Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 153 (2006) 35–53

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Automatic Complexity Analysis for Programs Extracted from Coq Proof

Jean-Pierre Jouannaud[1] Weiwen Xu[2]

LIX, École Polytechnique
91400 Palaiseau, France

Abstract

We describe an automatic complexity analysis mechanism for programs extracted from proofs carried out with the proof assistant Coq. By extraction, we mean the automatic generation of MiniML code [3]. By complexity analysis, we mean the automatic generation of a description of the time-complexity of a MiniML program in terms of the number of steps needed for its execution. This description can be a natural number for *closed program*, that is, programs coming along with their actual inputs. For programs per se, the description is given in terms of a set of recurrence relations which relate the number of steps of a computation in terms of the size of the inputs. Going from these recurrence relation to actual complexity functions is a hard task that requires the use of sophisticated tools for symbolic computations. This part is not implemented for the moment although we have manually used the MAPLE computer algebra system in some cases.

Keywords: theorem proving, program extraction, complexity analysis

1 Introduction

1.1 Background

A market is slowly building up in the area of proof development systems. Security applications in the banking business now require high level of confidence in computer applications, that can only be achieved by proving code correct with proof checkers. One of the most successful companies adressing this market is Trusted Logics, whose technology is based on Coq. For security

[1] This work was partly supported by the RNTL project AVERROES and by France-Telecom.

[2] Email: `xu@lix.polytechnique.fr`

doi:10.1016/j.entcs.2005.08.005

applications, proving some code correct with respect to its specification or extracting it from a proof of that specification is not enough. It must also be taken care of the environment. For smart cards applications in particular, this means a limited amount of ressources. Therefore, a correstness proof should be coupled with an analysis of the ammount of ressources needed for running a program.

Here, we consider programs extracted from Coq proofs. There are two reasons for our choice of extracted programs rather than proved programs). For the first, we know that the code is a well-defined subset the functional language MiniML. For the second, our intuition was that the proof carries more information than the program itself, since its properties are also described. Some of these properties could help in a complexity analysis. We have not exploited this intuition yet, but will elaborate on this in conclusion.

One may think of delegating the job of making the complexity analysis to the user, who would do a proof of the complexity properties of a program at the same time as proving the program correct with respect to its specification. We do not believe in this approach. Doing proofs is difficult, and the trend is to make the user's life easier, not harder. We therefore think that the complexity analysis should be automatic.

We could also fix a bound on the time spent in a given computation, and compute beforehand with a test set of inputs in order to estimate which ones yield computations which are safe with respect to the bound. This approach is very similar to testing, and it is well known that constructing complete test sets is very difficult, presumably as difficult as doing a proof, however resulting in a lower level of security.

On the other hand, a lot of work has been done already for the automatic analysis of imperative programs, especially for the average case analysis. The method is now well established. The behaviour of recursive programs is characterized by recurrence equations whose mathematical analysis, in case no closed form can be found that describes the expected complexity function, allows to study the assymptotic behaviour of the program.

It turns out that the statistical analysis of the computational behaviour of imperative programs has been investigated in much more depth than the worst case complexity of functional programs. One of the reasons is that functional programs do not have a widely recognized operational model. There is still some dispute in the community wether the evaluation strategy should be by value, by name, lazy, strict, etc. Besides, counting the number of steps in a program execution may be seen as a very rough approximation of the time spent at runtime, since different steps may take very different time. However, several authors have considered the problem. The most successful one is by Benzinger, who derives recurrence relations relating the number of steps needed in the execution of a functional program to the size of its inputs [2]. His implementation which was carried out in the NuPRL project, is however very limited. Only programs with nutural numbers or lists as inputs were considered.

1.2 Problem

Our program is to build general tools for analysing the complexity of functional programs written in MiniML. We are not only interested in time complexity, but also in space complexity. We know that the latter analysis should be more difficult than the former, since the space used by a program is extremely dependent from the compiler technology. Finally, we are not interested in programs whose behaviour is exponential (or more), but in programs whose actual complexity is bound by a polynomial of low degree.

1.3 Contribution

Our contribution is a formal framework and an implementation that allow us to compute a description of the complexity of a given MiniML program. As the operational semantics of a program, its complexity sould be compositional. This leads to decorate the rules describing the operational semantics of MiniML by some complexity information that will then allow to compute the complexity of a closed program by evaluating these rules. For terms with variables (or parameters), the complexity depends of course on the value of these variables. In case these variables are higher-order, it also depends from the complexity of the actual functions that will instantiate these variables. For such programs, our method generates a symbolic description of this complexity, which can then be transformed into a more convenient format: recurrence relations. This method is partly implemented in OCaml. The analysis of the recurrence relations by a computer algebra system is not done yet in general, but can be done in particular cases which are simple enough.

2 Annotated Semantics

The computational complexity refers to an asymptotic relation between the size of the input to a function and the time it takes to compute the output. Most formal definitions of computational complexity are based on the Turing machine or random access machine model and assign certain costs to a designated set of machine operations.

