Automatic Complexity Analysis

Mads Rosendahl*

Computer Laboratory
University of Cambridge
Cambridge CB2 3QG
England
Email: mro@uk.ac.cam.cl

Abstract

One way to analyse programs is to to derive expressions for their computational behaviour. A time bound function (or worst-case complexity) gives an upper bound for the computation time as a function of the size of input. We describe a system to derive such time bounds automatically using abstract interpretation. The semantics-based setting makes it possible to prove the correctness of the time bound function. The system can analyse programs in a first-order subset of Lisp and we show how the system also can be used to analyse programs in other languages.

Introduction

The computational behaviour of a program can be expressed with a time bound function—a function of input size that gives an upper bound for the computation time of the program. We describe a system to automatically derive time bound functions for programs in a first-order subset of Lisp.

Traditional complexity theory is only concerned with algorithms guaranteed to terminate, hence the time bound function can always give a maximum computation time for all input sizes. An automatic derivation system must be able to construct time bound functions for all programs but this means that it is not decidable whether the time bound function will return a finite value.

(*) The work was partially supported by the Danish Natural Science Research Council. Parts of the work was done while the author was at DIKU, University of Copenhagen, Denmark.

The derivation is made in two phases: from the program we derive a so-called *step-counting version*, a program which, when called with the same arguments as the original program, returns the computation time for the original program. In the next phase the time bound function is expressed as an abstract interpretation of the step-counting version. The soundness of the time bound function is proved by relating the abstract interpretation to the standard interpretation of the step-counting version. The system does not require any user-annotation of the program, but the user must specify the size measure that defines the "input size".

The system can be used to analyse programs in other languages as the only requirement is that we can derive a step-counting version in the first-order lisp subset. With suitable representation of data it is always be possible, since the language is universal.

This paper is mainly a rework of the author's M.Sc. thesis [Rosendahl 1986], but it also includes new results that are not in the thesis.

An outline of the paper can be found at the end of section 1.

1. Definitions

This section starts with definitions of the main concepts in the work. They are compared with definitions in the literature.

Time bound function. The objective is to generate automatically a *time bound function* for a program, defined as follows,

> A *time bound function* for a program is a partial function in input size that gives an upper bound for the computation time of the program for all inputs with the given size.

There are no constraints on the computation time if the time bound function is not defined for a given size. This means that the program may loop for some or all of the inputs with the given size. The definition differs slightly from some other definitions as we will discuss later.

Step-counting versions. An intermediate step to construct a time bound function is a *step-counting version* of a program, defined as follows,

> A *step-counting version* for a program is a program over the same domain and with the same termination properties as the original program, that outputs the computation time for the original program with the given input.

The units used to measure the computation time are not important in this context. It could be the computation time (in milliseconds) on a given machine but might as well be the number of basic operations performed or even simpler, the number of recursive calls.

The computation time is an *internal* property of the program, as it cannot be extracted from the result of the computation. The step-counting version makes this internal property *external*.

Time bound program. The step-counting version can be seen as a program with profiling commands inserted so it outputs the accumulated time instead of its normal output. With this definition the time bound function can be defined as computing an upper bound for the values of the step-counting version over the (usually infinite) set of inputs with a given size. In section 3 a method to construct a program to compute the time bound function is described. Such a program is naturally called a *time bound program.*

Time-complexity. The name "step-counting version" is taken from [Adachi, Kasai & Moriya 1979]. There does not seem to be any agreed name in the literature for this concept. [Wegbreit 1975] does not give any name for this intermediate program but calls the process of constructing it *local cost assignment.* [Métayer 1988] calls such a program a recursive time-complexity function and [Wadler 1988] with reference to [Métayer 1988], a time-complexity. The name *complexity function* is in both references used for the time bound function composed with the size measure.

According to [Aho, Hopcroft & Ullman 1974] a *worst-case complexity* (or just a *time complexity*) is a function of input size that gives the maximum computation time for all inputs of the given size. Our definition of a time bound function requires only a $\underline{\text{partial}}$ function, giving an upper bound rather than a $\underline{\text{maximum}}$. At least one of these relaxations is necessary for practical purposes as a $\underline{\text{total}}$ function that gives the $\underline{\text{maximum}}$ in general cannot be computed by a program—if it existed it could be used to solve the halting problem. Only one restriction needs to be relaxed but neither option will guarantee a more useful time-complexity.

If we choose a *partial function that gives the maximum* it is trivially achieved by the everywhere-undefined-function. An analysis to give the maximum computation time must take inter-conditional dependencies into account. As this is generally not possible the only safe time bound function will too often be the everywhere undefined function.

A *total function that gives an upper bound* must have an infinity symbol ∞ in its range to describe programs that do not terminate. A time bound function according to our definition can be transformed to a total function by extending its underlying algorithm with a clock so if an upper bound is not found after a given time (say two hours) it will return the ∞ symbol. This is a crude but practical way of showing how it can be achieved.

Occasionally the time complexity is defined as maximum/upper bound of computation time for all input with size *equal or less than* the given size. This type of time-complexity can be realised from a time bound function by taking the maximum of the *finite* set of input sizes equal or less than the given size, hence it does not impose computational problems or restrict the applications of the present approach. Actually we will claim that our definition gives a natural intermediate step when producing this type of time-complexity.

Related works

A way to estimate the efficiency of an algorithm is to construct its time-complexity. Work in complexity theory (see [Aho, Hopcroft & Ullman 1974] for a good introduction) has proven the computational difficulties of some problems and found more efficient algorithms for other problems.

The automatic construction of complexity functions has only been considered in relatively few articles in the last decade or two. A major piece of work was done by Wegbreit [Wegbreit 1975] with the METRIC system to analyse the performance of a first-order subset of Lisp. If the system terminates it will return a four tuple $\langle min, max, mean, variance\rangle$ of four closed form expressions (built from only basic operations, constants and the input size). *min* and *max* are best-case/worst-case complexities, *mean* gives the expected complexity using generating functions with *variance* as the variance. The complexities are not guaranteed to be correct as all tests are assumed to be independent and the system is only able to analyse "simple programs such as those which might be used as introductory exercises in Lisp programming". Nevertheless the system was the first of its kind and in many aspects is still the most advanced.

[Kasai et al 1980] describes a system, TIMER, that generates upper and lower bounds for the complexity of programs written in a small imperative C-like language.

The system is semi-automatic in that it requires some user annotation, and the estimates can be improved by further annotation of the program. As such it is a powerful tool for developing efficient algorithms, but it is not intended to be a program analysis system.

A program consists of a number of functions with a list of input and output variables where some of the input variables are specified as being *control variables*. The extracted complexity function will be a function in these control variables which will typically be a measure for the size of input. The function body consists of a number of statements (`IF, WHILE, FOR,`...) which if necessary can be annotated with expressions in the control variables specifying the rate at which the various branches are chosen.

The ACE system ([Métayer 1988] and [Métayer 1985]) is a system for automatic generation of complexity functions for a subset of FP. This system is based on program transformation, with a library of more than 1000 transformation rules which are applied to transform the step-counting version of a program to a composition of the complexity function and a size measure. An advantage of the system is that the main part is a rather general purpose source-to-source program transformation system satisfying a partial correctness property, though it is extended with approximating steps in which only upper bounds are preserved. It is a problem with the system that the termination and quality of the transformation are only given by the order in which the transformation rules are found in the library. Furthermore, the program transformation rules mirror a particular programming practice, so new programs may require new transformation rules.

A number of other works have addressed the topic: most recently [Wadler 1988] who describes a method of developing step-counting versions for programs under lazy evaluation. The transformation is rather trivial under eager evaluation (see section 2) but under lazy evaluation the computation time depends on the context. The step-counting version is constructed by extending the functions with an extra argument to describe which part of the result is needed.

[Talcott 1986] gives a general theory for externalising internal properties. A program can be extended to a program that returns the computation track of the evaluation. The computation time can then be expressed as the length of the computation track. The method can be used to analyse other internal properties such as maximum stack depth and number of function calls and the theory makes it possible to prove such extended programs correct.

[Flajolet 1987] examines generating functions to be used in a system for deriving expected complexities (average computation time). [Hickey & Cohen 1985] contains a number of results in average computation time analysis. The paper describes a performance compiler to analyse a simple procedureless imperative language and it also contains results on the analysis of FP programs.

[Adachi, Kasai & Moriya 1979] develops an algorithm to construct time bound programs for a class of imperative programs called simple loop programs.

The construction of time bound functions requires solution of recursive finite-difference equations. Systems to solve such equations automatically are described by [Cohen & Katcoff 1977] and [Cohen 1982].

