Resource Control for Synchronous Cooperative Threads

Roberto M. Amadio, Silvano Dal Zilio

Resource Control for Synchronous Cooperative Threads*

Roberto M. Amadio
Université Paris 7 †

Silvano Dal Zilio
CNRS Marseille‡

14th December 2005

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. We show that 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.

As a second contribution, we introduce a virtual machine and a related bytecode thus producing a precise description of the resources needed for the execution of a system. In this context, we present a suitable control flow analysis that allows to formulate the static analyses for resource control at byte code level.

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 [18] of polynomial time functions by bounded recursion on notation. Following work, see e.g.

*Work partially supported by ACI *Sécurité Informatique* CRISS.

†Laboratoire *Preuves, Programmes et Systèmes*, UMR-CNRS 7126.

‡Laboratoire d'Informatique Fondamentale de Marseille, UMR-CNRS 6166

[8, 19, 21, 23], has developed various inference techniques that allow for efficient analyses while capturing a sufficiently large range of practical algorithms.
Previous work [10, 24] 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.
This approach to resource control should be contrasted with traditional *worst case execution time* technology (see, e.g., [30]): the 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 the applications we have in mind (see below). In another direction, one may compare the approach with the one based on linear logic (see, e.g., [7]): while in principle the linear logic approach supports higher-order functions, it does not offer yet a user-friendly programming language.
In [3, 4], 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. As a starting point, we choose a basic 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., [28] 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 [9]. Boussinot *et al.* have proposed a weaker version of this feature where the reaction to the absence happens in the following instant [13] and they have implemented it in various programming environments based on C, Java, and Scheme [31]. Applications suited to this programming style include: event-driven applications, graphical user interfaces, simulations (e.g. N-bodies problem, cellular automata, ad hoc networks), web

services, multiplayer online games, ... Boussinot *et al.* 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. These applications rely on data structure such as lists and trees whose size needs to be controlled.

Recently, Boudol [12] has proposed a formalisation of this programming model. Our analysis will essentially focus on a small fragment of this model without higher-order functions, and where the creation of fresh memory cells (registers) and the spawning of new threads is 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 3.1). 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 3.2). A third step, is to combine the termination and size analyses. Here we show how to obtain polynomial bounds on the *space* and *time* 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 3.3).

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 2.1) that guarantees that each thread performs each read instruction (in its body code) at most once in an instant. This condition is instrumental to resource control. In particular, 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 arbitrarily many instants. This is a harder problem which in general requires an understanding of the *global* behaviour of the system and/or stronger restrictions on the programs we can write. For the language studied in this paper, we advocate a combination of our static analyses with a dynamic controller that at the end of each instant checks the size of the parameters of the system and may decide to stop some threads taking too much space.

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.

As a second contribution, we describe a virtual machine and the related bytecode for our programming model (Section 4). 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. More precisely, we de-

scribe a control flow analysis that allows to recover the conditions for termination and size bounds at bytecode level and we show that the control flow analysis is sufficiently liberal to accept the code generated by a rather standard compilation function.
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 and constructor definitions and (2) a list of mutually recursive function and behaviour definitions relying on pattern matching. In this respect, the resulting programming language is reminiscent of ERLANG [5], which is a *practical* language to develop concurrent applications. The set of instructions a behaviour can execute is rather minimal. Indeed, our language can be regarded as an *intermediate code* where, for instance, general pattern-matching has been compiled into a nesting of *if_then_else* constructs and complex control structures have been compiled into a simple tail-recursive form.

Types We denote *type names* with $t, t', \ldots$ and *constructors* with $\mathsf{c}, \mathsf{c}', \ldots$ We will also denote with $\mathsf{r}, \mathsf{r}', \ldots$ constructors of arity 0 and of 'reference' type (see equation of kind (2) below) and we will refer to them as *registers* (thus registers are constructors). The *values* $v, v', \ldots$ computed by programs are first order terms built out of constructors. Types and constructors are declared via recursive equations that may be of two kinds:

$$\begin{array}{lrcl} (1) & t & = & \ldots \mid \mathsf{c}\ \mathit{of}\ t_1, \ldots, t_n \mid \ldots \\ (2) & t & = & \mathit{Ref}(t')\ \mathit{with} \ldots \mid \mathsf{r} = v \mid \ldots \end{array}$$

In (1) we declare a type t with a constructor c of functional type $(t_1, \ldots, t_n) \to t$. In (2) we declare a type t of registers referencing values of type t' and a register r with initial value v. As usual, type definitions can be mutually recursive (functional and reference types can be intermingled) and it is assumed that all types and constructors are declared exactly once. This means that we can associate a unique type with every constructor and that with respect to this association we can say when a value is well-typed. For instance, we may define the type *nat* of natural numbers in unary format by the equation $\mathit{nat} = \mathsf{z} \mid \mathsf{s}\ \mathit{of}\ \mathit{nat}$ and the type *llist* of linked lists of natural numbers by the equations $\mathit{nlist} = \mathsf{nil} \mid \mathsf{cons}\ \mathit{of}\ (\mathit{nat}, \mathit{llist})$ and $\mathit{llist} = \mathit{Ref}(\mathit{nlist})\ \mathit{with}\ \mathsf{r} = \mathsf{cons}(\mathsf{z}, \mathsf{r})$. The last definition declares a register r of type *llist* with initial value the infinite (cyclic) list containing only z's.
Finally, we have a special *behaviour type*, *beh*. Elements of type *beh* do not return a value but produce side effects. We denote with β either a regular type or *beh*.

Expressions We let $x, y, \ldots$ denote *variables* ranging over values. The *size* $|v|$ of a value v is defined by $|\mathsf{c}| = 0$ and $|\mathsf{c}(v_1, \ldots, v_n)| = 1 + |v_1| + \cdots + |v_n|$. In the following, we will use the *vectorial notation* $\mathbf{a}$ to denote either a vector $a_1, \ldots, a_n$ or a sequence $a_1 \cdots a_n$

of elements. We use $\sigma, \sigma', \ldots$ to denote a *substitution* $[\mathbf{v}/\mathbf{x}]$, where $\mathbf{v}$ and $\mathbf{x}$ have the same length. A *pattern* p is a well-typed term built out of constructors and variables. In particular, a *shallow linear pattern* p is a pattern $\mathsf{c}(x_1, \ldots, x_n)$, where c is a constructor of arity n and the variables $x_1, \ldots, x_n$ are all distinct. *Expressions*, e, and *expression bodies*, eb, are defined as:

$$\begin{array}{rcl} e & ::= & x \mid \mathsf{c}(e_1, \ldots, e_k) \mid f(e_1, \ldots, e_n) \\ eb & ::= & e \mid \mathit{match}\, x\, \mathit{with}\, p\, \mathit{then}\, eb\, \mathit{else}\, eb \end{array}$$

where f is a functional symbol of type $(t_1, \ldots, t_n) \to t$, specified by an equation of the kind $f(x_1, \ldots, x_n) = eb$, and where p is a shallow linear pattern.
A closed expression body eb evaluates to a value v according to the following standard rules:

$$(\mathsf{e}_1)\ \frac{}{\mathsf{r} \Downarrow \mathsf{r}} \qquad (\mathsf{e}_2)\ \frac{\mathbf{e} \Downarrow \mathbf{v}}{\mathsf{c}(\mathbf{e}) \Downarrow \mathsf{c}(\mathbf{v})} \qquad (\mathsf{e}_3)\ \frac{\mathbf{e} \Downarrow \mathbf{v}, \quad f(\mathbf{x}) = eb, \quad [\mathbf{v}/\mathbf{x}]eb \Downarrow v}{f(\mathbf{e}) \Downarrow v}$$

$$(\mathsf{e}_4)\ \frac{[\mathbf{v}/\mathbf{x}]eb_1 \Downarrow v}{\left(\begin{array}{l}\mathit{match}\ \mathsf{c}(\mathbf{v})\ \mathit{with}\ \mathsf{c}(\mathbf{x}) \\ \mathit{then}\ eb_1\ \mathit{else}\ eb_2\end{array}\right) \Downarrow v} \qquad (\mathsf{e}_5)\ \frac{eb_2 \Downarrow v \quad \mathsf{c} \neq \mathsf{d}}{\left(\begin{array}{l}\mathit{match}\ \mathsf{c}(\mathbf{v})\ \mathit{with}\ \mathsf{d}(\mathbf{x}) \\ \mathit{then}\ eb_1\ \mathit{else}\ eb_2\end{array}\right) \Downarrow v}$$

Since registers are constructors, rule (e_1) is a special case of rule (e_2); we keep the rule for clarity.

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 registers. A behaviour may also affect the scheduling status of the thread executing it. We denote with $b, b', \ldots$ behaviours defined as follows:

$$\begin{array}{rcl} b ::= & & \mathit{stop} \mid f(\mathbf{e}) \mid \mathit{yield}.b \mid \mathit{next}.f(\mathbf{e}) \mid \varrho := e.b \mid \\ & & \mathit{read}\ \varrho\ \mathit{with}\ p_1 \Rightarrow b_1 \mid \cdots \mid p_n \Rightarrow b_n \mid [\_] \Rightarrow f(\mathbf{e}) \mid \\ & & \mathit{match}\, x\, \mathit{with}\, \mathsf{c}(\mathbf{x})\, \mathit{then}\, b_1\, \mathit{else}\, b_2 \end{array}$$

where: (i) f is a functional symbol of type $t_1, \ldots, t_n \to \mathit{beh}$, defined by an equation $f(\mathbf{x}) = b$, (ii) $\varrho, \varrho', \ldots$ range over variables and registers, and (iii) $p_1, \ldots, p_n$ are either shallow linear patterns or variables. We also denote with $[\_]$ a special symbol that will be used in the default case of *read* expressions (see the paragraph **Scheduler** below). Note that if the pattern p_i is a variable then the following branches including the default one can never be executed.

The *effect* of the various instructions is informally described as follows: *stop*, terminates the executing thread for ever; *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; $f(\mathbf{e})$ and $\mathit{next}.f(\mathbf{e})$ 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; $\mathit{read}\, \mathsf{r}\, \mathit{with}\, p_1 \Rightarrow b_1 \mid \cdots \mid p_n \Rightarrow b_n \mid [\_] \Rightarrow b$, waits until the value of r matches one of the patterns $p_1, \ldots, p_n$ (there could be no

delay) and yields the control otherwise; if at the end of the instant the thread is always stuck waiting for a matching value then it starts the behaviour b in the following instant; *match* v *with* p *then* b_1 *else* b_2 filters the value v according to the pattern p, it never blocks the execution. Note that if p is a pattern and v is a value there is at most one matching substitution σ such that $v = \sigma p$.

Behaviour reduction is described by the 9 rules below. A reduction $(b, s) \xrightarrow{X} (b', s')$ means that 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. A status is a value in $\{N, R, S, W\}$ that represents one of the four possible state of a thread — N stands for next (the thread will resume at the beginning of the next instant), R for run, S for stopped, and W for wait (the thread is blocked on a *read* statement).