Following Benzinger, we introduce in this section a general framework for reasoning about the computational complexity of a functional program relative to an operational semantics $\mathcal{O}$ by annotating each rule $t_1 \downarrow t_2$ in $\mathcal{O}$ with some complexity information n, written as

$$t_1 \downarrow^{\mathcal{A}} t_2 \text{ (in } n),$$

This yields an *annotated semantics* $\mathcal{A}$. The term t_2 need not be canonical (in case of a small step semantics is used), and n may be an arbitrary mathematical expression. In this paper, however, we will use a big step semantics, and therefore, t_2 will always be canonical.

Depending on the computational resource of interest and our assumptions about the underlying machine model, the annotations might model upper bounds,lower bounds, or exact quantities. As an example, consider the following constructor rule for succ:

$$\text{(succ)}\ \frac{u{\downarrow}k}{\text{succ}(u){\downarrow}\text{succ}(k)}$$

We can measure time complexity by defining

$$\text{(time)}\ \frac{u{\downarrow}k(\text{in } n)}{\text{succ}(u){\downarrow}k{+}1(\text{in } n{+}1)}$$

or space complexity by defining

$$\text{(space)}\ \frac{u{\downarrow}k(\text{in } n)}{\text{succ}(u){\downarrow}k{+}1(\text{in } n)}$$

2.1 Annotated semantics of Miniml

Miniml is functional language used for extracting programs from Coq proofs [3]. Its concrete syntax ressembles that of Ml, as shown by the following simple exemple:

```
let rec length = function
    | nil -> 0
    | Cons (a,m) -> S (length m)
```

while its abstract syntax uses a more compact form, closer to the syntax in Coq:

$$\begin{aligned} t ::= & \ x \mid \lambda x.t \mid [x:t]t \mid \ \text{apply}\ (t;t) \\ & \mid \ \ \text{Cases}\ t\ \text{of}\ u_1 \Rightarrow v_1, \cdots, u_n \Rightarrow v_n \text{end} \\ & \mid \ \text{ind}(t; v_1; \lambda z.v_2) := \ \text{case}\ t\ \text{of}\ \mathbf{b} \Rightarrow v_1, \mathbf{s}(t') \Rightarrow v_2[\text{ind}(t'; v_1; \lambda z.v_2)/z] \end{aligned}$$

where x is taken from a denumerable set of variables, $[x : u]v$ stands for the usual let construct `let x = u in v`, and b and s stand for the constructors of the inductive type to which the variable t belongs (we assume here for simplicity of notation that there are two constructors).

The semantics of Miniml is now given by the following set of annoted rules. Assuming A assigns a unit cost to each reduction step, we get the following rules:

$$\text{(canon)} \qquad w \downarrow w \text{ (in } 0)$$

$$\text{(abs)} \qquad \frac{u{\downarrow}v \text{ (in } n)}{\lambda x.u{\downarrow}\lambda x.v \text{ (in } n{+}1)}$$

$$\text{(apply)} \qquad \frac{f{\downarrow}\lambda x.b \text{ (in } n_1), u{\downarrow}w_0 \text{ (in } n_2), b[w_0/x]{\downarrow}w \text{ (in } n_3)}{\text{apply}(f;u){\downarrow}w \text{ (in } n_1{+}n_2{+}n_3{+}1)}$$

$$\text{(let-in)} \qquad \frac{a{\downarrow}w_0 \text{ (in } n_1), b[w_0/x]{\downarrow}w_2 \text{ (in } n_2)}{[x{:}a]b{\downarrow}w_2 \text{(in } n_1{+}n_2{+}1)}$$

$$\text{(cases)} \qquad \frac{t{\downarrow}w \text{ (in } n_1), \text{Match } (w,u_1,\cdots,u_p){\downarrow}(i,\eta) \text{ (in } n_2), v_i\eta{\downarrow}u \text{ (in } n_3)}{(\text{Case } t \text{ of } u_1{\Rightarrow}v_1,\cdots,u_p{\Rightarrow}v_p \text{ end}){\downarrow}u \text{ (in } n_1{+}n_2{+}n_3{+}1)}$$

$$\text{(induction)} \qquad \frac{t{\downarrow}w_0 \text{ (in } n_1), \text{Match}(w_0,\mathbf{b},\mathbf{s}){\downarrow}(i,\eta) \text{ (in } n_2), v_i\eta{\downarrow}w \text{ (in } n_3)}{\text{ind}(t;v_1;\lambda z.v_2){\downarrow}w \text{ (in } n_1{+}n_2{+}n_3{+}1)}$$

where the rules for Match are as follows, assuming that $u_1, \cdots, u_p$ are expressions of depth one and that Match$(w_0, \mathbf{b}, \mathbf{s})$ also returns (by convention) the appropriate value of z for the recursive call:

$$\text{(start)} \qquad \frac{\text{Match0}(w, 0, u_1, \cdots, u_n) \downarrow (i, \eta) \text{ (in } n)}{\text{Match}(w, u_1, \cdots, u_p) \downarrow (i, \eta) \text{ (in } n)}$$

$$\text{(failure)} \qquad \frac{\text{Match0}(f(\bar{w}), k+1, u_1, \cdots, u_p) \downarrow (i, \eta) \text{ (in } n)}{\text{Match0}(f(\bar{w}), k, g(\bar{v}), u_1, \cdots, u_p) \downarrow (i, \eta) \text{ (in } n+1)}$$

$$\text{(success)} \qquad \frac{}{\text{Match0}(f(\bar{w}), k, f(\bar{x}), u_1, \cdots, u_p) \downarrow (k, \{\bar{x} \mapsto \bar{w}\})}$$

The above rules allow to compute the number of steps taken by any expression of base type without free variable. Of course the computation of the number of steps for such an expression e forces its evaluation into its actual value. Unlike the usage, we have a rule for abstractions, making our life easier later.