Outline

The construction of time bound programs is done in several steps and the presentation will be interweaved with justification of these steps.

The first phase is to construct the step-counting version for the program. This is done according to a syntax-directed translation scheme. The scheme is justified by relating it to a step-counting interpretation of the language which gives a meaning to the concept "computation time". This first phase is described in section 2.

Section 3 describes an abstract interpretation of the step counting version. The interpretation is an abstraction of the collecting semantics and it approximates the execution of programs on sets of input. The interpretation is related to the standard semantics (given in section 2).

The construction of the time bound program is explained in section 4. The construction requires the size measure specified and it is shown that the inversion (the mapping of numbers to sets of data with a given size) can be approximated in a finite way. It can then proved that the composition of the abstract interpretation of the step-counting version and the approximation of the inverted size measure is a time bound function.

The time bound program is constructed by transforming the step counting version to a new program with the same standard interpretation as the abstract interpretation of the step-counting version. The abstract interpretation is, in other words, seen as a semantics of a new language and the step-counting version is compiled to the original language from this new language. The time bound program is the composition of this new program and the approximation of inverted size measure.

Extensions and other applications are discussed in section 5.

2. Step-counting version

In this section we develop an algorithm to construct step-counting versions of programs written in a first order subset of Lisp with call-by-value evaluation.

Language. A program in first-order lisp is a recursion equation system where expressions are built from constants, parameters, basic operations, conditionals and function calls. The underlying data structure is S-expressions of dotted pairs and atoms. Basic operations include *cons*, *car*, *cdr*, and *eq* as known from lisp as well as simple arithmetic operations. The program will often be identified as the first function in the program.

The abstract syntax of the language can be described as

$$
\begin{aligned}
\langle \mathit{program} \rangle \; &= F_1(x_1, \ldots, x_{i1}) = \langle \mathit{exp} \rangle_1 \\
&\;\;\vdots \\
&\;\; F_n(x_1, \ldots, x_{in}) = \langle \mathit{exp} \rangle_n
\end{aligned}
$$

and

$$
\begin{aligned}
\langle \mathit{exp} \rangle \; &= c \\
&\qquad \text{— constants} \\
&\mid x_i \\
&\qquad \text{— parameters} \\
&\mid \langle \mathit{op} \rangle \, (\langle \mathit{exp} \rangle_1, \ldots, \langle \mathit{exp} \rangle_i) \\
&\qquad \text{— basic operations} \\
&\mid F_j (\langle \mathit{exp} \rangle_1, \ldots, \langle \mathit{exp} \rangle_i) \\
&\qquad \text{— function calls} \\
&\mid \textbf{if } \langle \mathit{exp} \rangle_1 \textbf{ then } \langle \mathit{exp} \rangle_2 \textbf{ else } \langle \mathit{exp} \rangle_3 \\
&\qquad \text{— conditionals} \\
\langle \mathit{op} \rangle \; &= \mathit{car} \mid \mathit{cdr} \mid \mathit{cons} \mid \mathit{eq} \mid \mathit{add} \mid \\
&\;\;\; \mathit{sub} \mid \mathit{max} \mid \mathit{null} \mid \mathit{atom} \mid \mathit{and} \mid \mathit{or}
\end{aligned}
$$

The actual set of basic operations is not important in this context; other operations can be added without restricting the scope of the program analysis. The function names and parameter names are assumed to be indexed F's and x's for notational convenience but we will quickly forget this in the examples.

The semantics of the language is given as a fixed point semantics. The meaning of a program will be a function environment ($\phi : (D^* \to D_\perp)^n$) where D is the set of S-expressions ([McCarthy et al 1965]) and $D_\perp$ is the same set lifted to a flat domain with $\perp$ added as bottom element. Domain will here and throughout the paper mean a complete partial order. X^* and X^n are the cartesian product of a number of versions of a domain X, the number depending on the actual program whose meaning is to be defined. The argument domains in the function environment are not necessarily identical, hence the binding of * and ^{n} to particular values will be from outside-in. The input-output function for the program is the first function in this function environment.

The bottom element will denote non-termination, and it is vital for the construction of the time bound function that we distinguish between non-termination and other run-time errors. This means that any basic operation is given some sense (even if it is non-sense). *car* and *cdr* of atoms will return *nil*; if the test in a conditional expression is neither true nor false the value of the expression will be *nil*. Similar definitions can be made for arithmetic expressions with non-numeric expressions being interpreted as zero. A more realistic approach would have been to extend the set D with a new element *error* to be returned by such computations. For our purpose the simpler interpretation will be sufficient as we are only interested in the computation time and not in the values returned.

The description language is a simple denotational semantics meta-language ([Gordon 1979]). Since it is intuitively clear it will not be discussed or explained further here.

$$
\begin{aligned}
&\mathbf{U}[\![\langle \mathit{program} \rangle]\!] : (D^* \to D_\perp)^n \\
&\mathbf{E}[\![\langle \mathit{exp} \rangle]\!] : D^* \to (D^* \to D_\perp)^n \to D_\perp
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{U}[\![\langle \mathit{program} \rangle]\!] = \\
&\mathbf{U}[\![\; F_1(x_1, \ldots, x_{i1}) = E_1 \\
&\qquad \vdots \\
&\qquad F_n(x_1, \ldots, x_{in}) = E_n]\!] = \\
&\qquad\quad \mathit{fix}(\lambda\, \phi .\, \langle \mathbf{E}[\![E_1]\!]\, \phi, \ldots, \mathbf{E}[\![E_n]\!]\, \phi \rangle \,)
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{E}[\![c]\!]\, \phi\nu = c \\
&\mathbf{E}[\![x_i]\!]\, \phi\nu = \nu_i \\
&\mathbf{E}[\![\mathit{op}(E_1, \ldots, E_i)]\!]\, \phi\nu = \\
&\quad \mathit{op}(\mathbf{E}[\![E_1]\!]\, \phi\nu, \ldots, \mathbf{E}[\![E_i]\!]\, \phi\nu) \\
&\mathbf{E}[\![F_j(E_1, \ldots, E_i)]\!]\, \phi\nu = \\
&\quad \mathit{lift}(\phi_j)(\mathbf{E}[\![E_1]\!]\, \phi\nu, \ldots, \mathbf{E}[\![E_i]\!]\, \phi\nu) \\
&\mathbf{E}[\![\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3]\!]\, \phi\nu = \\
&\quad \textbf{case } \mathbf{E}[\![E_1]\!]\, \phi\nu \textbf{ of} \\
&\qquad \mathit{true} \quad \Rightarrow \mathbf{E}[\![E_2]\!]\, \phi\nu \\
&\qquad \mathit{false} \quad \Rightarrow \mathbf{E}[\![E_3]\!]\, \phi\nu \\
&\qquad \_ \quad \Rightarrow \mathit{nil} \\
&\quad \textbf{end}
\end{aligned}
$$

where

$$\mathit{lift} : (S^* \to X) \to (S_\perp^* \to X)$$

X and $S_\perp$ are domains

$$
\begin{aligned}
&\mathit{lift}(f)(x_1, \ldots, x_n) = \\
&\quad \textbf{if } x_i = \perp_S \textbf{ for any } i = 1, \ldots, n \textbf{ then } \perp_X \\
&\quad \textbf{else } f(x_1, \ldots, x_n)
\end{aligned}
$$

The *lift* function extends a function from a set to a function from a domain by mapping the bottom element to the bottom element. The *lift* function will in this case make function calls strict and we also assume that the basic operations (like cons) are strict.

The underscore (_) in the interpretation of the **if**-expression refers to any other value in D. If the test is undefined ($\perp$) then the value of the conditional is also $\perp$.

