Counterexample-Guided Partial Bounding for Recursive Function Synthesis

Azadeh Farzan and Victor Nicolet

University of Toronto, Canada

Abstract. Quantifier bounding (aka finitization) is a standard approach in inductive program synthesis to deal with unbounded domains. In this paper, we propose one such bounding method for the synthesis of recursive functions over recursive input data types. The synthesis problem is specified by an input reference (recursive) function and a *recursion skeleton*. The goal is to synthesize a recursive function equivalent to the input function whose recursion strategy is specified by the recursion skeleton. In this context, we illustrate that it is possible to *selectively* bound a *subset* of the (recursively typed) parameters, each by a proper bound. The choices of what parameters must be bounded and by how much are guided by counterexamples. The evaluation of our strategy on a broad set of benchmarks shows that it succeeds in efficiently synthesizing non-trivial recursive functions, where standard finitization would fail.

1 Introduction

Most computational tasks can be broken into logical units, many of which involve evaluating a function over a data collection. Recursively defined data types are broadly used to implement these collections. In functional languages, recursive functions implement computations over these recursive data types. Consider a typical scenario where a programmer has implemented a function f over a collection C by defining a recursive data type A and implementing f as a recursive function $\texttt{foo}_A$. Later, the programmer may need a different implementation $\texttt{foo}_B$ of f over a different data type B; perhaps B is better suited for an optimized implementation of f, or the programmer now needs an implementation of a new function g (in addition to f) over the collection C and the data type B is a much better choice than A for implementing g efficiently. Ideally, the programmer should not have to start from scratch implementing $\texttt{foo}_B$.

In this paper, we propose a generic and efficient algorithm for synthesizing recursive functions in such contexts. Our synthesis problem is specified by the following three components: (1) a recursive *reference implementation* that precisely defines the functionality, (2) a high level *recursion skeleton* that specifies a recursion strategy (i.e. a traversal plan over the new recursive data type) for the target code, and (3) a mapping, called *representation function*, that converts an instance of the new data type to one of the old data type (of the reference implementation), and establishes that the two are different implementations of the same concept.

Let us illustrate our problem setup with the aid of an example. Consider the standard A-labelled binary trees, recursively defined as $T \to Nil \mid Node(A, T, T)$ for an arbitrary type A, and the maximum in-order prefix sum (`mips`) function depicted on the right. `mips` maintains a pair of values: `sum`, which keeps track of the sum of the elements it has traversed so far, and `mps`, which maintains the maximum value over all such sums. This reference implementation precisely defines the functional specification for a function f.

```
let mips t = aux (0, 0) t
and aux s t =
    match t with
    | Nil -> sum, mps
    | Node(a,l,r) ->
        let sum, mps = aux s l in
        aux (sum + a, max (sum + a) mps) r
```

Fig. 1. Maximum in-order prefix sum

Suppose that the programmer needs an alternative implementation that can be efficiently parallelized, and therefore, opts for the divide-and-conquer *recursion skeleton* depicted on the right. The *partially defined* code specifies that the tree should be traversed in a manner that each subtree is processed separately, and then the results should be combined by a function `join`. It does not, however, specify what computation is performed; the implementation of `join` and the initial value for `s0` are *unknown*. In this example, labeled binary trees are the recursive data type for both the reference implementation and the target of synthesis. In cases like this, the representation function simply becomes the identity function.

```
let h t =
    match t with
    | Nil -> s0
    | Node(a,l,r) -> join a (h l) (h r)
```

Our algorithm reduces the problem to a set of recursion-free synthesis problems, which are solved using existing synthesis tools. It synthesizes the unknown computations for `join` and `s0`, and therefore produces the divide-and-conquer implementation of `mips` on binary trees:

```
let s0 = (0, 0)
let join a (s1, m1) (s2, m2) = a + s1 + s2, max m1 (m2 + a + s1)
```

At the high level, the problem of synthesizing a new recursive function can be framed as checking the validity of formulas of the type $\exists f \forall x : \theta.\phi(f, x, \dots)$ where θ is a recursive data type (i.e x ranges over a set of inductively defined terms), f is the target recursive function, and the ellipses stand in for all the relevant components of our specific problem statement as outlined before. Elements of type θ are unbounded in two different dimensions: the recursive structure can be of arbitrary size and each element of it belongs to an unbounded (data) domain. A straightforward way of under-approximating the unbounded specification is to bound the universal quantifier $\forall x : \theta$ in both dimensions. The synthesis problem is reformulated to synthesize the function from a bounded set of examples which are concrete bounded elements of the data type with concrete elements in them. This can be done by applying a counterexample-guided inductive synthesis (CEGIS) [33] algorithm in the straightforward way.

Alternatively, one can attempt to tackle the two dimensions independently. The quantifier $\forall x : \theta$ can be bounded in one dimension, i.e. recursive structures of

bounded size can be considered, and yet the elements of these bounded structures can range over unbounded domains. More formally, the universal quantification is instantiated over a finite set of bounded-depth terms, denoted by set T, and the resulting specification becomes $\exists f.\forall \boldsymbol{a} \in D. \bigwedge_{t \in T} \phi(f, t)$ where $\boldsymbol{a}$ are the free variables of the terms in T and of non-recursive type D. This bounding reduces the original problem to a standard functional synthesis problem (over unbounded data domains) that can be discharged to one of the many known solvers, employing a variety of techniques for it. The set of terms in T can still be discovered in a counterexample guided loop in the spirit of CEGIS, and therefore this algorithm can be viewed as a *symbolic* CEGIS variant.

The thesis of this paper is that forcing bounds on all recursively typed variables is unnecessary and can be avoided algorithmically. A subset of variables can retain their unbounded quantification and yet the problem can be reduced to a recursion-free functional synthesis instance. Recall the `mips` example. The `join` function takes two trees, `l` and `r`, and a value `a` as an input. The recursion-free specification for `join` can retain a universal quantifier on all trees for `l` and limit its bounded exploration to `r`. In other words, one can successfully synthesize the `join` function from examples enumerating a few small candidate trees for `r` and treating `h(l)` (i.e. the result of the computation on `l`) and not `l` itself for the inductive enumeration of examples for synthesis. We discuss in the paper how this information can be algorithmically derived from the specific components of our synthesis problem: the reference implementation, the recursion skeleton, and the representation function.

Beyond the decision on what quantifiers should be bounded, the synthesis algorithm also needs to determine a set of terms that are used to bound these quantifiers. We propose an algorithm that discovers these bounds guided by counterexamples in a refinement-style loop. We show that this algorithm is sound, satisfies the expected weak-progress property that other CEGIS instances have, and is *parsimonious* in a precise sense. We have implemented this algorithm as a prototype synthesis tool Synduce and demonstrate that Synduce can efficiently synthesize recursive functions from specifications.