3 Complexity of open expressions

We now move on to programs with variables. Programs are closed expressions fo the form $\lambda\overline{x}.e$, where the body e is an expression of arbitrary type, like the identity function $\lambda X : \alpha.X$, where α can be any type. The definition of a program will be made more precise later.

There is a big difference between the annotated semantics of a closed expression of base type and that of a program. In the first case, we can simply run the operational semantics and compute accordingly the output value and the number of steps it takes to reach the output value. In the second case, the program has input variables which will later be instantiated by arbitrary expressions. The number of steps needed for calculating the result of applying a program to particular input expressions will depend both on the complexity of calculating the values of these expressions and on the values themselves.

On the other hand, we can still run the annotated semantics until a variable is reached, or until an abstraction, a let-in, a case, or a recursion, or an application is blocked by a variable. Remember that applications (and let-expressions) commute with all other constructs (except with abstractions), after possibly renaming bound variables. Therefore,

Definition 3.1 An expression e is *blocked* if it is of the following form:

- $x \in \mathcal{X}$;
- $\lambda x.u$, where u is a blocked expression;
- $(u\ v)$, where u is a variable or a blocked application;;
- $[y : u]v$, where u is a variable or a blocked application;
- case t of $u_1 \Rightarrow v_1, \ldots, u_n \Rightarrow v_n$ end , where t is a blocked expression;
- ind $(t; v_1; \lambda z.v_2)$, where t is a blocked expression.

To an arbitrary expression e of type α and free variables $x_1, \ldots, x_n$, we associate a pair made of its value e^* and the number of steps $\overline{e}$ needed to reduce e to its value e^*. These notations should be thought of as relative to a valuation γ replacing the variables in $x_1, \ldots, x_n$ by appropriate expressions: relatively to this valuation γ, the notations e, e^* and $\overline{e}$ stand respectively for the expression $\gamma(e)$, the value of $\gamma(e)$ and the number of steps needed to reduce $\gamma(e)$ to its value. Intuitively:

$$e \downarrow e^* \text{ (in } \overline{e})$$

The complexity of the expression e will then be a function $\lambda x_1 \ldots x_n.Cpx(e)$ abstracting the number of steps from a particular valuation, that is satisfying $Cpx(e)(\gamma) = \overline{e}$, where $\overline{e}$ is relative here to this precise valuation γ.

To make this intuition precise, we need to define what is a valuation, this is related to the substitutivity property of complexity functions. Assume an expression depends on a free variable x of basic type. Then, its complexity will depend both on the value x^* and the number of steps $\overline{x}$ of x. In this case, a valuation should replace a variable x by the pair $(x^*, \overline{x})$. The case of a functionnal type will need a more complex valuation which will reflect the functional structure of the type.

3.1 Compositionality

For a structured expression, value, number of steps and complexity must be calculated from the corresponding values, number of steps and complexities for their subsexpressions, a principle called *compositionality.* For the case of complexity functions, our main principle is therefore that the complexity should be a transformation acting as a morphism from programs into complexity functions. For example, the complexity of an abstraction should be an abstraction over the complexity of its body. Besides, in the particular case where an expression e has no blocked subsexpression (hence no free variable), then both its value and number of steps should agree with the result obtained by applying the annotated semantics, and the complexity should be the number of steps itself. The difficulty of putting this principle into practice comes from expressions of higher type. In order to reduce the complexity of such an expression to the complexity of another simpler expression because it is of basic type, we will assume that programs are in η-expanded form.

Definition 3.2 A program is a well-typed, blocked, closed expression in which every subexpression of functional type is an abstraction or the left-argument of an application.

For example, to be considered as a program in our sense, the identity function $\lambda X : (\alpha \to \alpha) \to (\alpha \to \alpha).X$, where α is a base type, should be written as

$$\lambda X : (\alpha \to \alpha) \to (\alpha \to \alpha)\ \lambda y : \alpha\ \lambda f : \alpha \to \alpha.((X(\lambda x : \alpha.(fx)))y)$$

There is of course no real difference between the above identity function and the expression $((X(\lambda x : \alpha.(fx)))y)$ typable in the environment $\{X : (\alpha \to \alpha) \to (\alpha \to \alpha)\ f : \alpha \to \alpha\ y : \alpha\}$, or the expression $\lambda y : \alpha\ \lambda f : \alpha \to \alpha.((X(\lambda x : \alpha.(fx)))y)$ typable in the environment $\{X : (\alpha \to \alpha) \to (\alpha \to \alpha)\}$. In other words, the type of the complexity function of an expression e typable in an environment Γ depends both on its type, and on the type of its free variables as given in Γ. This remark will be put into pratice by removing outside abstractions and defining the complexity of open expressions as a function of their free variables seen as formal parameters.