Step-counting version. In this paragraph we describe a syntax directed translation scheme to construct a step-counting version. The program transformer is called $\mathcal{T}$ and is a total function on the set of programs $\mathcal{T} : \langle \textit{program} \rangle \rightarrow \langle \textit{t-program} \rangle$. The syntactic category $\langle \textit{t-program} \rangle$ is a subset of $\langle \textit{program} \rangle$ and is defined as the range of the function $\mathcal{T}$. Quasi-quotes $\ldots$ are used to construct the syntactic objects, as the quoted text should be taken literally except for variable substitution.

$$
\begin{aligned}
\mathcal{T}[\![\; & F_1(x_1, \ldots, x_{i1}) = E_1 \\
& \vdots \\
& F_n(x_1, \ldots, x_{in}) = E_n]\!] = \\
& T_1(x_1, \ldots, x_{i1}) = \; \mathcal{T}_\sigma[\![E_1]\!] \\
& \vdots \\
& T_n(x_1, \ldots, x_{in}) = \; \mathcal{T}_\sigma[\![E_n]\!] \\
& F_1(x_1, \ldots, x_{i1}) = E_1 \\
& \vdots \\
& F_n(x_1, \ldots, x_{in}) = E_n
\end{aligned}
$$

$$
\begin{aligned}
& \mathcal{T}_\sigma[\![c]\!] = \; 1 \\
& \mathcal{T}_\sigma[\![x_i]\!] = \; 1 \\
& \mathcal{T}_\sigma[\![op(E_1, \ldots, E_i)]\!] = \\
& \qquad add(1, \; \mathcal{T}_\sigma[\![E_1]\!] \; , \; \ldots \; , \; \mathcal{T}_\sigma[\![E_i]\!] \;) \\
& \mathcal{T}_\sigma[\![F_j(E_1, \ldots, E_i)]\!] = \\
& \qquad add(1, T_j(E_1, \ldots, E_i), \; \mathcal{T}_\sigma[\![E_1]\!] \; , \; \ldots \; , \\
& \qquad\qquad \mathcal{T}_\sigma[\![E_i]\!] \;) \\
& \mathcal{T}_\sigma[\![\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3]\!] = \\
& \qquad add(1, \; \mathcal{T}_\sigma[\![E_1]\!] \; , \\
& \qquad\qquad \textbf{if } E_1 \textbf{ then } \; \mathcal{T}_\sigma[\![E_2]\!] \; \textbf{ else } \; \mathcal{T}_\sigma[\![E_3]\!] \;)
\end{aligned}
$$

The resulting program consists of the original program plus new functions for all functions in the original program. These new functions are called with the same arguments as in the original program but return the number of sub-expressions to be evaluated with a call to the function in the original program. The time to compute constants and parameters is assumed to be the unit 1 ; for basic operations it is 1 plus the time to compute the arguments. For function calls it is 1 plus the time to compute the arguments plus the time to compute the function body; for conditionals it is 1 plus the time to compute the test plus the time to compute the branch. The original program is appended to these new functions because evaluation of tests and arguments of function calls may call functions in the original program.

Correctness. The computation time of a program is an internal property, hence we cannot express the correctness in terms of the denotational semantics given in the start of this section. The step-counting version mirrors a specific model of computation and a proof may be based on an operational semantics. We will use a different approach by introducing an *instrumented semantics* to make the computation time an external property. The semantics will give the meaning of programs as an extended function environment with results as pairs of value and time spent in the computation. The function environment (ϕ') will have type $(D^* \rightarrow (D \times \;)_\perp)^n$ with $\;_\perp$ as the flat domain of natural numbers.

The semantics functions for this instrumented semantics will be called $\mathbf{U}'$ for programs and $\mathbf{E}'$ for expressions.

$$
\begin{aligned}
& \mathbf{U}'[\![\; \langle \textit{program} \rangle \;]\!] = \\
& \mathbf{U}'[\![\; F_1(x_1, \ldots, x_{i1}) = E_1 \\
& \qquad \vdots \\
& \qquad F_n(x_1, \ldots, x_{in}) = E_n]\!] = \\
& \qquad\quad \mathit{fix}(\lambda\, \phi'.\, \langle \mathbf{E}'[\![E_1]\!]\,\phi', \ldots, \mathbf{E}'[\![E_n]\!]\,\phi' \rangle\,)
\end{aligned}
$$

$$
\begin{aligned}
& \mathbf{E}'[\![c]\!]\,\phi'\nu = \langle c, 1 \rangle \\
& \mathbf{E}'[\![x_i]\!]\,\phi'\nu = \langle \nu_i, 1 \rangle \\
& \mathbf{E}'[\![op(E_1, \ldots, E_i)]\!]\,\phi'\nu = \\
& \quad \textbf{let } \langle \alpha_j, \beta_j \rangle = \mathbf{E}'[\![E_j]\!]\,\phi'\nu \\
& \quad \textbf{in } \langle op(\alpha_1, \ldots, \alpha_i), 1 + \beta_1 + \cdots + \beta_i \rangle \\
& \mathbf{E}'[\![F_k(E_1, \ldots, E_i)]\!]\,\phi'\nu = \\
& \quad \textbf{let } \langle \alpha_j, \beta_j \rangle = \mathbf{E}'[\![E_j]\!]\,\phi'\nu \\
& \quad \textbf{in let } \langle \alpha_0, \beta_0 \rangle = \mathit{lift}(\phi_k)(\alpha_1, \ldots, \alpha_i) \\
& \quad \textbf{in } \langle \alpha_0, 1 + \beta_0 + \beta_1 + \cdots + \beta_i \rangle \\
& \mathbf{E}'[\![\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3]\!]\,\phi'\nu = \\
& \quad \textbf{case } \mathbf{E}'[\![E_1]\!]\,\phi'\nu \textbf{ of} \\
& \qquad \langle \mathit{true}, \beta_1 \rangle \quad \Rightarrow \textbf{let } \langle \alpha_2, \beta_2 \rangle = \mathbf{E}'[\![E_2]\!]\,\phi'\nu \\
& \qquad\qquad\qquad\qquad \textbf{in } \langle \alpha_2, 1 + \beta_1 + \beta_2 \rangle \\
& \qquad \langle \mathit{false}, \beta_1 \rangle \quad \Rightarrow \textbf{let } \langle \alpha_3, \beta_3 \rangle = \mathbf{E}'[\![E_3]\!]\,\phi'\nu \\
& \qquad\qquad\qquad\qquad \textbf{in } \langle \alpha_3, 1 + \beta_1 + \beta_3 \rangle \\
& \qquad \langle \_, \beta_1 \rangle \quad \Rightarrow \langle \mathit{nil}, 1 + \beta_1 \rangle \\
& \quad \textbf{end}
\end{aligned}
$$

The correctness property can then be expressed as

$$
\begin{aligned}
& \forall\, P \in \langle \textit{program} \rangle, j = 1, \ldots, n, \nu \in D^* : \\
& \quad (\mathbf{U}'[\![P]\!])_j \nu = \langle (\mathbf{U}[\![P]\!])_j \nu, (\mathbf{U}[\![\mathcal{T}[\![P]\!]]\!])_j \nu \rangle
\end{aligned}
$$

The expression $\mathbf{U}[\![\mathcal{T}[\![P]\!]]\!]$ denotes the meaning of the (step-counting version) program returned by transformation function $\mathcal{T}$.

Admittedly, this will be as much a definition of computation time as a proof of the step-counting version transformer $\mathcal{T}$. Any model of computation, however, will be a definition of computation time and the interpretation given by $\mathbf{U}'$ is as intuitive as any odd operational semantics.

The correctness proof can be stated by structural induction over expressions but will not be included here.

Semantics for step-counting versions. The introduction of the step-counting version means that we do not need the original program any more. The time bound program will be derived from the step-counting version. From now on the semantics functions will only be used for step-counting versions and as later interpretations will require different interpretations for expressions returning computation times and S-expressions we will introduce a specialised semantics to be used in the rest of this paper.

The interpretation differs from the previous semantics in that it uses the domain ${}^{\infty}$ instead of ${}_{\perp}$. We will justify the choice after introducing the domain and the semantics.

The domain ${}^{\infty}$. The set of natural numbers extended with an infinity symbol (∞) is organised as a domain called ${}^{\infty}$ with ordering $\preceq$.

$$a \preceq b \overset{\text{def}}{\Leftrightarrow} a \leq b \vee b = \infty$$

The infinity symbol (∞) is the top element and zero (0) the bottom element. The top element is a limit point and the domain has infinite height.

The least upper bound is called Max^{∞}

$$\begin{aligned} Max^{\infty}(S) = {} & \textbf{if } S = \quad \textbf{then } 0 \\ & \textbf{else if } S \text{ is infinite } \vee \infty \in S \textbf{ then } \infty \\ & \textbf{else } Max(S) \end{aligned}$$

and the dyadic least upper bound is

$$\begin{aligned} max^{\infty}(x, y) = {} & \textbf{if } x = \infty \textbf{ or } y = \infty \\ & \textbf{then } \infty \\ & \textbf{else if } x > y \textbf{ then } x \textbf{ else } y \end{aligned}$$

Addition is a continuous function in the domain ${}^{\infty}$.

$$\begin{aligned} add^{\infty}(x, y) = {} & \textbf{if } x = \infty \textbf{ or } y = \infty \textbf{ then } \infty \\ & \textbf{else } x + y \end{aligned}$$

We will write add^{∞} as an infix $+$ when the meaning is clear from the context.