2 Background & Notation

The notation introduced in this section is used for formalizing the result of applying recursive functions to symbolic inputs.

Terms. We make use of a set of *symbols* that are partitioned into *terminal symbols* Σ, *non-terminal symbols* $\mathcal{N}$, and an infinite set of typed *variables* $\mathcal{V}$. There is a unique symbol $\circ_?$ that stands for a *hole*. Terms are defined by the grammar $T \to x \mid T\ T$ where x is a symbol, and $T\ T$ is a function application. These are the relevant classes of terms:

- *Concrete terms* $T(\Sigma)$ are those containing only terminal symbols. Every concrete term can be interpreted and has a concrete value.
- *Symbolic terms* $T(\Sigma, \mathcal{V})$ are those containing terminal symbols or variables.

- *Closed terms* $T(\Sigma, \mathcal{N})$ are those containing terminal or non-terminal symbols, but no variables.
- *Applicative terms* $T(\Sigma, \mathcal{N}, \mathcal{V})$ are those containing any symbol except the hole symbol.
- *Contexts* $T(\Sigma, \mathcal{N}, \mathcal{V}, \circ_?)$ are those with at least one hole. A *one-hole context* $C[\,]$ is a context with a single occurrence of $\circ_?$, and $C[t]$ stands for the term formed by replacing the single hole in $C[\,]$ with the term t.

Two terms are equal, denoted by $t =_\alpha t'$ (standard alpha conversion), iff there exists two injective substitutions $\sigma : FV(t) \to \mathcal{V} \setminus (FV(t) \cup FV(t'))$ and $\sigma' : FV(t') \to \mathcal{V} \setminus (FV(t) \cup FV(t'))$ such that $\sigma t = \sigma' t'$ (i.e. syntactically equal).

A symbolic term t can be **expanded** into a term t' iff there exists a substitution $\sigma : FV(t) \to T(FV(t') \cup \Sigma)$ that substitutes the free variables of t for symbolic terms with the free variables of t' such that $t' = \sigma t$. The relation $\succeq$ over symbolic terms, is a partial order defined as, $t \succeq t'$ iff t can be expanded into t'. A single variable is the maximal element according to this partial order and concrete terms (of any depth) are minimal elements.

Recursive Functions. This paper focuses on recursive functions $f : \tau \to D$ with terms of a recursive type (τ or θ) as input, and an output of type D. These functions can be executed on concrete or symbolic input terms of type τ. We assume all functions can be translated to *recursion schemes* as defined below:

Definition 1 ([25]). *A* recursion scheme *is a tuple* $\mathcal{P} = \langle \Sigma, \mathcal{N}, \mathcal{R}, \Lambda \rangle$ *where:*

- Σ *is a ranked alphabet of* terminals
- $\mathcal{N}$ *is a finite set of typed* non-terminals.
- $\mathcal{R}$ *is a finite set of rewrite rules, each in one of the following shapes* ($m \geq 0$)*:*

$$\begin{array}{rl} \textit{(pure)} & F\ x_1\ \ldots\ x_m \to t \\ \textit{(pattern matching)} & F\ x_1\ \ldots\ x_m\ p \to t \end{array}$$

  where the x_i *are variables,* p *is a symbolic term,* t *is an applicative term in* $T(\Sigma \cup \mathcal{N} \cup \{x_1, \ldots, x_n\})$*, and* F *is a non-terminal.*
- $\Lambda : \tau \to D$ *is a distinguished non-terminal symbol whose defining rules are always pattern-matching rules.*

We associate with each recursion scheme $\mathcal{P}$ a notion of reduction. A *redex* is an applicative term of the form $F\ \sigma x_1\ \ldots\ \sigma x_m\ \sigma p$ for a substitution $\sigma : \mathcal{V} \to T(\Sigma, \mathcal{N}, \mathcal{V})$ and rule $F\ x_1\ \ldots\ x_m\ p \to t$ in $\mathcal{R}$. The *contractum* of the redex is σt. The one-step reduction relation $\mapsto \subseteq T(\Sigma, \mathcal{N}, \mathcal{V}) \times T(\Sigma, \mathcal{N}, \mathcal{V})$ is defined by $C[s] \mapsto C[t]$ whenever s is a redex, t is a contractum and $C[\,]$ is a one-hole context. A recursion scheme is *deterministic* iff for any redex $F\ s_1\ \ldots\ s_m$ there is exactly one rule $l \to r$ (in $\mathcal{R}$) which *matches* that redex, i.e. there exists a substitution θ such that $F\ s_1\ \ldots\ s_m = \theta\ l$.

Given a recursion scheme $\mathcal{P} = \langle \Sigma, \mathcal{N}, \mathcal{R}, \Lambda \rangle$ and a term $s \in T(\Sigma, \mathcal{N}, \mathcal{V})$, $\mathcal{L}(\mathcal{P}, s)$ denotes the language of $(\Sigma \cup \mathcal{N} \cup FV(s))$-labelled trees resulted from the maximal rewriting of the term s with the one-step reduction relation associated

to $\mathcal{P}$. If $\mathcal{P}$ is deterministic, then $\mathcal{L}(\mathcal{P}, s)$ is a singleton (the term s reduces to only one possible term), and $[\![s]\!]_{\mathcal{P}}$ denotes the unique resulting term. This notion of reduction is slightly different from the one used in [25], in that we do not require the substitution to be closed.

Symbolic evaluation. For any function f that can be defined as a recursion scheme, the symbolic evaluation of f on input s is simply $[\![s]\!]_f$. In other words, $f(s) = [\![s]\!]_f$. In this view, recursive functions and the corresponding recursion schemes are interchangeable. For a recursion scheme $\langle \Sigma, \mathcal{N}, \mathcal{R}, \Lambda \rangle$ representing a function f and a variable x, $f(x)$ and $\Lambda\ x$ become two different ways of referencing the same concept. In this paper, we assume that all recursion schemes to be deterministic total functions. Specifically, they terminate on all inputs; symbolic evaluation (or the equivalent reduction) of a symbolic term always terminates.

Types Notation. We use capital letters A, B, C, and D to refer to base types, which are scalar types $(\text{int}, \text{bool}, \text{char}, \ldots)$ or unlabeled products of scalar types (e.g. $\text{int} \times \text{int}$). Our focus is on functions that take as input elements of recursive variant (or sum) types denoted by $\tau, \theta, \ldots$. We denote by $\kappa_1, \ldots, \kappa_n$ the constructors of a variant type τ with n variants. Each constructor is assimilated to a terminal symbol $\tau_1 \times \ldots \times \tau_k \to \tau$, where $k \geq 0$. We assume that all recursive types define finite structures, that is, one can always construct a term of type τ with a finite number of constructors and elements of base type. $x : \tau$ denotes the judgement x is of type τ, and $\forall x : \tau$ denotes the universal quantification of all variables x of type τ.