Definition 3.3 We define the *arity* of a type α, written $ar(\alpha)$, to be the number n such that $\alpha = \alpha_1 \to \ldots \to \alpha_n \to \beta$ with β a basic type. The arity of an expression is the arity of its type.

In the previous example of the identity function of higher type, $ar(X) = 2, ar(f) = 1$ and $ar(x, y) = 0$. The arity of a type α will indeed be the arity of the complexity function of any *program* of type α. In particular, a program of arity zero is a ground expression of base type, hence can be evaluated into a value in a given number of steps. For programs of non-zero arity, we now define the type of the complexity function and the related notion of valuation:

Definition 3.4 Given a type α, we define the type $\underline{\alpha}$ as:

$$\text{if } \alpha \text{ is a base type:} \qquad \underline{\alpha} = (\alpha \times \mathbb{N})$$
$$\text{otherwise: } \underline{\alpha \rightarrow \beta} = (\alpha \times \underline{\alpha}) \rightarrow \underline{\beta}$$

Definition 3.5 Given a program e of type $\alpha = \alpha_1 \rightarrow \ldots, \alpha_n \rightarrow \beta$, we say that $Cpx(e) : \underline{\alpha}$ is a *complexity function* for e if

$$Cpx(e)(Cpx(u_1), \ldots, Cpx(u_n)) = m \text{ iff } (e\ u_1 \ldots u_n) \downarrow r\ (in\ m)$$

for arbitrary closed expressions in η-expanded form $u_1 : \alpha_1, \ldots, u_n : \alpha_n$. The mapping $\{x_1 \mapsto Cpx(u_1), \ldots, x_n \mapsto Cpx(u_n)\}$ is called a *valuation* of $x_1, \ldots, x_n$.

3.2 Computing complexity expressions

We now show how the complexity of a program is recursively defined according to the principle of compositionality. The reader is invited to check that types match. We will end up with the case of applications, which is the difficult one. We will use $Cpx(t).1$ for t^* and $Cpx(t).2$ for $\overline{t}$. We concentrate on the definition of $Cpx(t).2$.

3.2.1 Abstractions

Let $e = \lambda x_1 : \alpha_1 \ldots x_n : \alpha_n.u$ be a closed expression in which u is not an abstraction, hence is of basic type by assumption that programs are in η-expanded form.

$$Cpx(\lambda x_1 : \alpha_1 \ldots \lambda x_n : \alpha_n.u) = \lambda x_1 : \underline{\alpha_1}, \ldots, x_n : \underline{x_n}.Cpx(u)$$

When computing $Cpx(u)$, we will assume that the variable x_i free in u has value x_i^* and evaluates to its value in a number of steps equal to $\overline{x_i}$. This amounts to consider that $Cpx(u)$ is a function of the variables $x_1 : \underline{\alpha_1}, \ldots, x_n : \underline{x_n}$. This is why we say that the complexity of an expression depends on its free variables. Formally, this is not he case: the complexity of an expression depends on its type computed in an environment assigning types to its free variables.

3.2.2 Case expressions

$$Cpx(\text{case } t \text{ of } u_1 \Rightarrow v_1, \ldots, u_n \Rightarrow v_n \text{ end}).2 =$$
$$1 + Cpx(t).2 + \text{case } Cpx(t).1 \text{ of } u_1 \Rightarrow 1 + Cpx(v_1).2, \ldots, u_n \Rightarrow$$
$$n + Cpx(v_n).2 \text{ end}$$

Note that the property that $u_1, \ldots, u_n$ have the same type implies that $Cpx(u_1)$, ..., $Cpx(u_n)$ are all waiting for inputs of the types, hence the case expression is well typed.

3.2.3 Let expressions

$$Cpx([y : u]v).2 = 1 + [y : Cpx(u)](Cpx(v).2)$$

3.2.4 Recursion

$$Cpx(\text{ind } (u; v; \lambda z : \alpha.w)).2 =$$
$$Cpx(u).2 + \text{ind } (Cpx(u).1; 1 + Cpx(v).2; 2 + Cpx(\lambda z : \alpha.w).2)$$

3.3 Applications

We now come to the heart of the definition of complexities: the case of an application, and more precisely of a basic type expression u which is the η-expanded form of a program, that is, an expression $(e\ a_1 \ldots a_n)$, where $e : \alpha_1 \to \ldots \to \alpha_n \to \alpha$ has arity n, and $a_1, \ldots, a_n$ are the η-expanded forms of distinct variables $x_1 : \alpha_1, \ldots, x_n : \alpha_n$. The coming discussion covers the case where e is a base type variable, by letting $\alpha_{n+1} = \alpha$. It also covers the case where a_i is the η-expansion of any argument expression, not necessarily a variable.

Assume $n = 0$. Then e is of basic type α, hence $Cpx(e : \underline{\alpha}).2 = \overline{e}$.