The standard semantics. The following semantics is specialised to step-counting versions. There will be two types of semantic functions for expressions, the index σ is for expressions returning S-expressions and τ for expressions returning computation times:

$$\begin{aligned} & \mathbf{E}_{\tau}[\![\langle exp \rangle]\!] : \\ & \quad (D^* \to {}^{\infty})^n \to (D^* \to D_{\perp})^n \to D^* \to {}^{\infty} \\ & \mathbf{E}_{\sigma}[\![\langle exp \rangle]\!] : \\ & \quad (D^* \to D_{\perp})^n \to D^* \to D_{\perp} \end{aligned}$$

where $\mathbf{E}_{\sigma}$ is identical to $\mathbf{E}$ as defined earlier.

The meaning function for programs is

$$\mathbf{U}[\![\langle t\text{-}program \rangle]\!] : (D^* \to {}^{\infty})^n \times (D^* \to D_{\perp})^n$$

$$\begin{aligned} \mathbf{U}[\![\; & T_1(x_1, \ldots, x_{i1}) = E_{t1} \\ & \vdots \\ & T_n(x_1, \ldots, x_{in}) = E_{tn} \\ & F_1(x_1, \ldots, x_{i1}) = E_{s1} \\ & \vdots \\ & F_n(x_1, \ldots, x_{in}) = E_{sn}]\!] = \\ & \!\!\!\!\!\!\! \mathit{fix}(\lambda \langle \theta, \phi \rangle . \langle \langle \mathbf{E}_{\tau}[\![E_{t1}]\!]\theta\phi, \ldots, \mathbf{E}_{\tau}[\![E_{tn}]\!]\theta\phi \rangle , \\ & \qquad \langle \mathbf{E}_{\sigma}[\![E_{s1}]\!]\phi, \ldots, \mathbf{E}_{\sigma}[\![E_{sn}]\!]\phi \rangle \rangle) \end{aligned}$$

and

$$\begin{aligned} & \mathbf{E}_{\tau}[\![n]\!]\theta\phi\nu = n \\ & \mathbf{E}_{\tau}[\![add(E_1, \ldots, E_i)]\!]\theta\phi\nu = \\ & \quad \mathbf{E}_{\tau}[\![E_1]\!]\theta\phi\nu + \ldots + \mathbf{E}_{\tau}[\![E_i]\!]\theta\phi\nu \\ & \mathbf{E}_{\tau}[\![T_j(E_1, \ldots, E_i)]\!]\theta\phi\nu = \\ & \quad \mathit{lift}(\theta_j)(\mathbf{E}_{\sigma}[\![E_1]\!]\phi\nu, \ldots, \mathbf{E}_{\sigma}[\![E_i]\!]\phi\nu) \\ & \mathbf{E}_{\tau}[\![\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3]\!]\theta\phi\nu = \\ & \quad \textbf{case } \mathbf{E}_{\sigma}[\![E_1]\!]\phi\nu \textbf{ of} \\ & \qquad \mathit{true} \quad \Rightarrow \mathbf{E}_{\tau}[\![E_2]\!]\theta\phi\nu \\ & \qquad \mathit{false} \quad \Rightarrow \mathbf{E}_{\tau}[\![E_3]\!]\theta\phi\nu \\ & \qquad \_ \quad \Rightarrow 0 \\ & \quad \textbf{end} \end{aligned}$$

The lift function will here have functionality

$$\mathit{lift} : (D^* \to {}^{\infty}) \to (D^*_{\perp} \to {}^{\infty})$$

The lifted function will return the bottom element, zero, if one of its arguments is undefined. This means that a function application takes no time until all its arguments are defined.

The semantic functions for expressions can easily be seen to be continuous and the fixpoint will therefore be well-defined.

Discussion. It may seem strange to use the domain ${}^{\infty}$ with the "vertical" ordering because the recursive definitions of a function $f : D \to {}^{\infty}$

$$f(x) = f(x)$$

and

$$f(x) = 0$$

will both be interpreted as the constant function 0. This differs from the case where we use ${}_{\perp}$ but remember that we restrict ourselves to step-counting versions and for a program

$$F_1(x) = F_1(x)$$

we will have the step-counting version

$$T_1(x) = add(1, T_1(x), 1)$$
$$F_1(x) = F_1(x)$$

which will be interpreted as the constant function ∞ as one would expect from a looping program. Notice that no step-counting version can actually return the value 0.

The justification for the use of $\ ^{\infty}$ is that we can prove with $\mathbf{U}_{old}$ as the standard semantics defined in the start of the section that,

$$\forall\ T \in \langle \textit{t-program} \rangle, j = 1, \ldots, n, \nu \in D^* :$$
$$(\mathbf{U}_{old}[\![T]\!])_j \nu \simeq (\mathbf{U}[\![T]\!])_j \nu$$

where

$$x \simeq y \overset{\text{def}}{\Leftrightarrow} (x = \bot \wedge y = \infty) \vee (x = y \wedge x \in \quad)$$

This means that the new standard semantics is identical to the old except that we use ∞ to denote non-termination instead of $\bot$.

In the new semantics non-termination will be represented by the top element and as it is the limit point in a domain with infinite height it cannot be found by finite iteration. The interpretation is still computable and the functions add^{∞} and max^{∞} can be computed by the normal addition and maximum functions.

The difference between using $\ _{\bot}$ and $\ ^{\infty}$ is more than the place of "non-termination". For the intuition one can view the new semantics as a loud ticking clock where after each iteration we can see the computation time so far, and the old semantics as a silent clock where only the final result can be seen. In this way it will have many of the properties of a continuation-style semantics (see [Gordon 1979]) compared to the direct semantics using $\ _{\bot}$.

Time bound function criterion. Let

$$t(x_1, \ldots, x_{i1}) = (\mathbf{U}[\![\mathcal{T}[\![P]\!]]\!])_1 \langle x_1, \ldots, x_{i1} \rangle\ ,$$
$$P \in \langle \textit{program} \rangle$$

The function t is the input-output function for the step-counting version of a program P. This means that it will return the computation time of a call to the first function in the program P with the arguments $x_1, \ldots, x_{i1}$.

A time bound function $tb : \ ^* \rightarrow \ ^{\infty}$ for P must satisfy

$$tb(n_1, \ldots, n_{i1}) \succeq Max^{\infty}\{t(x_1, \ldots, x_{i1}) \,|\, \ell(x_j) = n_j\}$$

where

$$\ell : D \rightarrow \ _{\bot}$$

is a size function.

In the next section we will describe a method of constructing a program to compute the time bound function. Such a program will be called a time bound program and the criterion states that if the time bound program returns a value then this value will be an upper bound of the computation time for the program with inputs of the given sizes.

Example. The union function in lisp can be defined as

```
union(x,y) = if null(x) then y
             else if member(car(x),y)
             then union(cdr(x),y)
             else cons(car(x),union(cdr(x),y))

member(x,s) = if null(s) then false
              else if eq(x,car(s)) then true
              else member(x,car(s))
```

This function is interesting as an example because the METRIC system [Wegbreit 1975] was unable to produce a time bound function for it. The problem was that the test in the second conditional is computed by a user defined function with a variable computing time.

The step-counting version for the union function can easily be constructed (all additions of numerical constants have been simplified).

```
tunion(x,y)  =
     if null(x) then 4
     else add(tmember(car(x),y) ,
       if member(car(x),y)
       then add(11,tunion(cdr(x),y))
       else add(14,tunion(cdr(x),y)) )

tmember(x,s) =
     if null(s) then 4
     else if eq(x,car(s)) then 9
     else add(12,tmember(x,car(s)))

member(x,s) =
     if null(s) then false
     else if eq(x,car(s)) then true
     else member(x,car(s))
```

We have omitted the `union` function here as it cannot be called from `tunion`. In general the step-counting function is more complicated than the original program because it may contain all the functions from the original program. The next part of the construction of a time bound function will complicate the program even more but it is normally possible to simplify the result considerably.

3. Abstract interpretation of the step-counting version

The language we are using is first-order lisp with S-expressions as the underlying datatype. In this section we introduce an extension to S-expression of partially known structures. A partially known structure represents a subset of possible S-expressions and can as such be seen as an abstraction (or quotient) of the powerset of S-expressions. The set of partially known structures will be called $\tilde{D}$ and we introduce so-called tilde-extensions to the basic operations. We describe a tilde-interpretation of the step counting version which gives a safe time bound function.

