Type Inference for Complexity Analysis of Functional Programs

Maxime Lesourd
internship under the supervision of Patrick Baillot

1 Introduction

Complexity analysis is the study of the resources, time or space, required to run a progran given the size of its inputs. Here we are interested in automated time analysis for programs in a variant of λ-calculus. The goal would be to have an algorithm which gives human readable bounds for the execution time of programs written in a functional language. We use here an approach based on logic and which is transposed as a type system using the Curry-Howard isomorphism.

This internship builds on the work of Baillot *et al.*[3] on the $\mathsf{d}\ell\mathsf{T}$ type system in order to get a type inference algorithm which given a functional program computes a bound on its execution time.

$\mathsf{d}\ell\mathsf{T}$ is a type-based complexity analysis for a call-by-value, higher order calculus with primitive recursion. $\mathsf{d}\ell\mathsf{T}$ also includes primitives for writing and reading to memory but we will only consider its functional fragment here. The analysis proceeds in two steps, first we infer bounds on the size of the objects manipulated by the program and then we use these bounds to compute a bound on the number steps needed to evaluate the terms on an abstract machine. The result of this analysis comes as an annotated type giving bounds on the outputs of the program as a function of the size of its parameters. In [3] this is done by annotating types with first order terms which are specified by an equational program. Our work focused on providing an implementation for this analysis and exploring solutions for finding explicit closed-form bounds on the sizes specified by the equational programs.

The main contributions are :

- A full description of the type inference algorithm for the size analysis in $\mathsf{d}\ell\mathsf{T}$ for which some details were left out in [3].
- An implementation in OCaml of this algorithm allowing us use an external solver in order to find upper bounds on the equational programs.
- An analysis of the possibility to automatically compute these upper bounds using an existing solver for a similar problem : PUBS[1].
- An alternative algorithm to compute closed-form upper bounds tailored to the equational programs arising from type inference.

2 dlT

We will now describe the fragment of $\mathtt{d}\ell\mathsf{T}$ we will be working with. Since we mostly focus on the synthesis of upper bounds for equational programs we will not go over the evaluation rules for the language or the soundness of the type inference algorithm. These can be found in [3].

2.1 Setting

We consider a simply-typed, call-by-value lambda calculus with booleans, natural numbers and lists. Given a denumerable set of variables $\mathbb{V}$ (denoted by x, y or z), *terms* and *values* are defined as

$$\begin{array}{rcl} M, N & ::= & V \mid x \mid \mathtt{succ}(M) \mid \mathtt{cons}(M, N) \mid MN \\ V, W & ::= & \mathtt{zero} \mid \mathtt{succ}(V) \mid \mathtt{nil} \mid \mathtt{cons}(V, W) \mid \mathtt{tt} \mid \mathtt{ff} \\ & \mid & \lambda x.M \mid \mathtt{iter}(V, W) \mid \mathtt{fold}(V, W) \mid \mathtt{if}(V, W) \end{array}$$

The language constructs $\mathtt{zero}, \mathtt{succ}(V), \mathtt{nil}, \mathtt{cons}(V, W), \mathtt{tt}, \mathtt{ff}$ will be referred to as constructors and `iter`, `fold` as recursors.

Example 1. *We define the following terms which compute respectively the addition and multiplication of two natural numbers, the sum of a list of natural numbers and an implementation of addition using higher order iteration:*

$$\begin{array}{ll} \mathtt{add} := \lambda x.\mathtt{iter}(\lambda y.\mathtt{succ}(y), x) & \mathtt{mul} := \lambda x.\mathtt{iter}(\mathtt{add}\ x, \mathtt{zero}) \\ \mathtt{sum} := \mathtt{fold}(\mathtt{add}, \mathtt{zero}) & \mathtt{shift} := \mathtt{iter}(\lambda f x.f\ (\mathtt{succ}\ x), \lambda f.f) \end{array}$$

We start with an affine type system $\ell\mathsf{T}$ for the language which will be the basis of our analysis. It is a variant of system T with linearity constraints on function types. The *types* and *base types* are defined as:

$$\begin{array}{rcl} T & ::= & B \mid T \multimap T \\ B & ::= & \mathbf{B} \mid \mathbf{N} \mid \mathbf{L}(B) \end{array}$$

A *variable context* is denoted by Γ and is of the form $x_1 : T_1, \ldots, x_n : T_n$. A *ground* variable context is a variable context where the types are base types and is denoted by $\ell\Gamma$. Typing judgements are of the form $\Gamma \vdash M : T$. The union $\Gamma \uplus \Gamma'$ is only defined if the variables present both in Γ and Γ' are given the same *base* type. The typing rules of $\ell\mathsf{T}$ are similar to those of System T, we describe a few rules in which linearity plays a role in Figure 1. In the rule for term application we ensure that higher order variables are not duplicated, thus we know they will be used at most once. In the rules for the recursors the ground contexts ensure that no higher order variable is used when defining M and N.