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

$$(\mathsf{b}_4)\ \frac{([\mathbf{v}/\mathbf{x}]b_1, s) \xrightarrow{X} (b', s')}{\begin{pmatrix} match\ \mathsf{c}(\mathbf{v}) \\ with\ \mathsf{c}(\mathbf{x}) \quad , \ s \\ then\ b_1\ else\ b_2 \end{pmatrix} \xrightarrow{X} (b', s')} \qquad (\mathsf{b}_5)\ \frac{(b_2, s) \xrightarrow{X} (b', s'), \quad \mathsf{c} \neq \mathsf{d}}{\begin{pmatrix} match\ \mathsf{c}(\mathbf{v}) \\ with\ \mathsf{d}(\mathbf{x}) \quad , \ s \\ then\ b_1\ else\ b_2 \end{pmatrix} \xrightarrow{X} (b', s')}$$

$$(\mathsf{b}_6)\ \frac{\text{no pattern matches } s(\mathsf{r})}{(read\ \mathsf{r} \ldots, s) \xrightarrow{W} (read\ \mathsf{r} \ldots, s)} \qquad (\mathsf{b}_7)\ \frac{s(\mathsf{r}) = \sigma p, \quad (\sigma b, s) \xrightarrow{X} (b', s')}{(read\ \mathsf{r}\ with \cdots \mid p \Rightarrow b \mid \ldots, s) \xrightarrow{X} (b', s')}$$

$$(\mathsf{b}_8)\ \frac{\mathbf{e} \Downarrow \mathbf{v}, \quad f(\mathbf{x}) = b, \quad ([\mathbf{v}/\mathbf{x}]b, s) \xrightarrow{X} (b', s')}{(f(\mathbf{e}), s) \xrightarrow{X} (b', s')} \qquad (\mathsf{b}_9)\ \frac{e \Downarrow v, \quad (b, s[v/\mathsf{r}]) \xrightarrow{X} (b', s')}{(\mathsf{r} := e.b, s) \xrightarrow{X} (b', s')}$$

We denote with *be* either an expression body or a behaviour. All expressions and behaviours are supposed to be *well-typed*. As usual, all formal parameters are supposed to be distinct. In the *match* x *with* $\mathsf{c}(\mathbf{y})$ *then* be_1 *else* be_2 instruction, be_1 may depend on $\mathbf{y}$ but not on x while be_2 may depend on x but not on $\mathbf{y}$.

Systems We suppose that the execution environment consists of n threads and we associate with every thread a distinct identity that is an index in $\mathbf{Z}_n = \{0, 1, \ldots, n-1\}$. We let $B, B', \ldots$ denote *systems* of synchronous threads, that is finite mappings 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 that at the beginning of each instant the store is s_o, such that each register is assigned its default value. If B is a system and $i \in \mathbf{Z}_n$ is a valid thread index then we denote with $B_1(i)$ the behaviour executed by 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 $f(\mathbf{v})$ for all $i \in \mathbf{Z}_n$. System reduction is described by a relation $(B, s, i) \rightarrow (B', s', i')$: the system B with store s and

current thread (index) i runs an atomic sequence of actions and becomes (B', s', i').

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

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

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 $\mathcal{N}(\dots)\uparrow$) then no thread can be run in the current instant.

$$\begin{array}{ll}\text{If } \mathcal{N}(B, s, i) = k \text{ then} & B_2(k) = R \text{ or } (\ B_2(k) = W \text{ and} \\ & \quad B_1(k) = \mathit{read}\ \mathsf{r}\ \mathit{with}\ \cdots \mid p \Rightarrow b \mid \dots \text{ and some pattern} \\ & \quad \text{matches } s(\mathsf{r}) \text{ i.e., } \exists \sigma\ \sigma p = s(\mathsf{r})\ ) \\ \text{If } \mathcal{N}(B, s, i) \uparrow \text{ then} & \forall k \in \mathbf{Z}_n,\ B_2(k) \in \{N, S\} \text{ or } (\ B_2(k) = W, \\ & \quad B_1(k) = \mathit{read}\ \mathsf{r}\ \mathit{with}\ \dots, \text{ and no pattern matches } s(\mathsf{r})\ )\end{array}$$

When no more thread can run, the instant ends and the function $\mathcal{U}$ performs the following status transitions: $N \to R$, $W \to R$. We assume here that every thread in status W takes the $[\_] \Rightarrow \dots$ branch at the beginning of the next instant. Note that the function $\mathcal{N}$ is undefined on the updated system if and only if all threads are stopped.

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

Example 1 (channels and signals) *The read instruction allows to read a register subject to certain filter conditions. This is a powerful mechanism which recalls, e.g., Linda communication [15], 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* $Ref(ch(t))$ *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:*

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

(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* $Ref(fch(t))$ *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:*

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

(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* $Ref(sig)$ *with default value* abst, *meaning that a signal is originally absent:*

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

Example 2 (cooperative fragment) *The* cooperative *fragment of the model with no synchrony is obtained by removing the next instruction and assuming that for all read instructions the branch* $[\_] \Rightarrow f(\mathbf{e})$ *is such that* $f(\ldots) = stop$. *Then all the interesting computation happens in the first instant; threads still running in the second instant can only stop. By using the representation of fifo channels presented in Example 1(2) above, the cooperative fragment is already powerful enough to simulate, e.g., Kahn networks [20].*

Next, to make possible a compositional and functional analysis for resource control, we propose to restrict the admissible behaviours and we define a simple preliminary control flow analysis that guarantees that this restriction is met. We then rely on this analysis to define a symbolic representation of the states reachable by a behaviour. Finally, we extract from this symbolic control points suitable order constraints which are instrumental to our analyses for termination and value size limitation within an instant.

2.1 Read Once Condition

We require and statically check on the *call graph* of the program (see below) that threads can perform any given read instruction at most once in an instant.

1. We assign to every read instruction in a system a distinct fresh label, y, and we collect all these labels in an ordered sequence, $y_1, \ldots, y_m$. In the following, we will sometimes use the notation $read_{\langle y \rangle}\ \varrho\ with\ \ldots$ in the code of a behaviour to make visible the label of a *read* instruction.

2. With every function symbol f defined by an equation $f(\mathbf{x}) = b$ we associate the set $L(f)$ of labels of read instructions occurring in b.

3. We define a directed *call* graph $G = (N, E)$ as follows: N is the set of function symbols in the program defined by an equation $f(\mathbf{x}) = b$ and $(f, g) \in E$ if $g \in \mathit{Call}(b)$ where $\mathit{Call}(b)$ is the collection of function symbols in N that may be called in the current instant and which is formally defined as follows:

$$\mathit{Call}(\mathit{stop}) = \mathit{Call}(\mathit{next}.g(\mathbf{e})) = \emptyset \quad \mathit{Call}(f(\mathbf{e})) = \{f\}$$
$$\mathit{Call}(\mathit{yield}.b) = \mathit{Call}(\varrho := e.b) = \mathit{Call}(b)$$
$$\mathit{Call}(\mathit{match}\ x\ \mathit{with}\ p\ \mathit{then}\ b_1\ \mathit{else}\ b_2) = \mathit{Call}(b_1) \cup \mathit{Call}(b_2)$$
$$\mathit{Call}(\mathit{read}\ \varrho\ \mathit{with}\ p_1 \Rightarrow b_1 \mid \cdots \mid p_n \Rightarrow b_n \mid [\_] \Rightarrow b) = \textstyle\bigcup_{i=1,\ldots,n} \mathit{Call}(b_i)$$

We write fE^*g if the node g is reachable from the node f in the graph G. We denote with $R(f)$ the set of labels $\bigcup\{L(g) \mid fE^*g\}$ and with $\mathbf{y}_f$ the ordered sequence of labels in $R(f)$.

The definition of *Call* is such that for every sequence of calls in the execution of a thread within an instant we can find a corresponding path in the call graph.

Definition 3 (read once condition) *A system satisfies the read once condition if in the call graph there are no loops that go through a node f such that $L(f) \neq \emptyset$.*

Example 4 (alarm) *We consider the representation of signals as in Example 1(3). We assume two signals* sig *and* ring*. The behaviour $\mathit{alarm}(n, m)$ 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.*

$$\begin{aligned}\mathit{alarm}(x, y) = \quad & \mathit{match}\ y\ \mathit{with}\ \mathsf{s}(y') \\ & \mathit{then}\ \mathit{read}_{\langle u \rangle}\ \mathsf{sig}\ \mathit{with}\ \mathsf{prst} \Rightarrow \mathit{next}.\mathit{alarm}(x, x) \ \mid\ [\_] \Rightarrow \mathit{alarm}(x, y') \\ & \mathit{else}\ \mathsf{ring} := \mathsf{prst}.\mathit{stop}\end{aligned}$$

Hence u is the label associated with the read instruction and $L(\mathit{alarm}) = \{u\}$. Since the call graph has just one node, alarm, and no edges, the read once condition is satisfied.

To summarise, the read once condition is a checkable syntactic condition that safely approximates the semantic property we are aiming at.

Proposition 5 *If a system satisfies the read once condition then in every instant every thread runs every read instruction at most once (but the same read instruction can be run by several threads).*

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 6 (exponentiation) *We recall that nat* $= \mathsf{z} \mid \mathsf{s}$ *of nat is the type of tally natural numbers. The function* dble *defined below doubles the value of its parameter 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:*

$$\begin{array}{ll} dble(n) = & match\ n\ with\ \mathsf{s}(n')\ then\ \mathsf{s}(\mathsf{s}(dble(n')))\ else\ \mathsf{z} \\ exp(n) = & match\ n\ with\ \mathsf{s}(n') \\ & then\ read\ \mathsf{r}\ with\ m \Rightarrow \mathsf{r} := dble(m).exp(n') \\ & else\ stop \end{array}$$

The function exp does not *satisfy the read once condition since the call graph has a loop on the exp node. 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 does not appear to be a severe limitation on the expressiveness of a synchronous programming language. Intuitively, in most synchronous algorithms every thread reads some bounded number of variables before performing some action. Note that while the number of variables is bounded by a constant, the amount of information that can be read in each variable is not. Thus, for instance, a 'server' thread can just read one variable in which is stored the list of requests produced so far and then it can go on scanning the list and replying to all the requests within the same instant.

2.2 Control Points

From a technical point of view, 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. This fact is used to associate with a system satisfying the read once condition a *finite* number of control points.

A *control point* is a triple $(f(\mathbf{p}), be, i)$ where, intuitively, f is the currently called function, $\mathbf{p}$ represents the patterns crossed so far in the function definition plus possibly the labels of the read instructions that still have to be executed, 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 equation $f(\mathbf{x}) = eb$, then we associate with f the set $\mathcal{C}(f, \mathbf{x}, eb)$ defined as follows:

$$\begin{array}{ll} \mathcal{C}(f, \mathbf{p}, eb) = case\ eb\ of & \\ e & : \{(f(\mathbf{p}), eb, 0)\} \\ \begin{pmatrix} match\ x\ with\ \mathsf{c}(\mathbf{y}) \\ then\ eb_1\ else\ eb_2 \end{pmatrix} & : \{(f(\mathbf{p}), eb, 2)\} \cup \mathcal{C}(f, [\mathsf{c}(\mathbf{y})/x]\mathbf{p}, eb_1) \cup \mathcal{C}(f, \mathbf{p}, eb_2) \end{array}$$

On the other hand, suppose the function f is a behaviour defined by the equation $f(\mathbf{x}) = b$. Then we generate a fresh function symbol f^+ whose arity is that of f plus the size of $R(f)$, thus regarding the labels $\mathbf{y}_f$ (the ordered sequence of labels in $R(f)$) as part of the formal

parameters of f^+. The set of control points associated with f^+ is the set $\mathcal{C}(f^+, (\mathbf{x} \cdot \mathbf{y}_f), b)$ defined as follows:

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

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

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

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

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

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

$(\mathcal{C}_6)$ $\begin{pmatrix}\mathit{match}\ x\ \mathit{with}\ \mathsf{c}(\mathbf{y}) \\ \mathit{then}\ b_1\ \mathit{else}\ b_2\end{pmatrix}$: $\{(f^+(\mathbf{p}), b, 2)\} \cup \mathcal{C}(f^+, ([\mathsf{c}(\mathbf{y})/x]\mathbf{p}), b_1) \cup \mathcal{C}(f^+, \mathbf{p}, b_2)$

$(\mathcal{C}_7)$ $\begin{pmatrix}\mathit{read}_{\langle y \rangle}\ \varrho\ \mathit{with}\ p_1 \Rightarrow b_1 \mid \cdots \mid \\ p_n \Rightarrow b_n \mid [\_] \Rightarrow g(\mathbf{e})\end{pmatrix}$: $\{(f^+(\mathbf{p}), b, 2), (f^+(\mathbf{p}), g(\mathbf{e}), 2)\} \cup \mathcal{C}(f^+, ([p_1/y]\mathbf{p}), b_1) \cup \ldots \cup \mathcal{C}(f^+, ([p_n/y]\mathbf{p}), b_n)$

By inspecting the definitions, we can check that a control point $(f(\mathbf{p}), be, i)$ has the property that $\mathit{Var}(be) \subseteq \mathit{Var}(\mathbf{p})$.