The set of partially known structures. A partially known structure is an S-expression containing a special symbol to denote that the given substructure is unknown and therefore could be substituted for any S-expression. The atom *all* will be used as this special symbol and the set of partially known structures $\tilde{D}$ can be seen as an abstraction of the powerset of S-expressions ((D)).

$$\begin{tikzcd} \tilde{D}_\perp \arrow[r, shift left, "\gamma"] & \supseteq \arrow[l, shift left, "\alpha"] & (D) \end{tikzcd}$$

with

$$\begin{array}{lll} \gamma(x) = & & \textbf{if } x = \perp, \textbf{else} \\ & D & \textbf{if } x = \mathit{all}, \textbf{else} \\ & \{x\} & \textbf{if } x \in \mathit{Atoms}, \textbf{else} \\ & \{\langle a, b\rangle \mid a \in \gamma(x_1) \wedge b \in \gamma(x_2)\} & \\ & & \textbf{if } x = \langle x_1, x_2\rangle \end{array}$$

$$\begin{array}{lll} \alpha(x) = \perp & & \textbf{if } x = \quad, \textbf{else} \\ & e & \textbf{if } x = \{e\}, \textbf{else} \\ & \mathit{all} & \textbf{if } x \cap \mathit{Atoms} \neq \quad, \textbf{else} \\ & \langle \alpha(\{a \mid \langle a, b\rangle \in x\}), \alpha(\{b \mid \langle a, b\rangle \in x\})\rangle & \end{array}$$

Both these functions can be proved to satisfy the condition for abstraction and concretisation functions [Cousot & Cousot 1977],

$$\begin{array}{ll} \forall S \in \ (D) : & \gamma(\alpha(S)) \supseteq S \\ \forall x \in \tilde{D} : & \alpha(\gamma(x)) = x \end{array}$$

The set $\tilde{D}_\perp$ can be equipped with an ordering $\sqsubseteq$ and a least upper bound *Lub*.

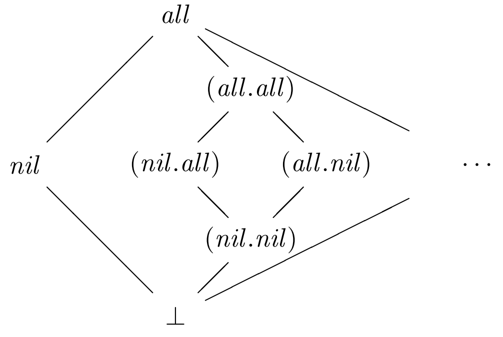
$$\forall x, y \in \tilde{D}_\perp \quad : x \sqsubseteq y \overset{\text{def}}{\Leftrightarrow} \gamma(x) \subseteq \gamma(y)$$

$$\forall S \subseteq \tilde{D}_\perp \quad : \mathit{Lub}(S) \overset{\text{def}}{=} \alpha\left(\bigcup_{y \in S} \gamma(y)\right)$$

With these definitions $\tilde{D}_\perp$ is a complete lattice with finite height (see [Rosendahl 1986]), and the functions α and γ are continuous. The dyadic least upper bound is

$$\begin{array}{l} \mathit{lub}(x, y) = \textbf{if } x = y \textbf{ then } x \\ \qquad \textbf{else if } x = \mathit{all} \textbf{ or } y = \mathit{all} \textbf{ or} \\ \qquad\qquad \mathit{atom}(x) \textbf{ or } \mathit{atom}(y) \textbf{ then } \mathit{all} \\ \qquad \textbf{else } \tilde{cons}(\,\mathit{lub}(\tilde{car}(x), \tilde{car}(y)), \\ \qquad\qquad \mathit{lub}(\tilde{cdr}(x), \tilde{cdr}(y))) \end{array}$$

where $\tilde{cons}$, $\tilde{car}$, and $\tilde{cdr}$ are defined below.



Some elements from the complete lattice $\tilde{D}$

Tilde-extensions. Functions on S-expressions can be extended to functions on $\tilde{D}_\perp$. We need such tilde-extensions of the basic operations in the abstract interpretation. Let $f : D^* \to D_\perp$ be a function on S-expressions. The continuous extension (see [Rosendahl 1986]) will be $\tilde{f} : \tilde{D}^* \to \tilde{D}_\perp$.

$$\tilde{f}(x_1, \ldots, x_i) = \\ \alpha\left(\{f(y_1, \ldots, y_i) \mid y_j \in \gamma(x_j), j = 1, \ldots, i\}\right)$$

The extensions for the basic operations are fairly obvious,

$$\begin{array}{llll} \tilde{car}(x) & = \mathit{all} & \textbf{if } x = \mathit{all}, \textbf{else} \\ & \quad a & \textbf{if } x = \langle a, b\rangle \\ \tilde{cons}(x, y) & = \langle x, y\rangle & \\ \tilde{eq}(x, y) & = \mathit{all} & \textbf{if } x = \mathit{all} \vee y = \mathit{all}, \textbf{else} \\ & \quad \mathit{true} & \textbf{if } x = y \wedge \mathit{atom}(x), \textbf{else} \\ & \quad \mathit{false} & \end{array}$$

Other basic operations can be extended in a similar way.

Functions from D to $\ ^\infty$ can also be extended to functions from $\tilde{D}$. Let

$$f : D^* \to \ ^\infty$$

We then define

$$\kappa f : \tilde{D}^* \to \ ^\infty$$

as

$$\kappa f(x_1, \ldots, x_i) = \mathit{Max}^\infty\{f(y_1, \ldots, y_i) \mid y_j \in \gamma(x_j)\}$$

The extension will give a continuous function.

Tilde-interpretation. The interpretation $\tilde{\mathbf{U}}$ of the step counting version is given here. The basic operations $\tilde{op}$ are the extensions defined in the previous paragraph and *max* takes the maximum of two numbers.

$$\tilde{\mathbf{U}}\llbracket\, \langle \textit{t-program} \rangle \,\rrbracket : (\tilde{D}^* \to \quad ^{\infty})^n \times (\tilde{D}^* \to \tilde{D}_\perp)^n$$

$$\begin{aligned}
&\tilde{\mathbf{U}}\llbracket\ T_1(x_1, \ldots, x_{i1}) = E_{t1} \\
&\qquad \vdots \\
&\qquad T_n(x_1, \ldots, x_{in}) = E_{tn} \\
&\qquad F_1(x_1, \ldots, x_{i1}) = E_{s1} \\
&\qquad \vdots \\
&\qquad F_n(x_1, \ldots, x_{in}) = E_{sn}\rrbracket = \\
&\mathit{fix}(\lambda \langle \theta, \phi \rangle . \langle\ \langle \tilde{\mathbf{E}}_\tau \llbracket E_{t1} \rrbracket \theta\phi, \ldots, \tilde{\mathbf{E}}_\tau \llbracket E_{tn} \rrbracket \theta\phi \rangle\ , \\
&\qquad\qquad\qquad \langle \tilde{\mathbf{E}}_\sigma \llbracket E_{s1} \rrbracket \phi, \ldots, \tilde{\mathbf{E}}_\sigma \llbracket E_{sn} \rrbracket \phi \rangle \,\rangle)
\end{aligned}$$