Example 2. *We can derive the following judgements in $\ell\mathsf{T}$:*

$$\begin{array}{ll} \vdash \mathtt{add} : \mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N} & \vdash \mathtt{mul} : \mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N} \\ \vdash \mathtt{sum} : \mathbf{L}(\mathbf{N}) \multimap \mathbf{N} & \vdash \mathtt{shift} : \mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N} \end{array}$$

$$\frac{\Gamma, x : T \vdash M : T'}{\Gamma \vdash \lambda x.M : T \multimap T'} \qquad \frac{\Gamma_1 \vdash M : T \multimap T' \qquad \Gamma_2 \vdash N : T}{\Gamma_1 \uplus \Gamma_2 \vdash MN : T'}$$

$$\frac{\ell\Gamma_1 \vdash M : T \multimap T \qquad \ell\Gamma_2 \vdash N : T}{\ell\Gamma_1 \uplus \ell\Gamma_2 \vdash \texttt{iter}(M, N) : \mathbf{N} \multimap T}$$

$$\frac{\ell\Gamma_1 \vdash M : B \multimap T \multimap T \qquad \ell\Gamma_2 \vdash N : T}{\ell\Gamma_1 \uplus \ell\Gamma_2 \vdash \texttt{fold}(M, N) : \mathbf{L}(B) \multimap T}$$

Figure 1: Linear Typing Rules — Selection

2.2 dlT

The $\mathsf{d}\ell\mathsf{T}$ type system allows us to keep track of the size of the objects manipulated by a program. This is done by decorating the types of $\ell\mathsf{T}$ with *index terms* which give us an upper bound on the size of objects. The notion of size we are concerned with here is the length of lists and the number of `succ` constructors in natural numbers. The index terms are defined as $I ::= f(a, \ldots, a) \mid a$ where f is a function symbol of arity n taken from a set $\mathcal{IF}$ and a denotes an index variable taken from a set $\mathcal{IV}$. For now we assume that $\mathcal{IF}$ contains symbols for the constant 0 and the successor written as $1+$. We use ϕ to denote sequences of index variables.

These symbols are interpreted using an *equational program* $\mathcal{A}$ specified as an orthogonal terminating rewriting system. An equational program defines the indices in positive position in a type in terms of those in negative position. We will actually work with a specific class of equational programs described later.

The *indexed types* and *indexed base types* are defined as follows:

$$\begin{array}{rcl} D, E & ::= & U \mid D \multimap D \\ U & ::= & \mathbf{B} \mid \mathbf{N}^I \mid \mathbf{L}^I(U) \end{array}$$

We say that a type or index is in positive (resp. negative) position in a type when it is to the left of an even (resp. odd) number of $\multimap$. For example in $(\mathbf{N}^{I_1} \multimap \mathbf{N}^{I_2}) \multimap \mathbf{N}^{I_3}$, I_1 and I_3 are in positive positions and I_2 is in negative position.

Definition 1 (Equational Programs)**.** *An* equational program *is an orthogonal, terminating, term rewriting system which allows us to interpret index functions as function from n-uples of natural numbers to natural numbers. We denote equational programs by $\mathcal{A}$. During type inference we will be working with* pseudo programs*, denoted by $\mathcal{E}$, in which some symbols are unspecified. Intuitively these symbols will correspond to those which occur in negative positions on a type. In order to keep descriptions simple, we will sometimes use index terms such as natural number constants or $1 + f(\phi)$ instead of giving an equational program.*

A completion of a pseudo program $\mathcal{E}$ fully specifies the symbols only appearing on the right hand side of the rules in $\mathcal{E}$.

Example 3. *The type $\mathbf{L}^3(\mathbf{N}^2)$ is the type of lists of length smaller than 3 built from integers smaller than 2. The type $\mathbf{N}^a \multimap \mathbf{N}^b \multimap \mathbf{N}^{f(a,b)}$ together with the program {f(a,0) = a; f(a, b+1) = 1 + f(a,b)} is the type of functions which*

$$\frac{}{\vdash^{\mathcal{E}} \mathbf{B} \sqsubseteq \mathbf{B}} \qquad \frac{\vdash^{\mathcal{E}} D' \sqsubseteq D \qquad \vdash^{\mathcal{E}} E \sqsubseteq E'}{\vdash^{\mathcal{E}} D \multimap E \sqsubseteq D' \multimap E'}$$

$$\frac{\models^{\mathcal{E}} I \leq J}{\vdash^{\mathcal{E}} \mathbf{N}^I \sqsubseteq \mathbf{N}^J} \qquad \frac{\models^{\mathcal{E}} I \leq J \qquad \vdash^{\mathcal{E}} D \sqsubseteq D'}{\vdash^{\mathcal{E}} \mathbf{L}^I(D) \sqsubseteq \mathbf{L}^J(D')}$$

Figure 2: Subtyping Rules

$$\frac{\vdash^{\mathcal{E}} D \sqsubseteq E}{\Gamma, x : D \vdash^{\mathcal{E}} x : E} \qquad \frac{\models^{\mathcal{E}} I + 1 \leq J \qquad \Gamma \vdash^{\mathcal{E}} M : \mathbf{N}^I}{\Gamma \vdash^{\mathcal{E}} \mathtt{succ}(M) : \mathbf{N}^J}$$

$$\frac{\vdash^{\mathcal{E}} D \sqsubseteq E \qquad \vdash^{\mathcal{E}} E \sqsubseteq E\{a+1/a\} \qquad \vdash^{\mathcal{E}} E\{I/a\} \sqsubseteq F \qquad \ell\Gamma_1 \vdash^{\mathcal{E}} M : D \multimap D\{a+1/a\} \qquad \ell\Gamma_2 \vdash^{\mathcal{E}} N : D\{0/a\}}{\ell\Gamma_1 \uplus \ell\Gamma_2 \vdash^{\mathcal{E}} \mathtt{iter}(M, N) : \mathbf{N}^I \multimap F}$$

Figure 3: Typing Rules — Selection

take two natural numbers of size at most a and b and return a natural number smaller than $a + b$.

Given an equational program $\mathcal{A}$ and a function symbol f defined in $\mathcal{A}$ we write $[\![f]\!]_{\mathcal{A}}$ for the interpretion of f in $\mathcal{A}$. Given a pseudo program $\mathcal{E}$ and indices $I = f(\phi)$ and $I' = g(\phi')$ we write $\models^{\mathcal{E}} I \leq J$ if for every completion $\mathcal{A}$ of $\mathcal{E}$ we have $[\![f]\!]_{\mathcal{A}}(\phi) \leq [\![g]\!]_{\mathcal{A}}(\phi')$ for every assignement of the variables in ϕ, ϕ'.