Definition 7 *An instance of a control point $(f(\mathbf{p}), be, i)$ is an expression body or a behaviour $be' = \sigma(be)$, where σ is a substitution mapping the free variables in be to values.*

The property of being an instance of a control point is preserved by expression body evaluation, behaviour reduction 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 evaluation and the reduction on pairs of control points and substitutions.

Proposition 8 *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 with control points:

Control point	Associated constraint
$(f(\mathbf{p}), e, 0)$	$f(\mathbf{p}) \succ_0 e$
$(f^+(\mathbf{p}), g(\mathbf{e}), 0)$	$f^+(\mathbf{p}) \succ_0 g^+(\mathbf{e}, \mathbf{y}_g)$
$(f^+(\mathbf{p}), e, 1)$	$f^+(\mathbf{p}) \succ_1 e$
$(f^+(\mathbf{p}), be, 2)$	*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 (cf. Section 3).

Example 9 *With reference to Example 4, we obtain the following control points:*

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

The triple $(alarm^+(x,y,u), \mathsf{prst}, 1)$ *is the only control point with a flag different from* 2*. It corresponds to the constraint* $alarm^+(x,y,u) \succ_1 \mathsf{prst}$*, where* u *is the label associated with the only read instruction in the body of alarm. 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 is also easily verified.*

In Example 2, we have discussed a possible representation of Kahn networks in the cooperative fragment of our model. In general Kahn networks 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., [22, 16, 17]). 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. Let us 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*.

Example 10 (read once vs. Lustre) *A* Lustre *network is composed of four types of nodes: the combinatorial node, the delay node, the when node, and the merge node. Each node may have several input streams and one output stream. The functional behaviour of each type of node is defined by a set of recursive definitions. For instance, the node When has one boolean input stream* b *— with values of type bool* $= \mathsf{false} \mid \mathsf{true}$ *— and one input stream* s *of values. A* When *node is used to output values from* s *whenever* b *is true. This behaviour may be described by the following recursive definitions:* $When(\mathsf{false} \cdot b, x \cdot s) = When(b,s)$*,* $When(\mathsf{true} \cdot b, x \cdot s) = x \cdot When(b,s)$*, and* $When(b,s) = \epsilon$ *otherwise. Here is a possible representation of the When node in our model, where the input streams correspond to one place channels* b, c *(cf. Example 1(1)), the output stream to a one place channel* c' *and at most one element in each input stream is processed per instant.*

$$
\begin{array}{l}
When() \ = \ read_{\langle u \rangle}\, \mathsf{b}\ with \\
\quad \mathsf{full}(\mathsf{true}) \Rightarrow read_{\langle v \rangle}\, \mathsf{c}\ with\ \mathsf{full}(x) \Rightarrow \mathsf{c}' := x.next.When() \mid [\_] \Rightarrow When() \\
\quad \mid \mathsf{full}(\mathsf{false}) \Rightarrow next.When() \\
\quad \mid [\_] \Rightarrow When()
\end{array}
$$

While the function When has no formal parameters, we consider the function $When^+$ *with two parameters* u *and* v *in our size and termination analyses.*

3 Resource Control

Our analysis goes in three main steps: first, we guarantee that each instant terminates (Section 3.1), second we bound the size of the computed values as a function of the size of the parameters at the beginning of the instant (Section 3.2), and third we combine the termination and size analyses to obtain polynomial bounds on space and time (Section 3.3). As we progress in our analysis, we refine the techniques we employ. Termination is reduced to the general problem of finding a suitable well-founded order over first-order terms. Bounding the size of the computed values is reduced to the problem of synthesizing a *quasi-interpretation*. Finally, the problem of obtaining polynomial bounds is attacked by combining *recursive path ordering* termination arguments with quasi-interpretations. We selected these techniques because they are well established and they can handle a significant spectrum of the programs we are interested in. It is to be expected that other characterisations of complexity classes available in the literature may lead to similar results.

3.1 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*, [6]).

Definition 11 (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 12 *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 13 (progress) *Let b be an instance of a control point. Then for all stores s, there exist X, b' and s' such that $(b, s) \xrightarrow{X} (b', s')$.*

Finally, we can guarantee 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)).

Theorem 14 (termination of the instant) *All sequences of system reductions involving only rule (s_1) are finite.*

Proposition 13 and Theorem 14 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 15 (monitor max value) *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 natural numbers received on* i *and assigns to* o *the greatest number in* x *and* l*.*

$$\begin{array}{lcl} f(x) & = & \mathit{yield}.\mathit{read}_{\langle i\rangle}\,\mathsf{i}\,\mathit{with}\,l \Rightarrow f_1(\mathit{maxl}(l,x)) \\ f_1(x) & = & \mathsf{o} := x.\mathit{next}.f(x) \\ \mathit{max}(x,y) & = & \mathit{match}\,x\,\mathit{with}\,\mathsf{s}(x') \\ & & \mathit{then}\,\mathit{match}\,y\,\mathit{with}\,\mathsf{s}(y')\,\mathit{then}\,\mathsf{s}(\mathit{max}(x',y'))\,\mathit{else}\,\mathsf{s}(x') \\ & & \mathit{else}\,\,y \\ \mathit{maxl}(l,x) & = & \mathit{match}\,l\,\mathit{with}\,\mathsf{cons}(y,l')\,\mathit{then}\,\mathit{maxl}(l',\mathit{max}(x,y))\,\mathit{else}\,x \end{array}$$

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

3.2 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 (cf. [10, 3, 4]).

Definition 16 (assignment) *Given a program, an* assignment q *associates with constructors and function symbols, functions over the non-negative reals* $\mathbf{R}^+$ *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 a function in* $(\mathbf{R}^+)^n \to \mathbf{R}^+$ *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 (name) with arity* n *then* $q_f : (\mathbf{R}^+)^n \to \mathbf{R}^+$ *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 $\mathit{Var}(e)$:

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

Here q_x is the identity function and, e.g., $q_{\mathsf{c}}(q_{e_1},\ldots,q_{e_n})$ is the functional composition of the function q_{c} with the functions $q_{e_1},\ldots,q_{e_n}$. It is easy to check that there exists a constant δ_q depending on the assignment q such that for all values v we have $|v| \leq q_v \leq \delta_q \cdot |v|$. Thus the quasi-interpretation of a value is always proportional to its size.

Definition 17 (quasi-interpretation) *An assignment is a* quasi-interpretation*, if for all constraints associated with the system of the shape $f(\mathbf{p}) \succ_i e$, with $i \in \{0,1\}$, the inequality $q_{f(\mathbf{p})} \geq q_e$ 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 18 *With reference to Examples 6 and 15, the following assignment is a quasi-interpretation (the parameter i corresponds to the label of the read instruction in the body of f). We give no quasi-interpretations for the function exp because it fails the read once condition:*

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

One can show [3, 4] 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:

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

We generalise this result to threads as follows.

Theorem 19 (bound on the size of the values) *Given a system of synchronous threads B, suppose that at the beginning of the instant $B_1(i) = f(\mathbf{v})$ for some thread index i. Then the size of the values computed by the thread i during an instant is bounded by $q_{f^+(\mathbf{v},\mathbf{u})}$ where $\mathbf{u}$ are the values contained in the registers at the time they are read by the thread (or some constant value, if they are not read at all).*

Theorem 19 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 bounded by a constant.

Corollary 20 *Let B be a system with m distinct read instructions and n threads. Suppose $B_1(i) = f_i(\mathbf{v}_i)$ 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 bounded by $h^{n \cdot m+1}(c)$.*

Example 21 *The $n \cdot m$ iterations of the function h predicted by Corollary 20 correspond to a tight bound, as shown by the following example. We assume n threads and one register, r, of type nat with default value* z*. The control of each thread is described as follows:*

$$\begin{aligned} f(x_0) \;=\; & \mathit{read}\ \mathsf{r}\ \mathit{with}\ x_1 \Rightarrow \mathsf{r} := \mathit{dble}(\mathit{max}(x_1, x_0)). \\ & \quad \mathit{read}\ \mathsf{r}\ \mathit{with}\ x_2 \Rightarrow \mathsf{r} := \mathit{dble}(x_2). \\ & \qquad \ldots\ldots \\ & \qquad \mathit{read}\ \mathsf{r}\ \mathit{with}\ x_m \Rightarrow \mathsf{r} := \mathit{dble}(x_m).\mathit{next}.f(\mathit{dble}(x_m))\ . \end{aligned}$$

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 $n \cdot m$ assignments to the register r *(m for every thread in the system) and that the value stored in* r *is $dble^{n \cdot m}(x_0)$ of size $2^{n \cdot m} \cdot |x_0|$.*

3.3 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 4). 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 [24, 10]. 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 [6].

Definition 22 *We say that a system terminates by LPO, if the reduction order associated with the system is a recursive path order where: (1) symbols are ordered so that function symbols are always bigger than constructor symbols and two distinct constructor symbols are incomparable; (2) the arguments of function symbols are compared with respect to the lexicographic order and those of constructor symbols with respect to the product order.*

Note that because of the hypotheses on constructors, this is actually a special case of the lexicographic path order. For the sake of brevity, we still refer to it as LPO.

Definition 23 *We say that a system admits a polynomial quasi-interpretation if it has a quasi-interpretation where all functions are bounded by a polynomial.*

The following property is a central result of this paper.

Theorem 24 *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 24 is based on Corollary 20 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 [10]. 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.

Example 25 *We can check that the order used in Example 15 for the functions f^+, f_1^+, max and maxl is indeed a LPO. Moreover, from the quasi-interpretation given in Example 18, we can deduce that the function $h(x)$ has the shape $a \cdot x + b$ (it is affine). In practice, many useful functions admit quasi-interpretations bound by an affine function such as the max-plus polynomials considered in [3, 4].*

The combination of LPO and polynomial quasi-interpretation actually provides a characterisation of PSPACE. In order to get to PTIME a further restriction has to be imposed. Among several possibilities, we select one proposed in [11]. We say that the system terminates by *linear* LPO if it terminates by LPO as in definition 22 and moreover if in all the constraints $f(\mathbf{p}) \succ_0 e$ or $f^+(\mathbf{p}) \succ_0 g^+(\mathbf{e})$ of index 0 there is at most one function symbol on the right hand side which has the same priority as the (unique) function symbol on the left-hand side. For instance, the Example 15 falls in this case. In op. cit., it is shown by a simple counting argument that the number of calls a function may generate is polynomial in the size of its arguments. One can then restate theorem 24 by replacing LPO with linear LPO and PSPACE with PTIME.
We stress that these results are of a *constructive* nature, thus beyond proving that a system 'runs in PSPACE (or PTIME)', we can extract a definite polynomial that bounds the size needed to run a system during an instant. In general, the bounds are rather rough and should be regarded as providing a *qualitative* rather than *quantitative* information.
In the purely functional framework, M. Hofmann [19] has explored the situation where a program is *non-size increasing* which means that the size of all intermediate results is bounded by the size of the input. Transferring this concept to a system of threads is attractive because it would allow to predict the behaviour of the system for arbitrarily many instants. However, this is problematic. For instance, consider again example 25. By Theorem 24, we can prove that the computation of a system running the behaviour $f(x_0)$ in an instant requires a space polynomial in the size of x_0. Note that the parameter of f is the largest value received so far in the register i. Clearly, bounding the value of this parameter for arbitrarily many instants requires a *global* analysis of the system which goes against our wish to produce a *compositional* analysis in the sense explained in the Introduction. An alternative approach which remains to be explored could be to develop linguistic tools and a programming discipline that allow each thread to control *locally* the size of its parameters.

4 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 synchronisation with

the *read* instruction and the detection of the end of an instant. A further motivation which is elaborated in Section 4.5 is the possibility to carry on the static analyses for resource control at bytecode level. The interest of bytecode verification is now well understood, and we refer the reader to [25, 26].

