'Informatique Fondamentale de Marseille (LIF),
CNRS and Université de Provence, France.

Abstract. 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. These two methods can be combined to obtain an explicit polynomial bound on the resources needed for the execution of the system during an instant.

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.*, [6,14,15,16], has developed various inference techniques that allow for efficient analyses while capturing a sufficiently large range of practical algorithms.

Previous work [9,17] 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. Thus, in a nutshell, resource control relies on termination and bounds on data size. In [3], we have considered the problem of automatically inferring quasi-interpretations in the space of multi-variate max-plus polynomials. In [2], 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.

* This work was partly supported by ACI Sécurité Informatique, project CRISS.

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 68–82, 2004.

Our approach to resource bound certification follows distinctive design decisions. First, we allow the space needed for the execution of a program to vary depending on the size of its arguments. This is in contrast to most approaches that try to enforce a constant space bound. While this latter goal is reasonable for applications targeting embedded devices, it is not always relevant in the context of mobile code. Second, our method is applicable to a large class of algorithms and does not impose specific syntactical restrictions on programs. For example, we depart from works based on a linear usage of variables [14].

Our approach to resource control should be contrasted with traditional *worst case execution time* technology (see, *e.g.*, [20]): our bounds are less precise but they apply to a larger class of algorithms and are functional in the size of the input, which seems more appropriate in the context of mobile code. In another direction, one may compare our approach with the one based on linear logic (see, *e.g.*, [11]). While in principle the linear logic approach supports higher-order functions, the approach does not offer yet a user-friendly programming language.

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.*, [19] 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.

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 [8]. Boussinot *et al.* have proposed a weaker version of this feature where the reaction to the absence happens in the following instant [7] and they have implemented it in various programming environments based on C, Java, and Scheme. 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 [5] has proposed a formalisation of this programming model. Our analysis will essentially focus on a small fragment of this model where higher-order functions are ruled out and dynamic thread creation, and dynamic memory allocation are only allowed at the very beginning of an instant. 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. Here we show how 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). We expect that one could derive polynomial bounds on *time* as well, by adapting the work in [17].

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.

Omitted proofs may be found in a long version of this paper [1] in which we describe our programming model up to the point where a bytecode for a simple virtual machine implementing our synchronous language is defined. The long version also provides a number of programming examples illustrating how some synchronous and/or concurrent programming paradigms can be represented in our model (some simple examples are given at the end of Section 2). These examples suggest that the constraints imposed by the static analyses are not too severe and that their verification can be automated.

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 [4], 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 $f(\boldsymbol{p}_1) = be_1, \ldots, f(\boldsymbol{p}_k) = be_k$, where be_i is either an expression or a thread behaviour (see below), and $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_k$ are sequences of length n of patterns. We follow the usual hypothesis that the patterns in $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_k$ 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 $\boldsymbol{v}$ matches exactly a sequence of patterns $\boldsymbol{p}_i$ 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).

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

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; $f(\boldsymbol{e})$ and $\mathsf{next}.f(\boldsymbol{e})$ 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; $\mathsf{match}\,\mathsf{r}\,\mathsf{with}\,p_1 \Rightarrow b_1 \mid \cdots \mid p_k \Rightarrow b_k \mid [x] \Rightarrow f(\boldsymbol{e})$, 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 rule filters v then start the next instant with the behaviour $[v/x]f(\boldsymbol{e})$. 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*, ranged over by $X, X', \ldots$, that is a value in $\{N, R, S, W\}$ — 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 $f(\boldsymbol{v})$. 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: (1) $e \Downarrow v$, the closed expression e evaluates to the value v; (2) $(b, s) \xrightarrow{X} (b', s')$, 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; during an instant, we can have the following status transitions in a thread: $R \to S, W, N$ and $W \to R$, the last transition corresponds to a thread blocked on the behaviour $\mathsf{match}\,\mathsf{r}\,\mathsf{with}\ldots$ and no filters match the value of r; (3) $(B, s, i) \to (B', s', i')$ the system B with store s and current thread (index) i runs an atomic sequence of actions (performed by $B_1(i)$) and becomes (B', s', i').