We can now define a subtyping relation $\sqsubseteq$ on indexed types in Figure 2 and using the notations of $\ell\mathsf{T}$ for typing context we define a typing judgement of the form $\Gamma \vdash^{\mathcal{E}} M : D$. We give a selection of typing rules for $\mathtt{d}\ell\mathsf{T}$ in Figure 3. The rule for typing variables states that in a context where $x : D$, x can be used with any E such that $D \sqsubseteq E$, for example if $x : \mathbf{N}^1$ we can use x with type $\mathbf{N}^2$, in this case we lose some information about the size of x. The `succ` rule states that the successor increases the size by 1. The rule for the iterator uses a free variable a in D to keep track of the number of iterations and an intermediary type E on which we impose a monotonicity constraint.

Definition 2 (Skeleton of a type)**.** *The* skeleton $[D]$ *of a type D is obtained by erasing all indices of D.*

Definition 3 (Primitive types)**.** *A type is* primitive *for a set of index variables ϕ when it is of the form $\mathbf{B}, \mathbf{N}^{f(\phi)}, \mathbf{L}^{f(\phi)}(U)$ or $D_1 \multimap D_2$ where U, D_1, D_2 are primitive for ϕ.*

We use the symbol p to denote polarities in $\{+, -\}$ and $\neg p$ to denote the opposite of a polarity p.

Definition 4 (Positive, Negative symbols)**.** *Given a type D primitive for ϕ its* positive *and* negative *symbols, denoted by D^+ and D^- are defined inductively*

as:

$$\begin{aligned} \mathbf{N}^{f(\phi)+} &:= \{f\} & \mathbf{N}^{f(\phi)-} &:= \varnothing \\ \mathbf{L}^{f(\phi)}(D)^{+} &:= \{f\} \cup D^{+} & \mathbf{L}^{f(\phi)}(D)^{-} &:= \varnothing \\ (D_1 \multimap D_2)^{p} &:= D_1^{\neg p} \cup D_2^{p} & \mathbf{B}^{p} &:= \varnothing \end{aligned}$$

As an example the type $D := \mathbf{N}^{f(a,b)} \multimap \mathbf{L}^{g(a,b)}(\mathbf{N}^{h(a,b)})$ is primitive for $\phi = a, b$ with skeleton $[D] = \mathbf{N} \multimap \mathbf{L}(\mathbf{N})$. Its positive symbols are g and h and its negative symbol is f.

Definition 5 (Specified Symbols)**.** *Given a pseudo program $\mathcal{E}$ and a set of function symbols $\mathcal{N}$ we say that a function symbol f is* $(\mathcal{E},\mathcal{N})$-specified *if f is well-defined by $\mathcal{E}$ completed with definitions for the symbols in $\mathcal{N}$. We say that f is* unspecified *in $\mathcal{E}$ if there is no rewriting rule for f in $\mathcal{E}$. We say that a type D is* $(\mathcal{E},\mathcal{N},p)$-specified *if the symbols in D^p are $(\mathcal{E},\mathcal{N} \cup D^{\neg p})$-specified and the symbols in $D^{\neg p}$ are unspecified. We say that a typing judgement $\Gamma \vdash^{\mathcal{E}} M : D$ is specified for ϕ when D and the types in Γ are primitive for ϕ and D is $(\mathcal{E},\mathcal{N},+)$-specified and the types in Γ are $(\mathcal{E},\mathcal{N},-)$-specified where $\mathcal{N}$ is the set of negative symbols of the judgement.*

Example 4. *With $\mathcal{E} := \{f() = g()\}$ the type $\mathbf{N}^{g()} \multimap \mathbf{N}^{f()}$ is $(\mathcal{E},\varnothing,+)$ specified.*

2.3 Inference algorithm

We present an inference algorithm for $\mathsf{d}\ell\mathsf{T}$ adapted from [3] and [4]. Given an $\ell\mathsf{T}$ type derivation π with conclusion $\Gamma \vdash t : T$ we want to produce a $\mathsf{d}\ell\mathsf{T}$ type derivation π' with conclusion $\Gamma' \vdash^{\mathcal{E}} t : D$ together with an equational program $\mathcal{E}$ giving meaning to the indices in positive position in Γ' and D. We get rid of subtyping by using a max operator in equational programs, thus we consider our $\ell\mathsf{T}$ derivations with subtyping constraints replaced by equalities.

$$\texttt{Infer}(\pi,\phi) = (D,\mathcal{E},\Gamma')$$

where π has conclusion $\Gamma \vdash M : T$ with the invariants :

- $\Gamma' \vdash^{\mathcal{E}} M : D$ is derivable and specified for ϕ
- $[\Gamma'] \subseteq \Gamma$, $[D] = T$

`Infer` is defined by induction on π :

- **case** $\pi = \dfrac{}{\Gamma \vdash x : T}$:

$$\begin{aligned} D &:= \alpha(T,\phi) & D' &:= \alpha(T,\phi) \\ \mathcal{E} &:= \texttt{Cut}(D',D) & \Gamma' &:= x : D \end{aligned}$$

- **case** $\pi = \dfrac{\pi'}{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda x.M : T_1 \multimap T_2}$:

$$(D_M,\mathcal{E}_M,\Gamma_M) := \texttt{Infer}(\pi',\phi)$$

if $\exists D_x, x : D_x \in \Gamma_M$ then

$$D := D_x \multimap D_M \qquad \mathcal{E} := \mathcal{E}_M \qquad \Gamma' := \Gamma_M \setminus x : D_x$$

else

$$(D_x, \mathcal{E}_x) := \texttt{Dummy}_{\_}(M_1, \phi) \qquad D := D_x \multimap D_M$$
$$\mathcal{E} := \mathcal{E}_x \uplus \mathcal{E}_M \qquad \Gamma' := \Gamma_M$$