4.1 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 name 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 thread in the system, and three local registers owned by the scheduler whose role will become clear in Section 4.3.
A thread identifier, t, is simply an index in $\mathbf{Z}_n$. 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 a sequence of frames, and a *frame* is a triple (f, pc, ℓ) composed of a function name, the value of the program counter (a natural number in $1..|f|$), and a *stack* of values $\ell = v_1 \cdots v_k$. We denote with $|\ell|$ the number of values in the stack. 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 index where to jump at the next instant if the thread does not resume in the current instant, and n is the (logical) time at which the thread is suspended (cf. Section 4.3).

4.2 Instructions

The set of *instructions* of the virtual machine together with their operational meaning is described in Table 1. 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`.
For every segment of bytecode, we require that the last instruction is either `return`, `stop` or `tcall` and that the jump index j in the instructions `branch c` j and `wait` j is within the segment.

4.3 Scheduler

In Table 2 we describe a simple implementation of the scheduler. The scheduler owns three registers: (1) tid 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 *scheduler* triggers the execution of the current instruction of the current thread, whose index is stored in tid, with a call to *run*(tid). The call returns the label X associated with the instruction in Table 1. By convention, take $X = \epsilon$ when no label is displayed. If $X \neq \epsilon$ then the scheduler must take some action. Assume tid stores the thread index t. We denote

Table 1: Bytecode instructions

$f[pc]$	Current memory Following memory
`load` k	$M \cdot (f, pc, \ell \cdot v \cdot \ell') \ \rightarrow \ M \cdot (f, pc+1, \ell \cdot v \cdot \ell' \cdot v), \ \lvert\ell\rvert = k-1$
`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	$M \cdot (f, pc, \ell \cdot v_1 \cdots v_n) \ \rightarrow \ M \cdot (f, pc, \ell \cdot v_1 \cdots v_n) \cdot (g, 1, v_1 \cdots v_n)$
`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`	$M \cdot (g, pc', \ell' \cdot \mathbf{v}') \cdot (f, pc, \ell \cdot v) \ \rightarrow \ M \cdot (g, pc'+1, \ell' \cdot v), \ ar(f) = \lvert\mathbf{v}'\rvert$
`read` r	$(M \cdot (f, pc, \ell), s) \ \rightarrow \ (M \cdot (f, pc+1, \ell \cdot s(\mathsf{r})), s)$
`read` k	$(M \cdot (f, pc, \ell \cdot \mathsf{r} \cdot \ell'), s) \ \rightarrow \ (M \cdot (f, pc+1, \ell \cdot \mathsf{r} \cdot \ell' \cdot s(\mathsf{r})), s), \ \lvert\ell\rvert = k-1$
`write` r	$(M \cdot (f, pc, \ell \cdot v), s) \ \rightarrow \ (M \cdot (f, pc+1, \ell), s[v/\mathsf{r}])$
`write` k	$(M \cdot (f, pc, \ell \cdot \mathsf{r} \cdot \ell' \cdot v), s) \rightarrow (M \cdot (f, pc+1, \ell \cdot \mathsf{r} \cdot \ell'), s[v/\mathsf{r}]), \lvert\ell\rvert = k-1$
`stop`	$M \cdot (f, pc, \ell) \ \stackrel{S}{\rightarrow} \ \epsilon$
`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	$M \cdot (f, pc, \ell \cdot v) \ \stackrel{W}{\rightarrow} \ M \cdot (f, j, \ell)$

pc_{tid} the program counter of the top frame (f, pc_t, ℓ) in M_t, if any, I_{tid} the instruction $f[pc_t]$ (the current instruction in the thread) and st_{tid} the state st_t of 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 the running status and all waiting threads were interrupted after the last store modification occurred.

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, as illustrated here, 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.

