Lower-bound Time-complexity Analysis of Logic Programs

Andy King[1], **Kish Shen**[2] and **Florence Benoy**[1]
[1]University of Kent at Canterbury, CT2 7NF, UK. {a.m.king, p.m.benoy}@ukc.ac.uk
[2]University of Manchester, M13 9PL, UK. kish@cs.man.ac.uk

Abstract

The paper proposes a technique for inferring conditions on goals that, when satisfied, ensure that a goal is sufficiently coarse-grained to warrant parallel evaluation. The method is powerful enough to reason about divide-and-conquer programs, and in the case of quicksort, for instance, can infer that a quicksort goal has a time complexity that exceeds 64 resolution steps (a threshold for spawning) if the input list is of length 10 or more. This gives a simple run-time tactic for controlling spawning. The method has been proved correct, can be implemented straightforwardly, has been demonstrated to be useful on a parallel machine, and, in contrast with much of the previous work on time-complexity analysis of logic programs, does *not* require any complicated difference equation solving machinery.

1 Introduction

Automatic time-complexity analysis is useful to the programmer for algorithmic considerations but has a special rôle in the development of efficient parallel programs [9, 6, 7, 12, 15]. The execution of a parallel program can break down into processes which are too fine-grained for a multiprocessor. This can present a mismatch of granularity between the program and the multi-processor which, in turn, can degrade performance. Time-complexity analysis enables fine-grained processes to be identified and coalesced into more coarse-grained units at run-time in a fully automatically way. This can unburden the programmer from awkward, machine-dependent and error-prone tactical programming decisions like deciding which processes to spawn.

Automatic time-complexity analysis was first suggested as a way of controlling granularity for logic programs in [21] where a simple, heuristic-based analysis was proposed. The analysis, however, was crude and did not satisfactorily model recursive predicates. Recursive predicates present difficulties because the quantity of computation (and therefore the granularity) is data-dependent and is therefore difficult to determine at compile-time. Useful complexity information can still be derived, however, by automatically inferring complexity expressions formulated as functions on the size of the data [7]. Once the size of the data is known at run-time, the time-complexity (and therefore the granularity) can be simply calculated. Specifically, the size of the data can be checked against a threshold to determine whether or not the goal should be evaluated in parallel.

```
Qs(l, s) <- Qs(l, s, []).
Qs([], l, l).
Qs([x | xs], h, t) <-
    Pt(xs, x, l, g),
    Qs(l, h, [x | m]),
    Qs(g, m, t).
```

```
Pt([], _, [], []).
Pt([x | xs], m, [x | l], g) <-
    x ≤ m, Pt(xs, m, l, g).
Pt([x | xs], m, l, [x | g]) <-
    m < x, Pt(xs, m, l, g).
```

To illustrate, consider the quicksort predicate implemented with difference-lists, and suppose that the first argument of `Qs/3` is known to be input. This, for example, might have been inferred through mode analysis. (Note that Gödel notation is used throughout: variables are denoted by identifiers beginning with a lower case letter whereas constants begin with an upper case letter.) The time-complexity of a `Qs/3` goal, t, depends on the length, l, of its first argument. To be more precise, if time is measured by counting the number of resolution steps, then $t_{min}(l) \leq t \leq t_{max}(l)$ where $t_{min}(l)$ and $t_{max}(l)$ are the lower- and upper-bounds on the time-complexity,

$$t_{min}(0) = 1 \qquad t_{max}(0) = 1$$
$$t_{min}(l) = 1 + l + t_{min}(\lfloor \tfrac{l-1}{2} \rfloor) + t_{min}(\lceil \tfrac{l-1}{2} \rceil) \qquad t_{max}(l) = 1 + l + t_{max}(0) + t_{max}(l-1)$$

and $\lfloor . \rfloor$ and $\lceil . \rceil$ denote the floor and ceiling integer rounding functions. Since the input list is ground, we assume perfect indexing between the `Pt/4` clauses so that failing computation paths do not need to be considered. Granularity can be controlled by clamping $t_{min}(l)$ and $t_{max}(l)$ with closed-form expressions, $l(\lfloor \log_2(\frac{l}{3}) \rfloor - \frac{2}{3}) \leq t_{min}(l)$ and $t_{max}(l) = 1 + \frac{l(l+5)}{2}$ (either derived by hand or derived automatically) and then only sequentialising goals for which $t_{max}(l) \leq d_{max}$ [7] where d_{max} is granularity spawning threshold that depends on the underlying machine architecture which, for example, relates to the cost of forking a process. Another strategy for throttling the granularity is to only spawn goals with $d_{min} \leq t_{min}(l)$ where d_{min} is another machine dependent threshold.

The $t_{max}(l) \leq d_{max}$ method is the dual of the $d_{min} \leq t_{min}(l)$ strategy [9]. Interestingly, if $t_{max}(l)$ is not a tight upper bound on d_{max}, then the first technique can still spawn fine-grained tasks. Thus there is no guarantee that the first strategy will actually improve the performance of a parallel system. In an extreme situation a parallel system might actually run slower than an equivalent sequential system. On the other hand, if $t_{min}(l)$ is not a tight lower bound on d_{min}, then there is no guarantee that any processes will be spawned with the second method. Note, however, that the parallel system is unlikely to lead to slowdown. The practicality of either technique depends on the inequalities $t_{max}(l) \leq d_{max}$ and $d_{min} \leq t_{min}(l)$ being solved for useful, non-trivial values of l.

Our contribution is to show how the $d_{min} \leq t_{min}(l)$ inequality can be solved straightforwardly for useful, non-trivial values of l by bottom-up abstract interpretation. For example, with $d_{min} = 64$, our analysis can infer for `Qs/2` that if $t_{min}(l) \leq d_{min}$ then $l \leq 9$. Interpreted negatively, this means that if $10 < l$ then $64 = d_{min} < t_{min}(l)$. This is not an exercise in aesthetics but has a number of important and practical implications:

precision – our analysis can straightforwardly solve $d_{min} \leq t_{min}(l)$ for useful values of l even for a number of divide-and-conquer problems, including quicksort, which are difficult to reason about requiring, for example, extra analysis machinery in the difference equation approach [9].

implementation – in terms of practicality, our analysis builds on the argument-size analysis of [2] and, like the analysis described in [2], the analysis can be implemented straightforwardly in a language with constraint support. In fact the initial prototype analyser is less than 200 lines of code and took just two weeks to code and debug. Furthermore, the analysis does not require difference equation support to solve the equations that normally arise in time-complexity analysis [6, 7]. The analysis reduces to solving and projecting systems of constraints and machinery for these operations is provided and already implemented in systems like CLP($\mathcal{R}$) and SICStus version 3.

correctness – time-complexity analysis is potentially very complicated and therefore the correctness of an analysis is a real issue. For the analysis described in this paper, safety has been formally proved through abstract interpretation. In more pragmatic terms it means that the thresholding conditions inferred by the analysis guarantee that fine-grained processes are never spawned.

Note, however, that not spawning fine-grained processes is not always enough to guarantee a speedup (or even a no slowdown) since even the largest parallel machine will eventually saturate!

The exposition is structured as follows. Section 2 outlines the analysis with a worked example. Section 3 presents some preliminary theory. Sections 4 and 5 describe the transformation and the fixpoint calculation that make up the body of the analysis. Section 6 describes how an implementation of the analysis has been used on a parallel machine and sections 7 and 8 present the related and future work. Finally section 9 summarises the work. The paper assumes some familiarity with the s-semantics for CLP [3].