- **case** $\pi = \dfrac{\overset{\pi_M}{\Gamma_M \vdash M : T^l_M \multimap T^r_M} \qquad \overset{\pi_N}{\Gamma_N \vdash N : T_N}}{\Gamma_M \uplus \Gamma_N \vdash MN : T^r_M}$:

$$(D^l_M \multimap D^r_M, \mathcal{E}_M, \Gamma_M) := \texttt{Infer}(\pi_M, \phi) \qquad (D_N, \mathcal{E}_N, \Gamma_N) := \texttt{Infer}(\pi_N, \phi)$$
$$D := D^r_M \qquad \mathcal{E}_1 := \texttt{Cut}(D_N, D^l_M)$$
$$(\Gamma', \mathcal{E}_2) := \texttt{Unify}(\phi, \Gamma_M, \Gamma_N) \qquad \mathcal{E} := \mathcal{E}_M \uplus \mathcal{E}_N \uplus \mathcal{E}_1 \uplus \mathcal{E}_2$$

- **case** $\pi = \dfrac{}{\Gamma \vdash \texttt{nil} : \mathbf{L}(T)}$:

$$(D_0, \mathcal{E}_0) := \texttt{Dummy}_+(T, \phi) \qquad D := \mathbf{L}^{n(\phi)}(D_0)$$
$$\mathcal{E} := \mathcal{E}_0 \uplus \{n(\phi) = 0\} \qquad \Gamma' := \varnothing$$

- **case** $\pi = \dfrac{\overset{\pi_M}{\Gamma_1 \vdash M : T} \qquad \overset{\pi_N}{\Gamma_2 \vdash N : \mathbf{L}(T)}}{\Gamma_1 \uplus \Gamma_2 \vdash \texttt{cons}(M, N) : \mathbf{L}(T)}$:

$$(D_M, \mathcal{E}_M, \Gamma_M) := \texttt{Infer}(\pi_M, \phi)$$
$$(\mathbf{L}^{f(\phi)}(D_N), \mathcal{E}_N, \Gamma_N) := \texttt{Infer}(\pi_N, \phi)$$
$$(D', \mathcal{E}') := \texttt{Max}(D_M, D_N)$$
$$\Gamma' := \texttt{Unify}(\phi, \Gamma_M, \Gamma_N)$$
$$D := \mathbf{L}^{f'(\phi)}(D')$$
$$\mathcal{E} := \mathcal{E}_M \uplus \mathcal{E}_N \uplus \mathcal{E}_0 \uplus \{f'(\phi) = 1 + f(\phi)\}$$

- **case** $\pi = \dfrac{\overset{\pi_s}{\ell\Gamma_s \vdash M : T \multimap T} \qquad \overset{\pi_b}{\ell\Gamma_b \vdash N : T}}{\ell\Gamma_b, \ell\Gamma_s \vdash \texttt{iter}(M, N) : \mathbf{N} \multimap T}$:

$$(D^l_s \multimap D^r_s, \mathcal{E}_s, \Gamma_s) := \texttt{Infer}(\pi_s, (\phi, a, b))$$
$$(D_b, \mathcal{E}_b, \Gamma_b) := \texttt{Infer}(\pi_b, (\phi, a))$$
$$\mathbf{N}^{n(\phi)} := \alpha(\mathbf{N}, \phi)$$
$$(D_r, \mathcal{E}_1) := \texttt{IterCut}_+(D_b, D^l_s, D^r_s, n)$$
$$(\Gamma', \mathcal{E}_2) := \texttt{Unify}(\phi, \Gamma_s, \Gamma_b)$$
$$D := \mathbf{N}^{n(\phi)} \multimap D_r$$
$$\mathcal{E} := \mathcal{E}_s \uplus \mathcal{E}_b \uplus \mathcal{E}_1 \uplus \mathcal{E}_2$$

with a; b fresh in ϕ.

We now define the $\texttt{IterCut}_p$ procedure. Let D_b, D_s^l, D_s^r be types such that $[D] = [E] = [F]$ and f be an index symbol, then

$$\texttt{IterCut}_p(D_b, D_s^l, D_s^r, f) = (\mathcal{E}, D_r)$$

$\texttt{IterCut}_p$ is defined by induction :

$\texttt{IterCut}_+(\mathbf{N}^{i(\phi,a)}, \mathbf{N}^{j(\phi,a,b)}, \mathbf{N}^{k(\phi,b)}, f) = (\mathbf{N}^{r(\phi)}, \mathcal{E})$

where $\mathcal{E}$ is

$\{j(\phi, a, b+1) = k(\phi, a+1, b)$

$j(\phi, a, 0) = i(\phi, a)$

$r'(\phi, b+1) = k(\phi, 0, b+1)$

$r'(\phi, 0) = i(\phi, 0)$

$r(\phi) = \max_{v \leq f(\phi)} r'(\phi, b)\}$

$\texttt{IterCut}_-(\mathbf{N}^{i(\phi,a)}, \mathbf{N}^{j(\phi,a,b)}, \mathbf{N}^{k(\phi,b)}, f) = (\mathbf{N}^{r(\phi)}, \mathcal{E})$

where $\mathcal{E}$ is

$\{k(\phi, a+1, b) = j(\phi, a, b+1)$

$k(\phi, 0, b) = r(\phi)$

$i(\phi, a+1) = j(\phi, a, 0)$

$i(\phi, 0) = r(\phi)\}$

$\texttt{IterCut}_p(D_b \multimap D_b', D_s^l \multimap D_s^{l'}, D_s^r \multimap D_s^{r'}, f) = (D_r \multimap D_r', \mathcal{E} \cup \mathcal{E}')$