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

Table 2: An implementation of the scheduler

```
for t in Z_n do { st_t := R; }                    (initialisation)
s := s_o; tid := time := wtime := 0;              (the initial thread is of index 0)
while (tid ∈ Z_n) {                               (loop until all threads are blocked)
 if I_tid = (write _) then wtime := time;         (record store modified)
 if I_tid = (wait j )
 then st_tid := W(pc_tid + 1, time);              (save continuation for next instant)
 X := run(tid);                                   (run current thread)
 if X ≠ ε then {
    if X ≠ W then st_tid := X;                    (update thread status)
    tid := N(tid, st);                            (compute index of next active thread)
    if tid ∈ Z_n                                  (test whether all threads are blocked)
    then { st_tid := R; time := time + 1; }       (if not, prepare next thread to run)
    else { s := s_o; wtime := time;               (else, initialisation of the new instant)
           tid := N(0, st);                       (select thread to run, starting from 0)
           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 $\mathcal{N}(\mathsf{tid}, st) = k \in \mathbf{Z}_n$ then $st_k = R$ or ($st_k = W(j, n)$ and $n < \mathsf{wtime}$)

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

Table 3: Compilation of source code to bytecode

Compilation of expression bodies:

$$
\begin{array}{lcl}
C(e,\eta) & = & C'(e,\eta)\cdot \texttt{return} \\
C\begin{pmatrix} \textit{match } x \textit{ with } \mathsf{c}(\mathbf{y}) \\ \textit{then } eb_1 \textit{ else } eb_2 \end{pmatrix},\eta\Big) & = & \begin{cases} (\texttt{branch c } j)\cdot C(eb_1,\eta'\cdot\mathbf{y})\cdot & \text{if } \eta=\eta'\cdot x \\ \quad (j:C(eb_2,\eta)) & \\ (\texttt{load } i(x,\eta))\cdot(\texttt{branch c } j)\cdot & \text{o.w.} \\ \quad C(eb_1,\eta\cdot\mathbf{y})\cdot(j:C(eb_2,\eta\cdot x)) & \end{cases}
\end{array}
$$

Auxiliary compilation of expressions:

$$
\begin{array}{lcl}
C'(x,\eta) & = & (\texttt{load } i(x,\eta)) \\
C'(\mathsf{c}(e_1,\ldots,e_n),\eta) & = & C'(e_1,\eta)\cdot\ldots\cdot C'(e_n,\eta)\cdot(\texttt{build c } n) \\
C'(f(e_1,\ldots,e_n),\eta) & = & C'(e_1,\eta)\cdot\ldots\cdot C'(e_n,\eta)\cdot(\texttt{call } f\ n)
\end{array}
$$

Compilation of behaviours:

$$
\begin{array}{lcl}
C(\mathit{stop},\eta) & = & \texttt{stop} \\
C(f(e_1,\ldots,e_n),\eta) & = & C'(e_1,\eta)\cdots C'(e_n,\eta)\cdot(\texttt{tcall } f\ n) \\
C(\mathit{yield}.b,\eta) & = & \texttt{yield}\cdot C(b,\eta) \\
C(\mathit{next}.f(\mathbf{e}),\eta) & = & \texttt{next}\cdot C(f(\mathbf{e}),\eta) \\
C(\varrho := e.b,\eta) & = & C'(e,\eta)\cdot(\texttt{write } i(\varrho,\eta))\cdot C(b,\eta) \\
C\begin{pmatrix} \textit{match } x \textit{ with } \mathsf{c}(\mathbf{y}) \\ \textit{then } b_1 \textit{ else } b_2 \end{pmatrix},\eta\Big) & = & \begin{cases} (\texttt{branch c } j)\cdot C(b_1,\eta'\cdot\mathbf{y})\cdot & \text{if } \eta=\eta'\cdot x \\ \quad (j:C(b_2,\eta)) & \\ (\texttt{load } i(x,\eta))\cdot(\texttt{branch c } j)\cdot & \text{o.w.} \\ \quad C(b_1,\eta\cdot\mathbf{y})\cdot(j:C(b_2,\eta\cdot x)) & \end{cases} \\
C\begin{pmatrix} \textit{read } \varrho \textit{ with } \cdots \mid \mathsf{c}_\ell(\mathbf{y}_\ell)\Rightarrow b_\ell \mid \\ \qquad\qquad \cdots y_k \Rightarrow b_k \cdots \end{pmatrix},\eta\Big) & = & \begin{pmatrix} j_0 : (\texttt{read } i(\varrho,\eta))\cdot\ldots\cdot \\ j_\ell : (\texttt{branch } \mathsf{c}_\ell\ j_{\ell+1})\cdot C(b_\ell,\eta\cdot\mathbf{y}_\ell)\cdot \\ j_{\ell+1} : \cdots j_k : C(b_k,\eta\cdot y_k) \end{pmatrix} \\
C\begin{pmatrix} \textit{read } \varrho \textit{ with } \cdots \mid \mathsf{c}_\ell(\mathbf{y}_\ell)\Rightarrow b_\ell \mid \\ \qquad\qquad \cdots \mid [\_]\Rightarrow g(\mathbf{e}) \end{pmatrix},\eta\Big) & = & \begin{pmatrix} j_0 : (\texttt{read } i(\varrho,\eta))\cdot\ldots\cdot \\ j_\ell : (\texttt{branch } \mathsf{c}_\ell\ j_{\ell+1})\cdot C(b_\ell,\eta\cdot\mathbf{y}_\ell)\cdot \\ j_{\ell+1} : \cdots j_n : (\texttt{wait } j_0)\cdot C(g(\mathbf{e}),\eta) \end{pmatrix}
\end{array}
$$

4.4 Compilation

In Table 3, we describe a possible compilation of the intermediate language into bytecode. We denote with η a sequence of variables. If x is a variable and η a sequence then $i(x, \eta)$ is the index of the rightmost occurrence of x in η. For instance, $i(x, x \cdot y \cdot x) = 3$. By convention, $i(\mathsf{r}, \eta) = \mathsf{r}$ if r is a register constant. We also use the notation $j : C(be, \eta)$ to indicate that j is the position of the first instruction of $C(be, \eta)$. This is just a convenient notation since, in practice, the position can be computed explicitly. With every function definition $f(x_1, \ldots, x_n) = be$ we associate the bytecode $C(be, x_1 \cdots x_n)$.

Example 26 (compiled code) *We show below the result of the compilation of the function alarm in Example 4:*

1 : `branch` s 12	6 : `load` 1	11 : `tcall` *alarm* 2
2 : `read` sig	7 : `tcall` *alarm* 2	12 : `build` prst 0
3 : `branch` prst 8	8 : `wait` 2	13 : `write` ring
4 : `next`	9 : `load` 1	14 : `stop`
5 : `load` 1	10 : `load` 2	

4.5 Control Flow Analysis Revisited

As a first step towards control flow analysis, we analyse the *flow graph* of the bytecode generated.

Definition 27 (flow graph) *The flow graph of a system is a directed graph whose nodes are pairs (f, i) where f is a function name in the program and i is an instruction index, $1 \leq i \leq |f|$, and whose edges are classified as follows:*

Successor: *An edge $((f, i), (f, i+1))$ if $f[i]$ is a* `load`*,* `branch`*,* `build`*,* `call`*,* `read`*,* `write`*, or* `yield` *instruction.*

Branch: *An edge $((f, i), (f, j))$ if $f[i] =$* `branch c` *j.*

Wait: *An edge $((f, i), (f, j))$ if $f[i] =$* `wait` *j.*

Next: *An edge $((f, i), (f, i + 1))$ if $f[i]$ is a* `wait` *or* `next` *instruction.*

Call: *An edge $((f, i), (g, 1))$ if $f[i] =$* `call` *g n or $f[i] =$* `tcall` *g n.*

The following is easily checked by inspecting the compilation function. Properties **Tree** and **Read-Wait** entail that the only cycles in the flow graph of a function correspond to the compilation of a *read* instruction. Property **Next** follows from the fact that, in a behaviour, an instruction *next* is always followed by a function call $f(\mathbf{e})$. Property **Read-Once** is a transposition of the read once condition (Section 2.1) at the level of the bytecode.

Proposition 28 *The flow graph associated with the compilation of a well-formed system satisfies the following properties:*

Tree: *Let G' be the flow graph without wait and call edges. Let G'_f be the full subgraph of G' whose nodes have the shape (f, i). Then G'_f is a* tree *with root $(f, 1)$.*

Read-Wait: *If $f[i] =$ `wait` j then $f[j] =$ `read` r and there is a unique path from (f, j) to (f, i) and in this path, every node corresponds to a* `branch` *instruction.*

Next: *Let G' be the flow graph without call edges. If $((f, i), (f, i + 1))$ is a next edge then for all nodes (f, j) accessible from $(f, i + 1)$, $f[j]$ is not a* `read` *instruction.*

Read-Once: *Let G' be the flow graph without wait edges and next edges. If the source code satisfies the read once condition then there is no loop in G' that goes through a node (f, i) such that $f[i]$ is a* `read` *instruction.*

In [1], we have presented a method to perform resource control verifications at bytecode level. This work is just concerned with the functional fragment of our model. Here, we outline its generalisation to the full model. The main problem is to reconstruct a symbolic representation of the values allocated on the stack. Once this is done, it is rather straightforward to formulate the constraints for the resource control. We give first an informal description of the method.

1. For every segment f of bytecode instructions with, say, formal parameters $x_1, \ldots, x_n$ and for every instruction i in the segment, we compute a sequence of expressions $e_1 \cdots e_m$ and a substitution σ.

2. The expressions $(e_i)_{i \in 1..m}$ are related to the formal parameters via the substitution σ. More precisely, the variables in the expressions are contained in $\sigma x_1, \ldots, \sigma x_n$ and the latter forms a linear pattern.

3. Next, let us look at the intended usage of the formal expressions. Suppose at run time the function f is called with actual parameters $u_1, \ldots, u_n$ and suppose that following this call, the control reaches instruction i with a stack ℓ. Then we would like that:

   - The values $u_1, \ldots, u_n$ match the pattern $\sigma x_1, \ldots, \sigma x_n$ via some substitution ρ.
   - The stack ℓ contains exactly m values $v_1, \ldots, v_m$ whose types are the ones of $e_1, \ldots, e_m$, respectively.
   - Moreover $\rho(e_i)$ is an *over-approximation* (w.r.t. size and/or termination) of the value v_i, for $i = 1, \ldots, m$. In particular, if e_i is a pattern, we want that $\rho(e_i) = v_i$.

We now describe precisely the generation of the expressions and the substitutions. This computation is called *shape analysis* in [1]. For every function f and index i such that $f[i]$ is a `read` instruction we assume a fresh variable $x_{f,i}$. Given a total order on the function

symbols, such variables can be totally ordered with respect to the index (f, i). Moreover, for every index i in the code of f, we assume a countable set $x_{i,j}$ of distinct variables.
We assume that the bytecode comes with annotations assigning a suitable type to every constructor, register, and function symbol. With every function symbol f of type $\mathbf{t} \to beh$, comes a fresh function symbol f^+ of type $\mathbf{t}, \mathbf{t}' \to beh$ so that $|\mathbf{t}'|$ is the number of read instructions accessible from f within an instant. Then, as in the definition of control points (Section 2.2), the extra arguments in f^+ corresponds to the values read in the registers within an instant. The order is chosen according to the order of the variables associated with the `read` instructions.
In the shape analysis, we will consider well-typed expressions obtained by composition of such fresh variables with function symbols, constructors, and registers. In order to make explicit the type of a variable x we will write x^t.
For every function f, the shape analysis computes a vector $\sigma = \sigma_1, \ldots, \sigma_{|f|}$ of substitutions and a vector $\mathbf{E} = E_1, \ldots, E_{|f|}$ of sequences of well-typed expressions. We let $\mathbf{E}_i$ and σ_i denote the sequence E_i and the substitution σ_i respectively (the i^{th} element in the vector), and $\mathbf{E}_i[k]$ the k^{th} element in $\mathbf{E}_i$. We also let $h_i = |\mathbf{E}_i|$ be the length of the i^{th} sequence. We assume $\sigma_1 = id$ and $\mathbf{E}_1 = x_{1,1}^{t_1} \cdots x_{1,n}^{t_n}$, if $f : t_1, \ldots, t_n \to \beta$ is a function of arity n.
The main case is the `branch` instruction:

$f[i] =$	Conditions
`branch` c j	$\mathtt{c} : \mathbf{t} \to t,\ \mathbf{E}_i = E \cdot e,\ e : t,$ and either $e = \mathtt{c}(\mathbf{e}),\ \sigma_{i+1} = \sigma_i,\ \mathbf{E}_{i+1} = E \cdot \mathbf{e}$ or $e = \mathtt{d}(\mathbf{e}),\ \mathtt{c} \neq \mathtt{d},\ \sigma_j = \sigma_i,\ \mathbf{E}_j = \mathbf{E}_i$ or $e = x^t,\ \sigma_j = \sigma_i,\ \mathbf{E}_j = \mathbf{E}_i,\ \sigma' = [\mathtt{c}(x_{i+1,h_i}^{t_1}, \ldots, x_{i+1,h_{i+1}}^{t_n})/x]$, $\sigma_{i+1} = \sigma' \circ \sigma_i,\ \mathbf{E}_{i+1} = \sigma'(E) \cdot x_{i+1,h_i} \cdots x_{i+1,h_{i+1}}$.

The constraints for the remaining instructions are given in Table 4, where it is assumed that $\sigma_{i+1} = \sigma_i$ except for the instructions `tcall` and `return` (that have no direct successors in the code of the function).

Example 29 *We give the shape of the values on the stack (a side result of the shape analysis) for the bytecode obtained from the compilation of the function f defined in Example 15:*

Instruction	*Shape*	*Instruction*	*Shape*
1 : `yield`	x	4 : `call` *maxl* 2	$x \cdot l \cdot x$
2 : `read` i	x	5 : `call` f_1 1	$x \cdot maxl(l, x)$
3 : `load` 1	$x \cdot l$	6 : `return`	$x \cdot f_1(maxl(l, x))$