2 Worked example

Consider a time-complexity analysis for the predicate `Qs/2` where $d_{min} = 16$. Analysis divides into two stages: a fixpoint calculation that characterises how the time complexity relates to argument sizes; and a post-processing phase that infers conditions for the time complexity of a goal to exceed d_{min} resolution steps. By applying program transformation (abstract compilation [13, 14]) time complexity analysis can be recast as the problem of inferring invariants of a CLP($\mathcal{R}$) program. Analysis then, in effect, reduces to evaluating the concrete (bottom-up) semantics of the CLP($\mathcal{R}$) program. The `Qs/2` program listed below, for example, is a CLP($\mathcal{R}$) program that is obtained from `Qs/2` by a syntactic transformation in which each term in the first program is replaced by its size with respect to list length. Note, however, that the first argument of each predicate in the CLP($\mathcal{R}$) (abstract) program corresponds to a counter, d, that records the time-complexity. d is the sum of the resolution steps required to solve the body goals with an increment for the single resolution step implicit in goal-head unification. d is clamped by the constraint $d \leq d_{min}$ to ensure that goals whose time-complexity exceeds d_{min} are not considered in bottom-up evaluation.

All arguments but the first of an abstract predicate define an n-ary tuple of argument sizes. The n-tuple represents the sizes of the n arguments of the corresponding (concrete) predicate. Time-complexity analysis is performed by inferring relationships between the time argument and the size arguments of the n-tuple. Other measures of term size, for instance, term depth, can also be used [10, 19] to generate the abstract program.

```
Qs(d, l, s) <-
    d ≤ 16, d = 1 + d1,
    Qs(d1, l, s, 0).

Qs(d, 0, l, l) <- d ≤ 16, d = 1.
Qs(d, 1 + xs, h, t) <-
    d ≤ 16, d = 1 + d1 + d2 + d3,
    Pt(d1, xs, _, l, g),
    Qs(d2, l, h, 1 + m),
    Qs(d3, g, m, t).
```

```
Pt(d, 0, _, 0, 0) <-
    d ≤ 16, d = 1.
Pt(d, 1 + xs, m, 1 + l, g) <-
    d ≤ 16, d = 1 + d1,
    Pt(d1, xs, m, l, g).
Pt(d, 1 + xs, m, l, 1 + g) <-
    d ≤ 16, d = 1 + d1,
    Pt(d1, xs, m, l, g).
```

Suppose that the abstract program is denoted $P^{\mathcal{A}}$. The fixpoint phase of the analysis amounts to computing $T_{Lin,P^{\mathcal{A}}} \uparrow \omega = \cup_{i=0} T_{Lin,P^{\mathcal{A}}} \uparrow i$ where $T_{Lin,P^{\mathcal{A}}}$ is the immediate consequence operator of the s-semantics for CLP instantiated for Herbrand equations and linear inequations [3] and $T_{Lin,P^{\mathcal{A}}} \uparrow i+1 = T_{Lin,P^{\mathcal{A}}}(T_{Lin,P^{\mathcal{A}}} \uparrow i)$. Each iteration in the fixpoint calculation takes an $T_{Lin,P^{\mathcal{A}}} \uparrow i$, dubbed an interpretation, as input and generates an $T_{Lin,P^{\mathcal{A}}} \uparrow i+1$, as output. $T_{Lin,P^{\mathcal{A}}} \uparrow 0 = \emptyset$ is the empty interpretation. To compute $T_{Lin,P^{\mathcal{A}}} \uparrow i+1$, the body atoms of each clause of the program are unified with the atom abstractions in interpretation $T_{Lin,P^{\mathcal{A}}} \uparrow i$. Since $T_{Lin,P^{\mathcal{A}}} \uparrow 0$ is empty, however, $T_{Lin,P^{\mathcal{A}}} \uparrow 1$ will represent only those argument and time-complexity relationships embodied in the clauses of $P^{\mathcal{A}}$ that do not have user-defined body atoms.

$$T_{Lin,P^{\mathcal{A}}} \uparrow 1 = T_{Lin,P^{\mathcal{A}}} \uparrow 0 \cup \left\{ \mathtt{Qs}(1,0,x_2,x_3) \leftarrow x_2 = x_3. \quad \mathtt{Pt}(1,0,x_2,0,0) \leftarrow \mathit{true}. \right\}$$

The relationships asserts that if a `Qs/3` goal can be solved in one resolution step, then the first argument must (ultimately) be bound to `[]` and the second and third arguments must (ultimately) be of equal length. Similarly, for the `Pt/4` goal to solved in one step, the first, third and fourth must be bound to `[]`. Since the `Pt/4` and `Qs/2` predicates each have only one unit clause, only one abstract atom for `Pt/4` and `Qs/2` is included in $T_{Lin,P^{\mathcal{A}}} \uparrow 1$. In calculating $T_{Lin,P^{\mathcal{A}}} \uparrow 2$, however, two abstract atoms are generated from the two recursive clauses of `Pt/4`.

$$\left\{ \begin{array}{cc} \mathtt{Qs}(2,0,0) \leftarrow \mathit{true}. & \mathtt{Pt}(2,1,x_2,1,0) \leftarrow \mathit{true}. \\ \mathtt{Qs}(4,1,x_2,x_3) \leftarrow x_2 = 1 + x_3. & \mathtt{Pt}(2,1,x_2,0,1) \leftarrow \mathit{true}. \end{array} \right\}$$

To keep the size of each $T_{Lin,P^{\mathcal{A}}} \uparrow i$ small and manageable, the sets of inequalities for each predicate are collected and approximated by an over-estimate, the convex hull. The convex hull can itself be expressed as a single set of inequalities so that $T_{Lin,P^{\mathcal{A}}} \uparrow i$ needs only to maintain one set of inequalities for each predicate at each depth. For example, to calculate $T_{Lin,P^{\mathcal{A}}} \uparrow 2$ the convex hull is computed for two equation sets that define the argument sizes for `Pt/4` at depth 2, that is,

$$\text{hull}\left(\left\{ \begin{array}{c} x_1 = 1, x_3 = 1, \\ x_4 = 0 \end{array} \right\}, \left\{ \begin{array}{c} x_1 = 1, x_3 = 0, \\ x_4 = 1 \end{array} \right\} \right) = \left\{ \begin{array}{cc} x_1 = 1, & 0 \leq x_3, \\ x_3 \leq 1, & x_4 = 1 - x_3 \end{array} \right\}$$

The first equation set $x_1 = 1, x_3 = 1, x_4 = 0$ defines the second, fourth and fifth arguments in the first abstract `Pt/5` atom whereas $x_1 = 1, x_3 = 0, x_4 = 1$ defines the arguments in the second atom. The two equation sets are over-approximated with a single equation set thereby leading to

$$T_{Lin,P^{\mathcal{A}}} \uparrow 2 = T_{Lin,P^{\mathcal{A}}} \uparrow 1 \cup \left\{ \begin{array}{c} \mathtt{Qs}(2,0,0) \leftarrow \mathit{true}. \\ \mathtt{Qs}(4,1,x_2,x_3) \leftarrow x_2 = 1 + x_3. \\ \mathtt{Pt}(2,1,x_2,x_3,x_4) \leftarrow 0 \leq x_3, x_3 \leq 1, x_4 = 1 - x_3. \end{array} \right\}$$

Although the convex hull operation computes an approximation, useful argument size relationships are still preserved since the convex hull corresponds to the smallest convex space enclosing the spaces defined by the sets of inequalities. In the case of `Pt/4`, for example, $T_{Lin,P^{\mathcal{A}}} \uparrow 2$ asserts that if a `Pt/4` goal can be solved in exactly two resolution steps then the first argument is a list of length one, and the third and fourth arguments are lists of length either zero or one. (Interestingly, the convex hull operation often produces deep and unexpected argument size relationships [2].) The convex hull calculation is used in the ensuing iterates.