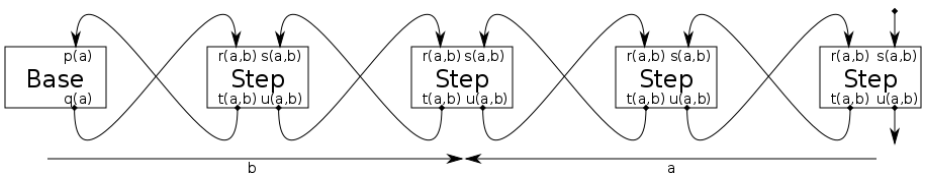
where

$\texttt{IterCut}_{\neg p}(D_b, D_s^l, D_s^r, f) = (\mathcal{E}, D_r)$

$\texttt{IterCut}_p(D_b', D_s^{l'}, D_s^{r'}, f) = (\mathcal{E}', D_r')$

The case for $\mathbf{L}(D)$ is similar to $\mathbf{N}$.

The definition of the procedures used in the the previous definitions can be found in [3] and in the implementation. The intuition behind the definition of `IterCut` can be illustrated by the following diagram. The boxes represent the action of the step and base terms on the size of the data we iterate on. The parameters a and b can be seen as counters which vary in opposite directions during the iteration. In this example we iterate on the type $(\mathbf{N}^{t(a,b)} \multimap \mathbf{N}^{s(a,b)}) \multimap (\mathbf{N}^{r(a,b)} \multimap \mathbf{N}^{u(a,b)})$.



2.4 The equational program

We now describe a restricted syntax for the equational programs generated by `Infer`. This will allow us to translate these equations in order to solve them using an external solver.

$$\begin{array}{rcl} t & ::= & 0 \mid v \mid 1+v \\ r & ::= & 0 \mid v \mid 1+f(\phi) \mid f(\bar{t}) \\ & \mid & \max(f(\phi), g(\phi)) \mid \max_{v \leq f(\phi)} g(\phi) \end{array}$$

ϕ (resp. $\bar{t}$) denotes a sequence of v (resp. t)

$$\mathcal{E} = \{f(\phi_1, 0, \phi_2) = g(\bar{t}_1); f(\phi_1, v+1, \phi_2) = h(\bar{t}_2)\}_{f \in I_1} \cup \{f(\phi) = r\}_{f \in I_2}$$

Where I_1 and I_2 are disjoint subsets of $\mathcal{IF}$.

Definition 6 (Alias)**.** *We say that $f(\phi)$* aliases to *r in $\mathcal{E}$ if $f(\phi) = r$ is in $\mathcal{E}$ or $f(\phi) = g(\phi')$ is in $\mathcal{E}$ and $g(\phi')$ aliases to r.*

3 Solving the equational program

3.1 PUBS

The PUBS solver was developed by Albert et al.[1] as a backend to a complexity analysis tool for Java programs. It is a promising candidate to solve equational programs because it handles multivariate recurrence relations and handles nondetermism, allowing us to easily translate the *max* on the right hand side of equations. These features are not so common in upper bound synthesis from a set of equations.

The PUBS solver is able to infer bounds for a system of *cost relations*. These are similar to our equations but they can deal with nondetermism.

Definition 7 (Cost Relation System)**.** *A* cost relation system *(CRS) is a set $\mathcal{C}$ of rewriting rules of the form*

$$f(\bar{t}) = c + \sum_{i=0}^{n} v_i + \sum_{i=0}^{m} f_i(\bar{t})$$

where the terms in $\bar{t}$ are linear combinations of variables and c is a natural number.

We provide a partial translation from equational programs to cost relations. The issue is that we cannot encode equational programs which require a form of functional composition. This problem arises when translating right hand sides of the form $\max_{v \leq f(\phi)} g(\phi)$ for which $f(\phi)$ is not a linear combination of negative occurences because in cost relations function symbols can only be applied to linear combinations of variables. In order to further simplify the implementation we restrict ourselves to programs in which such an $f(\phi)$ aliases to a variable. This restriction could be lifted but as we will see this would not allow us to treat a significantly larger set of programs.

For the translation we rely on a primitive $\texttt{CR}$ such that $\texttt{CR}(\bar{v}, f, \bar{t}, r)$ produces cost relations encoding $f(\bar{t}) = r$, the variables in $\bar{v}$ are used to produce a result which is parametrized with regard to the indices in negative position. $\texttt{CR}$ is defined by case analysis on r:

- **case** $r = 1 + g(\phi)$:

$$\texttt{CR}(\bar{v}, f, \bar{t}, r) := \{f(\bar{v}, \bar{t}) = 1 + g(\bar{v}, \phi)\}$$

  The cases for $r = 0, v, f(\bar{t})$ are straightforward.

- **case** $r = \max(g(\phi_1), h(\phi_2))$:

$$\texttt{CR}(\bar{v}, f, \bar{t}, r) := \{f(\bar{v}, \bar{t}) = g(\bar{v}, \phi_1); f(\bar{v}, \bar{t}) = h(\bar{v}, \phi_2)\}$$

- **case** $r = \max\limits_{v \leq g(\phi)} h(\phi')$:

  As explained above we assume that $g(\phi)$ aliases to some variable v', then

$$\begin{aligned}\texttt{CR}(\bar{v}, f, \bar{t}, r) := \{&f(\bar{v}, \bar{t}) = f'(\bar{v}, \bar{t}, v');\\ &f'(\bar{v}, \bar{t}, a) = h(\bar{v}, \phi\{a/v\});\\ &f'(\bar{v}, \bar{t}, a + 1) = f'(\bar{v}, \bar{t}, a)\}\end{aligned}$$

