GUBS Upper Bound Solver

Martin Avanzini　　Michael Schaper

1 Introduction

Synthesizing functions that obey certain constraints, in the form of (in)equalities that the functions should obey, is a fundamental task in program analysis. For instance, this is important for the termination and complexity analysis. To this end, the program verification community predominantly adopts techniques either based on LP solvers [10], or dedicated tools like PUBS [1]. The former approach is usually restricted to the synthesis of linear functions. The latter approach is based on solving recurrence relations and particularly useful for the synthesis of ranking functions in the context of imperative programs. Recurrence relations are limited in scope however, for instance, since function composition cannot be directly expressed.

To overcome these restrictions, we have developed the *GUBS Upper Bound Solver* (GUBS for short). Given a set of inequalities over arithmetical expressions and uninterpreted functions, GUBS tries to find a model in $\mathbb{N}$, i.e., an interpretation of all the functions that make the given inequalities hold. This tool is currently used in our inference machinery for sized-types[1] and, experimentally, in our complexity tool TCT[2] for the synthesis of specific ranking function. GUBS itself is heavily inspired by methods developed in the context of synthesising linear and non-linear interpretations in the context of rewriting. The main novel aspect of GUBS is the modular approach it rest upon, inspired by the framework underlying TCT. To date, it incorporates an adaption of the synthesis technique from [5] which reduces polynomial inequalities to SMT (with respect to the theory of *quantifier free non-linear integer arithmetic*), various syntactic simplification techniques and a per-SCC analysis. GUBS is open sourced and available from

https://github.com/ComputationWithBoundedResources/gubs .

Although developed foremost for the complexity analysis of (higher-order) rewrite systems, we are convinced that GUBS has potential applications outside rewriting. E.g., for the synthesis of ranking function, the inference of certain dependent type systems such as the one of Dal Lago and Petit [4], the complexity analysis of concurrent programs as in [7], or various systems developed by the implicit computational complexity community (we name just [3]).

In the following we briefly outline our tool GUBS. More specific, we introduce the problem tacked by GUBS formally, we outline the central synthesis techniques and report on the experience that we have collected so far.

2 Constraint System over the Naturals

Let $\mathcal{V}$ denote a countably infinite set of *variables*, and let $\mathcal{F}$ denote a *signature*, i.e., a set of *function symbols*, disjoint from $\mathcal{V}$. Each function symbol $\mathtt{f} \in \mathcal{F}$ is equipped with a natural number $\mathtt{ar}(\mathtt{f})$, its *arity*. We always use $x, y, \ldots$ to denote variables, whereas $\mathtt{f}, \mathtt{g}, \ldots$ denote function symbols. The set

[1] See https://cl-informatik.uibk.ac.at/users/zini/software/hosa.

[2] See http://cl-informatik.uibk.ac.at/software/tct.

Submitted to:
DICE-FOPARA 2017

of *arithmetical terms* $\mathcal{T}(\mathcal{F},\mathcal{V})$ over function symbols $\mathcal{F}$ and variables $\mathcal{V}$ is generate inductively from $x \in \mathcal{V}$, $\mathtt{f} \in \mathcal{F}$, $n \in \mathbb{N}$ and pre-defined arithmetical operations $\oplus \in \{+,*,\max\}$ according to the following grammar:

$$a,b ::= x \mid n \mid a \oplus b \mid \mathtt{f}(a_1,\dots,a_{\mathsf{ar}(\mathtt{f})}) \; .$$

A *constraint system* $\mathcal{C}$ (over $\mathbb{N}$) is a finite set of *constraints* $a \geqslant b$.