Note that the code has no `branch` *instruction, hence the substitution σ is always the identity. Once the shapes are generated it is rather straightforward to determine a set of constraints that entails the termination of the code and a bound on the size of the computed values. For instance, assuming the reduction order is a simplification order, it is enough to require that $f^+(x, l) > f_1(maxl(l, x))$, i.e. the shape of the returned value, $f_1(maxl(l, x))$, is less than the shape of the call, $f^+(x, l)$.*

Table 4: Shape analysis at bytecode level

$f[i] =$	Conditions
`load` k	$k \in 1..h_i,\ \mathbf{E}_{i+1} = \mathbf{E}_i \cdot \mathbf{E}_i[k]$
`build` `c` n	$\mathtt{c} : \mathbf{t} \to t,\ \mathbf{E}_i = E \cdot \mathbf{e},\ \lvert\mathbf{e}\rvert = n,\ \mathbf{e} : \mathbf{t},\ \mathbf{E}_{i+1} = E \cdot \mathtt{c}(\mathbf{e})$
`call` g n	$g : \mathbf{t} \to t,\ \mathbf{E}_i = E \cdot \mathbf{e},\ \lvert\mathbf{e}\rvert = n,\ \mathbf{e} : \mathbf{t},\ \mathbf{E}_{i+1} = E \cdot g(\mathbf{e})$
`tcall` g n	$g : \mathbf{t} \to \beta,\ \mathbf{E}_i = E \cdot \mathbf{e},\ \lvert\mathbf{e}\rvert = n,\ \mathbf{e} : \mathbf{t}$
`return`	$f : \mathbf{t} \to t,\ \mathbf{E}_i = E \cdot e,\ e : t$
`read` `r`	$\mathtt{r} : \mathit{Ref}(t),\ \mathbf{E}_{i+1} = \mathbf{E}_i \cdot x^t_{f,i}$
`read` k	$k \in 1..h_i,\ \mathbf{E}_i[k] : \mathit{Ref}(t),\ \mathbf{E}_{i+1} = \mathbf{E}_i \cdot x^t_{f,i}$
`write` `r`	$\mathtt{r} : \mathit{Ref}(t),\ \mathbf{E}_i = E \cdot e,\ e : t,\ \mathbf{E}_{i+1} = E$
`write` k	$k \in 1..h_i,\ \mathbf{E}_i[k] : \mathit{Ref}(t),\ \mathbf{E}_i = E \cdot e,\ e : t,\ \mathbf{E}_{i+1} = E$
`yield`	$\mathbf{E}_{i+1} = \mathbf{E}_i$
`next`	$\mathbf{E}_{i+1} = \mathbf{E}_i$
`wait` j	$\mathbf{E}_i = \mathbf{E}_j \cdot x^t_{f,j},\ \mathbf{E}_{i+1} = \mathbf{E}_j,\ \sigma_i = \sigma_j$

If one can find a reduction order and an assignment satisfying the constraints generated from the shape analysis then one can show the termination of the instant and provide bounds on the size of the computed values. We refrain from developing this part which is essentially an adaptation of Section 3 at bytecode level. Moreover, a detailed treatment of the functional fragment is available in [1]. Instead, we state that the shape analysis is always successful on the bytecode generated by the compilation function described in Table 3 (see Appendix B.8). This should suggest that the control flow analysis is not overly constraining though it can certainly be enriched in order to take into account some code optimisations.

Theorem 30 *The shape analysis succeeds on the compilation of a well-formed program.*

5 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 more complicated control structures.

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. Moreover, it is quite possible to perform them 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 [12].

Acknowledgements and Publication History We would like to thank the referees for their valuable comments. Thanks to G. Boudol and F. Dabrowski for comments and discussions on a preliminary version of this article that was presented at the 2004 International Conference on Concurrency Theory. In the present paper, we consider a more general model which includes references as first class values and requires a reformulation of the control flow analysis. Moreover, we present a new virtual machine, a number of examples, and complete proofs not available in the conference paper.

References

[1] R. Amadio, S. Coupet-Grimal, S. Dal-Zilio, and L. Jakubiec. A functional scenario for bytecode verification of resource bounds. In *Proceedings of CSL – International Conference on Computer Science Logic*, Lecture Notes in Computer Science 3210, Springer, 2004.

[2] R. Amadio, S. Dal-Zilio. Resource control for synchronous cooperative threads. In *Proceedings CONCUR – 15th International Conference on Concurrency Theory*, Lecture Notes in Computer Science 3170, Springer, 2004.

[3] R. Amadio. Max-plus quasi-interpretations. In *Proceedings of TLCA – 6th International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 2701, Springer, 2003.

[4] R. Amadio. Synthesis of max-plus quasi-interpretations. In *Fundamenta Informaticae*, 65(1-2):29-60, 2005.

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

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

[7] P. Baillot and V. Mogbil, Soft lambda calculus: a language for polynomial time computation. In *Proceedings of FOSSACS – 7th International Conference on Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science 2987, Springer, 2004.

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

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

[10] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On termination methods with space bound certifications. In *Proceedings Perspectives of System Informatics*, Lecture Notes in Computer Science 2244, Springer, 2001.

[11] G. Bonfante, J.-Y. Marion, J.-Y. Moyen. Quasi-interpretations. Internal report LORIA, November 2004, available from the authors.

[12] G. Boudol, ULM, a core programming model for global computing. In *Proceedings of ESOP – 13th European Symposium on Programming*, Lecture Notes in Computer Science 2986, Springer, 2004.

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

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

[15] N. Carriero and D. Gelernter. Linda in Context. *Communication of the ACM*, 32(4): 444-458, 1989.

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

[17] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *Proceedings of ICFP – ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices 31(6), ACM Press, 1996.

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

[19] M. Hofmann. The strength of non size-increasing computation. In *Proceedings of POPL – 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 2002.

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

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

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

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

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

[25] 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.

[26] G. Necula. Proof carrying code. In *Proceedings of POPL – 24th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 1997.

[27] M. Odersky. Functional nets. In *Proceedings of ESOP – 9th European Symposium on Programming*, Lecture Notes in Computer Science 1782, Springer, 2000.

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

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

[30] P. Puschner and A. Burns (eds.), *Real time systems* 18(2/3), Special issue on Worst-case execution time analysis, 2000.

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

A 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 $full(v)$, hence we use function parameters as a kind of private memory (to compare with registers that model shared memory).

$$\begin{array}{lcl} empty() & = & read\ \mathsf{put}\ with\ \mathsf{full}(x) \Rightarrow next.full(x)\ \mid\ [\_] \Rightarrow empty() \\ full(x) & = & read\ \mathsf{get}\ with\ \mathsf{prst} \Rightarrow \mathsf{result} := x.yield.empty()\ \mid\ [\_] \Rightarrow full(x) \end{array}$$

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 [27]. 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).
We define a new algebraic datatype 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 to 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. An endRead and endWrite request is always immediately acknowledged.
The thread protecting the resource starts with the behaviour $onlyreader(\mathsf{z})$, defined in Table 5, meaning the system has no pending requests for reading or writing. 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 endRead requests before acknowledging the request to write.
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 $askRead(id).read$ allow $with\ id \Rightarrow \cdots$ where $askRead(id)$ is a shorthand for $read$ req $with$ none $\Rightarrow$ 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 31 *8 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.*

Table 5: Code for the Readers-Writers pattern

$$
\begin{array}{lcl}
\mathit{onlyreader}(x) & = & \mathit{match}\ x\ \mathit{with}\ \mathsf{s}(x')\ \mathit{then\ read}\ \mathsf{req}\ \mathit{with} \\
& & \quad \mathsf{endRead} \Rightarrow \mathit{next}.\mathit{onlyreader}(x') \\
& & \mid \mathsf{startWrite}(y) \Rightarrow \mathit{next}.\mathit{pwrite}(y, \mathsf{s}(x')) \\
& & \mid \mathsf{startRead}(y) \Rightarrow \mathit{next}.\mathsf{allow} := y.\mathit{onlyreader}(\mathsf{s}(\mathsf{s}(x'))) \\
& & \mid [\_] \Rightarrow \mathit{onlyreader}(\mathsf{s}(x')) \\
& & \mathit{else\ read}\ \mathsf{req}\ \mathit{with} \\
& & \quad \mathsf{startWrite}(y) \Rightarrow \mathit{next}.\mathsf{allow} := y.\mathit{pwrite}(y, \mathsf{z}) \\
& & \mid \mathsf{startRead}(y) \Rightarrow \mathit{next}.\mathsf{allow} := y.\mathit{onlyreader}(\mathsf{s}(\mathsf{z})) \\
& & \mid [\_] \Rightarrow \mathit{onlyreader}(\mathsf{z}) \\
\mathit{pwrite}(id, x) & = & \mathit{match}\ x\ \mathit{with}\ \mathsf{s}(x')\ \mathit{then} \\
& & \quad \mathit{match}\ x'\ \mathit{with}\ \mathsf{s}(x'')\ \mathit{then\ read}\ \mathsf{req}\ \mathit{with} \\
& & \qquad \mathsf{endRead} \Rightarrow \mathit{next}.\mathit{pwrite}(id, \mathsf{s}(x'')) \\
& & \quad \mid [\_] \Rightarrow \mathit{pwrite}(id, \mathsf{s}(\mathsf{s}(x''))) \\
& & \quad \mathit{else\ read}\ \mathsf{req}\ \mathit{with} \\
& & \qquad \mathsf{endRead} \Rightarrow \mathit{next}.\mathsf{allow} := id.\mathit{pwrite}(id, \mathsf{z}) \\
& & \quad \mid [\_] \Rightarrow \mathit{pwrite}(id, \mathsf{s}(\mathsf{z})) \\
& & \mathit{else\ read}\ \mathsf{req}\ \mathit{with} \\
& & \quad \mathsf{endWrite} \Rightarrow \mathit{next}.\mathit{onlyreader}(\mathsf{z}) \\
& & \mid [\_] \Rightarrow \mathit{pwrite}(id, \mathsf{z})
\end{array}
$$

PROOF. Let $(f(\mathbf{p}), be, i)$ be a control point of an expression body or of a behaviour. In Table 6, we reformulate the evaluation and the reduction by replacing expression bodies or behaviours by triples $(f(\mathbf{p}), be, \sigma)$ where $(f(\mathbf{p}), be, i)$ is a control point and σ is a substitution mapping the variables in $\mathbf{p}$ to values. By convention, we take $\sigma(\mathsf{r}) = \mathsf{r}$ if r is a register.
We claim that the evaluation and reduction in Table 6 are equivalent to those presented in Section 2 in the following sense:

1. $(f(\mathbf{p}), e_0, \sigma) \Downarrow v$ iff $\sigma e_0 \Downarrow v$.

2. $(f^+(\mathbf{p}), b_0, s, \sigma) \xrightarrow{X} (g^+(\mathbf{q}), b'_0, s', \sigma')$ iff $\sigma b_0 \xrightarrow{X} \sigma' b'_0$.

In the following proofs we will refer to the rules in Table 6. The revised formulation makes clear that if b is an instance of a control point and $(b, s) \xrightarrow{X} (b', s')$ then b' is an instance. It remains to check that being an instance is a property preserved at the level of system reduction. We proceed by case analysis on the last reduction rule used in the derivation of $(B, s, i) \to (B', s', i')$.

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

(s_2) One of the threads performs one step. Moreover, the threads in waiting status take the

Table 6: Expression body evaluation and behaviour reduction revised

$$(\mathsf{e}_0)\ \frac{}{(f(\mathbf{p}), x, \sigma) \Downarrow \sigma(x)} \qquad (\mathsf{e}_1)\ \frac{}{(f(\mathbf{p}), \mathsf{r}, \sigma) \Downarrow \mathsf{r}}$$

$$(\mathsf{e}_2)\ \frac{(f(\mathbf{p}), e_i, \sigma) \Downarrow v_i \quad i \in 1..n}{(f(\mathbf{p}), \mathsf{c}(\mathbf{e}), \sigma) \Downarrow \mathsf{c}(\mathbf{v})} \qquad (\mathsf{e}_3)\ \frac{\begin{array}{c}(f(\mathbf{p}), e_i, \sigma) \Downarrow v_i \quad i \in 1..n, \\ g(\mathbf{x}) = eb, \quad (g(\mathbf{x}), eb, [\mathbf{v}/\mathbf{x}]) \Downarrow v\end{array}}{(f(\mathbf{p}), g(\mathbf{e}), \sigma) \Downarrow v}$$

$$(\mathsf{e}_4)\ \frac{\begin{array}{c}\sigma(x) = \mathsf{c}(\mathbf{v}), \\ (f([\mathsf{c}(\mathbf{x})/x]\mathbf{p}), eb_1, [\mathbf{v}/\mathbf{x}] \circ \sigma) \Downarrow v\end{array}}{\left(f(\mathbf{p}), \begin{array}{c} match\ x \\ with\ \mathsf{c}(\mathbf{x}) \\ then\ eb_1\ else\ eb_2 \end{array}, \sigma\right) \Downarrow v} \qquad (\mathsf{e}_5)\ \frac{\begin{array}{c}\sigma(x) = \mathsf{d}(\ldots), \\ (f(\mathbf{p}), eb_2, \sigma) \Downarrow v\end{array}}{\left(f(\mathbf{p}), \begin{array}{c} match\ x \\ with\ \mathsf{c}(\mathbf{x}) \\ then\ eb_1\ else\ eb_2 \end{array}, \sigma\right) \Downarrow v}$$