We can now define the translation $\texttt{CRS}(\mathcal{E}, D)$ of an equational program $\mathcal{E}$ giving meaning to symbols in some indexed type D:

$$\begin{aligned}&\texttt{CRS}(\mathcal{E}, D) = \mathcal{C} \cup \mathcal{C}'\\ &\quad\text{where}\\ &\qquad var \text{ is an injection from } D^- \text{ to fresh symbols in } \mathbb{V}\\ &\qquad \bar{v} := (var(f))_{f \in D^-}\\ &\qquad \mathcal{C}' := f(\bar{v}) = var(f)_{f \in D^-}\\ &\qquad \mathcal{C} := \bigcup_{(f(\bar{t}) = r) \in \mathcal{E}} \texttt{CR}(\bar{v}, f, \bar{t}, r)\end{aligned}$$

In the translation the equations in $\mathcal{C}'$ map indices in negative positions to PUBS variables and the equations in $\mathcal{C}$ are translated from the equational program.

Example 5. *Here is how we would translate an equational program $\mathcal{E}$ giving meaning to symbols in $D = \mathbf{N}^{a()} \multimap \mathbf{N}^{b()} \multimap \mathbf{N}^{c()} \multimap \mathbf{N}^{g()}$:*

$\mathcal{E}$	$\texttt{CRS}(\mathcal{E}, D)$
$g() = \max_{x \leq a()} f(x)$	$a(a,b,c) = a$
$f(x+1) = 1 + f(x)$	$b(a,b,c) = b$
$f(0) = \max(b(), c())$	$c(a,b,c) = c$
	$g(a,b,c) = g'(a,b,c,a)$
	$g'(a,b,c,x+1) = g'(a,b,c,x)$
	$g'(a,b,c,x) = f(a,b,c,x)$
	$f(a,b,c,x+1) = 1 + f(a,b,c,x)$
	$f(a,b,c,0) = b()$
	$f(a,b,c,0) = c()$

3.2 Evaluating PUBS

The first attempts at usign PUBS as a backend solver were very promising. We are able to infer precise bounds for addition using both `add` and `shift` terms and several terms using simple recursion on a higher order type such as $\mathbf{N} \multimap \mathbf{N}$. The first issues arose when we tried to infer bounds for multiplication.

3.2.1 Compositionality problems

A natural implementation of multiplication as iterated addition in $\ell\mathsf{T}$ would be $\texttt{mul}_1 := \lambda x.\texttt{iter}(\texttt{iter}(\lambda y.\texttt{succ}\ y, x), \texttt{zero})$ from Example 1 which can be rewritten as
$\lambda x.\texttt{iter}(\texttt{add}\ x, \texttt{zero})$. The resulting equational program does not satisfy the restriction required for our translation into a CRS because of the following equations :

$$\begin{aligned} r(A,B) &:= \max_{C \leq s(A,B)} r'(A,B,C) \\ s(A,0) &:= 0 \\ s(A,B+1) &:= r(A+1,B) \\ &\vdots \end{aligned}$$

Here the bound $s(A,B)$ does not alias to a negative index. Intuitively, the issue is that the size of the second argument to `add` is used to drive the iteration and its size $s(A,B)$ changes at every step of the outer iteration. In practice this limitation prevents us from applying a term built from `iter` to a complex term or using it as a step function. At this point the question that remains is whether we can express some interesting functions for which we can produce an equational program that can be translated to PUBS.

3.2.2 Cover point problems

It is possible to represent multiplication with these restrictions as
$\texttt{mul}_2 := \lambda x.\texttt{iter}(\lambda y.\texttt{iter}(\lambda z.\texttt{succ}\ z, y)\ x, \texttt{zero})$ which is equivalent to
$\lambda x.\texttt{iter}(\lambda y.\texttt{add}\ y\ x, \texttt{zero})$ and we can actually produce terms representing any polynomial function with natural coefficients. Not only is this restriction slightly less natural, it is actually less efficient. This comes from the fact that the time complexity for evaluating $\texttt{add}\ x\ y$ is the size of y which remains constant in $\texttt{mul}_1$ and increases by the size of x every step in $\texttt{mul}_2$. Trying to solve the CRS

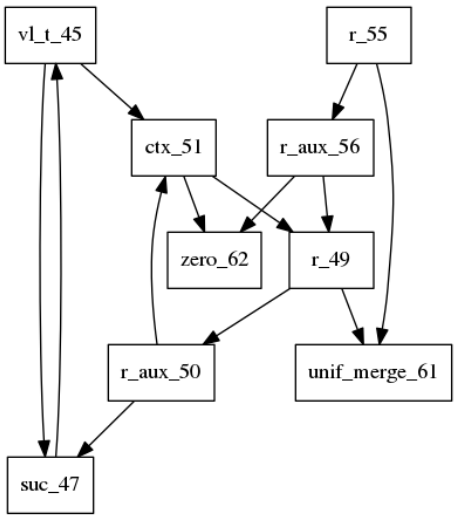


Figure 4: Dependency Graph for $\texttt{mul}_2$

for $\texttt{mul}_2$ allowed us to discover a fundamental limitation of the PUBS solver for our purpose.

We now need to introduce a few notions to talk about the structure of equational programs in order to understand this limitation.

Definition 8 (Dependency Graph)**.** *The* dependency graph *for an equational program $\mathcal{E}$ is the graph which has the indices appearing in $\mathcal{E}$ as nodes and an edge between f and g if the definition of f in $\mathcal{E}$ mentions g. We can define a similar notion for cost relation systems.*

Definition 9 (Strongly Connected Component)**.** *In a directed graph $G = (E, V)$ a* strongly connected component *is a subset U of E such that for any two nodes u, v in U there is a path in G from u to v.*

Definition 10 (Cover Point)**.** *A* cover point *for a strongly connected component U is a node p in U such that every cycle in U contains p.*

PUBS requires that every strongly connected component in the dependency graph has at least one cover point. This condition is mentioned in [1] but the authors argue that it is always satisfied in the cost relations generated from the imperative programs they study.