Informally, a constraint system $\mathcal{C}$ is *satisfiable* if we can interpret each symbol $\mathtt{f} \in \mathcal{F}$ with a function $\mathtt{f}_{\mathcal{M}} : \mathbb{N}^{\mathsf{ar}(\mathtt{f})} \to \mathbb{N}$ so that all the constraints in $\mathcal{C}$ holds. Here, constraints and arithmetical operations are interpreted in the natural way. Consider for instance the constraint system $\mathcal{C}_1$ consisting of the following two constraints:

$$\mathtt{r}(\mathtt{n},y) \geqslant 1 \qquad\qquad \mathtt{r}(\mathtt{c}(x,y),z) \geqslant 1 + \mathtt{r}(x,\mathtt{c}(y,z)) \; .$$

This system is satisfiable, for instance, by taking $\mathtt{n}_{\mathcal{M}} = 1$ for $\mathtt{r}_{\mathcal{M}}(x,y) = x$ and $\mathtt{c}_{\mathcal{M}}(x,y) = y+1$, as we have

$$\mathtt{r}_{\mathcal{M}}(\mathtt{n}_{\mathcal{M}},y) = 1 \geqslant 1 \quad \text{and} \quad \mathtt{r}_{\mathcal{M}}(\mathtt{c}_{\mathcal{M}}(x,y),z) = y+1 \geqslant 1+y = 1+\mathtt{r}(y,\mathtt{c}(x,z)) \; .$$

More formally, an *interpretation* $\mathcal{M}$ over symbols $\mathcal{F}$ (into the naturals) is a mapping that assigns to each symbol $\mathtt{f} \in \mathcal{F}$ a function $\mathtt{f}_{\mathcal{M}} : \mathbb{N}^{\mathsf{ar}(\mathtt{f})} \to \mathbb{N}$. Let $\oplus_{\mathbb{N}} : \mathbb{N}^2 \to \mathbb{N}$ for $\oplus \in \{+,*,\max\}$ denote addition, multiplication and the maximum function on natural numbers, respectively. The interpretation $[\![t]\!]^{\alpha}_{\mathcal{M}}$ of a term $t \in \mathcal{M}$ with respect to $\mathcal{M}$ and variable assignment $\alpha \colon \mathcal{V} \to \mathbb{N}$ is then defined in the expected way:

$$[\![a]\!]^{\alpha}_{\mathcal{M}} = \begin{cases} \alpha(a) & \text{if } a \in \mathcal{V} \; , \\ a & \text{if } a \in \mathbb{N} \; , \\ a_1 \oplus_{\mathbb{N}} a_2 & \text{if } a = a_1 \oplus a_2 \in \mathbb{N} \text{ and } \oplus \in \{+,*,\max\} \; , \\ \mathtt{f}_{\mathcal{M}}([\![a_1]\!]^{\alpha}_{\mathcal{M}},\dots,[\![a_n]\!]^{\alpha}_{\mathcal{M}}), & \text{if } a = \mathtt{f}(a_1,\dots,a_n) \text{ and } \mathtt{f} \in \mathcal{F} \; . \end{cases}$$

We say that $\mathcal{M}$ is a *model* of a constraint system $\mathcal{C}$, in notation $\mathcal{M} \models \mathcal{C}$, if $[\![a]\!]^{\alpha}_{\mathcal{M}} \geqslant_{\mathbb{N}} [\![b]\!]^{\alpha}_{\mathcal{M}}$ holds for all assignments α and constraints $a \geqslant b \in \mathcal{C}$. For instance, we have $\mathcal{M} \models \mathcal{C}_1$ for the constraint system $\mathcal{C}_1$ and interpretation $\mathcal{M}$ depicted above. With our tool GUBS, we give a sound, but necessarily incomplete procedure to the following undecidable problem.

Definition 1 (Model Synthesis)**.** *Given a constraint system $\mathcal{C}$, the* model synthesis problem *asks for an interpretation $\mathcal{M}$ with $\mathcal{M} \models \mathcal{C}$.*

3 Implementation

GUBS is written in the functional programming language Haskell. The source contains approximately 2000 lines of code, spread over 25 modules. GUBS comes along as a stand-alone executable as well as a Haskell library. For usage information and installation instructions, we kindly refer the reader to the homepage of GUBS. Here, we just provide a short outline of two central methods implemented in GUBS.