Scheduler. The reduction relation, see Table 1, relies on the function $\mathcal{N}$ that computes the index of the next thread that should run in the current instant and the function $\mathcal{U}$ that updates the status of the thread at the end of an instant.

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 following status transitions take place $N \to R$, $W \to R$. 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 $[x] \Rightarrow f(\boldsymbol{e})$ 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 examples in [1]).

Example 1 (channels and signals). As shown in our informal presentation of behaviours, the match instruction allows one to read a register subject to certain filter conditions. This is a powerful mechanism which recalls, *e.g.*, Linda communication [12], 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:

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

(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:

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

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

(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:

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

Expression evaluation:

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

Behaviour reduction:

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

$$\frac{\text{no pattern matches } s(\mathsf{r})}{(\mathsf{match}\ \mathsf{r}\ \mathsf{with} \ldots, s) \xrightarrow{W} (\mathsf{match}\ \mathsf{r}\ \mathsf{with} \ldots, s)}$$

$$\frac{\sigma p = s(\mathsf{r}), \quad (\sigma b, s) \xrightarrow{X} (b', s')}{(\mathsf{match}\ \mathsf{r}\ \mathsf{with} \cdots \mid p \Rightarrow b \mid \ldots, s) \xrightarrow{X} (b', s')}$$

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

System reduction:

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

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

Conditions on the scheduler:

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

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

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

Table 1. Operational semantics

3 Control Flow Analysis

To bound the resources needed for the execution of a system and make possible a compositional analysis, a preliminary control flow analysis is required. 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.

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 $dble(\mathsf{s}(n)) = \mathsf{s}(\mathsf{s}(dble(n)))$ 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:

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

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 given in Section 3.1 to allow the execution of a read instruction at most a constant number of times.

3.1 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:

$$\text{if } (f(\boldsymbol{p}_i) = b_i)_{i\in 1..n} \text{ are the rules of } f \text{ then } R(f) =_{\text{def}} R(f) \cdot \textstyle\bigcup_{i\in 1..n} R(b_i)\ ,$$

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

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

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

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). If the first situation occurs, then for every function symbol f that returns a behaviour we can obtain a list of registers $\mathbf{r}_f$ 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 the second condition 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.

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

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

3.2 Control Points

We define 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 $(f(\boldsymbol{p}), be, i)$ where, intuitively, f is the currently called function, $\boldsymbol{p}$ 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. We associate with a system satisfying the read once condition a *finite* number of control points. If the function f returns a value and is defined by the rules $f(\boldsymbol{p}_1) = e_1, \ldots, f(\boldsymbol{p}_n) = e_n$, then we associate with f the set $\{(f(\boldsymbol{p}_1), e_1, 0), \ldots, (f(\boldsymbol{p}_n), e_n, 0)\}$.

On the other hand, if the function f is a behaviour defined by the rules $f(\boldsymbol{p}_1) = b_1, \ldots, f(\boldsymbol{p}_n) = b_n$ 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 $\mathbf{r}_f$ of registers that may be read within an instant starting from f. (The vector $\mathbf{r}_f$ 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 $\mathbf{r}_f$ and we regard the registers as part of the formal parameters of f^+. Then from the definition of f we produce the set $\bigcup_{i \in 1..n} \mathcal{C}(f^+, (\boldsymbol{p}_i, \mathbf{r}_f), b_i)$, where $\mathcal{C}(f^+, \boldsymbol{p}, b)$ is defined inductively on b as follows:

$$
\begin{array}{l}
\mathcal{C}(f^+, \boldsymbol{p}, b) = \text{case } b \text{ of} \\
\quad \mathsf{stop} \quad\quad : \{(f^+(\boldsymbol{p}), b, 2)\} \\
\quad g(\boldsymbol{e}) \quad\quad : \{(f^+(\boldsymbol{p}), b, 0)\} \\
\quad \mathsf{yield}.b' \quad : \{(f^+(\boldsymbol{p}), b, 2)\} \cup \mathcal{C}(f^+, \boldsymbol{p}, b') \\
\quad \mathsf{next}.g(\boldsymbol{e}) : \{(f^+(\boldsymbol{p}), b, 2), (f^+(\boldsymbol{p}), g(\boldsymbol{e}), 2)\} \\
\quad \mathsf{r} := e.b' \quad : \{(f^+(\boldsymbol{p}), b, 2), (f^+(\boldsymbol{p}), e, 1)\} \cup \mathcal{C}(f^+, \boldsymbol{p}, b') \\
\quad \mathsf{match}\ \mathsf{r}\ \mathsf{with}\ p_1 \Rightarrow b_1 \mid \cdots \mid p_n \Rightarrow b_n \mid [x] \Rightarrow g(\boldsymbol{e}) \ : \ \{(f^+(\boldsymbol{p}), b, 2), \\
\qquad (f^+([x/\mathsf{r}]\boldsymbol{p}), g(\boldsymbol{e}), 2)\} \cup \mathcal{C}(f^+, ([p_1/\mathsf{r}]\boldsymbol{p}), b_1) \cup \ldots \cup \mathcal{C}(f^+, ([p_n/\mathsf{r}]\boldsymbol{p}), b_n)
\end{array}
$$

By inspecting the definitions, we can check that a control point $(f(\boldsymbol{p}), be, i)$ has the property that $Var(be) \subseteq Var(\boldsymbol{p})$. The read once condition is instrumental to this property. For instance, (i) in case $g(\boldsymbol{e})$, we know that if g can read some register r then r could not have been already read by f and (ii) in the case of the match operator, 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 $\boldsymbol{p}$.

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 1. *An instance of a control point $(f(\boldsymbol{p}), b, i)$ 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.

Proposition 1. *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.*

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:	$(f(\boldsymbol{p}), e, 0)$,	$(f^+(\boldsymbol{p}), g(\boldsymbol{e}), 0)$,	$(f^+(\boldsymbol{p}), e, 1)$,	$(f^+(\boldsymbol{p}), be, 2)$
Constraint:	$f(\boldsymbol{p}) \succ_0 e$,	$f^+(\boldsymbol{p}) \succ_0 g^+(\boldsymbol{e}, \mathbf{r}_g)$,	$f^+(\boldsymbol{p}) \succ_1 e$,	*no constraints*

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 5. 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*, [10]).

Definition 2 (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 2. *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 3 (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 1 (termination of the instant). *All sequences of system reductions involving only rule (s_1) are finite.*

Proposition 3 and Theorem 1 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 6. 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. A suitable notion of quasi-interpretation [17,3] provides a modular solution to this problem.

Definition 3 (assignment). *Given a program, an* assignment q *associates with constructors and function symbols, functions over the positive 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 4 (quasi-interpretation). *An assignment is a* quasi-interpretation*, if for all constraints associated with the system of the shape* $f(\boldsymbol{p}) \succ_i e$*, with* $i \in \{0, 1\}$*, the inequality* $q_{f(\boldsymbol{p})} \geq q_e$ *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 7. With reference to Examples 2 and 6, the following assignment is a quasi-interpretation (we give no quasi-interpretations for the function *exp* because it fails the read once condition):

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

One can show [3] 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:

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

We generalise this result to threads as follows.