In this setting, where we distinguish base types and recursive types, we differentiate **bounded terms**, which are symbolic terms where all free variables are of base type (in $\mathcal{V}_B$), and **unbounded terms** where some variables can be of recursive type. An unbounded term t is a symbolic term of finite size, but there are infinitely many bounded terms that are expansions of t.

3 Formal Definition of the Synthesis Problem

The synthesis problem solved in this paper is defined by three components: a reference recursive function $f : \tau \to D$, a representation function $r : \theta \to \tau$ that maps inputs of the target function to those of f, and a recursion skeleton for the target function. All three components are formally modelled by recursion schemes (Definition 1). f and r are standard recursive functions representable by deterministic recursion schemes. The recursion scheme for the recursion skeleton $\mathcal{S}[\Xi] : \theta \to D$ includes a special set Ξ of symbols as a subset of its terminal symbols, which correspond to the unknown components for synthesis. These unknowns stand for constants or functions that have to be synthesized.

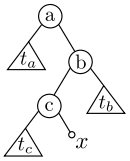
At the high level, the solution to the synthesis problem is the definition of a new recursive function. At the low level, each of the unknowns in Ξ need to be given a definition. In each problem instance, it is assumed that f and $\mathcal{S}[\Xi]$ use a common set of terminal symbols Σ that belong to a background theory $\mathcal{T}$ (e.g. linear integer arithmetic). Formally, the solution is identified by a mapping Z

from the unknowns Ξ to function definitions $\lambda x_1 \ldots \lambda x_n.t$ where $n \geq 0$ and t is a symbolic term in $T(\Sigma, \{x_1, \ldots, x_n\})$ (a concrete term if $n = 0$). Let $\mathcal{S}[\Xi/Z]$ be the recursion scheme obtained by replacing the unknowns Ξ by their definition in Z. Any solution Z that satisfies the following specification is a valid solution:

$$\Psi \equiv \forall x : \theta, \mathcal{S}[\Xi/Z](x) = (f \circ r)(x)$$

Example 1. We use a problem instance with the goal of synthesizing a recursive function on *tree paths* as a running example of this paper. Recall the `mips` function given in Figure 1. Suppose that we want to transform it to a function on *tree paths*[1] as an alternative data type to labelled binary trees. For an A-labelled tree (of type $Tree$), $Path$ is a datatype defined by the following grammar:

$$Path \rightarrow Top \mid Zip((\top|\bot), A, Tree, Path)$$



Intuitively, a path decomposes a tree as shown on the right. The path $Zip(\top, a, t_a, Zip(\bot, b, t_b, Zip(\top, c, t_c, x)))$, from the root to a leaf decomposes the tree into the subtrees t_a, t_b, and t_c.

The synthesis problem is specified by three recursion schemes. The recursion scheme f, on the right, models the function `mips` from Figure 1.

$$f : \begin{cases} \Lambda_f\ t & \rightarrow G\ (0,0)\ t \\ G\ \ s\ \ Nil & \rightarrow s \\ G\ \ s\ \ Node(a,l,r) & \rightarrow G\ (L\ a\ (G\ s\ l))\ r \\ L\ \ a\ \ (s,m) & \rightarrow (s+a, max(s+a, m)) \end{cases}$$

Λ_f is the non-terminal corresponding to the main function `mips` and G is an auxiliary function. An additional non-terminal L is used to mirror the tuple decomposition done by the let-binding in the code of `mips`.

The second recursion scheme is the representation function r from paths to trees. The input path is recursively decomposed by the rewrite rules, and *Node* is constructed recursively on the right or on the left depending on the first value contained in the *Zip* constructor.

$$r : \begin{cases} \Lambda_r\ \ Top & \rightarrow Nil \\ \Lambda_r\ \ Zip(\top, a, t, z) & \rightarrow Node(a, t, \Lambda_r\ z) \\ \Lambda_r\ \ Zip(\bot, a, t, z) & \rightarrow Node(a, \Lambda_r\ z, t) \end{cases}$$

The last recursion scheme specifies the recursion skeleton of the target function with unknowns s_0, g_l and g_r. It traverses the input path, making recursive calls $(\Lambda_S\ z)$ on paths, and calling the reference function on subtrees $(\Lambda_f\ t)$. The goal is then to synthesize implementations of s_0, g_l and g_r such that $\mathcal{S}[s_0, g_l, g_r]$ is equivalent to $f \circ r$.

$$\mathcal{S}[s_0, g_l, g_r] : \begin{cases} \Lambda_S\ \ Top & \rightarrow s_0 \\ \Lambda_S\ \ Zip(\top, a, t, z) & \rightarrow g_l\ a\ (\Lambda_f\ t)\ (\Lambda_S\ z) \\ \Lambda_S\ \ Zip(\bot, a, t, z) & \rightarrow g_r\ a\ (\Lambda_f\ t)\ (\Lambda_S\ z) \end{cases}$$

4 Recursion-Free Approximations

A *system of recursion-free equations* models an approximation of the full functional specification Ψ for a recursive synthesis problem instance.

[1] This example is from [23], which calls this data type *zipper*.

Definition 2. *Given two sets of terminals Σ and Ξ, a* system of recursion-free equations *is a finite set of constraints $\{e_i = e'_i\}$ where $e, e' \in T(\Sigma \cup \Xi, \mathcal{V}_B)$.*

We denote by $\{e_i = e'_i\}_{i \in I}$ the set of constraints of the system, and $\{x_j\}_{1 \leq j \leq n} \equiv \bigcup_{i \in I} FV(e_i) \cup FV(e'_i)$ are the free variables in the system. The above system defines a synthesis problem where Σ is the signature of some theory $\mathcal{T}$ and Ξ is the set of unknowns to be synthesized. A solution Z to this synthesis problem is a mapping from Ξ to function definitions. Z is valid iff the following formula is *valid*:

$$\forall x_1 : D_1 \ldots \forall x_n : D_n. \bigwedge_{i \in I} (e_i = e'_i)[\Xi / Z]$$

where $(e_i = e'_i)[\Xi/Z]$ denotes the term in which the unknowns Ξ have been replaced by their definition in Z. In the rest of the paper, we consider systems of recursion-free equations where the set of terminals Σ and the set of unknowns Ξ are fixed and the same as in the main synthesis problem of Section 3. We say that a system $\mathcal{E}'$ is a sound approximation of a system $\mathcal{E}$ ($\mathcal{E}' \gtrsim \mathcal{E}$) (or the synthesis problem Ψ) when any solution of $\mathcal{E}$ (or Ψ) is also a solution of $\mathcal{E}'$.

4.1 Partially Bounded Quantification

Consider the formal definition of the synthesis problem in Section 3. Bounding the quantifiers consists in expressing the problem on a finite set of bounded terms. This bounding effectively eliminates recursion; recursive calls can be inlined a bounded number of times. Yet, since the free variables of the bounded term are universally quantified over an infinite base domain, a bounded term t of type θ represents an infinite set of concrete inputs (of bounded size).