Assume $n > 0$. The call by-value evaluation of the expression $(e\ a_1 \ldots a_n)$ proceeds as follows:

$$(e\ a_1 \ldots a_n)^* = (\ldots((e^* a_1^*)^* a_2^*)^* \ldots a_n^*)^*$$

The structure of this computation is described by naming the subexpressions successively occuring in this computation:

Definition 3.6 To an expression $(e\ e_1 \ldots e_n)$ in η-expanded form, we associate its *decomposition*, which is a sequence of named expressions $\{e_i : \alpha_{i+1} \to$

$\ldots \to \alpha_n \to \alpha \mid i \in [0..n]\}$ defined as:

$$e_0 = e \quad \{e_i = (e^*_{i-1}\ a^*_i) \mid i \in [1..n]\} \quad \text{hence } (e\ a_1 \ldots a_n)^* = e^*_n$$

Decompositions play a fundamental role in the next section. In terms of evaluation steps, we get:

$$\frac{e \downarrow e^* \text{ (in } \overline{e}) \quad a_i \downarrow a^*_i \text{ (in } \overline{a_i}) \quad \{e_i \downarrow e^*_{i-1} \text{ (in } \overline{e_{i-1}}) \mid i \in [1..n]\}}{(e\ a_1 \ldots a_n) \downarrow e^*_n \text{ (in } \overline{e} + \Sigma_{i=1}^{i=n}(\overline{a_i} + \overline{e_i} + 1))} \tag{1}$$

In case a_i is of basic type, its number of steps $\overline{a_i}$ is a primitive quantity. Otherwise, it can be recursively decomposed into a sum of more primitive quantities by applying the same technique. We will see such an example in the next paragraph.

Our goal now is to express these numbers of steps in terms of complexities. To this end, we consider the expression $(e\ x_1 \ldots x_n)$, where $x_1, \ldots, x_n$ are distinct fresh variables of respective types $\alpha_1, \ldots, \alpha_n$. Let

$$\gamma_{i-1} \text{ be the valuation } \{x_i \mapsto (a^*_i, \overline{a_i}), \ldots, x_n \mapsto (a^*_n, \overline{a_n})\}$$

By our interpretation, $\overline{e_i} = Cpx(e_i).2(\gamma_i)$, and $\overline{a_i} = Cpx(a_i).2$ since a_i is ground. Substituting back complexity functions into (1) yields We omit the .2 everywhere):

$$\frac{e \downarrow e^* \text{ (in } Cpx(e)(\gamma_n)) \ \{e_{i-1} \downarrow e^*_{i-1} \text{ (in } Cpx(e_{i-1})(\gamma_{i-1})),\ a_i \downarrow a^*_i \text{ (in } Cpx(a_i)) \mid i \in [1..n]\}}{(e\ a_1 \ldots a_n) \downarrow e^*_n \text{ (in } Cpx(e)(\gamma_0) + \Sigma_{i=1}^{i=n}(Cpx(a_i) + Cpx(e_i)(\gamma_i) + 1))}$$

3.4 Variables

We now apply the previous discussion to the case of variables. The complexity of the variable X will be the number of steps needed for evaluating the expression $X\ a_1 \ldots a_n$, assuming that $a_1, \ldots, a_n$ are the η-expanded forms of distinct variables of the appropriate type. We will stick to number of steps here, although we could replace them by complexities as well.

Example 3.7 Consider a variable x of type α, where α is a base type. Then x is η-expanded, $x^* : \alpha$, and $Cpx(x) = \overline{x_0} : \mathbb{N}$ is defined as satisfying

$$x \downarrow x^*_0 \text{ (in } \overline{x_0})$$

Example 3.8 Consider now a variable f of type $\alpha \to \alpha$, where α is a base type. By our assumption, f occurs as left-argument of an application $(f\ a)$

where $a : \alpha$. Using our previous indexed notation $f_0 = f$, and $f_1 = f^*a^*$, we have:

$$\frac{f \downarrow f^* \text{ (in } \overline{f_0}), a \downarrow a^* \text{ (in } \overline{a}), f^*a^* \downarrow (fa)^* \text{ (in } \overline{f_1})}{fa \downarrow (fa)^* \text{ (in } \overline{f_0} + \overline{a} + \overline{f_1} + 1)}$$

Therefore,

$$(f\ a) \downarrow (f\ a)^* \text{ (in } \ Cpx(f)(a^*, \overline{a}))$$

with

$$Cpx(f)(a^*, \overline{a}) = \overline{f_0} + \overline{f_1} + \overline{a} + 1$$

Example 3.9 Consider finally a variable F of type $(\alpha \to \alpha) \to \alpha$, where α is a base types. By our assumption, F occurs as left-argument of the application $(F\ \lambda x : \alpha.(f\ x))$ where $f : \alpha \to \alpha$. Using again our previous indexed notation, we have:

$$\frac{\begin{array}{c} F \downarrow F^*(\text{in } \overline{F_0}) \\ f \downarrow f^*(\text{in } \overline{f_0}) \quad x \downarrow x^*(\text{in } \overline{x}) \quad (f^*\ x^*) \downarrow (f\ x)^*(\text{in } \overline{f_1}) \\ (F^*\ \lambda x.(f\ x)^*) \downarrow (F.\lambda x.(f\ x))^*(\text{in } \overline{F_1}) \end{array}}{(F\ \lambda x.(f\ x)) \downarrow (F.\lambda x.(f\ x))^*(\text{in } (\overline{F_0} + (\overline{f_0} + \overline{x} + \overline{f_1} + 1) + \overline{F_1} + 1))}$$

3.5 Main property

Lemma 3.10 *Given a program P with free variables $x_1, \ldots, x_n$, and a ground substitution $\sigma : x_1 \mapsto v_1, \ldots, x_n \mapsto v_n$, then*

$$Cpx(P\sigma) = Cpx(P)(x_1 \mapsto Cpx(v_1), \ldots, x_n \mapsto Cpx(v_n)) = \overline{P\sigma}$$

That is, $Cpx(t)$ satisfies our definition of a complexity function for t.