$$\begin{array}{ll} T_{Lin,P^{\mathcal{A}}} \uparrow 3 = T_{Lin,P^{\mathcal{A}}} \uparrow 2 \cup \left\{ \begin{array}{c} \mathtt{Qs}(5,1,1) \leftarrow \mathit{true}. \\ \mathtt{Qs}(8,2,x_2,x_3) \leftarrow x_2 = 2 + x_3. \\ \mathtt{Pt}(3,2,x_2,x_3,x_4) \leftarrow 0 \leq x_4, x_4 \leq 2, x_3 = 2 - x_4. \end{array} \right\} \\ \quad \ldots \\ T_{Lin,P^{\mathcal{A}}} \uparrow 16 = T_{Lin,P^{\mathcal{A}}} \uparrow 15 \cup \left\{ \mathtt{Pt}(16,15,x_2,x_3,x_4) \leftarrow 0 \leq x_4, x_4 \leq 15, x_3 = 15 - x_4 \right\} \\ T_{Lin,P^{\mathcal{A}}} \uparrow 17 = T_{Lin,P^{\mathcal{A}}} \uparrow 16 \end{array}$$

The iteration sequence will always converge within $d_{min} + 1$ iterations because of the $d \leq d_{min}$ constraints and since there are a finite number of clauses in the abstract program. Thus fixpoint termination techniques like widening are not required [4]. To use $T_{Lin,P^{\mathcal{A}}} \uparrow \omega$ to control spawning, however, we want to deduce conditions that guarantee that the time complexity of a goal exceeds d_{min} resolution steps. A bounding box approximation of $T_{Lin,P^{\mathcal{A}}} \uparrow \omega$ makes these conditions explicit.

$$\left\{ \begin{array}{l} \texttt{Qs}(d, x_1, x_2) \leftarrow \\ \quad 0 \leq d, \quad d \leq 14, \quad 0 \leq x_1, \quad x_1 \leq 3, \quad 0 \leq x_2, \quad x_2 \leq 3. \\ \texttt{Qs}(d, x_1, x_2, x_3) \leftarrow \\ \quad 0 \leq d, \quad d \leq 13, \quad 0 \leq x_1, \quad x_1 \leq 3, \quad 0 \leq x_2, \quad x_2 \leq 3. \\ \texttt{Pt}(d, x_1, x_2, x_3, x_4) \leftarrow \\ \quad 0 \leq d, \quad d \leq 16, \quad 0 \leq x_1, \quad x_1 \leq 15, \quad 0 \leq x_3, \quad x_3 \leq 15, \quad 0 \leq x_4, \quad x_4 \leq 15. \end{array} \right\}$$

Note how each argument, including d, is approximated as an interval so that the argument sizes are represented as a box in the space $\mathbb{R}^n$. The abstraction asserts (among other things) that if the time-complexity of a `Qs/2` goal is less or equal to d_{min} steps, then the first and second arguments must ultimately be bound to lists with a length of less than four. Put another way, if the length of argument is known to be greater or equal to four, then the computation must either exceed d_{min} resolution steps or fail. Possible failure (or equivalently definite non-failure) can be detected with a query-dependent non-failure analysis [8]. Thus if the program is queried with a `Qs/2` goal where the first argument is known to be a list of integers, say, then non-failure can be deduced [8]. Hence, if the argument is also known to have a length of greater or equal to four, then the goal is guaranteed to lead to a computation that exceeds d_{min} resolution steps.

3 Preliminaries

Syntax of logic programs Let $Func$, $Pred$ and Var respectively denote the set of function symbols, predicate symbols and a denumerable set of variables. The non-ground term algebra over $Funct$ and Var is denoted $Term$, where the set of atoms constructed from the predicate symbols $Pred$ is denoted $Atom$. A goal is a sequence of atoms. A logic program is a finite set of clauses. A clause has the form $h \leftarrow \vec{b}$ where h, the head, is a atom and $\vec{b}$, the body, is a finite sequence of atoms. Also $var(o)$ denotes the set of variables in a syntactic object o, :: denotes concatenation, whereas $\pi_i(.)$ denotes vector projection, that is, $\pi_i(\langle x_1, \ldots, x_n \rangle) = x_i$.

The set of idempotent substitutions from Var to $Term$ is denoted Sub and the set of renamings (which are bijective substitutions) is denoted Ren. A substitution ϕ will sometimes be represented as a finite set of pairs $\phi = \{u_1 \mapsto t_1, \ldots, u_n \mapsto t_n\}$. Sub and Ren extend in the usual way from functions from variables to terms, to functions from terms to terms, to functions from atoms to atoms, and to functions from clauses to clauses. Syntactic objects, o and o', are variants of one another, denoted $o \approx o'$, if there exists $\rho \in Ren$ such that $\rho(o) = o'$. The equivalence class of o under $\approx$ is denoted $[o]_{\approx}$. The restriction of a substitution ϕ to a set of variables U and the composition of two substitutions ϕ and φ, are denoted by $\phi \restriction U$ and $\phi \circ \varphi$ respectively, and defined such that: $\phi \restriction U = \{u \mapsto t \in \phi \mid u \in U\}$ and $(\phi \circ \varphi)(u) = \phi(\varphi(u))$.

An equation is an equality constraint of the form $a = b$ where a and b are terms or atoms. Let Eqn denote the set of finite sets of equations. There is a natural mapping from substitutions to equations, that is, $eqn(\phi) = \{u = t \mid u \mapsto t \in \phi\}$, and $mgu(E)$ denotes the set of most general unifiers for an equation set E.

Operational semantics of logic programs An operational semantics is introduced to argue correctness. The semantics is described in terms of a transition system that defines reductions between states. The set of states is defined by $State = Atom^* \times Sub$.

Definition 3.1 Let P be a logic program. The transition system $\langle State, \rightarrow \rangle$ where $\rightarrow \subseteq State \times State$ is the least relation such that

$$s \rightarrow s' \Leftrightarrow \begin{cases} s = \langle \vec{a}, \phi \rangle \quad \wedge \quad h \leftarrow \vec{b} \in P \quad \wedge \quad h \leftarrow \vec{b} \approx h' \leftarrow \vec{b}' & \wedge \\ var(h' \leftarrow \vec{b}') \cap var(s) = \emptyset \quad \wedge \quad \varphi = mgu(\{\phi(a_i) = h'\}) & \wedge \\ s' = \langle \langle a_1, \ldots, a_{i-1} \rangle :: \vec{b}' :: \langle a_{i+1}, \ldots, a_n \rangle, \varphi \circ \phi \rangle \end{cases}$$

The notion of answer for a goal g are defined in terms of a transition system. Depth corresponds to the number of resolution steps and is used as a measure of computational complexity, that is, if $\langle \vec{a}_1, \phi_1 \rangle \rightarrow \langle \vec{a}_2, \phi_2 \rangle \rightarrow \langle \vec{a}_3, \phi_3 \rangle \ldots$ then $\langle \vec{a}_1, \phi_1 \rangle \rightarrow^d \langle \vec{a}_{1+d}, \phi_{1+d} \rangle$.

Definition 3.2 (answers and partial answers at a depth)

- A goal g has a partial answer g' at depth d iff $\langle g, \epsilon \rangle \rightarrow^d \langle g', \phi \rangle$ and $g' \approx \phi(g)$;
- A goal g has an answer g' at depth d iff $\langle g, \epsilon \rangle \rightarrow^d \langle true, \phi \rangle$ and $g' \approx \phi(g)$.

Fixpoint s-semantics of constraint logic programs The semantics of the abstract program is formalised in terms of the concrete s-style semantics for constraint logic programs [3] and therefore, to make the paper reasonably self-contained, the semantics is summarised below. The semantics is parameterised over a computational domain, C, of constraints. We write $c \models c'$ iff c entails c' and also $c = c'$ iff $c \models c'$ and $c' \models c$. The interpretation base B_C for the language defined by a program P is the set of unit clauses of the form $p(\vec{x}) \leftarrow c$ quotiented by equivalence. Equivalence, again denoted $\approx$, is defined by: $p(\vec{x}) \leftarrow c \approx p(\vec{x'}) \leftarrow c'$ iff $c \restriction var(\vec{x}) = (c' \wedge (\vec{x} = \vec{x'})) \restriction var(\vec{x})$ where $\restriction$ denotes projection. If C is the domain of equations over Herbrand terms, $Herb$ say, then $\approx$ is variance and $\restriction$ is restriction. The fixpoint semantics of a program P is defined in terms of an immediate consequence operator like so: $\mathcal{F}_C[\![P]\!] = lfp(T_{C,P})$.