We propose a different strategy for bounding the quantifiers: we aim to instantiate the quantifier on a finite set of bounded and *unbounded* terms such that the resulting specification is not recursive. To start, we instantiate the universal quantifier by a finite set of arbitrary symbolic terms T. Our first approximation then becomes the set of constraints:

$$E(T) = \{\mathcal{S}[\Xi](t) = (f \circ r)(t) \mid t \in T\} \tag{1}$$

The set of constraints $E(T)$ can be seen as a synthesis problem where free variables are universally quantified and the unknowns in Ξ are to be synthesized. $E(T)$ is not guaranteed to be a system of recursion-free equations for all choices of T. For an arbitrary symbolic term t, calls to recursive functions may appear in subterms of $\mathcal{S}[\Xi](t)$ and $(f \circ r)(t)$. Restricting T to bounded terms would yield a recursion-free system after symbolic evaluation of both sides of the equation.

This, however, is too restrictive. There may exist unbounded terms t where the equation $\mathcal{S}[\Xi](t) = (f \circ r)(t)$ can be *rewritten* to an equivalent recursion-free equation. Intuitively, in an applicative term (resulting from the symbolic evaluation of a recursive function f) the simple subterms of the form $f(x)$ where x is a variable can be eliminated by replacing $f(x)$ with a single variable a of type D which now stands for the result of the invocation of f on any x.

Definition 3. *A symbolic term t is maximally reducible (t is a MR-term) by a recursion scheme $\mathcal{P} = (\Sigma, \mathcal{N}, \mathcal{R}, \Lambda)$ iff $[\![t]\!]_{\mathcal{P}}$ is an applicative term in $T(\Sigma, \mathcal{N}, \mathcal{V})$ such that replacing all subterms of the form $(\Lambda\ x)$ (where $x \in \mathcal{V}$) by a fresh variable $x' \notin FV(t)$ yields a symbolic term.*

Example 2. The term $z = Zip(\top, a, t, Top)$ where a is an integer and t is of type $Tree$ is maximally reducible by $f \circ r$ and $\mathcal{S}[s_0, g_l, g_r]$ (cf. Example 1). First we have $r(z) = [\![z]\!]_r = Node(a, t, Nil)$ and $(f \circ r)(z) = G\ (L\ a\ (\Lambda_f\ t))\ Nil$. If $\Lambda_f\ t$ is replaced by (a_1, a_2) (of type int $\times$ int), then the term can be reduced further to $(a_1 + a, max(a_1 + a, a_2))$. For the other function, we have $\mathcal{S}[s_0, g_l, g_r](z) = g_l\ a\ (\Lambda_f\ t)\ s_0$. If $\Lambda_f\ t$ is also replaced by (a_1, a_2), then the term reduces to the symbolic term $g_l\ a\ (a_1, a_2)\ s_0$. Note that z is an unbounded term, since t is a variable representing a tree of arbitrary depth.

If every term in T is maximally reducible by both $(f \circ r)$ and $\mathcal{S}[\Xi]$, then every call to a recursive function can be eliminated in $E(T)$. Note that this new *sufficient* condition for $E(T)$ to be recursion free is strictly weaker than the condition of having the terms in T to be bounded; a maximally reducible term need not be a bounded term.

Definition 4. *A set of constraints $E(T) = \{\mathcal{S}[\Xi](t) = (f \circ r)(t)\ |t \in T\}$ is well-formed iff every $t \in T$ is maximally reducible by $f \circ r$ and $\mathcal{S}[\Xi]$.*

A well-formed set of constraints $E(T)$ can be transformed to a system of recursion-free equations. For each free variable $x : \theta$ in $E(T)$, a fresh variable $a : D$ is added and the subterms $(f \circ r)(x)$ and $\mathcal{S}[\Xi](x)$ are replaced by a in every constraint. We call this rewriting step *recursion elimination* over D. Note that the calls to $f \circ r$ and $\mathcal{S}[\Xi]$ are both replaced by the same variable, since their equivalence is part of the specification of the synthesis problem.

The transformation described above produces a recursion-free system of equations, but it does not always yield a *sound abstraction*, specifically when $f \circ r$ is *not onto* D. There may exist a solution of Ψ that is not a solution of the resulting system of equations. This can be fixed by having additional constraints (invariants) on the fresh variables. Let $Im_f : D \to$ bool a predicate such that $f \circ r$ is onto $\{c \mid c : D \wedge Im_f(c)\}$. Then, the abstraction is sound if the choices for $a : D$ are limited to when $Im_f(a)$ holds.

Example 3. Recall Example 1. The maximum in-order prefix sum is not onto int $\times$ int, since the second element of the pair is always a positive integer. The constraint $Im_f(x, y) = y \geq 0$ is required to make the function onto. In Example 2, a_2 must be a positive integer.

Definition 5. *Let T be a set of maximally reducible terms by $f \circ r$ and $\mathcal{S}[\Xi]$, and Im_f a predicate such that $f \circ r$ is* onto $\{c \mid c : D \wedge Im_f(c)\}$. *We denote by $\mathcal{E}(T)$ the equation system obtained by rewriting each constraint in $E(T)$ to a recursion free equation, through recursion elimination over $\{c \mid c : D \wedge Im_f(c)\}$.*

In the synthesis problem defined by $\mathcal{E}(T)$, the variables introduced by recursion elimination are universally quantified over their restricted range. The exact encoding of the range restriction by Im_f depends on the implementation of a synthesis oracle.

Proposition 1. *Z is a solution of $\mathcal{E}(T)$ iff Z is a solution of $E(T)$.*

The proof follows from the construction of $\mathcal{E}(T)$ based on $E(T)$. Combining this with the fact that $E(T)$ results from bounding the universal quantifications in Ψ, we can conclude that $\mathcal{E}(T)$ approximates Ψ.

Theorem 1 (Sound approximation). *If T is a set of maximally reducible terms by $f \circ r$ and $\mathcal{S}[\Xi]$, $\mathcal{E}(T)$ is a sound approximation of Ψ.*

By construction, any solution of the functional specification Ψ is a solution of the system of equations $\mathcal{E}(T)$.