In the case of $\texttt{mul}_2$ we get the dependency graph in Figure 4. We can see that the cycles {`vl_t_45 suc_47`} and {`ctx_51 r_49 r_aux_50`} are disjoint and belong to the same strongly connected component. In this case the PUBS solver fails to produce an upper bound and gives the following error message :

```
CRS r_55(A)

  * Non Asymptotic Upper Bound: c(failed(cover_point,[scc=1,cr=r_49/3]))
```

3.3 Beyond PUBS

Since PUBS does not seem to be able to solve our equational programs even on simple examples we have tried to find a heuristic tailored to these programs. The idea is that the dependence graph is a coarse reprensentation of the actual dependencies arising in equational programs.

This heuristic is not yet fully defined but the intuition can be explained with the following example equational program $\mathcal{E}$ which is adapted from the equational program from $\texttt{mul}_2$:

$$\begin{aligned} f(a,b) &= f'(a,b,x()) \\ f'(a,b,0) &= g(a,b) \\ f'(a,b,c+1) &= 1 + h(a,b,0,d+1) \\ g(a,0) &= 0 \\ g(a,b+1) &= s(a+1,b) \\ h(a,b,c,0) &= g(a,b) \\ h(a,b,c,d+1) &= 1 + h(a,b,c+1,d) \end{aligned}$$

The dependency graph for $\mathcal{E}$ is strongly connected and it has two disjoint cycles (f, f', g) and (h) thus PUBS is of no help to us. If we were to solve it by hand we would notice that the dependency from h to g is only present in the base case. We could replace the last two equations by $h(a,b,c,d) = g(a,b) + d$ and from there we could either solve it using PUBS or try to find another similar pattern which we could simplify.

In order to turn this intuition into a proper algorithm we would need a way to bound simple recursions of the form $f(\bar{v}, x+1) = F(\bar{v}, f(\bar{v}, x)), f(\bar{v}, 0) = G(\bar{v})$ for some functions F, say linear or polynomial functions with natural coefficients. Then we would need to define an ordering on indices taking into account these weak dependencies arising in the base case of recursions.

The first part could be handled by a computer algebra system for some classes of functions F. Unfortunately we were not able to find a way to express the desired order during this internship. Using this approach by hand we were able to find tight upper bounds for all the examples presented here and for an implementation of $x \mapsto 2^x$.

4 Implementation

We provide an implementation of the inference algorithm in OCaml. This implementation was used to output the equational program needed to perform type inference on several examples and to experiment with the PUBS solver. In order to save time we provide an embedded domain specific language (EDSL) instead of a parser, this way we can input terms directly in OCaml.

The inference algorithm is implemented as an OCaml library which provides the following functionalities :

- An EDSL to input $\ell\mathsf{T}$ terms
- Type inference from $\ell\mathsf{T}$ to $\mathsf{d}\ell\mathsf{T}$
- Alias reduction for equational programs
- Translation of equational programs to the PUBS format

The implementation of the type inference algorithm outputs the equational program in a solver-independent format which can then be translated to the PUBS format. This approach allows us to output the equational programs which cannot be translated to the PUBS format and solve them by hand or use an alternative solver. It also allows us to perform alias reduction which is needed for the translation and makes equational programs more readable for the user. The data type used to represent the equations is as close as possible to the format described is section 2.4, this way the implementation of the translation to the PUBS format is very close to the definition of `CRS`.

We heavily rely on the OCaml type system when handling $\ell\mathsf{T}$ typed terms and indexed types by representing them as a *generalized algebraic data type* indexed by the corresponding $\ell\mathsf{T}$ type. We will use a simplified version of the datatypes used in the implementation in the following examples. Here the type `nat` is used as a type marker for $\mathbf{N}$ and the OCaml function type `->` for $\multimap$. We introduce the datatypes for types and indexed types, the type parameter is used to record the base type. For example the type $\mathbf{N}^{f(\phi)} \multimap \mathbf{N}^{g(\phi)}$ would be represented as
`IA (IN (index f` ϕ`), IN (index g` ϕ`)) : (nat -> nat) ityp`

```
type nat

type _ typ =
  | N : nat typ
  | A : 'a typ * 'b typ -> ('a -> 'b) typ

type _ ityp =
  | IN : index -> nat ityp
  | IA : 'a ityp * 'b ityp -> ('a -> 'b) ityp
```

Instead of writing a parser for untyped terms we opted for an EDSL to input $\ell\mathsf{T}$ terms which are typeable by construction. The only part of $\ell\mathsf{T}$ type checking which is not performed when constructing terms is the linearity constraints, these are checked during $\mathsf{d}\ell\mathsf{T}$ type inference. Another advantage is that we can write OCaml functions which can be used as macros to write terms.

```
type _ term =
  | Var : string * 'a typ -> 'a term
  | Lam : string * 'a typ * 'b term -> ('a -> 'b) term
  | App : ('a -> 'b) term * 'a term -> 'b term
  | Zero : nat term
  | Suc : nat term -> nat term
  | Iter : ('a -> 'a) term * 'a term -> (nat -> 'a -> 'a) term

(* lam : string -> 'a typ -> ('a term -> 'b term) -> ('a -> 'b) term *)
let lam v typ f = Lam (v, typ, f (Var (v, typ)))
```

The `lam` function is used to build terms from OCaml functions. This ensures that variables are properly scoped and used with the right type. In the following example we use `lam` to implement the identity function which can be specialised to any type. The term resulting from the evaluation of `id typ` is equivalent to `Lam ("x", typ, Var ("x", typ))`.

```
(* id : 'a tag -> ('a -> 'a) term *)
let id typ = lam "x" N (fun x -> x)
```

The main downside of using an EDSL is that recompilation is needed everytime the user wants to specify a different term to analyse. During our experimentation we did not find this to be a problem since compilation time is very short and by leveraging the OCaml language we were able to quickly write complex terms including an OCaml function which takes a list of coefficients describing a polynomial and outputs a term computing the associated polynomial function. A big step interpreter is also provided and was used to test that complex terms compute the right function.