Definition 3.3 (fixpoint s-semantics for CLP [3]) The immediate consequence operator $T_{C,P} : B_C \rightarrow B_C$ is defined by:

$$T_{C,P}(I) = \left\{ [p(\vec{x}) \leftarrow c']_{\approx} \;\middle|\; \begin{array}{ll} w \in P \quad \wedge \quad w = p(\vec{t}) \leftarrow c, p_1(\vec{t_1}), \ldots, p_n(\vec{t_n}) & \wedge \\ [w_i]_{\approx} \in I \quad \wedge \quad w_i = p_i(\vec{x}_i) \leftarrow c_i & \wedge \\ \forall i. var(w) \cap var(w_i) = \emptyset \quad \wedge \quad \forall i \neq j. var(w_i) \cap var(w_j) = \emptyset & \wedge \\ c' = \wedge_{i=1}^{n} (\vec{x}_i = \vec{t}_i \wedge c_i) \wedge (\vec{x} = \vec{t}) \wedge c \quad \wedge \quad c' \text{ is solvable} \end{array} \right\}$$

For the abstract programs of the analysis the domain C is Lin, that is, sets of (non-strict) inequalities between linear expressions and equations between Herbrand terms. Thus, for example, $\{f(a) = f(b), x \leq y + z\} \in Lin$.

Fixpoint depth semantics for logic programs Correctness of the analysis is argued in terms of the depth semantics of [1] since, although it was originally devised to reason about termination, the semantics also expresses a natural notion of complexity. Again, to keep the paper self-contained, we briefly summarise the relevant aspects of the depth semantics [1]. The interpretation base, denoted B_{Time} for clarity, is the set of depth and clause pairs where clauses are quotiented by variance, that is, $\langle d, [h \leftarrow \vec{b}]_{\approx} \rangle$. Informally, the pair $\langle d, [h \leftarrow \vec{b}]_{\approx} \rangle$

represents a partial (incomplete) computation from the atomic goal h to the goal $\vec{b}$ in d steps. Empty partial computations correspond to the set $\Phi_P = \{\langle 0, [p(\vec{x}) \leftarrow p(\vec{x})]_\approx \rangle \mid p \in Pred\}$ [1]. The set of partial answers for a depth d can be characterised with another immediate consequence operator $T_{Time,P}$.

Definition 3.4 (fixpoint clausal semantics with depth [1]) The immediate consequence operator $T_{Time,P} : B_{Time} \rightarrow B_{Time}$ is defined by:

$$T_{Time,P}(I) = \left\{ \langle d, [\varphi(h \leftarrow \vec{b_1} :: \ldots :: \vec{b_n})]_\approx \rangle \;\middle|\; \begin{array}{ll} w \in P & \wedge\, w = h \leftarrow \vec{b} \quad \wedge \\ \langle d_i, [w_i]_\approx \rangle \in I \cup \Phi_P & \wedge\, w_i = h_i \leftarrow \vec{b_i} \wedge \\ \forall i.var(w) \cap var(w_i) = \emptyset & \qquad\qquad \wedge \\ \forall i \neq j.var(w_i) \cap var(w_j) = \emptyset & \qquad\qquad \wedge \\ d = 1 + \sum_{i=1}^{n} d_i & \qquad\qquad \wedge \\ \varphi = mgu(\{\vec{b} = \langle h_1, \ldots, h_n \rangle\}) & \end{array} \right\}$$

$T_{Time,P}$ is continuous and defines the fixpoint semantics of a program P like so $\mathcal{F}_{Time}[\![P]\!] = lfp(T_{Time,P})$. $\mathcal{F}_{Time}[\![P]\!]$ is consistent with $\mathcal{F}_{Herb}[\![P]\!]$ in that $\mathcal{F}_{Herb}[\![P]\!] = \{[h]_\approx \mid \langle d, [h \leftarrow true]_\approx \rangle \in \mathcal{F}_{Time}[\![P]\!]\}$ [1]. The following theorem, adapted from [1], formally asserts the relationship between partial answers and the fixpoint semantics.

Theorem 3.1 Let P be a logic program. A goal g is a partial answer at depth d for g' iff there exists $\langle d_i, [h_i \leftarrow true]_\approx \rangle \in \mathcal{F}_{Time}[\![P]\!]$ such that $\forall i.var(h_i) \cap var(g') = \emptyset$ and $\forall i \neq j.var(h_i) \cap var(h_j) = \emptyset$, $\varphi = mgu(\{g' = \vec{h}\})$, $g \approx \varphi(g')$ and $d = \sum_{i=1} d_i$.

4 Abstract compilation

By applying program transformation (abstract compilation [13, 14]) the problem of inferring how time complexity depends on argument size is recast as the problem of inferring the invariants of a CLP($\mathcal{R}$) program. Our transformation is dubbed α. Size, as usual [10, 19], is expressed in terms of norms that map terms to (possibly non-ground) constraint in Lin. In the case of the list length norm [22, 19], for example, $\|[]\|_{\text{leng}} = 0$, $\|[x]\|_{\text{leng}} = 1$ and $\|[x|y]\|_{\text{leng}} = 1 + y$. In addition, to ensure that the norm is always defined, if t cannot be instantiated to a list we define $\|t\|_{\text{leng}} = z$ where z is a free (fresh) variable.

Definition 4.1 (program abstraction α)

$$\begin{array}{l} \alpha[\![w_1, \ldots, w_m]\!] = \\ \quad \alpha_{\text{clause}}[\![w_1]\!], \ldots, \alpha_{\text{clause}}[\![w_m]\!] \end{array}$$

$$\begin{array}{l} \alpha_{\text{clause}}[\![p(\vec{t}) \leftarrow p_1(\vec{t_1}), \ldots, p_m(\vec{t_m})]\!] = \\ \quad \left\{ \begin{array}{l} p(d :: \vec{x}) \leftarrow \\ \quad d \leq d_{min}, d = 1 + \sum_{i=1}^{m} d_i, \\ \quad \alpha_{\text{eqns}}[\![eqn(mgu(\vec{x} = \vec{t} \wedge \vec{x_1} = \vec{t_1} \wedge \ldots \wedge \vec{x_m} = \vec{t_m}))]\!], \\ \quad p_1(d_1 :: \vec{x}_1), \ldots, p_m(d_m :: \vec{x}_m) \end{array} \right. \end{array}$$

$$\begin{array}{l} \alpha_{\text{eqns}}[\![e_1, \ldots, e_m]\!] = \\ \quad \alpha_{\text{eqn}}[\![e_1]\!], \ldots, \alpha_{\text{eqn}}[\![e_m]\!] \end{array}$$

$$\begin{array}{l} \alpha_{\text{eqn}}[\![x = t]\!] = \\ \quad \left\{ \begin{array}{l} x = \vec{y}, \\ \quad x_1 = \vec{y_1}, \ldots, x_m = \vec{y_m}, \\ \quad \|\phi_1(x)\|_1 = \|\phi_1(t)\|_1, \ldots, \|\phi_n(x)\|_n = \|\phi_n(t)\|_n \end{array} \right. \end{array}$$

where w_i and e_i respectively denote a clause and an equation, $var(t) = \{x_1, \ldots, x_m\}$, $\phi_i = \{x \mapsto \pi_i(\vec{y}), x_1 \mapsto \pi_i(\vec{y_1}), \ldots, x_m \mapsto \pi_i(\vec{y_m})\}$ and the variables d, d_i and vectors of variables $\vec{x}$, $\vec{x}_i$, $\vec{y}$ and $\vec{y}_i$ are fresh and distinct. The arities of $\vec{y}$ and $\vec{y}_i$ are both n.