Proof. The proof should be by induction on the structure of P (not done).
□ □

3.6 Distance to the target

So far, we have not progressed very much. The complexity of a program is now expressed as a program whose parameters satisfy sets of equations. Of course, if we could solve these equations, we would get the complexity of the entire program. For example, if we know in advance the complexity of the program arguments, then the complexity of the program for these particular arguments would be described as a stand-alone program. We will not elaborate on this idea that we have not explored yet, and concentrate instead on trying to *guess* the complexity of a program and *verify* that the guess makes sense. For that, we will follow Benzinger's idea and consider the program itsef as a higher-order variable applied to its formal arguments, and try to apply the previous analysis described for applicative terms.

4 Expressing complexities as recurrence equations

Our language for calculating complexities does not really give us much insight about the possible polynomial growth (of a certain degree) of a given program. To achieve such a goal, we need to approximate these complexity functions. Approximations work themselves as morphisms. The approximation of a case expression is the maximum of the approximations of all branches. The approximation of a composition is the composition of the approximations. So is the case of a let expression. The difficulty comes with the inductive construct, since approximating the complexities involved in the recursive call does not suffice to obtain an approximation for the fixpoint itself. To handle this case, we introduce an intermediate step using recurrence relations. Going from recurrence relations to mathematical functions has been studied in depth and implemented in various computer algebra systems such as, for example, Mathematica or Mapple.

Besides the formal arguments of the program, the inductive term has possibly several inductive arguments.

Notation and Assumptions 1. We denote by $r(m)$ the recursive term

$$\mathrm{ind}(m; v; \lambda z.w[z]) := \mathrm{case}\ m\ \mathrm{of}\ \mathbf{b} \Rightarrow v,\ \mathbf{s}(t) \Rightarrow w[\mathrm{ind}(t, v, \lambda z.w[z])]$$

of type β in the environment $\{m : \alpha\}$ typing its inductive argument m. We assume that there is exactly one inductive argument for each recursive term (embedded recursions are ruled out). We also assume that m is in normal form, that is, $m = m^*$ since the complexity $\overline{m}$ of evaluating m is simpoly added to the resulting complexity $\overline{r}(m^*)$ as seen from the discussion in paragraphp:app. For simplicity of notations, we will use e' for the complexity (number of steps) of the expression e.

Since $r(m), v$ and $w[z]$ have the same type in their respective environments, their call-by-value evaluations have the same structure described in Paragraph 3.3, hence their decompositions are similar. Assume $r(m)$ has type $\alpha \to \beta$ in the environment $\{m : \alpha\}$. We then use $r_0(m)$ for $r(m)$, $r_1(m)$ for $r(m)^*\ x^*$, where $x : \alpha$, v_0 for v, v_1 for $v^*\ x^*$, $w_0[z]$ for $w[z]$ and $w_1[z]$ for

$w[z]^*\ x^*$. We have the following equations:

$$r(m) = \text{case } m \text{ of } \mathbf{b} \Rightarrow v,\ \mathbf{s}(k) \Rightarrow w[r(k)]$$
$$((r(m))^*\ x^*) = \text{case } m^* \text{ of } \mathbf{b} \Rightarrow (v^*\ x^*),\ \mathbf{s}(k) \Rightarrow (w[r(k)]^*\ x^*)$$

Since $r(m)^* = r^*(m^*) = r^*(m)$, the last equation becomes:

$$(r^*(m)\ x^*) = \text{case } m \text{ of } \mathbf{b} \Rightarrow (v^*\ x^*),\ \mathbf{s}(k) \Rightarrow (w[r(k)]^*\ x^*)$$

Applying our rules for computing complexities and lemma 3.10, we get:

$$r'_0(m) = \text{case } m \text{ of } \mathbf{b} \Rightarrow 1 + v'_0,\ \mathbf{s}(k) \Rightarrow 2 + w'_0[r'(k)]$$
$$r'_1(m) = \text{case } m \text{ of } \mathbf{b} \Rightarrow 1 + v'_1,\ \mathbf{s}(k) \Rightarrow 2 + w'_1[r'(k)]$$

which can be written as recurrence equations as follows:

$$r'_0(m) = \begin{cases} 1 + v'_0 & \text{if } m = \mathbf{b} \\ 2 + w'_0[r'_0(k)] & \text{if } m = \mathbf{s}(k) \end{cases} \qquad r'_1(m) = \begin{cases} 1 + v'_1 & \text{if } m = \mathbf{b} \\ 2 + w'_1[r'1(k)] & \text{if } m = \mathbf{s}(k) \end{cases}$$

Of course, when m is a natural number, then $k = m - 1$. Similarly, when m is a flat list $cons(a, k)$ of size n, then k is a flat list of size $n - 1$. The case of trees leads to clear difficulties, since we do not have any clue about the size and complexity of the left and right subtrees in terms of the whole tree. A worst case approximation is needed in this case. This leads to our second assumption:

Assumption 2. We assume that the inductive type α considered does not have a constructor whose type contains more than two occurrences of α. Natural numbers and lists satisfy this restriction. These are the only two inductive types accepted so far by our implementation.