Synthesis of Models via SMT. Conceptually, we follow the method presented in [5]. In this approach, each k-ary symbol $\mathtt{f} \in \mathcal{F}$ is associated with a *template max-polynomial*, i.e., an expression over k variables and connectives $\{+,*,\max\}$, and undetermined coefficient variables $\vec{\mathsf{c}}$. For instance, a linear template for a binary symbol f employed by GUBS is

$$\mathtt{f}_{\mathcal{A}}(x,y) = \max(\mathsf{c}_1 \cdot x + \mathsf{c}_2 \cdot y + \mathsf{c}_3, \mathsf{d}_1 \cdot x + \mathsf{d}_2 \cdot y + \mathsf{c}_3) \; .$$

To find a concrete model based on these templates, we search for a solution to $\exists \vec{\mathtt{c}}.\ \bigwedge_{a \geqslant b \in \mathcal{C}} \forall \vec{x} \in \mathcal{V}.\ [\![a]\!]^{\mathcal{A}}_{\mathcal{M}} \geq [\![b]\!]^{\mathcal{A}}_{\mathcal{M}}$, in two steps. First, we eliminate max according to the following rules.

$$\begin{aligned} e \geq C[\max(f_1, f_2)] \quad &\Rightarrow \quad e \geq C[f_1] \wedge e \geq C[f_2]\ , \\ C[\max(e_1, e_2)] \geq f \quad &\Rightarrow \quad C[e_1] \geq f \vee C[e_2] \geq f\ . \end{aligned}$$

Notice that this elimination procedure is sound as our max-polynomial algebra allows the formation of weakly monotone expressions only. Once all occurrences of max are eliminated, we reduce the resulting formula to *diophantine constraints* over the coefficient variables $\vec{\mathtt{c}}$, via the so called *absolute positiveness check*, see also [5]. The diophantine constraints are then given to an SMT-solver that support quantifier-free non-linear integer arithmetic, from its assignment and the initially fixed templates `GUBS` then computes concrete interpretations. To get more precise bounds, `GUBS` minimises the obtained model by making use of the incremental features of current SMT-solvers, essentially by putting additional constraints on coefficients.

Limitations: The main limitation of this approach is that the shape of interpretations is fixed to that of the templates, noteworthy, the degree of the interpretation is fixed in advance. As the complexity of the absolute positiveness check depends not only on the size of the given constraint system but to a significant extend also on the degree of interpretation functions, our implementation searches iteratively for interpretations of increasing degrees.

Also notice that our max-elimination procedure is incomplete, for instance, it cannot deal with the constraint $\max(2x, 2y) \geqslant x + y$, which is reduced to $2x \geqslant x + y \vee 2x \geqslant x + y$. In contrast, in [6] a complete procedure is proposed. However, our experimental assessment concluded that this encoding introduces too many auxiliary variables, which turned out as a significant bottleneck of the overall procedure.

Separate SCC Analysis. Synthesis of models via SMT gets impractical on large constraint systems. To overcome this, `GUBS` divides the given constraint system $\mathcal{C}$ into its *strongly connected components* (*SCCs* for short) $\mathcal{C}_1, \dots, \mathcal{C}_n$, topologically sorted, and finds a model for each SCC $\mathcal{C}_i$ iteratively. Here, the underlying *call graph* is formed as follows. The nodes are given by the constraints in $\mathcal{C}$. Let $a_1 \geqslant b_1$ to $a_2 \geqslant b_2$ be two constraints in $\mathcal{C}$, where wlog. $a_1 = C[\mathtt{f}_1(\vec{a}_1), \dots, \mathtt{f}_n(\vec{a}_n)]$ for a context C without function symbols. Then there is an edge from $a_1 \geqslant b_1$ to $a_2 \geqslant b_2$ if any of the symbols occurring in $\vec{a}_1, \dots, \vec{a}_n, b_1$ occurs in a_2. The intuition is that once we have found a model for all the successors of $a_1 \geqslant b_1$, we can interpret the terms $\vec{a}_i$ and b_1 within this model. We can then extend this model by finding a suitable interpretation for $\mathtt{f}_1, \dots, \mathtt{f}_n$, thereby obtaining a model that satisfies $a \geqslant b_1$.