Example 4. Let $T = \{Top, Zip(\top, a, t, Top), Zip(\bot, a, Nil, z)\}$ be a set of terms, where $a : int$, $t : Tree$ and $z : Path$. Top is a concrete term, therefore maximally reducible. We saw in Example 2 that $Zip(\top, a, t, Top)$ is a MR-term. With a similar reasoning, one can conclude that $Zip(\bot, a, Nil, z)$ is a MR-term; note how the term differs in which subterm is unbounded depending on the first component of the Zip. Therefore, $E(T)$ is a well-formed set of constraints and by substituting $\Lambda_f\ t$ and $\Lambda_S\ z$ for (a_1, a_2) (where $a_1 : \text{int}$ and $a_2 \in \{v : \text{int} | v \geq 0\}$), we obtain the following recursion-free system of equations:

$$\mathcal{E}(T) = \begin{cases} 0, 0 = s_0, \\ a_1 + a, max(a_1 + a, a_2) = g_l\ a\ (a_1, a_2)\ s_0 \\ a_1 + a, max(a_1 + a, a_2) = g_r\ a\ s_0\ (a_1, a_2) \end{cases}$$

with free variables $a : int$, $a_1 : int$ and $a_2 \in \{v : int | v \geq 0\}$.

In contrast to a canonical CEGIS setting, where the approximation is the specification projected over a finite set of concrete terms, our abstraction is over an infinite set of concrete terms represented by a finite set of symbolic terms. In the original functional specification, the equational constraint $(f \circ r)(x) = \mathcal{S}[\Xi](x)$ ranges over all possible terms x of type θ. In the abstraction $\mathcal{E}(T)$, the universally quantified variables are the free variables of the terms in the equations, which correspond to the variable symbols of scalar type used in the symbolic terms of T, modulo the introduction of fresh variables during the rewriting of the set of constraints $E(T)$ to the system of equations $\mathcal{E}(T)$.

4.2 Refining Systems of Equations

Our approximation, the system of equations $\mathcal{E}(T)$, is parametric on a set of maximally reducible terms T. This approximation can be refined by adding terms to T, since for any two set of terms R and T such that $R \subseteq T$, $\mathcal{E}(R) \gtrsim \mathcal{E}(T)$.

The convergence of the refinement process depends on the terms added at each step. We present our refinement algorithm in the next section, but the main insights behind it, not tied to specific algorithmic choices, are captured by Propositions 2 and 3.

Proposition 2. *Let T be a set of MR-terms and Z be a solution of $\mathcal{E}(T)$. Then for any term t' such that there exists $t \in T$ s.t $t \succeq t'$, Z is a solution of $\mathcal{E}(T \cup \{t'\})$.*

This proposition implies that if Z is a spurious solution, then a counterexample term showing that it is not a solution of Ψ is necessarily not expanded from a term in T. We also learn that T should ideally be an antichain of $\succeq$ at every refinement round, since adding expanded terms does not strengthen the approximation.

Proposition 3. *Given two terms t and t' such that $t \succeq t'$ (i.e. t' is an expansion of t) and a set of MR-terms T such that $\forall x \in T, \neg(x \succeq t \wedge t \succeq x)$, we have $\mathcal{E}(T \cup \{t\}) \precsim \mathcal{E}(T \cup \{t'\})$.*

Adding the less expanded term (i.e. t) yields both a more general approximation and a more compact one. In other words, given a choice, always choose the least expanded term as the counterexample for refinement.

5 Synthesis Algorithm

Our synthesis algorithm computes a sequence of approximations of the functional specification Ψ from Section 3. Each approximation is a system of equations of the form $\mathcal{E}(T)$ (Definition 5). The approximations are incrementally refined until the synthesis solution for one is also a valid solution for the synthesis problem specified by Ψ.

Figure 2 illustrates the work flow of our algorithm. At the beginning of each iteration, a solution of the system of recursion-free equations $\mathcal{E}(T)$ is synthesized. If no solution is found, then there is no solution for the original synthesis problem, since the $\mathcal{E}(T)$ is guaranteed to be a sound approximation (Theorem 1). If a solution Z is found, then Z is verified against Ψ and if it passes, then it is returned as a solution. Otherwise, the verifier returns a counterexample term x_C. By Proposition 2, x_C cannot be an expansion of any term in T, and new terms related to x_C have to be added to T in the spirit of refinement.

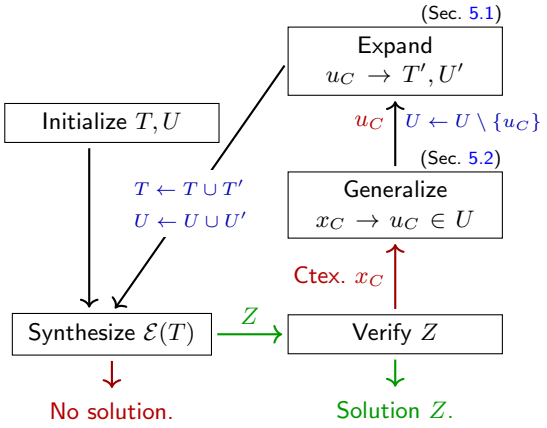
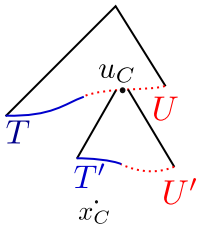


Fig. 2. Approximation refinement algorithm.

The algorithm additionally keeps track of a set U of non-maximally reducible terms, which intuitively represents the set of inputs not *covered* by the current approximation. The sets T and U are *complementary* in a precise sense: $T \cup U$ is always a *boundary of* $\succeq$. A boundary (of a partial order) is an antichain C such that for any bounded term t, there is some c in C such that $c \succeq t$.

The counterexample x_C is necessarily an expansion of some term $u_C \in U$. But since u_C is by definition not maximally reducible, one cannot just remove it from U and add it to T. The Expand step takes u_C as an input and produces two sets T' and U' to update the current sets T and U and repair the boundary before the loop restarts.

The figure on the right is a graphical representation of the boundary repair. The sets T (in blue) and U (in red) initially form a boundary. This boundary is updated by removing the term u_C and adding U' and T' (the results of the Expand step) to form a new boundary. The fact that $T \cup U$ always forms a boundary is a required invariant of this refinement loop: (i) T, as a parameter of $\mathcal{E}(T)$, is required to be an antichain (as discussed in Section 4.2), and (ii) the Generalize step relies on the assumption that U is an antichain containing all the terms not yet sufficiently expanded to be in T.



We rely on existing tools/techniques for the steps Synthesize and Verify of Figure 2. In the following, we describe the Initialize, Expand, and Generalize steps of the algorithm.

Initialization. There is a straightforward way to initialize T and U: apply the Expand component to a single variable x of type θ and take the resulting sets T of maximally reducible terms and U of non-maximally reducible terms. The Expand step is described in the next section. For Example 1, a variable x of type *Path* is expanded to produce $T = \{Top\}$ and $U = \{Zip(\bot, a, t, z), Zip(\top, a, t, z)\}$ with variables a, t, and z of the appropriate types.

5.1 Expand : Producing Maximally Reducible Terms

Given an input term u_C, Expand generates two sets T' and U' such that the terms in T' are maximally reducible by both $f \circ r$ and $\mathcal{S}[\Xi]$. The algorithm on the right illustrates the process. At each step, a term u_0 is picked from the set of non-maximally reducible terms U'. This term is expanded once, by a call to ExpandOnce (which is described later). The resulting set of terms is then partitioned into a set of maximally reducible terms T' and a set of non-maximally reducible terms U''; the latter is used to update U'.