$$\begin{aligned}
&\tilde{\mathbf{E}}_\tau \llbracket n \rrbracket \theta\phi\nu = n \\
&\tilde{\mathbf{E}}_\tau \llbracket \mathit{add}(E_1, \ldots, E_i) \rrbracket \theta\phi\nu = \\
&\quad \tilde{\mathbf{E}}_\tau \llbracket E_1 \rrbracket \theta\phi\nu + \cdots + \tilde{\mathbf{E}}_\tau \llbracket E_i \rrbracket \theta\phi\nu \\
&\tilde{\mathbf{E}}_\tau \llbracket T_j(E_1, \ldots, E_i) \rrbracket \theta\phi\nu = \\
&\quad \mathit{lift}(\theta_j)(\tilde{\mathbf{E}}_\sigma \llbracket E_1 \rrbracket \phi\nu, \ldots, \tilde{\mathbf{E}}_\sigma \llbracket E_i \rrbracket \phi\nu) \\
&\tilde{\mathbf{E}}_\tau \llbracket \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3 \rrbracket \theta\phi\nu = \\
&\quad \textbf{case } \tilde{\mathbf{E}}_\sigma \llbracket E_1 \rrbracket \phi\nu \textbf{ of} \\
&\qquad \mathit{all} \quad \Rightarrow \mathit{max}^\infty(\tilde{\mathbf{E}}_\tau \llbracket E_2 \rrbracket \theta\phi\nu, \tilde{\mathbf{E}}_\tau \llbracket E_3 \rrbracket \theta\phi\nu) \\
&\qquad \mathit{true} \quad \Rightarrow \tilde{\mathbf{E}}_\tau \llbracket E_2 \rrbracket \theta\phi\nu \\
&\qquad \mathit{false} \quad \Rightarrow \tilde{\mathbf{E}}_\tau \llbracket E_3 \rrbracket \theta\phi\nu \\
&\qquad \_ \quad \Rightarrow 0 \\
&\quad \textbf{end} \\
&\tilde{\mathbf{E}}_\sigma \llbracket c \rrbracket \phi\nu = c \\
&\tilde{\mathbf{E}}_\sigma \llbracket x_i \rrbracket \phi\nu = \nu_i \\
&\tilde{\mathbf{E}}_\sigma \llbracket \mathit{op}(E_1, \ldots, E_i) \rrbracket \phi\nu = \\
&\quad \tilde{\mathit{op}}(\tilde{\mathbf{E}}_\sigma \llbracket E_1 \rrbracket \phi\nu, \ldots, \tilde{\mathbf{E}}_\sigma \llbracket E_i \rrbracket \phi\nu) \\
&\tilde{\mathbf{E}}_\sigma \llbracket F_j(E_1, \ldots, E_i) \rrbracket \phi\nu = \\
&\quad \mathit{lift}(\phi_j)(\tilde{\mathbf{E}}_\sigma \llbracket E_1 \rrbracket \phi\nu, \ldots, \tilde{\mathbf{E}}_\sigma \llbracket E_i \rrbracket \phi\nu) \\
&\tilde{\mathbf{E}}_\sigma \llbracket \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3 \rrbracket \phi\nu = \\
&\quad \textbf{case } \tilde{\mathbf{E}}_\sigma \llbracket E_1 \rrbracket \phi\nu \textbf{ of} \\
&\qquad \mathit{all} \quad \Rightarrow \mathit{lub}(\tilde{\mathbf{E}}_\sigma \llbracket E_2 \rrbracket \phi\nu, \tilde{\mathbf{E}}_\sigma \llbracket E_3 \rrbracket \phi\nu, \mathit{nil}) \\
&\qquad \mathit{true} \quad \Rightarrow \tilde{\mathbf{E}}_\sigma \llbracket E_2 \rrbracket \phi\nu \\
&\qquad \mathit{false} \quad \Rightarrow \tilde{\mathbf{E}}_\sigma \llbracket E_3 \rrbracket \phi\nu \\
&\qquad \_ \quad \Rightarrow \mathit{nil} \\
&\quad \textbf{end}
\end{aligned}$$

It can easily be seen that $\tilde{\mathbf{E}}_\tau \llbracket \langle \mathit{exp} \rangle \rrbracket \theta\phi$ and $\tilde{\mathbf{E}}_\sigma \llbracket \langle \mathit{exp} \rangle \rrbracket \phi$ are defined as continuous functions for continuous functions θ and ϕ. This makes the fixpoint well-defined.

Correctness. We would like to prove that if the $\tilde{\mathbf{U}}$ interpretation of a program (a step-counting version) returns a value then the standard interpretation of the program will return a value for all arguments described by the input to the $\tilde{\mathbf{U}}$-interpretation and the result will be less than or equal to the result of the $\tilde{\mathbf{U}}$ interpretation. In other words (or symbols),

$$\begin{aligned}
&\forall\, T \in \langle \textit{t-program} \rangle\ \forall\, \nu \in \tilde{D}^* : \\
&(\tilde{\mathbf{U}}\llbracket T \rrbracket)_1 \nu \succeq \\
&\qquad \mathit{Max}^\infty\{(\mathbf{U}\llbracket T \rrbracket)_1 \nu' \mid \nu'_j \in \gamma(\nu_j), j = 1, \ldots, i\}
\end{aligned}$$

There may not exist a tuple $\nu' \in D^*$ such that $(\mathbf{U}\llbracket T \rrbracket)_1 \nu'$ is the result of the $\tilde{\mathbf{U}}$ interpretation, hence we can only prove that the left hand side gives an *upper bound* for the computation time—in accordance with the definition in the introduction. The proof will not be included in this version of the paper.

It is an improvement over the result in [Rosendahl 1986] where only a weak correctness was stated:

$$\begin{aligned}
&\mathit{tb}(n_1, \ldots, n_i) \succeq \\
&\qquad \mathit{Max}^\infty\{\mathbf{U}\llbracket T \rrbracket \nu' \mid \mathbf{U}\llbracket T \rrbracket \nu' \neq \infty \wedge \ell(\nu'_j) = n_j\}
\end{aligned}$$

The weak correctness proof was based on an intermediate collecting semantics using Hoare power domains. Coincidentally this is exactly the correctness criterion stated in [Métayer 1988]. Only weak correctness is guaranteed because the transformation system changes the semantics from eager to lazy evaluation so more powerful transformation rules can be applied.

4. Construction of the time bound program

The time bound function *tb* must satisfy the criterion

$$\mathit{tb}(n_1, \ldots, n_i) \succeq \mathit{Max}^\infty\{t(x_1, \ldots, x_i) \mid \ell(x_j) = n_j\}$$

where

$$t(x_1, \ldots, x_i) = (\mathbf{U}\llbracket \mathcal{T}\llbracket P \rrbracket \rrbracket)_1(x_1, \ldots, x_i)$$

for a program $P \in \langle \mathit{program} \rangle$, and

$$\ell : D \to \quad _\perp$$

is a size function

The $\tilde{\mathbf{U}}$-interpretation gives a candidate:

$$\begin{aligned}
&(\tilde{\mathbf{U}}\llbracket \mathcal{T}\llbracket P \rrbracket \rrbracket)_1(\,\alpha(\{x_1 \mid \ell(x_1) = n_1\}), \ldots, \\
&\qquad\qquad \alpha(\{x_i \mid \ell(x_i) = n_i\})) \\
&\quad \succeq \mathit{Max}^\infty\{\,\mathbf{U}\llbracket \mathcal{T}\llbracket P \rrbracket \rrbracket \nu' \mid \\
&\qquad\qquad \nu'_j \in \gamma(\alpha(\{x_j \mid \ell(x_j) = n_j\}))\} \\
&\quad \succeq \mathit{Max}^\infty\{\,\mathbf{U}\llbracket \mathcal{T}\llbracket P \rrbracket \rrbracket \nu' \mid \\
&\qquad\qquad \nu'_j \in \{x_j \mid \ell(x_j) = n_j\}\} \\
&\quad = \mathit{Max}^\infty\{\mathbf{U}\llbracket \mathcal{T}\llbracket P \rrbracket \rrbracket \nu' \mid \ell(\nu'_j) = n_j\}
\end{aligned}$$

This will not necessarily be a computable time bound function as $\lambda\, n.\alpha(\{x \mid \ell(x) = n\})$ in general cannot be computed by a program. Luckily it is always possible to make an approximation that in a certain sense safely approximates the inverted length function.

$$\ell 1 : \quad \rightarrow \tilde{D}_\perp$$
$$\ell 1(n) \sqsupseteq \alpha(\{x \mid \ell(x) = n\})$$

A trivial solution is $\ell 1$ as the constant function *all* but there will normally be much better approximations.

Example. For the length function in lisp

$$\begin{aligned} \mathit{length}(x) = {} & \textbf{if } \mathit{null}(x) \textbf{ then } 0 \\ & \textbf{else } 1 + \mathit{length}(\mathit{cdr}(x)) \end{aligned}$$

An approximation (in this case exact) to the inverted length function is

$$\begin{aligned} \mathit{length}1(n) = {} & \textbf{if } n = 0 \textbf{ then } \mathit{nil} \\ & \textbf{else } \mathit{cons}(\mathit{all}, \mathit{length}1(n-1)) \end{aligned}$$

This is exact since $\gamma(\mathit{length}1(n))$ gives the set of lists with length n.

A computable time bound function can be constructed as

$$\begin{aligned} & tb(n_1, \ldots, n_i) = \\ & \quad (\tilde{\mathbf{U}}[\![\mathcal{T}[\![P]\!]]\!])_1 \ \langle \ell 1(n_1), \ldots, \ell 1(n_i) \rangle \end{aligned}$$

The time bound function may return infinity to denote possible nontermination of the original program. The infinity symbol is a limit point in the domain $\quad^\infty$ and can therefore not always be found after finite fixpoint iteration. All other results can be found after finite iteration and can therefore be found by a program.

We will derive a program to compute the time bound function. This time bound program will terminate if and only if the time bound function returns values different from the infinity symbol.