4 Case Studies

In this section we briefly outline our experience collected so far, in the two contexts where `GUBS` is currently employed.

Sized-Types Synthesis. Various successful approaches to automatic verification of termination properties of higher-order functional programs are based on *sized-types* [9]. Here, a type carries not only some information about the *kind* of each object, but also about its *size*, hence the name. This information is then exploited when requiring that recursive calls are done on arguments of *strictly smaller* size.

In recent work [2] we have taken a fresh look at sized-types, with particular emphasis towards application in runtime analysis and automation. A result of this work is the tool `HoSA`, which given a program

```
       map : ∀ijk.(∀ l.L[l](a) →L[f4(l,i)](a)) → L[k](L[j](a)) → L[f6(i,j,k)](L[f5(i,j,k)](a))
    append : ∀ij.L[i](a) →L[j](a) → L[f1(i,j)](a)
prependAll : ∀ijk.L[i](a) → L[k](L[j](a)) → L[f3(i,j,k)](L[f2(i,j,k)](a))
```

(a) Template sized-types assigned by HoSA to the main function f and auxiliary functions.

```
(>= (f1 0 (var x)) (var x))                             (>= (f71 (var x)) (var x))
(>= (f1 (+ (var x) 1) (var y))                          (>= (f72 (var x)) (var x))
    (+ (f78 (var y) (var x)) 1))                        (>= (f74 (var x)) (var x))
(>= (f2 (var x) (var y) (var z))                        (>= (f75 (var x)) (var x))
    (f5 (f73 (var x)) (f75 (var y)) (f74 (var z))))     (>= (f76 (var x)) (var x))
(>= (f3 (var x) (var y) (var z))                        (>= (f77 (var x)) (var x))
    (f6 (f73 (var x)) (f75 (var y)) (f74 (var z))))     (>= (f78 (var x) (var y))
(>= (f4 (var x) (f73 (var y)))                              (f1 (f76 (var y)) (f77 (var x))))
    (f1 (f71 (var y)) (f72 (var x))))                   (>= (f79 (var x)) (var x))
(>= (f4 (var x) (f82 (var y)))                          (>= (f80 (var w) (var x) (var y) (var z))
    (f4 (f81 (var x)) (var y)))                             (f4 (f79 (var x)) (var w)))
(>= (f5 (var x) (var y) (+ (var z) 1))                  (>= (f80 (var w) (var x) (var y) (var z))
    (f80 (var x) (var y) (var z) (var w)))                  (f5 (f82 (var w)) (f84 (var x)) (f83 (var y))))
(>= (f5 (var x) (var y) 0) (f86 ()))                    (>= (f81 (var x)) (var x))
(>= (f6 (var x) (var y) 0) 0)                           (>= (f83 (var x)) (var x))
(>= (f6 (var x) (var y) (+ (var z) 1))                  (>= (f84 (var x)) (var x))
    (+ (f85 (var x) (var y) (var z) (var w)) 1))        (>= (f85 (var w) (var x) (var y) (var z))
                                                            (f6 (f82 (var w)) (f84 (var x)) (f83 (var y))))
```

(b) Constraint system generated from HoSA.

```
f1(x,y) = x + y        f71(x) = x         f77(x) = x               f83(x) = x
f2(x,y,z) = x + y      f72(x) = x         f78(x,y) = x + y         f84(x) = x
f3(x,y,z) = z          f73(x) = x         f79(x) = x               f85(x,y,z,w) = z
f4(x,y) = x + y        f74(x) = x         f80(x,y,z,w) = x + y     f86 = 0
f5(x,y,z) = x + y      f75(x) = x         f81(x) = x
f6(x,y,z) = z          f76(x) = x         f82(x) = x
```

(c) Model inferred by GUBS on the generated constraints.

```
       map : ∀ijk.(∀ l.L[l](a) →L[l+i](a)) → L[k](L[j](a)) → L[k](L[i+j](a))
    append : ∀ij.L[i](a) →L[j](a) → L[i+j](a)
prependAll : ∀ijk.L[i](a) → L[k](L[j](a)) → L[k](L[i+j](a))
```

(d) Inferred size type obtained by instantiating the template types with the model computed by GUBS.