Note that the transform is parameterised by the machine dependent granularity constant d_{min} and the n norms $\|.\|_1, \ldots, \|.\|_n$. Multiple norms are useful when a unique norm cannot be matched to an argument position, for example, because of a lack of type declarations or because a type analysis is imprecise. The worked example corresponds to a (simplified) special case for when $n = 1$ and $\|.\|_1 = \|.\|_{\text{leng}}$. Abstracting equations, α_{eqn}, is the most subtle part of the program abstraction α and so example 4.1 illustrates how α_{eqn} is applied.

Example 4.1 Consider $\alpha_{\text{eqn}}[\![x = [x_1|x_2]]\!]$ when $n = 2$ and in particular $\|.\|_1 = \|.\|_{\text{leng}}$. and $\|.\|_2 = \|.\|_{\text{size}}$ where $\|.\|_{\text{size}}$ counts the number of function symbols in a term. If $\vec{y} = \langle y_1, y_2\rangle$, $\vec{y_1} = \langle y_{1,1}, y_{1,2}\rangle$ and $\vec{y_2} = \langle y_{2,1}, y_{2,2}\rangle$ then $\phi_1 = \{x \mapsto y_1, x_1 \mapsto y_{1,1}, x_2 \mapsto y_{2,1}\}$ and similarly $\phi_2 = \{x \mapsto y_2, x_1 \mapsto y_{1,2}, x_2 \mapsto y_{2,2}\}$ so that $(\|\phi_1(x)\|_1 = \|\phi_1([x_1|x_2])\|_1)$ = $(\|y_1\|_{\text{leng}} = \|[y_{1,1}|y_{2,1}]\|_{\text{leng}})$ = $(y_1 = 1 + y_{2,1})$ and $(\|\phi_2(x)\|_2 = \|\phi_2([x_1|x_2])\|_2)$ = $(\|y_2\|_{\text{size}} = \|[y_{1,2}|y_{2,2}]\|_{\text{size}})$ = $(y_2 = 1 + y_{1,2} + y_{2,2})$. Hence

$$\alpha_{\text{eqn}}[\![x = [x_1|x_2]]\!] = \left\{ \begin{array}{lll} x = \langle y_1, y_2\rangle, & x_1 = \langle y_{1,1}, y_{1,2}\rangle, & x_2 = \langle y_{2,1}, y_{2,2}\rangle, \\ y_1 = 1 + y_{2,1}, & y_2 = 1 + y_{1,2} + y_{2,2} & \end{array} \right\}$$

In practise, the abstract programs generated by α tend to include equations that can be eliminated, combined or simplified. Since the clauses of $\alpha[\![P]\!]$ are used repeatedly to compute a fixpoint, we have found it beneficial to simplify $\alpha[\![P]\!]$ in a partial evaluation (local simplification) phase that precedes the fixpoint calculation.

Example 4.2 Consider the `Leng/2` predicate, listed in the left-hand column, which computes the length of a list. Its (partially evaluated) abstract program is listed in the right-hand column. The two norms are $\|.\|_{leng}$ and $\|.\|_{num}$ where the latter gives the numeric value of an integer. By using both norms together useful time complexity can often be inferred even in an absence of type information [10, 19]. Our prototype analyser, for example, does not perform type analysis and simply measures size with a set of pre-defined norms.

`Leng([], 0).`
`Leng([_ | ys], l) <-`
`    Leng(ys, ls),`
`    l = ls + 1.`
	`Leng(1, ⟨0,_⟩, ⟨_,0⟩).`
	`Leng(d, ⟨z1,_⟩, ⟨_,z2⟩) <-`
	`    d ≤ dmin, d = 2 + d1, z1 = 1 + z3, z2 = 1 + z4,`
	`    Leng(d1, ⟨z3,_⟩, ⟨_,z4⟩).`

Note that the depth equation $d = 2 + d_1$ reflects the presence of the builtin `=/2` in the clause. Each builtin requires one addition resolution step. The partial evaluation phase has applied the equations to the head and body of the clause to reduce the numbers of equations that have to be solved at analysis time. This explains, for example, why the arguments of the heads are not variables.

To formalise the relationship between a concrete program and its abstract program, the concretisation mapping is introduced.

Definition 4.2 (γ) Concretisation $\gamma : \wp(B_{Lin}) \to \wp(B_{Time})$ is defined by:

$$\gamma(I) = \left\{ \langle d, [p(\vec{t})]_\approx\rangle \;\middle|\; \begin{array}{l} [p(x' :: \vec{x}) \leftarrow c \wedge (\wedge_{i=1} \pi_i(\vec{x}) = \vec{y_i})]_\approx \in I \quad \wedge \\ x' = d \wedge (\wedge_{i=1} \wedge_{j=1}^{n} \pi_j(\vec{y_i}) = \|t_i\|_j) \;\models c \end{array} \right\}$$

Example 4.3 Suppose $n = 2$ where $\|.\|_1 = \|.\|_{\text{leng}}$. and $\|.\|_2 = \|.\|_{\text{num}}$. If $c = (x' = y_{1,1} + 1) \wedge (x_1 = \langle y_{1,1}, y_{1,2}\rangle) \wedge (y_{2,2} = y_{1,1}) \wedge (x_2 = \langle y_{2,1}, y_{2,2}\rangle)$ then

$$\begin{array}{l} \gamma(\{[\mathtt{Leng}(x', x_1, x_2) \leftarrow c]_\approx\}) = \\ \quad \{\langle d, [\mathtt{Leng}(t_1, t_2)]_\approx\rangle \mid d = |t_1|_{leng} + 1 \wedge |t_2|_{num} = |t_1|_{leng}\} \end{array}$$

The concretisation mapping is used to link $\mathcal{F}_{Time}[\![P]\!]$ with $\mathcal{F}_{Lin}[\![\alpha[\![P]\!]]\!]$ in the following safety theorem. The theorem explains how the abstract program can be used to characterise the time behaviour of the concrete program.

Theorem 4.1 (safety I)

$$\{\langle d, [h \leftarrow true]_\approx\rangle \in \mathcal{F}_{Time}[\![P]\!] \mid d \leq d_{min}\} \subseteq \gamma(\mathcal{F}_{Lin}[\![\alpha[\![P]\!]]\!])$$

Because each clause in the abstract program includes the constraint $d \leq d_{min}$, $\mathcal{F}_{Lin}[\![\alpha[\![P]\!]]\!]$ can be finitely computed within $d_{min} + 1$ iterations. Thus termination techniques, like widening [4], are not required to induce iteration. Finally, the corollary relates $\mathcal{F}_{Lin}[\![\alpha[\![P]\!]]\!]$ to the operational semantics.

Corollary 4.1 (safety II) Let P be a logic program. If an atomic goal g had an answer g' at depth d then there exists $\langle d, [h \leftarrow true]_\approx\rangle \in \gamma(\mathcal{F}_{Lin}[\![\alpha[\![P]\!]]\!])$ such that $var(h) \cap var(g) = \emptyset$, $\varphi = mgu(\{g = h\})$ and $g' \approx \varphi(g)$.

5 Fixpoint computation

Although $\mathcal{F}_{Lin}[\![\alpha[\![P]\!]]\!]$ can always be computed within $d_{min} + 1$ iterations, the number of atoms in an interpretation (iterate) can become large. Thus, to constrain the growth of interpretations, the sets of inequalities for each predicate are collected together and approximated by their convex hull. To be more precise, the convex hull is used to over-approximate the argument sizes for atoms at the same depth.