Translation. The $\tilde{\mathbf{U}}$-interpretation is based on the set $\tilde{D}$ where the symbol *all* has a special interpretation. If we want to compile the step-counting version to a program that computes the same function in the standard semantics we assume that *all* is a unique atom not used by any of the user defined functions. With this encoding of the *all* symbol from $\tilde{D}$ the translation to the standard semantics is reasonably straight forward. We describe it here as a syntax-directed translation scheme similar to the $\mathcal{T}$ function but we will not prove it correct or justify it further. The translation function is called $\mathbf{C} : \langle \textit{t-program} \rangle \rightarrow \langle \textit{program} \rangle$.

$$\begin{aligned} \mathbf{C}[\![\ & T_1(x_1, \ldots, x_{i1}) = E_{t1} \\ & \vdots \\ & T_n(x_1, \ldots, x_{in}) = E_{tn} \\ & F_1(x_1, \ldots, x_{i1}) = E_{s1} \\ & \vdots \\ & F_n(x_1, \ldots, x_{in}) = E_{sn}]\!] = \end{aligned}$$

$$\begin{aligned} & T_1(x_1, \ldots, x_{i1}) = \ \mathbf{C}_\tau[\![E_{t1}]\!] \\ & \vdots \\ & T_n(x_1, \ldots, x_{in}) = \ \mathbf{C}_\tau[\![E_{tn}]\!] \\ & F_1(x_1, \ldots, x_{i1}) = \ \mathbf{C}_\sigma[\![E_{s1}]\!] \\ & \vdots \\ & F_n(x_1, \ldots, x_{in}) = \ \mathbf{C}_\sigma[\![E_{sn}]\!] \\ & \mathit{car}1(x) = \textbf{if } \mathit{eq}(x, \mathit{all}) \textbf{ then } \mathit{all} \\ & \qquad\qquad \textbf{else } \mathit{car}(x) \\ & \mathit{lub}1(x, y) = \textbf{if } \mathit{and}(\mathit{eq}(x, y), \mathit{eq}(x, \mathit{nil})) \\ & \qquad\qquad \textbf{then } x \\ & \qquad\qquad \textbf{else if } \mathit{or}(\mathit{eq}(x, \mathit{all}), \mathit{or}(\mathit{eq}(y, \mathit{all}), \\ & \qquad\qquad\qquad \mathit{or}(\mathit{atom}(x), \mathit{atom}(y)))) \\ & \qquad\qquad \textbf{then } \mathit{all} \\ & \qquad\qquad \textbf{else } \mathit{cons}(\ \mathit{lub}1(\mathit{car}(x), \mathit{car}(y)), \\ & \qquad\qquad\qquad \mathit{lub}1(\mathit{cdr}(x), \mathit{cdr}(y))) \\ & \vdots \end{aligned}$$

where

$$\begin{aligned} & \mathbf{C}_\tau[\![n]\!] = \ n \\ & \mathbf{C}_\tau[\![\mathit{add}(E_1, \ldots, E_i)]\!] = \\ & \quad \mathit{add}(\ \mathbf{C}_\tau[\![E_1]\!] \ , \ \ldots \ , \ \mathbf{C}_\tau[\![E_i]\!] \ ) \\ & \mathbf{C}_\tau[\![T_j(E_1, \ldots, E_i)]\!] = \\ & \quad T_j(\ \mathbf{C}_\tau[\![E_1]\!] \ , \ \ldots \ , \ \mathbf{C}_\tau[\![E_i]\!] \ ) \\ & \mathbf{C}_\tau[\![\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3]\!] = \\ & \quad \textbf{if } \mathit{eq}(\ \mathbf{C}_\sigma[\![E_1]\!] \ , \mathit{all}) \\ & \quad \textbf{then } \mathit{max}(\ \mathbf{C}_\tau[\![E_2]\!] \ , \ \mathbf{C}_\tau[\![E_3]\!] \ ) \\ & \quad \textbf{else if } \ \mathbf{C}_\sigma[\![E_1]\!] \\ & \quad \textbf{then } \ \mathbf{C}_\tau[\![E_2]\!] \ \textbf{ else } \ \mathbf{C}_\tau[\![E_3]\!] \end{aligned}$$

$$\begin{aligned} & \mathbf{C}_\sigma[\![c]\!] = \ c \\ & \mathbf{C}_\sigma[\![x_i]\!] = \ x_i \\ & \mathbf{C}_\sigma[\![\mathit{car}(E)]\!] = \ \mathit{car}1(\ \mathbf{C}_\sigma[\![E]\!] \ ) \\ & \mathbf{C}_\sigma[\![\mathit{cons}(E_1, E_2)]\!] = \\ & \quad \mathit{cons}(\ \mathbf{C}_\sigma[\![E_1]\!] \ , \ \mathbf{C}_\sigma[\![E_2]\!] \ ) \\ & \vdots \\ & \mathbf{C}_\sigma[\![F_j(E_1, \ldots, E_i)]\!] = \\ & \quad F_j(\ \mathbf{C}_\sigma[\![E_1]\!] \ , \ \ldots \ , \ \mathbf{C}_\sigma[\![E_i]\!] \ ) \\ & \mathbf{C}_\sigma[\![\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3]\!] = \\ & \quad \textbf{if } \mathit{eq}(\ \mathbf{C}_\sigma[\![E_1]\!] \ , \mathit{all}) \\ & \quad \textbf{then } \mathit{lub}1(\ \mathbf{C}_\sigma[\![E_2]\!] \ , \ \mathbf{C}_\sigma[\![E_3]\!] \ ) \\ & \quad \textbf{else if } \ \mathbf{C}_\sigma[\![E_1]\!] \\ & \quad \textbf{then } \ \mathbf{C}_\sigma[\![E_2]\!] \ \textbf{ else } \ \mathbf{C}_\sigma[\![E_3]\!] \end{aligned}$$

The functions *car*1 and *lub*1 should really have been called F_{n+1} and F_{n+2} according to our function-name convention.

To construct the time bound program we now only need to compose the translated step-counting version with the inverted length function.

Example. In section 2 we used the union function from lisp as example

```
timebound(n,m) = tunion(length1(n),length1(m))

length1(n) = if eq(n,0) then nil
     else cons(all,length1(sub(n,1)))

tunion(x,y)  =
     if eq(null1(x),all) then
     max(4, add(tmember(car1(x),y) ,
       if eq(member(car1(x),y),all)
       then max(add( 8,tunion(cdr1(x),y)),
            add(11,tunion(cdr1(x),y)) )
       else if member(car1(x),y)
            then add( 8,tunion(cdr1(x),y))
            else add(11,tunion(cdr1(x),y)) )
     )
     else if null1(x) then 4
     else add(tmember(car1(x),y) ,
       if eq(member(car1(x),y),all)
       then max(add( 8,tunion(cdr1(x),y)),
            add(11,tunion(cdr1(x),y)) )
       else if member(car1(x),y)
            then add( 8,tunion(cdr1(x),y))
            else add(11,tunion(cdr1(x),y)) )
     )

tmember(x,s) =
     if eq(null1(s),all) then max(4,
        if eq(eq1(x,car1(s)),all)
        then max(9,add(12,tmember(x,car1(s)))
        else if eq1(x,car1(s)) then 9
             else add(12,tmember(x,car1(s)))
     )
     else if null1(s) then 4
     else if eq(eq1(x,car1(s)),all)
          then max(9,add(12,tmember(x,car1(s)))
          else if eq1(x,car1(s)) then 9
               else add(12,tmember(x,car1(s)))

member(x,s) =
     if eq(null1(s),all) then lub1(false,
        if eq(eq1(x,car1(s)),all)
        then lub1(true,member(x,car1(s)))
        else if eq1(x,car1(s)) then true
             else member(x,car1(s))
     )
     else if null1(s) then false
     else if eq(eq1(x,car1(s)),all)
          then lub1(true,member(x,car1(s)))
          else if eq1(x,car1(s)) then true
               else member(x,car1(s))

lub1(x,y) = if and(eq(x,y),eq(x,nil)) then x
            else if or(eq(x,all),or(eq(y,all),
                 or(atom(x),atom(y)))) then all
            else cons(lub1(car1(x),car1(y)),
                      lub(cdr1(x),cdr1(y)))

null1(x) = if eq(x,all) then all else null(x)

car1(x)  = if eq(x,all) then all else car(x)

cdr1(x)  = if eq(x,all) then all else cdr(x)

eq1(x,y) = if or(eq(x,all),eq(y,all))
           then all else eq(x,y)
```

There is nothing like computer generated programs!

This program is a real gold mine for optimisations. With rather simple transformations preserving total correctness the same program can be written as

```
timebound(n,m) =
  add(4,add(mul(19,n),mul(12,mul(n,m))))
```

or more readably

$$\mathit{timebound}(n,m) = 4 + 19 * n + 12 * n * m$$

This is the time bound program for the union function.