Figure 1: Sized-type inference carried out by HoSA on `prependAll` *xs* ys = `map` (`append` *xs*) ys.

written in a simple typed higher-order language, annotates all types with size information. GUBS is a central ingredient of HoSA.

Consider the function, `prependAll` *xs* ys = `map` (`append` *xs*) ys, which prepend a given list *xs* to all elements of it second argument *ys*, itself a list of lists. Sized-type inference with HoSA works by first decorating datatypes occurring in the types of functions with size indices, resulting in so called *template types*, see Figure 1a. Size indices are annotated here in square brackets. These template types make reference to so far undetermined functions. Inference then amounts simply to type-checking, where however, type-checking emits constraints relating the different size indices. The emitted constraint system is depicted in Figure 1b in S-expression notation, the notation expected GUBS. A model found by GUBS on the resulting constraint system, such as the one depicted in Figure 1c, is then used to construct concrete sized-types from the initial template types, see Figure 1d.

It is worthy of note that in this example, the inferred types are precise and thus informative, not least,

```
dub(nil) → []          dub(x : xs) → x : x : dub(xs)
```

(a) TRS that duplicates the elements of a given list.

```
(>= (dup (nil)) (+ 1 (nil)))                               (>= (cons (var x) (var xs)) (var xs))
(>= (dup (cons (var x) (var xs)))                          (>= (+ (var x) (var xs) (k))
    (+ 1 (cons (var x) (cons (var x) (dup (var xs))))))        (cons (var x) (var xs)))
```

(b) Constraints generated by TcT on the considered TRS.

```
cons(x0,x1) = x1 + 1      dub(x0) = 2*x0      nil = 1      k = 1
```

(c) Model obtained by GUBS on the considered TRS.

Figure 2: Synthesis of polynomial interpretations in TcT.

because of the minimisation techniques incorporated in GUBS, as outlined in Section 3. Also worthy of note, including the time spend by GUBS, the depicted sized-types were found in a quarter of a second, on one of the authors laptops.

HoSA is also capable of instrumenting a given program, by threading through the computation a program counter. This way, the runtime of a program is reflected in its sized-type and thus HoSA can give quantitative information on the runtime of programs. HoSA is able to analyse the runtime of a series of examples, fully automatically, which cannot be handled by most competitor methodologies (see e.g., [8]). Noteworthy, HoSA can deal with a variety of examples whose runtime is not linear, e.g., sorting algorithms and non-trivial list functions. GUBS is capable of dealing with reasonably sized constraint systems in this context. For instance, the runtime analysis of the function which computes the cross-product of two lists in quadratic time, itself defined in terms of two folds, relies on 87 functions related by 85 constraints. To this end, GUBS analyses 10 SCC individually. The computed bounds are tight, the overall procedure takes just three quarters of a second on this example.

Synthesis of Polynomial Interpretation. *Term rewriting systems* (*TRSs* for short) provide an abstract model of computation that is at the heart of functional and declarative programming. In the last decade termination and resource behaviour of TRSs have been investigated actively. *Polynomial interpretation* provide an elementary method in these context. In its most simple setting, a polynomial interpretation $\mathcal{M}$, i.e. interpretation over polynomials as above, *orients* a TRS $\mathcal{R}$ if $[\![l]\!]_{\mathcal{M}}^{\alpha} >_{\mathbb{N}} [\![r]\!]_{\mathcal{M}}^{\alpha}$ holds for all rules $l \to r \in \mathcal{R}$. Together with certain monotonicity constraints on the interpretation, orientability implies termination. When the interpretation of constructors are additionally sufficiently constrained, quantitative information on the runtime of $\mathcal{R}$ can be obtained. Notably, orientation constraints and monotonicity constraints are straight forward translated into a constraint system. Consider for instance the TRS consisting of the two rules depicted in Figure 2a, corresponding constraints are given in Figure 2b. In Figure 2b, symbols dub, cons and nil denote the interpretation of the corresponding functions from the input rewrite system. The two constraints on the left enforce the orientation. The first constrain on the right enforces sufficient monotonicity constraints on the interpretation, viz, cons should be monotone in its second argument. The final rule enforces that cons is interpreted by a strongly linear interpretation $cons_{\mathcal{M}}(x,xs) \leqslant x + xs + \mathtt{k}$ for some constant $\mathtt{k} \in \mathbb{N}$, thereby relating the interpretation of lists linearly to its size. The so computed interpretation witnesses that the runtime complexity, i.e., the number of reduction steps as measured in the size of the input list, is linear.