Finally, we can easily derive the total complexity of the inductive definition by summing up the obtained complexities as explained in Paragraph 3.3. Examples are carried out in the next section.

5 Guessing complexities

For solving recurrence relations, computer algebra systems provide tools for solving systems of recurrence relations in one variable, wich explains our restriction that recursive programs depend up a single inductive variable. Unfortunately, even with this restriction, our method generates systems of recurrence relations depending upon several variables (or parameters). We therefore need to transform these systems of equations to cope with the possibilities of these systems.

5.1 *Handling parameterized linear recurrences*

The general form of our parameterized recurrence equations is

$$R(m, p_1, \cdots, p_n) = \begin{cases} F(p_1, \ldots, p_n) & \text{if } m = 0 \\ G(p_1, \ldots, p_n, m, R) & \text{if } m > 0 \end{cases} \tag{2}$$

where m is called the *argument* and $\bar{p}$ are the parameters of R. We say the equation is a linear equation if all parameters are scalars. We try to solve such recurrence equations by means of conventional methods. The main idea (at the same time, the main limitation) of our method is that we presuppose a particular form for the closed solution to the recurrence equations. More precisely, we assume that the closed solution is a linear combination of the parameters $\bar{p}$, and that the coefficients $\bar{c}$ of this combination are functions of m.

$$R^*(m, p_0, \cdots, p_n) := c_0(m) + c_1(m)p_0 + \cdots + c_{n+1}(m)p_n \tag{3}$$

Substituting R^* with R yields two polynomial over $\bar{p}$ with coefficients $c_i^*(0)$ and $c_i^*(m)$, respectively, which form a system of linear recurrence equations depending upon the single variable m.

$$c_i(m) = \begin{cases} c_i^*(0) & \text{if } m = 0 \\ c_i^*(m) & \text{if } m > 0 \end{cases} \tag{4}$$

Such a system can now be easily solved. Substituting the closed solutions for c_i back into R^* yields a closed solution for R.

As an example,

$$R(m, p_0, p_1, p_2, p_3) = \begin{cases} 1 & \text{if } m = 0 \\ p_1 + a + b + R(m-1, p_0, p_1, p_2, p_3) + p_3 & \text{if } m > 0 \end{cases} \tag{5}$$

We assume the form of closed solution is

$$R^*(m, p_0, p_1, p_2, p_3) = c_0(m) + c_1(m)p_0 + c_2(m)p_1 + c_3(m)p_2 + c_4(m)p_3 \tag{6}$$

Matching coefficient c_i on both sides of the equaiton, we obtain the system of recurrence equations.

$$\begin{aligned}
c_0(m) &= a+b+1+c_0(m-1); c_0(0) = 1 \\
c_1(m) &= c_1(m-1) \qquad ; c_1(0) = 0 \\
c_2(m) &= 2+c_2(m-1) \qquad ; c_2(0) = 0 \\
c_3(m) &= c_3(m-1) \qquad ; c_3(0) = 0 \\
c_4(m) &= 1+c_4(m-1) \qquad ; c_4(0) = 0
\end{aligned} \tag{7}$$

Solve these recurrence equations with Maple V9, we get the solution:

$$\begin{aligned}
c_0(m) &:= -a-b+(a+b+1)(m+1) \\
c_1(m) &:= 0 \\
c_2(m) &:= 2m \\
c_3(m) &:= 0 \\
c_4(m) &:= m
\end{aligned} \tag{8}$$

therefore

$$R(0,p_0,p_1,p_2,p_3) = \begin{cases} 1 & \text{if } m=0 \\ -a-b+(1+a+b)(m+1)+2mp_1+mp_3 & \text{if } m>0 \end{cases} \tag{9}$$

5.2 Implementation

Our system automatically generates recurrence relations in a format which is convenient for the computer algebraic system Maple V9. So far, the communication between both systems goes through a script file which can be read by Mapple. Once the output script is ready, it is feeded back to the system which uses the result to generate the closed form of the complexity expression of the program to be analysed.

5.3 Examples

We present now two examples illustrating the method. These examples have been obtained with a complexity model which departs slightly from the one described here. In particular, matching in case of a Case expression takes constant time 0, meaning that the selection of the appropriate branch is done by using an appropriate data structure based on indexing.

5.3.1 Higher Order Term

```
Require Import Sorting.
Extraction sort_rec.

(** val sort_rec: ‘a2 -> (‘a1 -> ‘a1 list -> __ -> ‘a2 -> __ -> ‘a2)
                      -> ‘a1 list -> ‘a2 **)

let rec sort_rec y h h0 = match y with
  | Nil -> h
  | Cons (a,l)  -> h0 a l __ (sort_rec h h0 l) __
```

where, y is argument and h h0 are parameters.