$T' = \emptyset, U' = \{u_C\}$;
while $T' = \emptyset$ **do**
  Pick u_0 in U';
  $S = \mathsf{ExpandOnce}(u_0)$;
  $T', U'' = \mathsf{Partition}(S)$;
  $U' = (U' \setminus u_0) \cup U''$;
end
return T', U'

The choice of u_0 at the first line of the loop is important for the termination of the algorithm. There may be an infinite sequence of expansions if the u_0's are adversarially chosen. There always exists a finite sequence of expansions yielding bounded terms which are by definition maximally reducible. A breadth-first exploration of all expansions is one such strategy that ensures the termination of the algorithm.

ExpandOnce. The input of ExpandOnce is a term u_0 that is not maximally reducible. The following proposition characterizes u_0 and the reason for its non-reducibility:

Proposition 4. *Let $u_0 \in T(\Sigma, \mathcal{V})$ and $g = (\Sigma, \mathcal{N}, \mathcal{R}, \Lambda)$ a recursion scheme. u_0 is not maximally reducible by g iff there exists a subterm of $[\![u_0]\!]_g$ of the form $s = F\ t_1\ \ldots t_n\ x$, where $F \in \mathcal{N}$ and $F \neq \Lambda$, the terms $t_1 \ldots t_n$ are applicative terms, and $x \in FV(u_0)$.*

The proof by cases on the subterms of u_0 is given in Appendix A.2. In order to take a step towards making u_0 maximally reducible, the variable x needs to be expanded. Expanding x into a term guarantees some rule $F\ x_1\ \ldots x_n\ p \to t \in \mathcal{R}$ can be used to reduce u_0 further. Such a rule is guaranteed to exist for a recursion scheme representing a total function.

Next, we define how u_0 is expanded at a variable x identified by Proposition 4. u_0 can be written as $C[x]$ for some one-hole context $C[\ ]$. Assume the type β of x has constructors $\kappa_1, \ldots \kappa_n$ where each κ_i has type $\gamma_i \to \beta$. The *pointwise expansion* of u_0 at x is the set of terms $\{C[\kappa_1(x_1)], \ldots, C[\kappa_n(x_n)]\}$ where each x_i is a variable (or a tuple) of variables of type γ_i.

In summary, ExpandOnce first identifies a variable x in u_0 (Proposition 4) that needs to be expanded and then performs the *pointwise expansion* of u_0 at x and returns the resulting set of terms.

One important feature of ExpandOnce is that terms are expanded only where needed. Proposition 4 identifies the precise location (i.e. x) where expanding is necessary and ignores locations where it is not.

Example 5. Recall Example 1. Suppose $u_0 = \mathit{Zip}(\top, a, t, \underline{z})$ is a (symbolic) path and an input to ExpandOnce, where a is an integer, t is of type Tree, and z is of type Path. u_0 is not maximally reducible and has to be expanded. Note that $r(u_0) = \mathit{Node}(a, t, \Lambda_r\ \underline{z})$ and therefore $(f \circ r)(u_0) = G\ (L\ a\ (\Lambda_f\ t))\ (\Lambda_r\ \underline{z})$. The subterm $(\Lambda_r\ \underline{z})$ blocks the reduction of the term starting with G, because $\underline{z}$ blocks the reduction of $\Lambda_r\ \underline{z}$ and therefore, u_0 has to be expanded at $\underline{z}$. The pointwise expansion of u_0 at $\underline{z}$ yields the terms $u_1 = \mathit{Zip}(\top, a, t, \underline{\mathit{Top}})$, $u_2 = \mathit{Zip}(\top, a, t, \underline{\mathit{Zip}(\top, a', t', z')})$, and $u_3 = \mathit{Zip}(\top, a, t, \underline{\mathit{Zip}(\bot, a', t', z')})\}$. Note that the tree element t need not be expanded; we showed in Example 2 that u_1 is maximally reducible and therefore, the expansion loop stops and returns $T' = \{u_1\}$ and $U' = \{u_2, u_3\}$.

Consider the symmetric term $\mathit{Zip}(\bot, a, \underline{t}, z)$ acquired by replacing the $\top$ in u_0 with $\bot$. The expansion of this term yields $T' = \mathit{Zip}(\bot, a, \underline{\mathit{Nil}}, z)$ and $U' = \{\mathit{Zi}(\bot, a, \underline{\mathit{Node}(a', l, r)}, z)\}$. Note that unlike the case for u_0, the tree element of the path has to be expanded and the path element need not be expanded.

5.2 Counterexample Generalization

The generalization of the counterexample x_C is the unique term $u_C \in U$ such that $u_C \succeq x_C$. The term u_C is guaranteed to exist because the algorithm maintains the invariant that $T \cup U$ is a *boundary*, and it is unique since U is always an antichain.

Example 6. After initialization, the synthesis solver attempts to find a solution for the system of equations given in Example 4. One possible solution is

$$s_0 = (0, 0) \qquad g_l(a, (s_1, m_1), (s_2, m_2)) = a + s_1, max(m_1, a + s_1)$$

together with a similar solution for g_r. But the solution for g_l is incorrect; the first component should be $a + s_1 + s_2$ (i.e. the sum of both partial sums and the label of the node). The verifier returns a counterexample of the form $x_c = Zip(\top, 1, Node(?), Zip(\top, -2, Node(?), ?))$ where the question marks stand for concrete subterms of the appropriate type. These subterms are ignored. The counterexample is generalized by selecting $u_2 = Zip(\top, a, t, Zip(\top, a', t', z'))$ (where $u_2 \succeq x_C$), the term that was stored in U after the expansion described in Example 5. This determines where the algorithm must unfold the path one more time to build a stronger approximation.

We report in Section 7 that SYNDUCE succeeds in finding a solution for this example with 3 refinement rounds in 1.57s, whereas the symbolic CEGIS (described in Section 1) times out after 10min over 6 refinement rounds.

5.3 Algorithm Properties

Soundness. Under the assumption that the steps Synthesize and Verify are soundly implemented, the overall algorithm is sound. By construction, T is always a set of maximally reducible terms. Therefore, $\mathcal{E}(T)$ is a guaranteed to be a sound approximation of Ψ by Theorem 1. The soundness of the verification oracle guarantees that any returned solution is in fact a solution of the synthesis problem specified by Ψ.