So far, we did not conduct a thorough investigation of the strength of GUBS in this context. We expect however an increase in strength and execution time due to the incorporation of GUBS in TcT.

5 Conclusion and Future Work

We have described GUBS, an open source tool for synthesising functions over the naturals that satisfy a given set of inequalities. The development of GUBS was motivated by the lack of dedicated constraint solvers for polynomial inequalities.

In future work, we would like to extend GUBS in various directions. In the short term, we would like to improve up the methods that are currently implemented. This includes dedicated synthesis techniques for certain subclasses of constraint systems or models, e.g., via reductions to *linear programming*. It also includes the search for suitable divide-and-conquer methods. Model synthesis is in general not modular, however for certain classes, e.g. disjoint systems, synthesis is modular. The aforementioned SCC analysis is a first step into this direction. Another direction for future work is to extend upon the set of models that can be found in GUBS. Currently, GUBS is only able to synthesise weakly monotone functions over the naturals. It would be interesting, for instance, to include support of integers and rationals, together with non-monotone functions such as subtraction and division. This is clearly feasible with the current toolset underlying GUBS. Related, an extension of GUBS to an SMT solver for dealing with non-linear integer arithmetic could also be worthwhile. Finally, and maybe most importantly, we would like to see further applications of our tool.

References

[1] E. Albert, P. Arenas, S. Genaim & G. Puebla (2008): *Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis*. In: *Proc. of 15th SAS*, Springer, pp. 221–237.

[2] M. Avanzini & U. Dal Lago (2017): *Complexity Analysis by Polymorphic Sized Type Inference and Constraint Solving*. Technical Report, Universities of Bologna and Innsbruck. Available at `http://cl-informatik.uibk.ac.at/users/zini/CAPSTICS.pdf`.

[3] G. Bonfante, J.-Y. Marion & J.-Y. Moyen (2011): *Quasi-interpretations: A Way to Control Resources*. *TCS* 412(25), pp. 2776–2796.

[4] U. Dal Lago & Barbara Petit (2013): *The Geometry of Types*. In: *Proc. of 40th POPL*, ACM, pp. 167–178.

[5] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann & H. Zankl (2007): *SAT Solving for Termination Analysis with Polynomial Interpretations*. In: *Proc. 10th of SAT*, pp. 340–354.

[6] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann & H. Zankl (2008): *Maximal Termination*. In: *Proc. of 19th RTA*, 5117, Springer, pp. 110–125.

[7] E. Giachino, E. Broch Johnsen, C. Laneve & K. I. Pun (2016): *Time Complexity of Concurrent Programs - - A Technique Based on Behavioural Types -*. In: *Proc. of 12th FACS*, *LNCS* 9539, Springer, pp. 199–216.

[8] J. Hoffmann, A. Das & S.-C. Weng (2017): *Towards Automatic Resource Bound Analysis for OCaml*. In: *Proc. of 44th POPL*, ACM, pp. 359–373.

[9] J. Hughes, L. Pareto & A. Sabry (1996): *Proving the Correctness of Reactive Systems Using Sized Types*. In: *Proc. of 23rd POPL*, ACM, pp. 410–423.

[10] A. Podelski & A. Rybalchenko (2004): *A Complete Method for the Synthesis of Linear Ranking Functions*. In: *Proc. 5th VMCAI*, pp. 239–251.