Theorem 2. *Given a system of synchronous threads B, suppose that at the beginning of the instant $B_1(i) = f(\boldsymbol{v})$ for some thread index i. Then the size of the values computed by the thread i during an instant is bound by $q_{f^+(\boldsymbol{v},\boldsymbol{u})}$ where $\boldsymbol{u}$ are the values contained in the registers $\mathbf{r}_f$ when they are read by the thread (or some constant value, otherwise).*

Theorem 2 is proven by showing that quasi-interpretations satisfy a suitable invariant. In general, a value computed and written by a thread can be read by another thread. However, at each instant, we have a bound on the number of threads and the number of reads that can be performed. We can then derive a bound on the size of the computed values which depends only on the size of the parameters at the beginning of the instant.

Corollary 1. *Let B be a system with m registers and n threads. Suppose $B_1(i) = f_i(\boldsymbol{v}_i)$ 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 8. The $n \cdot m$ iterations of the function h predicted by Corollary 1 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 long version of this paper [1]). Unfortunately, quasi-interpretations provide a bound on the size of the frames but not on their number (at least not in a direct implementation that does not rely on memoization). One way to cope with this problem is to combine quasi-interpretations with various families of reduction orders [9,17]. In the following, we provide an example of this approach based on *recursive path orders* which is a widely used and fully mechanisable technique to prove termination [10].

Definition 5. *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 6. *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 3. *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 3 is based on Corollary 1 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 [9]. The point is that the depth of such nested calls is polynomial in the size of the values, which allows us 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 depends on the quasi-interpretation and that bounds the size needed to run a system during an instant.

Example 9. With reference to Example 6, we can check that the order used there is indeed a LPO. From the quasi-interpretation in Example 7, 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 [3]. 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 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 [2] and the virtual machine presented in [1], 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 [5].

References

1. R. Amadio and S. Dal-Zilio. Resource control for synchronous cooperative threads. Research Report LIF 22-2004, 2004.
2. 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.
3. R. Amadio. Max-plus quasi-interpretations. In Proc. *TLCA*, Springer LNCS 2701, 2003.
4. J. Armstrong, R. Virding, C. Wikström, M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall 1996.
5. G. Boudol, ULM, a core programming model for global computing. In Proc. *ESOP*, Springer LNCS 2986, 2004.
6. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
7. F. Boussinot and R. De Simone, The SL Synchronous Language. *IEEE Trans. on Software Engineering*, 22(4):256–266, 1996.
8. G. Berry and G. Gonthier, The Esterel synchronous programming language. *Science of computer programming*, 19(2):87–152, 1992.
9. G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On termination methods with space bound certifications. In Proc. *PSI*, Springer LNCS 2244, 2001.
10. F. Baader and T. Nipkow. *Term rewriting and all that*. CUP, 1998.
11. P. Baillot and V. Mogbil, Soft lambda calculus: a language for polynomial time computation. In Proc. *FoSSaCS*, Springer LNCS 2987, 2004.
12. N. Carriero and D. Gelernter. Linda in Context. *CACM*, 32(4): 444-458, 1989.
13. A. Cobham. The intrinsic computational difficulty of functions. In Proc. *Logic, Methodology, and Philosophy of Science II*, North Holland, 1965.
14. M. Hofmann. The strength of non size-increasing computation. In Proc. *POPL*, ACM Press, 2002.
15. N. Jones. *Computability and complexity, from a programming perspective*. MIT-Press, 1997.
16. D. Leivant. Predicative recurrence and computational complexity i: word recurrence and poly-time. *Feasible mathematics II, Clote and Remmel (eds.)*, Birkhäuser:320–343, 1994.
17. J.-Y. Marion. *Complexité implicite des calculs, de la théorie à la pratique*. Habilitation à diriger des recherches, Université de Nancy, 2000.
18. M. Odersky. Functional nets. In Proc. *ESOP*, Springer LNCS 1782, 2000.
19. J. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at the USENIX Technical Conference, 1996.
20. P. Puschner and A. Burns (eds.), *Real time systems* 18(2/3), special issue on Worst-Case Execution Time Analysis, 2000.