Weak Progress. Consider the naive algorithm that would expand T by simply adding the counterexample x_C to it; x_C is a maximally reducible term after all. This naive algorithm satisfies a weak progress property, namely that, the spurious solution Z from any round will not be a solution in any future round. Our algorithm does something more sophisticated and therefore it has to be argued that the same weak progress property holds. First, Expand satisfies the following property that guarantees $T \cup U$ to always be a boundary:

Proposition 5. *Let t be some symbolic term and T', U' be the results of the call to Expand(t). Then $T' \cup U'$ is a boundary of the set $\{t' | t \succeq t'\}$.*

Let u_C be the generalization of x_C. Proposition 5 guarantees that Expand computes and adds all possible expansions of u_C to T. This in turn implies that there always exists a term $t \succeq x_C$ in the updated set T (after the call to Expand), which rules x_C out as a spurious solution in all future rounds. Note that the algorithm relies on the existence of u_C in U. For this, it requires $T \cup U$ to be a boundary.

Parsimony. Finally, we can show that our algorithm is parsimonious with the selection of the terms for T in the following precise way:

Theorem 2. *[Parsimony] Let us assume (T, U) is a boundary that our algorithm reaches in some round, then (T, U) is optimal in the following two senses:*

- *for every $t \in T \cup U$ there is no MR-term t' such that $t' \succeq t$.*
- *there is no non-empty subset T' of T and set U' such that $(T \setminus T') \cup U'$ is a boundary and $\mathcal{E}(T \setminus T') \precsim \mathcal{E}(T)$.*

Intuitively, all the terms in T are expanded to the extent necessary and no proper subset of T can form a boundary that maintains the same precise approximation that $T \cup U$ induces. A proof is given in Appendix A.5.

6 Implementation

Our approach is implemented in Synduce [35], a tool written in OCaml [21], and the inputs are recursive functions and datatypes written in Caml.

6.1 Verification and Synthesis Oracles

Synduce uses bounded model checking to implement Verify from Figure 2. A bounded check for the validity of a synthesis solution Z is encoded as the validity of the formula $\wedge_{t \in T} \forall a \in FV(t). \mathcal{S}[\Xi/Z](t) = (f \circ r)(t)$ for a set of *bounded* terms T. Z3 [24] is used as the backend SMT solver, which produces a counterexample in the form of a term for which at least one equality constraint is invalid.

Synduce spends most of its time in the Synthesize box of Figure 2. Since the input to Synthesize is guaranteed to be a recursion-free synthesis specification, any off-the-shelf syntax-guided synthesis (SyGuS) [4] solver that supports the standard language [28] can be used to implement Synthesize. We use CVC4 [5] for the results presented in this section.

A SyGuS problem is specified by a grammar describing the space of programs to be synthesized and a set of constraints. In this case, the grammar is generated from the type of the functions to be synthesized (the unknowns in Ξ), which can be inferred from the constraints where they appear. Instances of generic grammars for integers and booleans can be found in the SyGuS language specification [28], and these grammars for base types can be combined into tuples in a straightforward manner. The constraints are the equations of the system, with the addition of the predicates constraining the domain of the variables, i.e. Im_f from Definition 5. Each recursion-free equation $e = e'$ is translated to a constraint of the form $\neg(\bigwedge_{v \in FV(e) \cup FV(e')} Im_f(v)) \vee e = e'$ where $Im_f(v)$ is the predicate associated to the variable v.

6.2 Baseline Method

The goal of our experimentation is to evaluate the efficiency and efficacy of the proposed partial quantifier bounding approach for synthesis of recursive programs. Since there is no available (automated) tool that solves the specific problem posed in this paper, we implemented the symbolic CEGIS technique (as outlined in Section 1) to serve as a *baseline*. To be precise, the algorithm of Figure 2 is modified by removing the Generalize and Expand steps; the symbolic counterexample returned by the verification at each step is added directly to the set of terms instead of being generalized. The set T is also initialized as a set of bounded terms of some minimal depth, depending on the particular definition of the data type. Note that since the baseline method is counterexample-guided, it is better than the more straightforward finitization techniques, for example, manual finitization by a preset bound.

We also implemented the concrete CEGIS method (outlined in Section 1) to confirm that the symbolic CEGIS is the better choice. Symbolic CEGIS solves 6 more benchmarks than concrete CEGIS, and does better time-wise in the vast majority of the rest. The detailed results of the comparison between the three algorithms are given in Appendix B.2.

6.3 Optimizations

We implemented a few simple, straightforward and generic (i.e. they can be incorporated in any SyGuS solver) optimizations. These aim to compensate for the brittleness of the SyGuS solvers, which can fail for very simple constraints for no good reason. Here is a brief overview of these optimizations, which are applicable to any system of equations (baseline's and ours):

- *Syntactic definitions*, which are those that define an unknown function ξ unequivocally in the form of $\xi(x_1, \ldots, x_n) = t$, can be identified quickly and eliminated from the synthesis task to simplify it.
- A system of equations can be split into *independent subsystems* by identifying an independent subsets of equations. A subset of equations is independent if it constrains a subset of the unknowns that does not appear in the rest of the set of equations. Identification of independent subsystems generates simpler subproblems.
- Instead of starting from a default initial state, we can start from a set of terms that makes for an interesting first round and consequently saves a few refinement rounds from the solution. We form a set of initial terms by using the Expand routine to expand enough terms such that each unknown appears in at least one constraint in the approximation for the first round.

These optimizations are applied to both the baseline method and our algorithm for the purpose of evaluation. Appendix B.3 includes more detailed evaluation of them and experimental results illustrating their precise impact on each algorithm.

7 Evaluation

We evaluate SYNDUCE on a broad set of benchmarks. Our benchmarks are grouped into six categories. Table 1 lists all the benchmarks, grouped accordingly. Each category, shares the same representation function and *polymorphic* recursion skeleton, but a different reference implementation is used to specify the synthesis problem. The recursion skeletons (and the representation functions) are polymorphic and therefore reusable. Only 9 different skeletons and 4 different representation functions were used across our 43 benchmarks. More details about the benchmarks appear in Appendix B.1, including the skeletons used in each category.

7.1 Case Studies

Changing Tree Traversals. An example of this category is the `mips` example used in the introduction. The reference function is a natural implementation of a function with a post- or in-order traversal of a binary tree. The target is an equivalent implementation corresponding to the divide-and-conquer tree homomorphism style recursion.
From Trees to Paths. A tree path (zipper in [23]) is a data structure used to represent a tree together with a subtree that is the focus of attention. Our running example belongs in this category. The other benchmarks in this category are from [23].
Enforcing Tail Recursion. In this category, the reference implementation is a direct-style recursion on the data structure, while the recursion skeleton specifies that an accumulator should be used to make the function tail-recursive. Tail recursive functions generally compile to more efficient code.
Combining Traversals. Suppose a collection of existing implementations computes different functions with different traversals of the same data structure. If in some larger context all of these functions need to be computed, *combining* them can lower the amortized cost. In this set of benchmarks, we synthesize automatically the implementation that corresponds to traversing the data structure with a single recursion strategy, combining the computations into one.
Tree Flattening. These benchmarks target the synthesis of an implementation on the more complex *plane tree* data structure from a reference implementation on the simpler binary tree data structure.
Parallelizing Functions on Lists. Parallelizing a function on lists can be seen as the translation of a recursive function on cons-lists to a homomorphic function on lists built with the concatenation operator. These benchmarks are from [7,8,22].