5. Extensions and efficiency

The last section described a way to construct a time bound program. It is a program in the standard semantics that computes an upper bound for the computation time of the original program from the input size. We can not guarantee that the program will be in closed form but this is a standard source-to-source program transformation/optimisation problem. Unlike the automatic complexity analysis there is a large amount of literature on this problem.

A system to make source-code optimisation is described in [Rosendahl 1986] and it is strong enough to transform the time bound program of the union function to closed form (see previous example). The system is based on three different techniques: an abstract interpretation based on data set definitions as described by [Reynolds 1969] to simplify unnecessary tests for the value *all*. Driving as described by [Turchin 1986] together with solution of finite-difference equations is used to remove the intermediate lists produced by the inverted length functions.

We will not discuss the system further here but instead show some other applications and improvements.

Step-counting versions for other languages. The construction of time bound functions can be used for other languages provided it is possible to construct step-counting versions written in first-order lisp. This has been done successfully both for an imperative language and for a lazy functional language. In both cases it was done by writing a step-counting interpretation (like $\mathbf{U}'$) for the language in first-order lisp. This means that even though the interpretation is more time consuming than running the original program, the notion of execution time will not be altered.

The claim that we can analyse programs in other languages is formally meaningless as the execution time is an internal property and normally not part of the definition of a language. For all practical purposes there is a natural meaning like the number of basic operation or the number of reductions performed under the evaluation.

Size-complexity. The normal meaning of size-complexity for lisp-like languages is a function that gives an upper bound of the size of the results as a function of input size (cp. [Métayer 1988] and [Wegbreit 1975]). This complexity function can be constructed as the tilde-semantics of the original program composed with the size function. This is easier to construct than the time-complexity because the size is an external property.

The cons-complexity is another type of storage-complexity. It will give an upper bound of the number of cons-operations as a function of the input size. To construct it we can use the same method as for the time-complexity except that the step-counting version should only count 1 for cons-operations and zero for all other operations.

The real storage complexity for a lisp program should give an upper bound of the necessary storage needed to evaluate the program. This will be less than or equal to the cons-complexity (hence the cons-complexity is a safe approximation) with the overhead being the number of storage cells that can be removed by garbage collection. The necessary-storage complexity is much more difficult to construct than the previous two. It can be done by making an instrumented reference-count semantics—a semantics that returns a pair of the result and the maximum number of cells referenced by parameters under the evaluation. This is far from trivial so the cons-complexity will in most situations be the most useful. Notice by the way that the size-complexity in certain situations can give higher values than the cons-complexity.

Better approximation of the power set. The central idea in the construction of time bound programs is the use of the domain $\tilde{D}$ to approximate the power set of S-expressions. The method works because the domain has finite height unlike (D) and that makes the time bound program computable.

The approximation can be improved by extending the domain $\tilde{D}$ with more special elements like *all* (which denotes the set of all S-expressions),

atoms	the set of atoms
numbers	the set of numbers
evennums	the set of even numbers
oddnums	the set of odd numbers
bool	the set $\{true, false\}$
$\vdots$	

Such an extension will make *lub*, γ, and α more complicated but they can still be continuous and the domain will have finite height.

6. Conclusion

This paper describes a method to construct time bound programs for programs in a first-order lisp subset. The method has been proved correct in the sense that if the time bound program returns a value then the original program will terminate for all inputs of the given size and the computation time will be less than or equal to the value returned by the time bound program.

All the algorithms described in this paper have been realised in Franz Lisp. Simplification of the resulting time bound program is described in [Rosendahl 1986] and these algorithms are also implemented in Franz Lisp.

Applications. The complexity function for a program can be useful as an estimate of the efficiency of the algorithm. The system could be used by the programmer to construct more efficient algorithms. A more interesting application would be to use it to guide a program transformation system. This idea was proposed in [Wegbreit 1976] where the transformations leading to more efficient algorithms are preferred. [Sarkar & Hennessy 1986] discuss compile-time scheduling of parallel programs. Their method uses profile data to assign computations to processors so as to minimise the parallel execution time. The complexity function could naturally be used for such purposes.

Restrictions. The method has its limitation—of course it cannot solve the halting problem. The method has been successfully used on a number of programs including matrix algorithms, sorting programs and a deterministic parser. Due to the precision of the approximation of the power set (D) with the domain $\tilde{D}$ it only works for programs where the recursion is controlled by structural constraints, like the length of list. This can be improved by extending the set as discussed in the last section.

An extended $\tilde{D}$-domain can improve the approximations but there are certain situations where the approach will fail miserably. If the input size measure is the size-function (the function that gives the number of cells in a structure) then the inverted size function will be the constant function *all* and the time bound will be independent of their size for all input.

It is normally desirable to have the time bound program in closed form, an expression in the input size not containing any user defined functions. We have not addressed the issue in this paper—it is part of the more general source-code program optimisation problem. In a sense this work can be seen as reducing the problem of automatic program analysis to the well studied program optimisation problem.

Acknowledgements. Thanks to Neil D Jones, Alan Mycroft, Juanito Camilleri, Troy Ferguson, and numerous other people for helpful discussions and comments on drafts of this paper.

References

[Adachi, Kasai & Moriya 1979] A Adachi, T Kasai and E Moriya. *A Theoretical Study on the Time Analysis of Programs.* LNCS **74**, pp. 201–207. Springer-Verlag, 1979.

[Aho, Hopcroft & Ullman 1974] A V Aho, J E Hopcroft and J D Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.

[Cohen & Katcoff 1977] J Cohen and J Katcoff. *Symbolic Solution of Finite-Difference Equations.* ACM Trans. Math. Software **3**(3), pp. 261–271, Sept., 1977.

[Cohen 1982] J Cohen. *Computer-Assisted Microanalysis of Programs.* C. ACM **25**(10), pp. 724–733, Oct., 1982.

[Cousot & Cousot 1977] P Cousot and R Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.* In *4th POPL, Los Angeles, CA*, pp. 238–252, Jan., 1977.

[Flajolet 1987] P Flajolet. *Mathematical Tools for Automatic Program Analysis.* Tech. Rep. 603. Rocquencourt, Feb., 1987.

[Gordon 1979] M J C Gordon. *The Denotational Description of Programming Languages: An Introduction.* Springer-Verlag, 1979.

[Hickey & Cohen 1985] Hickey and Cohen. *Automatic Program Analysis.* J. ACM. **35**(1), pp. 185–220, 1985.

[Kasai et al 1980] Kasai et al. *An Automatic Time Analysis System.* Tech. Rep. RIMS-335. Kyoto Univ., May, 1980.

[McCarthy et al 1965] J McCarthy et al. *Lisp 1.5 Programmers Manual*, second edition. MIT Press, 1965.

[Métayer 1985] D L Métayer. *Mechanical Analysis of Program Complexity.* In *SIGPLAN '85 Conference, Seattle, WA*, pp. 69–73. Volume 20(7) of ACM SIGPLAN Not., July, 1985.

[Métayer 1988] D L Métayer. *ACE: An Automatic Complexity Evaluator.* ACM TOPLAS **10**(2), pp. 248–266, Apr., 1988.

[Reynolds 1969] J C Reynolds. *Automatic Computation of Data Set Definitions.* In *IFIP'68*, pp. 456–461, 1969.

[Rosendahl 1986] M Rosendahl. *Automatic Program Analysis.* M.Sc. Thesis. DIKU, Univ. of Copenhagen, Denmark, 1986.

[Sarkar & Hennessy 1986] V Sarkar and J Hennessy. *Compile-time Partitioning and Scheduling of Parallel Programs.* In *SIGPLAN '86 Symposium on Compiler Construction, Palo Alto, California*, pp. 17–26. Volume 21(7) of ACM SIGPLAN Not., July, 1986.

[Talcott 1986] C Talcott. *Derived properties and derived programs.* Internal Report. Univ. of Stanford, May, 1986.

[Turchin 1986] V F Turchin. *The Concept of a Supercompiler.* ACM TOPLAS **8**(3), pp. 292–325, July, 1986.

[Wadler 1988] P Wadler. *Strictness Analysis Aids Time Analysis.* In *15th POPL, San Diego, California.* ACM Press, Jan., 1988.

[Wegbreit 1975] B Wegbreit. *Mechanical Program Analysis.* C. ACM **18**(9), pp. 528–539, Sept., 1975.

[Wegbreit 1976] B Wegbreit. *Goal-Directed Program Transformation.* IEEE Trans. Software Engrg. **SE-2**(2), pp. 69–80, June, 1976.