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

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

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

$$(\mathsf{b}_4)\ \frac{\sigma(x) = \mathsf{c}(\mathbf{v}), \quad (f^+([\mathsf{c}(\mathbf{x})/x]\mathbf{p}), b_1, [\mathbf{v}/\mathbf{x}] \circ \sigma, s) \xrightarrow{X} (f_1^+(\mathbf{p}'), b', \sigma', s')}{\left(f^+(\mathbf{p}), \begin{array}{l} match\ x\ with\ \mathsf{c}(\mathbf{x}) \\ then\ b_1\ else\ b_2 \end{array}, \sigma, s\right) \xrightarrow{X} (f_1^+(\mathbf{p}'), b', \sigma', s')}$$

$$(\mathsf{b}_5)\ \frac{\sigma(x) = \mathsf{d}(\ldots), \quad \mathsf{c} \neq \mathsf{d}, \quad (f^+(\mathbf{p}), b_2, \sigma, s) \xrightarrow{X} (f_1^+(\mathbf{p}'), b', \sigma', s')}{\left(f^+(\mathbf{p}), \begin{array}{l} match\ x\ with\ \mathsf{c}(\mathbf{x}) \\ then\ b_1\ else\ b_2 \end{array}, s, \sigma\right) \xrightarrow{X} (f_1^+(\mathbf{p}'), b', \sigma', s')}$$

$$(\mathsf{b}_6)\ \frac{\text{no pattern matches } s(\sigma(\varrho))}{(f^+(\mathbf{p}), read\ \ \varrho\ with \ldots, \sigma, s) \xrightarrow{W} (f^+(\mathbf{p}), read\ \ \varrho\ with \ldots, \sigma, s)}$$

$$(\mathsf{b}_7)\ \frac{\sigma_1(p) = s(\sigma(\varrho)), \quad (f^+([p/y]\mathbf{p}), b, \sigma_1 \circ \sigma, s) \xrightarrow{X} (f_1^+(\mathbf{p}'), b', \sigma', s')}{(f^+(\mathbf{p}), read_{\langle y \rangle}\ \varrho\ with\ \cdots \mid p \Rightarrow b \mid \ldots, \sigma, s) \xrightarrow{X} (f_1^+(\mathbf{p}'), b', \sigma', s')}$$

$$(\mathsf{b}_8)\ \frac{\begin{array}{c}\sigma\mathbf{e} \Downarrow \mathbf{v}, \quad g(\mathbf{x}) = b, \\ (g^+(\mathbf{x}, \mathbf{y}_g), b, [\mathbf{v}/\mathbf{x}], s) \xrightarrow{X} (f_1^+(\mathbf{p}'), b', \sigma', s')\end{array}}{(f^+(\mathbf{p}), g(\mathbf{e}), \sigma, s) \xrightarrow{X} (f_1^+(\mathbf{p}'), b', \sigma', s')}$$

$$(\mathsf{b}_9)\ \frac{\sigma e \Downarrow v, \quad (f^+(\mathbf{p}), b, \sigma, s[v/\sigma(\varrho)]) \xrightarrow{X} (f_1^+(\mathbf{p}'), b', \sigma', s')}{(f^+(\mathbf{p}), \varrho := e.b, \sigma, s) \xrightarrow{X} (f_1^+(\mathbf{p}'), b', \sigma', s')}$$

$[\_] \Rightarrow g(\mathbf{e})$ branch of the *read* instructions that were blocking. A thread *read* ϱ $\ldots \mid [\_] \Rightarrow g(\mathbf{e})$ in waiting status is an instance of a control point $(f^+(\mathbf{p}), \mathit{read}\ \varrho\ \ldots \mid [\_] \Rightarrow g(\mathbf{e}_0), j)$. By $(\mathcal{C}_7)$, $(f^+(\mathbf{p}), g(\mathbf{e}_0), 2)$ is a control point, and $g(\mathbf{e})$ is one of its instances. □

B.2 Evaluation of Closed Expressions

Proposition 32 *12 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.*

As announced, we refer to the rules in Table 6. We recall that the order $>$ or $\geq$ refers to the reduction order that satisfies the constraints of index 0. We start by proving the following working lemma.

Lemma 33 *For all well formed triples, $(f(\mathbf{p}), eb, \sigma)$, there is a value v such that $(f(\mathbf{p}), eb, \sigma) \Downarrow v$. Moreover, if eb is an expression then $\sigma(eb) \geq v$ else $f(\sigma\mathbf{p}) \geq v$.*

Proof. We proceed by induction on the pair $(f(\sigma\mathbf{p}), eb)$ ordered lexicographically from left to right. The first argument is ordered according to the reduction order and the second according to the structure of the expression body.

$eb \equiv x$. We apply rule (e_0) and $\sigma(x) \geq \sigma(x)$.

$eb \equiv \mathsf{r}$. We apply rule (e_1) and $\sigma(\mathsf{r}) = \mathsf{r} \geq \mathsf{r}$.

$eb \equiv \mathsf{c}(e_1, \ldots, e_n)$. We apply rule (e_2). By inductive hypothesis, $(f(\mathbf{p}), e_i, \sigma) \Downarrow v_i$ for $i \in 1..n$ and $\sigma e_i \geq v_i$. By definition of reduction order, we derive $\sigma(\mathsf{c}(e_1, \ldots, e_n)) \geq \mathsf{c}(v_1, \ldots, v_n)$.

$eb \equiv f(e_1, \ldots, e_n)$. We apply rule (e_3). By inductive hypothesis, $(f(\mathbf{p}), e_i, \sigma) \Downarrow v_i$ for $i \in 1..n$ and $\sigma e_i \geq v_i$. By the definition of the generated constraints $f(\mathbf{p}) > g(\mathbf{e})$, which by definition of reduction order implies that $f(\sigma\mathbf{p}) > g(\sigma\mathbf{e}) \geq g(\mathbf{v}) = g([\mathbf{v}/\mathbf{x}]\mathbf{x})$. Thus by inductive hypothesis, $g(\mathbf{x}, eb, [\mathbf{v}/x]) \Downarrow v$. We conclude by showing by case analysis that $g(\sigma\mathbf{e}) \geq v$.

- eb is an expression. By the constraint we have $g(\mathbf{x}) > eb$, and by inductive hypothesis $[\mathbf{v}/\mathbf{x}]eb \geq v$. So $g(\sigma\mathbf{e}) \geq g(\mathbf{v}) > [\mathbf{v}/\mathbf{x}]eb \geq v$.
- eb is not an expression. Then by inductive hypothesis, $g(\mathbf{v}) \geq v$ and we know $g(\sigma\mathbf{e}) \geq g(\mathbf{v})$.

$eb \equiv \mathit{match}\ x\ \mathit{with}\ \mathsf{c}(\mathbf{x}) \ldots$. We distinguish two cases.

- $\sigma(x) = \mathsf{c}(\mathbf{v})$. Then rule (e_4) applies. Let $\sigma' = [\mathbf{v}/\mathbf{x}] \circ \sigma$. Note that $\sigma'([\mathsf{c}(\mathbf{x})/x]\mathbf{p}) = \sigma\mathbf{p}$. By inductive hypothesis, we have that $(f([\mathsf{c}(\mathbf{x})/x]\mathbf{p}), eb_1, \sigma') \Downarrow v$. We show by case analysis that $f(\sigma\mathbf{p}) \geq v$.

– eb_1 is an expression. By inductive hypothesis, $\sigma'(eb_1) \geq v$. By the constraint, $f([\mathsf{c}(\mathbf{x})/x]\mathbf{p}) > eb_1$. Hence, $f(\sigma\mathbf{p}) = f(\sigma'[\mathsf{c}(\mathbf{x})/x]\mathbf{p}) > \sigma'(eb_1)$.

– eb_2 is not an expression. By inductive hypothesis, we have that $f(\sigma\mathbf{p})$ equals $f(\sigma'[\mathsf{c}(\mathbf{x})/x]\mathbf{p}) \geq v$.

- $\sigma(x) = \mathsf{d}(\ldots)$ with $\mathsf{c} \neq \mathsf{d}$. Then rule ($\mathsf{e}_5$) applies and an argument simpler than the one above allows to conclude. □

Relying on Lemma 33 we can now prove Proposition 12, that if e is a closed expression and $e \Downarrow v$ then $e \geq v$ in the reduction order. Proof. We proceed by induction on the structure of e.

e **is value** v**.** Then $v \Downarrow v$ and $v \geq v$.

$e \equiv \mathsf{c}(e_1, \ldots, e_n)$**.** By inductive hypothesis, $e_i \Downarrow v_i$ and $e_i \geq v_i$ for $i \in 1..n$. By definition of reduction order, $\mathsf{c}(\mathbf{e}) \geq \mathsf{c}(\mathbf{v})$.

$e \equiv f(e_1, \ldots, e_n)$**.** By inductive hypothesis, $e_i \Downarrow v_i$ and $e_i \geq v_i$ for $i \in 1..n$. Suppose $f(\mathbf{x}) = eb$. By Lemma 33, $(f(\mathbf{x}), eb, [\mathbf{v}/\mathbf{x}]) \Downarrow v$ and either $f(\mathbf{v}) \geq v$ or $f(\mathbf{x}) > eb$ and $\sigma(eb) \geq v$. We conclude by a simple case analysis. □

B.3 Progress

Proposition 34 *13 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 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. Moreover, let $ln(b)$ be the *length* of b inductively defined as follows:

$$ln(\mathtt{stop}) = ln(f(\mathbf{e})) = 0 \quad ln(yield.b) = ln(\varrho := e.b) = 1 + ln(b) \quad ln(next.f(\mathbf{e})) = 2$$
$$ln(match\ x\ with\ \mathsf{c}(\mathbf{x})\ then\ b_1\ else\ b_2) = 1 + max(ln(b_1), ln(b_2))$$
$$ln(read\ \varrho\ with \ldots \mid p_i \Rightarrow b_i \mid \ldots \mid [\_] \Rightarrow f(\mathbf{e})) = 1 + max(\ldots, ln(b_i), \ldots)$$

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

$$\mu(b, \gamma) \ =_{\text{def}} \ (nr(b), f^+(\sigma\mathbf{p}), ln(b)) \ .$$

We assume that measures are lexicographically ordered from left to right, where the order on the first and third component is the standard order on natural numbers and the order on the second component is the reduction order considered in study of the termination conditions. 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.

$b \equiv stop$**.** Rule (b_1) applies, with $X = S$, and the measure stays constant.

$b \equiv \mathit{yield}.b'$. Rule ($\mathsf{b}_2$) applies, with $X = R$, and the measure decreases because $ln(b)$ decreases.

$b \equiv \mathit{next}.b'$. Rule ($\mathsf{b}_3$) applies, with $X = N$, and the measure decreases because $ln(b)$ decreases.

$b \equiv \mathit{match}\ \ldots$. Rules ($\mathsf{b}_4$) or ($\mathsf{b}_5$) apply and the measure decreases because $ln(b)$ decreases.

$b \equiv \mathit{read}\ \ldots$. If no pattern matches then rule (b_6) applies and the measure is left unchanged. If a pattern matches then rule (b_7) applies and the measure decreases because $nr(b)$ decreases and then the induction hypothesis applies.

$b \equiv g(\mathbf{e})$. Rule (b_8) applies to $(f^+(\mathbf{p}), g(\mathbf{e}_0), \sigma)$, assuming $\mathbf{e} = \sigma\mathbf{e}_0$. By Proposition 12, we know that $\mathbf{e} \Downarrow \mathbf{v}$ and $\mathbf{e} \geq \mathbf{v}$ in the reduction order. Suppose g is associated to the declaration $g(\mathbf{x}) = b$. The constraint associated with the control point requires $f^+(\mathbf{p}) > g^+(\mathbf{e}_0, \mathbf{y}_g)$. Then using the properties of reduction orders we observe:

$$f^+(\sigma\mathbf{p}) \ > g^+(\sigma\mathbf{e}_0, \mathbf{y}_g) \ = g^+(\mathbf{e}, \mathbf{y}_g) \ \geq g^+(\mathbf{v}, \mathbf{y}_g)$$

Thus the measure decreases because $f^+(\sigma\mathbf{p}) > g^+(\mathbf{v}, \mathbf{y}_g)$, and then the induction hypothesis applies.

$b \equiv \varrho := e.b'$. By Proposition 12, we have $e \Downarrow v$. Hence rule (b_9) applies, the measure decreases because $ln(b)$ decreases, and then the induction hypothesis applies. □

Remark 35 *We point out that in the proof of proposition 13, if $X = R$ then the measure decreases and if $X \in \{N, S, W\}$ then the measure decreases or stays the same. We use this observation in the following proof of Theorem 14.*

B.4 Termination of the Instant

Theorem 36 *14 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 defined in the proof of Proposition 13. Thus $\mu'(i)$ can be regarded as a quadruple with a lexicographic order from left to right. With a system B of n threads we associate the measure $\mu_B =_{\text{def}} (\mu'(0), \ldots, \mu'(n-1))$ that is a tuple. We compare such tuples using the product order. We prove that every system reduction sequence involving only rule (s_1) terminates by proving that this measure decreases during reduction. We recall the rule below:

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

Let $B'' = B'[(B'_1(k), R)/k]$. We proceed by case analysis on X and $B'_2(k)$.
If $B'_2(k) = R$ then $\mu'(k)$ is left unchanged. The only other case is $B'_2(k) = W$. In this case the conditions on the scheduler tell us that $i \neq k$. Indeed, the thread k must be blocked on a *read* r instruction and it can only be scheduled if the value stored in r has been modified, which means than some other thread than k must have modified r. For the same reason, some pattern in the *read* r instruction of $B_1(k)$ matches $s'(\mathsf{r})$, which means that the number of reads that $B_1(k)$ may perform in the current instant decreases and that $\mu'(k)$ also decreases.
By hypothesis we have $(B_1(i), s) \xrightarrow{X} (b', s')$, hence by Remark 35, $\mu'(i)$ decreases or stays the same. By the previous line of reasoning $\mu'(k)$ decreases and the other measures $\mu'(j)$ stay the same. Hence the measure μ_B decreases, as needed. □

B.5 Bounding the Size of Values for Threads

Theorem 37 *19 Given a system of synchronous threads B, suppose that at the beginning of the instant $B_1(i) = f(\mathbf{v})$ for some thread index i. Then the size of the values computed by the thread i during an instant is bounded by $q_{f^+(\mathbf{v},\mathbf{u})}$ where $\mathbf{u}$ are the values contained in the registers at the time they are read by the thread (or some constant value, if they are not read at all).*

In Table 6, we have defined the reduction of behaviours as a *big step* semantics. In Table 7 we reformulate the operational semantics following a *small step* approach. First, note that there are no rules corresponding to (b_1), (b_3) or (b_6) since these rules either terminate or suspend the computation of the thread in the instant. Second, the reduction makes abstraction of the memory and the scheduler. Instead, the reduction relation is parameterized on an assignment δ associating values with the labels of the read instructions.