Example 5.1 Returning to the worked example, recall that the convex hull operation collapses together the constraints for the two argument relationships for depth 2

$$\text{hull}\left(\left\{\begin{array}{l} [\mathtt{Pt}(x' :: \vec{x}) \leftarrow x' = 1, x_1 = 0, x_3 = 0, x_4 = 0]_\approx \\ [\mathtt{Pt}(x' :: \vec{x}) \leftarrow x' = 2, x_1 = 1, x_3 = 1, x_4 = 0]_\approx \\ [\mathtt{Pt}(x' :: \vec{x}) \leftarrow x' = 2, x_1 = 1, x_3 = 0, x_4 = 1]_\approx \end{array}\right\}\right) =$$
$$\left\{\begin{array}{l} [\mathtt{Pt}(x' :: \vec{x}) \leftarrow x' = 1, x_1 = 0, x_3 = 0, x_4 = 0]_\approx \\ [\mathtt{Pt}(x' :: \vec{x}) \leftarrow x' = 2, x_1 = 1, 0 \leq x_3, x_3 \leq 1, x_4 = 1 - x_3]_\approx \end{array}\right\}$$

The hull operator is defined in terms of convex hull operation on sets of constraint sets.

Definition 5.1 (hull) The approximation operator hull : $B_{Lin} \to B_{Lin}$ is defined by:

$$\text{hull}(I) = \{[p(x' :: \vec{x}) \leftarrow c_{p,n}]_\approx \mid p \in Pred \wedge n \in \mathbb{N}\}$$

where $c_{p,n} = \text{hull}_{var(x'::\vec{x})}(\{c \restriction var(x' :: \vec{x}) \mid [p(x' :: \vec{x}) \leftarrow c]_\approx \in I \wedge c \models (x' = n)\})$, $\text{hull}_X(\emptyset) = false$ and $\text{hull}_X(\{c_1, \ldots, c_n\}) = \text{hull}_X(c_1, \text{hull}_X(\{c_2, \ldots, c_n\}))$.

The binary hull_X can be computed straightforwardly with a relaxation adapted from disjunctive constraint logic programming [5]. For simplicity, consider calculating

$\text{hull}_{var(\vec{x})}(c_1, c_2)$ where $\vec{x}$ is an n-ary vector and the constraints c_i are represented in standard form $A_i\vec{x} \leq \vec{b_i}$, where A_i is an $m \times n$ matrix and $\vec{b_i}$ in an m-ary vector. The convex hull of the spaces defined by c_1 and c_2 can be computed by:

$$\left\{ \begin{array}{c} \vec{x} = \vec{x_1} + \vec{x_2} \ \wedge \ \sigma_1 + \sigma_2 = 1 \ \wedge \\ A_1\vec{x_1} \leq \sigma_1\vec{b_1} \wedge A_2\vec{x_2} \leq \sigma_2\vec{b_2} \wedge \\ -\sigma_1 \leq 0 \quad \wedge \quad -\sigma_2 \leq 0 \end{array} \right\} \upharpoonright var(\vec{x})$$

Since the system of inequations is linear, the convex hull can be calculated by simply imposing the equations on the store of a constraint language and then applying projection [2].

Example 5.2 Continuing with example 5.1, combining the $x_1 = 1, x_3 = 1, x_4 = 0$ and $x_1 = 1, x_3 = 0, x_4 = 1$ equations for the `Pt/5` atoms amounts to solving:

$$\left\{ \begin{array}{ccc} \vec{x} = \vec{x_1} + \vec{x_2} & \wedge & \sigma_1 + \sigma_2 = 1 \qquad \wedge \\ \begin{bmatrix} 1, 0, & 0, & 0 \\ -1, 0, & 0, & 0 \\ 0, 0, & 1, & 0 \\ 0, 0, & -1, & 0 \\ 0, 0, & 0, & 1 \\ 0, 0, & 0, & -1 \end{bmatrix} \vec{x_1} \leq \sigma_1 \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \\ 0 \\ 0 \end{bmatrix} & \wedge & \begin{bmatrix} 1, 0, & 0, & 0 \\ -1, 0, & 0, & 0 \\ 0, 0, & 1, & 0 \\ 0, 0, & -1, & 0 \\ 0, 0, & 0, & 1 \\ 0, 0, & 0, & -1 \end{bmatrix} \vec{x_2} \leq \sigma_2 \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0 \\ 1 \\ -1 \end{bmatrix} \wedge \\ -\sigma_1 \leq 0 & \wedge & -\sigma_2 \leq 0 \end{array} \right\} \upharpoonright var(\vec{x})$$
$$= \{\ x_1 = 1, 0 \leq x_3, x_3 \leq 1, x_4 = 1 - x_3\ \}$$

The post-processing phase of the analysis boils down to computing a bounding box abstraction for the fixpoint that defines the maximum and minimum sizes of the arguments that can occur for goals with a complexity between 1 and d_{min} resolution steps. The bounding box approximation is the obvious lifting of a bounding box operator on sets of constraint sets to interpretations.

Definition 5.2 (box) The approximation operator box : $B_{Lin} \to B_{Lin}$ is defined by:

$$\begin{array}{l} \text{box}(I) = \{[p(\vec{x}) \leftarrow c_p]_{\approx} \,|\, p \in Pred\} \\ \qquad \text{where } c_p = \text{box}_{var(\vec{x})}(\{c \upharpoonright var(\vec{x}) \,|\, [p(\vec{x}) \leftarrow c]_{\approx} \in I\}) \end{array}$$

As with the convex hull, $\text{box}_X(\emptyset) = false$. There are several ways of calculating box_X. One tactic that can be coded very simply in a constraint language offering projection and an entailment check is to use $\text{box}_{var(\vec{x})}(c, c') = \wedge_{i=1}(\{e \,|\, e \in c \upharpoonright var(\pi_i(\vec{x})) \wedge c' \models e\} \cup \{e' \,|\, e' \in c' \upharpoonright var(\pi_i(\vec{x})) \wedge c \models e'\})$. Note that c and c' are themselves regarded as sets of inequations e and e'. The final safety theorem states that the convex hull and bounding box approximations do not compromise safety. When combined with the earlier safety results, the theorem gives an efficient way of characterising fine-grained goals.

Theorem 5.1 (safety III)

$$\gamma(\mathcal{F}_{Lin}[\![P^{\mathcal{A}}]\!]) \subseteq \gamma(\text{box}(\text{hull}(T_{Lin,P^{\mathcal{A}}}) \uparrow \omega))$$

Thresholding tests can then be inferred to test whether the input arguments of a goal permit the goal to be a member of $\gamma(\text{box}(\text{hull}(T_{Lin,P^{\mathcal{A}}}) \uparrow \omega))$ and therefore possibly a member of $\{\langle d, [h \leftarrow true]_{\approx}\rangle \in \mathcal{F}_{Time}[\![P]\!] \,|\, d \leq d_{min}\}$. If not, then the goal must either lead to a computation that exceeds d_{min} steps or the goal must eventually fail. Input arguments can be deduced with mode analysis whereas the non-failing goals can be detected by non-failure analysis [8]. Thus the program can be annotated with granularity thresholding tests that ensure that goals are only spawned when their granularity is guaranteed to exceed d_{min}.

6 Experimental results

The purpose of the experiment presented here is to study the effect of different granularity sizes has on many programs, under different configurations of queries, and number of processors used to run the program. The analysis was implemented in SICStus Prolog, and used to infer thresholding tests for grains sizes of 16, 64, 256 and 1024 resolution steps for the Fibonacci, Hanoi and quicksort programs. `Fib(`n`, `f`)` calculates the n'th Fibonacci number f; `Hanoi(`n`, `l`)` computes a list of moves, l, for n disks in the towers of Hanoi problem; and `Qsort(`l`, `s`)` quicksorts a random list l of length n to give s. Hanoi and Fibonacci are good candidates for granularity control since the parallelism is fine-grained whereas quicksort is less predictable generating both fine-grained and course-grained processes. The programs were hand annotated with the thresholding tests, and then timings where taken on a Sequent Symmetry for 1, 2, 4 and 9 processors. We have used similar benchmark programs to [12], and the same 20MHz 80386 processor Sequent. The and-parallel Prolog system DASWAM [20] was used. The programs used were limited to independent and-parallelism, because suspension complicates the granularity question for general dependent and-parallelism. The programs were executed with different queries, which affected the execution times of the program, but not the relationship between threshold and grain-sizes.