This representation is useful in order to implement the type inference algorithm as a patently total function. In many of the auxilliary procedures used in `Infer` one of the invariant is that several type arguments ought to have the same skeleton which is used when pattern patching on these arguments. For example in the following example we do not have to consider the cases for which the types do not have the same skeleton and the compiler can check that the pattern matching is exhaustive. The price we have to pay in order to get exhaustiveness checks is that we need to give explicit type signatures for the functions handling these types because the OCaml compiler cannot infer them.

```
let rec cut : type a. a ityp -> a ityp -> prog =
  fun context typ1 typ2 ->
    match typ1, typ2 with
    | IN f, IN g -> ... (* equation for (f = g) *)
    | IA (t1, t1'), IA (t2, t2') ->
       union (cut t2 t1) (cut t1' t2')
```

Example 6. *We will now see how to use our implementation and the PUBS solver*[1] *to infer the* $\mathsf{d}\ell\mathsf{T}$ *type for the* `shift` *example of Example 1.*

The first step is to input the term in the file `examples.ml` using the helper functions for $\ell\mathsf{T}$ terms and modify the file `main.ml` to perform the analysis of the term `shift`. In our case we want to output the cost relation system to the file `shift.ces`.

```
(* examples.ml *)
let shift =
  iter (lam "f" (arr nat nat) (fun f ->
        lam "x" nat (fun x ->
        app f (suc x))))
       (id N)

(* main.ml *)
let _ =
  driver ~config:{default with pubs = Some "shift.ces"} Examples.shift
```

We can now compile the program and run the analysis with `make; ./main`. The output includes the type `nat[n_44()] -> nat[r_45()] -> nat[r_46()]`

[1] A linux executable for the PUBS solver is available at `http://samir.fdi.ucm.es/~genaim/tmp/pubs/`

which is inferred for our term and an equational program which gives meaning to the indices in the indexed type. The file `shift.ces` contains the translation of the equational program, we are interested in the first two sets of equations.

```
% shift.ces
% Entries

entry('r_46'(A,B):[]).

% Equations

eq('n_44'(A,B),nat(A),[],[]).
eq('r_45'(A,B),nat(B),[],[]).
...
```

Under the entries header we can find the positive indices, in this case `'r_46'(A,B):[]`. The first set of equations after the equations header relates PUBS variables and indices in negative position, for example `n_44` is referred to as `A` in the equations. We can now call the PUBS solver using `pubs_static -file shift.ces -entry "'r_46'(A,B):[]"` to get a bound on `r_46()`.

```
CRS r_46(A,B)

  * Non Asymptotic Upper Bound: 1+nat(A-1)+nat(B)
```

We have to substitute the variables `A` and `B` for the indices they refer to to get the final bound which is `1+n_44()+n_45()`.

5 Related work

There are several approaches to time complexity analysis for higher order functional programs. This work tries to improve on an approach which involves reducing the problem of inferring time or size bounds on functional programs to solving recurrence relations. Most of the time these recurrence relations as in Danner et al.[5] or Dal Lago and Gaboardi [7] are the end result of the analysis. The main limitation is that the obtained recurrences are not always guaranteed to be terminating and the user still has to find a way to infer an upper bound on the recurrence.

Another approach is to restrict the analysis to a restricted class of upper bounds. Resource Aware ML[6] can infer polynomial size and space bounds for higher order functional programs. We were not able to produce a comparison between our approach and RAML during this internship but given an adequate solver our approach should allow us to infer non polynomial bounds.

The literature concerning complexity of imperative programs is vast and uses different approaches to infer explicit bounds. A similar approach to the one we used here is used in the COSTA analyser[2] to produce size and time bounds for Java programs. The PUBS solver was developed as a backend for COSTA.

6 Perspectives

We have seen that the PUBS solver is not suited for solving our equational programs and on way we believe that we gained some insight on a method which would allow us to exploit their structure to infer upper bounds for a wider range of programs. We still have to properly define our tentative heuristic and find a class of recurrences for which we could use an existing tool or devise our own.

We have implemented the inference algorithm for $\mathsf{d}\ell\mathsf{T}$ in a way which allows us to use a tool of our choice to bound the equational program. We could try to find another existing method for bounding the equational programs and see how well it fares on our examples to compare it to the heuristic. Once we find a reliable way to bound the equational programs, the implementation should be extended to produce the time bounds as described in [3].

Acknowledgments

I would like to thank Patrick Baillot and Ugo Dal Lago for their advice and suggestions during this internship.

References

[1] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

[2] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer, 2008.

[3] Patrick Baillot, Gilles Barthe, and Ugo Dal Lago. Implicit Computational Complexity of Subrecursive Definitions and Applications to Cryptographic Proofs (Long version). Research report, ENS Lyon, September 2015.

[4] Ugo Dal lago and Barbara Petit. The geometry of types. *SIGPLAN Not.*, 48(1):167–178, January 2013.

[5] Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. Denotational cost semantics for functional languages with inductive types. *CoRR*, abs/1506.01949, 2015.

[6] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource Aware ML. In *24rd International Conference on Computer Aided Verification (CAV'12)*, volume 7358 of *Lecture Notes in Computer Science*, pages 781–786. Springer, 2012.

[7] Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. *Logical Methods in Computer Science*, 8(4), 2011.