Our symbolic evaluation get the complexity description

```
 y_c+ f1(y H H_c H0 H0_c)
```

In order to solve `y_c+ f1(y H H_c H0  H0_c)`, our system generate the recurrection relations,

```
0 : f0(y ) =1
1 : f0(y ) =1

0 : f1(y H0 H0_c H H_c ) =1
1 : f1(y H0 H0_c H H_c ) =H0_c+aL_c+l_c+1 +f1(y-1 H H_c H0 H0_c)
                           +H0_c+H_c
```

As we discussed before, this kind of recurrence equations is parameterized REs. In order to solve it, we first assume the closed solution of it is as follows,

```
Transformation of recurrence equations :
f1(y H0 H0_c H H_c ) =C0(0)+C1(0)H0+C2(0)H0_c+C3(0)H+C4(0)H_c
f1(y H0 H0_c H H_c ) =C0(n)+C1(n)H0+C2(n)H0_c+C3(n)H+C4(n)H_c
```

Substitute this solution back into the previous equations,

```
C0(n)+C1(n)H0+C2(n)H0_c+C3(n)H+C4(n)H_c = H0_c+aL_c+l_c+1
+C0(n-1)+C1(n-1)H0+C2(n-1)H0_c+C3(n-1)H+C4(n-1)H_c+ +H0_c+H_c
```

Matching the both sides of the equations, we obtain the appendix equations.

```
#Appendix recurrence euqations
init4:=C4(0)=0 ;
init3:=C3(0)=0 ;
init2:=C2(0)=0 ;
init1:=C1(0)=0 ;
init0:=C0(0)=1 ;
```

```
#Appendix recurrence euqations
eq4:=C4(n)=C4(n-1)+1 ;
eq3:=C3(n)=C3(n-1);
eq2:=C2(n)=1 +C2(n-1)+1 ;
eq1:=C1(n)=C1(n-1);
eq0:=C0(n)=aL_c+l_c+1 +C0(n-1);
```

which is solved by Maple version 9, get the following solution,

```
s0:=rsolve({eq0,init0},C0);
s1:=rsolve({eq1,init1},C1);
s2:=rsolve({eq2,init2},C2);
s3:=rsolve({eq3,init3},C3);
s4:=rsolve({eq4,init4},C4);
```

We then substitute them back to the original equation, we get the final result,

```
f1(y H0 H0_c H H_c ) =1
f1(y H0 H0_c H H_c ) = -aL_c - l_c + (1 + aL_c + l_c) (n + 1)
                        + 2 nH0_c+ nH_c
```

So substitute it back to the original equation, we get the final complexity descrition

```
Y_c -aL_c - l_c + (1 + aL_c + l_c) (n + 1)+ 2 nH0_c+ nH_c
```

5.4 First Order Term:plus

```
(** val plus : nat -> nat -> nat **)

let rec plus n m =
  match n with
    | O -> m
    | S p -> S (plus p m)
```

Symbolic evaluation gets the complexity description as

```
 n_c+ f1(n m m_c )
```

The corresponding recurrence equations

```
 f1(n m m_c ) = m_c
 f1(n m m_c ) = S_c+ 0+ f1(n-1 m m_c ) + m_c
```

In order to solve such equation, we first generate an appendix recurrence relations.

```
#Appendix recurrence euqations
init2:=C2(0)=1 ;
init1:=C1(0)=0 ;
init0:=C0(0)=0 ;

#Appendix recurrence euqations
eq2:=C2(n)=C2(n-1)+1 ;
eq1:=C1(n)=C1(n-1);
eq0:=C0(n)=S_c+0 +C0(n-1);
```

And the solution solved by Maple as follows,

```
writeto(result);
s0:=rsolve({eq0,init0},C0);
s1:=rsolve({eq1,init1},C1);
s2:=rsolve({eq2,init2},C2);
```

Substitute it back to the original equations get

```
f1(n m m_c ) =1
f1(n m m_c ) = -aL_c - l_c + (1 + aL_c + l_c) (n + 1)+ 2nm_c
```

So the final complexity description is

```
  m_c+-aL_c - l_c + (1 + aL_c + l_c) (n + 1)+ 2 nm_c
```

6 Conclusion

Our system is able to compute complexities for simple recursive definitions following our assumptions: the induction should operate on natural numbers or lists, and there should be one recursive call only. The method is justified with respect to a formal complexity model for functional computations using a call by value semantics. Complexity functions are extracted from sets of linear recurrence relations expressing their input-output behaviour by the computer algebra system Maple. In general, the method generates sets of parameterized recurrence relations which cannot be solved directly. In this case, we eliminate them by guessing the form of the solution before to call Maple.

Many of our ideas have been inspired by the work of Benzinger [2,1], done in the NuPRL project, that we have simplified and generalized. The last example, however, could not be taken care of by Benzinger's work.

There are a lot of problems to be solved. In particular, it appears essential to be able to consider programs with several recursive calls, such as quicksort. We are far from making an analysis of these. Another strong limitation of the method is the assumption that the closed solution to the set of parameterized equations is a linear combination of the parameters.

Acknowledgement

The authors thank Drs Pierre Letouzey and Hugo Herbelin for their help.

References

[1] Ralph Benzinger. *Automated complexity analysis.* PhD thesis, 2001.

[2] Ralph Benzinger. Complexity analysis. *Journal of Functionnal Programming*, 11(1):3–31, 2001.

[3] Pierre Letouzey. Programmation fonctionnelle certifiée – l'extraction de programmes dans l' asistant coq. Technical report, 2004.