#	none	16	64	256	1024
	fib(17), 1108.1±0.3				
1	1702.9±1.4	1688.2±4.4	1338.0±4.5	1177.1±1.4	1139.3±0.9
2	862.4±5.6 (1.97×)	848.8±4.0 (1.99×)	670.1±1.8 (2.00×)	589.9±1.4 (2.00×)	583.8±12.3 (1.95×)
4	440.2±1.6 (3.87×)	429.6±1.1 (3.92×)	341.1±1.1 (3.92×)	300.5±2.1 (3.92×)	331.1±0.2 (3.44×)
9	203.4±1.2 (8.37×)	201.0±3.4 (8.40×)	156.9±0.2 (8.53×)	140.5±0.2 (8.38×)	167.9±1.0 (6.79×)
	fib(19), 2898.5±0.4				
1	4470.8±1.9	5048.4±0.2	3476.1±3.3	3040.4±3.0	2949.2±1.1
2	2257.5±7.1 (1.98×)	2544.4±4.3 (1.98×)	1761.1±1.9 (1.97×)	1536.9±1.3 (1.98×)	1483.2±0.2 (1.99×)
4	1140.0±4.6 (3.92×)	1277.9±1.8 (3.95×)	882.1±1.7 (3.94×)	775.9±2.0 (3.92×)	766.2±6.5 (3.85×)
9	516.5±1.0 (8.66×)	577.8±2.0 (8.74×)	398.8±17.0 (8.72×)	351.4±2.5 (8.65×)	365.4±6.2 (8.07×)
	hanoi(10), 441.7±0.3				
1	727.0±4.0	522.0±2.6	466.3±1.4	454.5±0.4	453.1±0.1
2	363.1±2.0 (2.00×)	260.2±0.1 (2.01×)	230.9±0.2 (2.02×)	224.4±0.6 (2.02×)	223.1±0.1 (2.03×)
4	185.7±2.3 (3.92×)	132.1±0.2 (3.95×)	117.4±0.3 (3.97×)	114.5±0.4 (3.97×)	223.4±0.8 (2.03×)
9	87.8±1.1 (8.28×)	63.9±0.4 (8.17×)	60.6±0.3 (7.69×)	59.5±0.1 (7.63×)	224.2±1.9 (2.02×)
	hanoi(16), 28061.6±11.6				
1	46509.5±60.8	33054.6±11.5	29391.0±12.8	28656.4±14.8	28323.3±4.8
2	23210.6±138.1 (2.00×)	16522.3±2.7 (2.00×)	14730.9±4.2 (2.00×)	14273.7±3.4 (2.01×)	14322.9±16.5 (1.98×)
4	11594.8±15.3 (4.01×)	8265.5±9.8 (4.00×)	7376.2±4.5 (3.98×)	7147.9±2.7 (4.01×)	7092.9±4.3 (3.99×)
9	5209.6±14.5 (8.93×)	3694.6±3.4 (8.95×)	3287.6±2.1 (8.94×)	3191.9±2.4 (8.98×)	3298.3±2.8 (8.59×)
	qsort(300), 816.8±1.5				
1	909.5±0.1	912.6±2.9	888.9±2.3		
2	509.7±0.6 (1.78×)	512.7±3.2 (1.78×)	498.5±0.7 (1.78×)		
4	330.6±11.8 (2.75×)	334.2±9.0 (2.73×)	332.3±17.8 (2.68×)		
9	272.1±0.6 (3.34×)	274.8±0.9 (3.32×)	278.0±0.8 (3.20×)		
	qsort(3200), 12239.1±3.1				
1	13474.8±29.9	14083.9±3.2	13197.2±2.4		
2	7452.1±3.5 (1.81×)	7701.4±3.2 (1.83×)	7337.8±1.4 (1.80×)		
4	4822.6±9.8 (2.79×)	4925.5±26.5 (2.86×)	4771.9±19.9 (2.77×)		
9	3724.0±10.1 (3.62×)	3766.6±29.8 (3.74×)	3741.6±4.7 (3.53×)		

The table summarises our results. Timings, in milliseconds, were averaged over five runs and are given with the standard deviation. Entries are not given for quicksort for grain sizes of 256 and 1024 because the prototype analyser cannot infer the thresholds within a minute and, we believe that for an optimisation to be practical, it should be reasonably fast. The problem stems from the repeated computations in the fixpoint calculation. We believe that this overhead can be removed by considering the strongly connected components of the call graph of the program. More usually, thresholds can be inferred within a minute even for the larger grain sizes for the `benchmark3.tar.Z` programs obtained from UPM Madrid.

The execution time for DASWAM running in sequential mode is given in the headings. The results confirm those of [12], showing that granularity control is useful even for a Sequent Symmetry which has relatively low task creation overheads. The results show, as expected, that controlling the granularity has two main effects:

- It reduces the total execution time for the program by reducing the frequency of parallel execution and thus parallel overheads. The larger the granularity threshold, the smaller the parallel overhead. The limit is the sequential case, with no parallel overhead at all.
- It reduces the amount of available parallelism. The larger the granularity threshold, the lesser the available parallelism.

Reducing the parallel overhead tends to improve the total amount of computation (work and overhead), but at the same time, it reduces parallelism. These two factors need to be balanced to give the best results. For any program, the best granularity size can be affected by the particular query being solved and the number of workers the system is using. In addition, the best size changes from program to program, and we also expect it to change from system to system. For some programs, such as quicksort, the overhead of testing for the threshold can be sufficiently expensive so that it actually degrades performance instead of improving it. Thus what is best for one configuration is not necessarily best for another.

It may be possible to take some of these factors into account (such as the type of threshold test being performed), but some factors cannot be controlled, such as what query the user want to solve, and to a lesser extent, how many workers the user choose to use. Thus, a compromise threshold has to be chosen that works well (but not best) for a range of configurations. Looking at the results in general, if the grain size is set too high, say 1024 resolution steps, then the granularity control mechanism limits the parallelism to the extent that the processors are not properly utilised. hanoi(10) on 9 processors is one extreme example. Conversely, if the grain size is set too low, say 16 resolution steps, then the cost of the threshold check is not repaid by reduced task creation, so the overall performance is worse with granularity control than without. The grain size should thus balance machine utilisation against reduced task creation overheads. For the Sequent and the programs that we have analysed and tested, grain sizes of around 64 resolution steps seem to give consistently good results for our granularity control scheme. Moreover, since the Sequent has low task creation overheads, a granularity control scheme is also likely to be useful (and perhaps even more useful) on a more coarse-grained multiprocessor such as a loosely-coupled system.

7 Related work

Imperative programming Cousot and Halbwachs [4] mention how extra counters can be added to loops and how polyhedral abstractions might be used to infer bounds for the number of iterations of a loop. The link with time-complexity analysis is not reported.

Functional programming Most similar to our work is that of Huelsbergen, Larus and Aiken [15] in the context of parallel functional programming. The analysis reported in [15], like ours, is based on an abstract semantics that calculates lower bounds on the time complexity by instrumenting the semantics with counters. Termination, again, is not an issue since the depth of computations is bounded. Coincidently, a granularity control experiment is reported for quicksort, coded in SML, on an eight processor Sequent Symmetry. Our work adds weight to theirs since Huelsbergen *et al.* conclude that with lower bound time complexity analysis "large reductions ($> 20\%$) in execution time" are possible. Interestingly, the thresholds used in the experiment do not seem to be derived automatically [15]. Our experiments suggest that this is because non-trivial thresholds cannot be derived without convex hull approximations. Convex hull approximations are, in fact, essential if the analysis is to be collapsed into something manageable. Furthermore, we have shown how an analysis for logic programs can be formulated elegantly as abstract compilation, established correctness, and shown how it can be implemented straightforwardly in a language with constraint support.