The assignment δ is a kind of *oracle* that provides the thread with the finitely many values (because of the read once condition) it may read within the current instant. The assignment δ provides a *safe* abstraction of the store s used in the transition rules of Table 6. Note that the resulting system represents *more* reductions than can actually occur in the original semantics within an instant. Namely, a thread can write a value v in r and then proceed to read from r a value different from v without yielding the control. This kind of reduction is impossible in the original semantics. However, since we do not rely on a precise monitoring of the values written in the store, this loss of precision does not affect our analysis.
Next we prove that if $(f^+(\mathbf{p}), b, \sigma) \rightarrow_\delta (g^+(\mathbf{q}), b', \sigma')$ then $q_{f^+(\sigma'' \circ \sigma(\mathbf{p}))} \geq q_{g^+(\sigma'(\mathbf{q}))}$ over the non-negative reals, where σ'' is either the identity or the restriction of δ to the label of the *read* instruction in case (b'_7).
Proof. By case analysis on the small step rules. Cases (b'_2), (b'_5) and (b'_9) are immediate.

(b'_4) The assertion follows by a straightforward computation on substitutions.

(b'_7) Then $\sigma''(y) = \delta(y) = [\sigma_1(p)/y]$ and recalling that patterns are linear, we note that: $f^+((\sigma'' \circ \sigma)(\mathbf{p})) = f^+((\sigma_1 \circ \sigma)[p/y](\mathbf{p}))$.

Table 7: Small step reduction within an instant

(b'_2) $(f^+(\mathbf{p}), \mathit{yield}.b, \sigma) \to_\delta (f^+(\mathbf{p}), b, \sigma)$

(b'_4) $\big(f^+(\mathbf{p}), \begin{array}{l}\mathit{match}\ x\ \mathit{with}\ \mathsf{c}(\mathbf{x})\\ \mathit{then}\ b_1\ \mathit{else}\ b_2\end{array}, \sigma\big) \to_\delta (f^+([\mathsf{c}(\mathbf{x})/x]\mathbf{p}), b_1, [\mathbf{v}/\mathbf{x}] \circ \sigma)$ if (1)

(b'_5) $\big(f^+(\mathbf{p}), \begin{array}{l}\mathit{match}\ x\ \mathit{with}\ \mathsf{c}(\mathbf{x})\\ \mathit{then}\ b_1\ \mathit{else}\ b_2\end{array}, \sigma\big) \to_\delta (f^+(\mathbf{p}), b_2, \sigma)$ if $\sigma(x) = \mathsf{d}(\ldots), \mathsf{c} \neq \mathsf{d}$

(b'_7) $(f^+(\mathbf{p}), \mathit{read}_{\langle y \rangle}\, \varrho\ \mathit{with}\ \cdots \mid p \Rightarrow b \mid \ldots, \sigma) \to_\delta (f^+([p/y]\mathbf{p}), b, \sigma_1 \circ \sigma)$ if (2)

(b'_8) $(f^+(\mathbf{p}), g(\mathbf{e}), \sigma) \to_\delta (g^+(\mathbf{x}, \mathbf{y}_g), b, [\mathbf{v}/\mathbf{x}])$ if $\sigma\mathbf{e} \Downarrow \mathbf{v}$ and $g(\mathbf{x}) = b$

(b'_9) $(f^+(\mathbf{p}), \varrho := e.b, \sigma) \to_\delta (f^+(\mathbf{p}), b, \sigma)$ if $\sigma e \Downarrow v$

where: (1) $\equiv \sigma(x) = \mathsf{c}(\mathbf{v})$ and (2) $\equiv \sigma_1(p) = \delta(y)$.

(b'_8) By the properties of quasi-interpretations, we know that $q_{\sigma(\mathbf{e})} \geq q_{\mathbf{v}}$. By the constraints generated by the control points, we derive that $q_{f^+(\mathbf{p})} \geq q_{g^+(\mathbf{e},\mathbf{y}_g)}$ over the non-negative reals. By the substitutivity property of quasi-interpretations, this implies that $q_{f^+(\sigma(\mathbf{p}))} \geq q_{g^+(\sigma(\mathbf{e},\mathbf{y}_g))}$. Thus we derive, as required: $q_{f^+(\sigma(\mathbf{p}))} \geq q_{g^+(\sigma(\mathbf{e},\mathbf{y}_g))} \geq q_{g^+(\mathbf{v},\mathbf{y}_g)}$. □

It remains to support our claim that all values computed by the thread i during an instant have a size bounded by $q_{f(\mathbf{v},\mathbf{u})}$ where $\mathbf{u}$ are either the values read by the thread 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 $(f^+(\mathbf{p}, \mathbf{y}_f), b, \sigma) \to^*_\delta (g^+(\mathbf{q}), \varrho := e.b', \sigma')$ by performing a series of reads recorded by the substitution σ''. Then the invariant we have proved above implies that: $q_{f^+((\sigma'' \circ \sigma)(\mathbf{p},\mathbf{y}_f))} \geq q_{g^+(\sigma'\mathbf{q})}$ over the non-negative reals. If some of the variables in $\mathbf{y}_f$ are not instantiated by the substitution σ'', then we may replace them by some constant. Next, we observe that the constraint of index 1 associated with the control point requires that $q_{g^+(\mathbf{q})} \geq q_e$ and that if $\sigma(e) \Downarrow v$ then this implies $q_{g^+(\sigma'(\mathbf{q}))} \geq q_{\sigma'(e)} \geq q_v \geq |v|$.

Recursive call Suppose $(f^+(\mathbf{p}, \mathbf{y}_f), b, \sigma) \to^*_\delta (g^+(\mathbf{q}), h(\mathbf{e}), \sigma')$ by performing a series of reads recorded by the substitution σ''. Then the invariant we have proved above implies that: $q_{f^+((\sigma'' \circ \sigma)(\mathbf{p},\mathbf{y}_f))} \geq q_{g^+(\sigma'(\mathbf{q}))}$ over the non-negative reals. Again, if some of the variables in $\mathbf{y}_f$ are not instantiated by the substitution σ'', then we may replace them by some constant value. Next we observe that the constraint of index 0 associated with the control point requires that $q_{g^+(\mathbf{q})} \geq q_{h^+(\mathbf{e},\mathbf{y}_h)}$. Moreover, if $\sigma'(\mathbf{e}) \Downarrow \mathbf{v}$ then $q_{g^+(\sigma'(\mathbf{q}))} \geq q_{h^+(\sigma'(\mathbf{e},\mathbf{y}_h))} \geq q_{h^+(\mathbf{v},\mathbf{y}_h)} \geq q_{v_i} \geq |v_i|$, where v_i is any of the values in $\mathbf{v}$. The last inequation relies on the monotonicity property of assignments, see property (3) in Definition 16, that is $q_{h^+}(z_1, \ldots, z_n) \geq z_j$ for all $j \in 1..n$. □

B.6 Bounding the Size of Values for Systems

Corollary 38 *20 Let B be a system with m distinct read instructions and n threads.*

Suppose $B_1(i) = f_i(\mathbf{v}_i)$ 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 bounded 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 reads the system has performed so far to prove that the size of the values is bounded by $h^{k+1}(c)$.

$k = 0$ If no read has been performed, then Theorem 19 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 bounded by $h^k(c)$. Theorem 19 says that all the values that can be computed before performing a new read have a size bound by $h(h^k(c)) = h^{k+1}(c)$. □

B.7 Combination of LPO and Polynomial Quasi-interpretations

Theorem 39 *24 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 20. Hence, h^{nm+1} is also a polynomial. This shows that the size of all the values computed by the system is bounded 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 [10]. The idea is to exploit the characteristics of the LPO order: a nested sequence of recursive calls $f_1(\mathbf{v}_1), \ldots, f_n(\mathbf{v}_n)$ must satisfy $f_1(\mathbf{v}_1) > \cdots > f_n(\mathbf{v}_n)$, where $>$ is the LPO order on terms. Because of the polynomial bound on the size of the values and the characteristics of the LPO on constructors, 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. □

B.8 Compiled Code is Well-shaped

Theorem 40 *30 The shape analysis succeeds on the compilation of a well-formed program.*

Let *be* be either a behaviour or an expression body, η be a sequence of variables, and E be a sequence of expressions. We say that the triple (be, η, E) is *compatible* if for all variables x free in *be*, the index $i(x, \eta)$ is defined and if $\eta[k] = x$ then $E[k] = x$. Moreover,

we say that the triple is *strongly compatible* if it is compatible and $|\eta| = |E|$. In the following we will neglect typing issues that offer no particular difficulty. First we prove the following lemma.

Lemma 41 *If (e, η, E) is compatible then the shape analysis of $C'(e, \eta)$ starting from the shape E succeeds and produces a shape $E \cdot e$.*

Proof. By induction on the structure of e.

$e \equiv x$ Then $C'(x, \eta) = \texttt{load}\ i(x, \eta)$. We know that $i(x, \eta)$ is defined and $\eta[k] = x$ implies $E[k] = x$. So the shape analysis succeeds and produces $E \cdot x$.

$e \equiv \texttt{c}(e_1, \ldots, e_n)$ Then $C'(\texttt{c}(e_1, \ldots, e_n), \eta) = C'(e_1, \eta) \cdots C'(e_n, \eta)(\texttt{build c } n)$. We note that if e' is a subexpression of e, e'' is another expression, and (e, η, E) is compatible then $(e', \eta, E \cdot e'')$ is compatible too. Thus we can apply the inductive hypothesis to $e_1, \ldots, e_n$ and derive that the shape analysis of $C'(e_1, \eta)$ starting from E succeeds and produces $E \cdot e_1$,..., and the shape analysis of $C'(e_n, \eta)$ starting from $E \cdot e_1 \cdots e_{n-1}$ succeeds and produces $E \cdot e_1 \cdots e_n$. Then by the definition of shape analysis of `build` we can conclude.

$e \equiv f(e_1, \ldots, e_n)$ An argument similar to the one above applies. □

Next we generalise the lemma to behaviours and expression bodies.

Lemma 42 *If (be, η, E) is strongly compatible then the shape analysis of $C(be, \eta)$ starting from the shape E succeeds.*

Proof. *be* $\equiv e$ We have that $C(e, \eta) = C'(e, \eta) \cdot \texttt{return}$ and the shape analysis on $C'(e, \eta)$ succeeds, producing at least one expression.

be $\equiv$ *match* x *with* $\texttt{c}(\mathbf{y})$ *then* eb_1 *else* eb_2 Following the definition of the compilation function, we distinguish two cases:

- $\eta \equiv \eta' \cdot x$: Then $C(be, \eta) = (\texttt{branch c } j) \cdot C(eb_1, \eta' \cdot \mathbf{y}) \cdot (j : C(eb_2, \eta)\ )$. By the hypothesis of strong compatibility, $E \equiv E' \cdot x$ and by definition of shape analysis on `branch` we get on the *then* branch a shape $[\texttt{c}(\mathbf{y})/x]E' \cdot \mathbf{y}$ up to variable renaming. We observe that $(eb_1, \eta' \cdot \mathbf{y}, [\texttt{c}(\mathbf{y})/x]E' \cdot \mathbf{y})$ are strongly compatible (note that here we rely on the fact that η' and E' have the same length). Hence, by inductive hypothesis, the shape analysis on $C(eb_1, \eta' \cdot \mathbf{y})$ succeeds. As for the else branch, we have a shape $E' \cdot x$ and since $(eb_2, \eta' \cdot x, E' \cdot x)$ are strongly compatible we derive by inductive hypothesis that the shape analysis on $C(eb_2, \eta)$ succeeds.
- $\eta \not\equiv \eta' \cdot x$: The compiled code starts with $(\texttt{load}\ i(x, \eta))$ which produces a shape $E \cdot x$. Then the analysis proceeds as in the previous case.

be $\equiv$ *stop* The shape analysis succeeds.

be $\equiv f(e_1, \ldots, e_n)$ By lemma 41, we derive that the shape analysis of $C'(e_1, \eta) \cdot \ldots \cdot C'(e_n, \eta)$

succeeds and produces $E \cdot e_1 \cdots e_n$. We conclude applying the definition of the shape analysis for `tcall`.

$be \equiv yield.b$ The instruction `yield` does not change the shape and we can apply the inductive hypothesis on b.

$be \equiv next.g(\mathbf{e})$ The instruction `next` does not change the shape and we can apply the inductive hypothesis on $g(\mathbf{e})$.

$be \equiv \varrho := e.b$ By lemma 41, we have the shape $E \cdot e$. By definition of the shape analysis on `write`, we get back to the shape E and then we apply the inductive hypothesis on b.

$be \equiv match \ldots$ The same argument as for expression bodies applies.

$be \equiv read\ \varrho\ with\ \mathtt{c}_1(\mathbf{y}_1) \Rightarrow b_1 \mid \ldots \mid \mathtt{c}_n(\mathbf{y}_n) \Rightarrow b_n \mid [\_] \Rightarrow g(\mathbf{e})$ We recall that the compiled code is:

$$j_0 : (\mathtt{read}\ i(\varrho, \eta)) \cdot (\mathtt{branch}\ \mathtt{c}_1\ j_1) \cdot C(b_1, \eta \cdot \mathbf{y}_1) \cdots$$
$$j_{n-1} : (\mathtt{branch}\ \mathtt{c}_n\ j_n) \cdot C(b_n, \eta \cdot \mathbf{y}_n) \cdot j_n : (\mathtt{wait}\ j_0) \cdot C(g(\mathbf{e}), \eta)$$

The `read` instruction produces a shape $E \cdot y$. Then if a positive branch is selected, we have a shape $E \cdot \mathbf{y}_k$ for $k \in 1..n$. We note that the triples $(b_k, \eta \cdot \mathbf{y}_k, E \cdot \mathbf{y}_k)$ are strongly compatible and therefore the inductive hypothesis applies to $C(b_k, \eta \cdot \mathbf{y}_k)$ for $k \in 1..n$. On the other hand, if the last default branch $[\_]$ is selected then by definition of the shape analysis on `wait` we get back to the shape E and again the inductive hypothesis applies to $C(g(\mathbf{e}), \eta)$. The case where a pattern can be a variable is similar.

To conclude the proof we notice that for every function definition $f(\mathbf{x}) = be$, taking $\eta = \mathbf{x} = \mathbf{E}$ we have that (be, η, E) are strongly compatible and thus by lemma 42 the shape analysis succeeds on $C(be, \eta)$ starting from E. □