Logic programming Time-complexity analyses [6, 7] for logic programs have tended to cast the problem of inferring argument relationships in terms of solving difference equations. The analyses focus on deriving upper bound time complexity expressions like, for example $1 + \frac{l(l+5)}{2}$, for quicksort. Mode analysis is first applied to trace the input and output arguments of a clause and derive a data dependency graph for the clause literals. Prototype difference equations are then extracted from the recursive clauses, boundary conditions derived from the non-recursive clauses, and finally a difference equation is solved to yield a closed-form time-complexity expression. Although, the difference equation method is potentially useful, it requires sophisticated machinery just to manipulate and solve the equations. By way of contrast, our approach is formulated in terms of linear constraints. Also, divide and conquer algorithms can be particularly difficult to reason about with difference equations requiring special techniques [9, 18]. Moreover, additive argument size relationships, like $x_1 = x_3 + x_4$ for `Pt/4`, cannot be expressed [18]. On the other hand, the efficiency of the difference equation approach does not depend on the grain size whereas our approach may well become inefficient for coarse-grained systems, like multi-processor farms, that may require a very high granularity threshold.

A lower-bound time-complexity calculation for `Fib/2` is sketched in [12]. Difference equations are used to derive a non-trivial time complexity expression but what is not clear is the extent to which the method can be automated. The paper also discusses the cost analysis for or-parallelism, and reports some granularity control experiments. The authors justify granularity analysis by demonstrating that it is possible to obtain improved performance on a fixed configuration of four processors for different grain sizes [16]. The question of variations with different program sizes and range of number of processors and programs, does not seem to have been considered. For a configuration of four processors Sequent, a threshold test of `n > 15` (which corresponds to $\gg 4096$ resolution steps if builtins are assumed to have a non-zero cost) was found to be useful for solving the query `Fib(19, f)`. Our experiments have found that although the threshold is close to ideal for `Fib(19, f)` on four processors, it gives practically no parallelism for `Fib(17, f)` and severely limits parallelism for `Fib(19, f)` on more than four processors. Our work suggests that (for DASWAM at least) a much lower threshold is necessary if granularity control is to be useful across a range of processors numbers and goal sizes, and that these thresholds can be inferred automatically. The lower-bound time-complexity work of [12] is further developed in [9] which develops some special tactics for reasoning about divide-and-conquer algorithms. The methodology can infer a useful lower bound of $4n + 1$ for quicksort (where n is the length of the first argument). It is not yet clear whether or not our method can improve on this degree of precision.

A technique for reducing the cost of calculating term size is proposed in [17]. The technique is based on finding predicates which are called before a term size test and which traverses the terms whose size need to be determined. As we basically use the same annotation methods as [12], we expect this technique to be applicable to our work.

Very recently, Gallagher and Lafave [11] have shown abstract programs can instrumented with trace terms that abstract the shape of the computation to derive control flow information for program specialisation. The depth counter, d, is another way to abstract the shape of the computation.

8 Future work

The prototype analyser cannot (yet) infer thresholds for coarse grained loosely coupled systems very quickly and future work is required on the implementation to make the approach fast and efficient. Furthermore, integration with a norm derivation analysis [10] and a non-failure analysis is required [8]. Orders of magnitude speedup are possible, however, by carefully limiting the size of the interpretation. The tradeoff is between speed and safety. This unusual tradeoff is possible since, when used for granularity control, the analysis does not affect program correctness only program efficiency. All that matters is that the threshold is reasonably precise. Future work will examine how limitating the enumeration impacts on precision and analysis time. This is likely to be a study within itself. We also suspect that computation size alone may not be the best metric to use for controlling granularity, and we intend to research into other metrics. Finally, we shall also investigate how the method can be adapted to infer closed-form time-complexity expressions.

9 Summary

An analysis has been presented for inferring size conditions on goals that, when satisfied, ensure that a goal is sufficiently coarse-grained to warrant parallel evaluation. The analysis is precise enough to infer useful thresholding conditions even for a number of problematic divide-and-conquer programs, can be implemented straightforwardly in a language with constraint support, and, finally, has been proved correct.

Acknowledgements

Thanks are due to Nai-Wei Lin, Jon Martin and Andy Verden for stimulating discussions that motivated the investigation; Pedro López García for his comments and suggestions; Manuel Hermenegildo and Vítor Santos Costa for hosting some of the work; and Mats Carlsson and Christian Holzbaur for their invaluable help with SICStus.

References

[1] R. Barbuti, M. Codish, R. Giacobazzi, and M. Maher. Oracle Semantics for Prolog. *Information and Computation*, 22(2):178–200, 1992.

[2] F. Benoy and A. King. Inferring Argument Size Relationships with CLP($\mathcal{R}$). In *LOPSTR'96*. Springer-Verlag, 1996.

[3] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: theory and applications. *Journal of Logic Programming*, 1991.

[4] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL'78*, pages 84–97, 1978.

[5] B. De Backer and H. Beringer. A CLP language handling disjunctions of linear constraints. In *ICLP'93*, pages 550–563. MIT Press, 1993.

[6] S. Debray and N.-W. Lin. Cost Analysis for Logic Programs. *ACM TOPLAS*, July 1992.

[7] S. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *PLDI'90*, White Plains, New York, 1990. ACM.

[8] S. Debray, P. López García, and M. Hermenegildo. Non-Failure Analysis of Logic Programs. In *ICLP'97*. MIT Press, 1997.

[9] S. Debray, P. López García, M. Hermenegildo, and N. Lin. Lower Bound Cost Estimation for Logic Programs. Technical Report TR Number CLIP20/95.0, T.U. of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, 1995.

[10] S. Decorte, D. De Schreye, and M. Fabris. Automatic Inference of Norms: A Missing Link in Termination Analysis. In *ICLP'93*, pages 420–436. MIT Press, 1993.

[11] J. Gallagher and L. Lafave. Regular Approximation of Computational Paths in Logic and Functional Languages. In *Partial Evaluation*, pages 115–136. Springer-Verlag, 1996.

[12] P. López García, M. Hermenegildo, and S.K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computing*, 11(3–4):217–242, 1996.

[13] R. Giacobazzi, S. K. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *Journal of Logic Programming*, 3(25):191–248, 1995.

[14] M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *JLP*, 13(4):349–366, 1992.

[15] L. Huelsbergen, J. R. Larus, and A. Aiken. Using the Run-Time Sizes of Data Structures to Guide Parallel-Thread Creation. In *LFP'94*. ACM Press, 1994.

[16] P. López García. Personal communication on granularity control for &-Prolog with a Sequent. December 1996.

[17] P. López García and M. Hermenegildo. Efficient Term Size Computation for Granularity Control. In *ICLP'95*, pages 647–661. MIT Press, 1995.

[18] P. López García and N.-W. Lin. E-mail exchanges on argument size analysis and time complexity analysis of divide and conquer algorithms. September 1996.

[19] J. Martin, A. King, and P. Soper. Typed Norms for Typed Logic Programs. In *LOPSTR'96*. Springer-Verlag, 1996.

[20] K. Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *JLP*, 29(1–3), 1996.

[21] E. Tick. Compile-time Granularity Analysis for Parallel Logic Programming Languages. *New Generation Computing*, 7:325–337, 1990.

[22] A. Van Gelder. Deriving constraints among argument sizes in logic programs. *Annals of Mathematics and Artificial Intelligence*, 3, 1991.