7.2 Experimental Results

To best of our knowledge, there are no available tools that can be directly compared against SYNDUCE. We can transform our specification to a format that can be accepted by LEON [17]. However, the latter does not succeed in solving

Table 1. Experimental Results. Benchmarks are grouped by categories introduced in Section 7.1. # steps indicates the number of refinement rounds. T_{last} is the elapsed time before the last call to the SyGuS solver in the last refinement step before timeout. All times are in seconds. The best time is highlighted in bold font. A '-' indicates timeout (> 10 min). The "Inv" column indicates if codomain constraints were required. Experiments are run on a laptop with 16G memory and an i7-8750H 6-core CPU at 2.20GHz running Ubuntu 19.10.

Class	Benchmark	Inv.	SYNDUCE			Baseline Method		
			time	# steps	T_{last}	time	# steps	T_{last}
Changing Tree Traversals	sum	no	**0.03**	2	0.01	0.04	3	0.02
	max	no	**0.33**	1	0.00	0.34	2	0.01
	max 2	no	**0.25**	1	0.00	0.34	2	0.01
	min	no	**0.23**	1	0.00	0.32	2	0.01
	min-max	no	**0.85**	3	0.15	73.16	3	0.06
	max weighted path	no	0.09	3	0.03	**0.07**	3	0.02
	sorted in-order	no	**0.01**	1	0.00	43.97	4	1.98
	pre-order poly.	no	**16.09**	2	0.06	-	4	0.97
	`mips`	yes	**0.29**	2	0.04	-	4	2.70
	in-order mts	yes	**0.41**	2	0.04	-	4	4.84
	post-order mps	yes	**132.14**	4	82.56	-	6	39.29
From Tree to Path	sum	no	0.07	2	0.02	**0.06**	3	0.02
	height	no	**0.90**	1	0.00	1.24	5	0.43
	max weighted path	no	0.15	2	0.03	**0.12**	3	0.03
	max w. path (hom)	no	**0.01**	1	0.00	1.42	4	0.69
	leftmost odd	no	**0.01**	1	0.00	-	4	0.27
	`mips`	yes	**1.57**	3	0.50	-	7	322.45
Enforcing Tail Recursion	sum	no	**0.02**	2	0.01	0.03	3	0.02
	mts	no	**5.86**	2	0.02	115.58	3	0.06
	mps	no	1.68	2	0.02	**0.34**	3	0.03
Combining Traversals	mts + sum	no	9.71	2	0.02	**5.42**	3	0.03
	sum + mts + mps	yes	**0.26**	3	0.12	-	3	0.04
Tree Flattening	sum	no	**0.07**	3	0.04	0.07	2	0.01
	product	no	**0.07**	2	0.01	0.16	2	0.01
	max of heads	no	0.21	2	0.02	**0.18**	3	0.03
	max of lasts	no	**0.21**	2	0.02	0.33	3	0.03
	max sibling sum	no	5.26	2	0.03	**2.72**	3	0.04
Parallelizing Functions on Lists	sum	no	**0.08**	1	0.00	0.30	3	0.04
	sum of even elts.	no	**0.10**	1	0.00	0.39	3	0.04
	length	no	**0.07**	1	0.00	0.22	4	0.05
	last	no	**0.01**	1	0.00	0.03	2	0.01
	product	no	**0.07**	1	0.00	0.31	3	0.04
	polynomial	no	**0.07**	1	0.00	0.71	5	0.10
	hamming	no	**0.10**	1	0.00	0.46	3	0.04
	min	no	**0.02**	1	0.00	0.08	2	0.01
	is sorted	no	3.45	2	0.11	**3.12**	4	0.14
	linear search	no	**0.08**	1	0.00	0.35	3	0.04
	line of sight	no	**0.86**	2	0.09	7.67	4	0.34
	mts	yes	**0.10**	1	0.00	4.80	4	0.08
	mps	yes	**0.09**	1	0.00	4.73	4	0.08
	mts and mps combined	yes	**0.38**	2	0.11	210.84	6	36.77
	mss	yes	**4.82**	3	1.53	-	6	24.23
	count max elements	no	**138.20**	1	0.00	-	3	0.46

even the simplest of our benchmarks (e.g. *sum* in the list function parallelization category), likely due to the fact that the required deductive rules are missing. We comment on the rest of the available tools in Section 8.

Table 1 presents the results of comparing SYNDUCE against the baseline method. Both techniques use symbolic counterexamples, and therefore, the comparison can highlight the performance impact of our partial bounding algorithm. The most important point of comparison is the overall synthesis time. In 9 out of 43 benchmarks, the baseline method times out. In another 5 cases, it outperforms the baseline by two orders of magnitude. In the easiest of the benchmarks, i.e. when the overall synthesis time of the baseline is in tens of milliseconds, the two methods are equally good within a small margin of error. The bold number in each row highlights the fastest synthesis time.

Amongst the 9 benchmarks for which the baseline algorithm times out, 7 are cases where SYNDUCE takes advantage of partial bounding by leaving some quantifiers unbounded. The baseline algorithm in these cases requires more terms and terms of higher complexity in the finite approximations. Two of the 9 benchmarks (*post-order mps* and *sum + mts + mps*) are cases where the set of maximally reducible terms is exactly the set of bounded terms (i.e. one cannot take advantage of partial bounding), but SYNDUCE still outperforms the baseline because it adds smaller terms to the abstraction through generalization and produces less complex problems for the backend synthesis oracle. In summary, both counterexample generalization and the partial bounding yield big practical advantages in comparison with the baseline symbolic CEGIS algorithm.

It is noteworthy that whenever an instance is hard, the majority of the time is spent in the Synthesize step. This becomes nearly 100% of the time for the baseline algorithm whenever it times out. The weakness of the baseline method lies in the fact that the recursion-free instances generated by it are too difficult to solve by the backend solver. The timeout occurs within a few refinement rounds (at most 7) when the baseline algorithm gets stuck in the Synthesize step attempting to solve a prohibitively difficult recursion-free synthesis instance.

Across all benchmarks, our algorithm generally requires fewer refinement rounds than the baseline method. The few exceptions are the cases where the synthesis oracle gets lucky in producing a good solution when the target programs are very simple, for example in the case of the *sum* and *product* benchmarks of the flat tree category.

Finally, to isolate the precise contribution of the partial bounding idea, we evaluated the effect of each optimization on each algorithm. The applicability of a particular optimization highly depends on the particular set of constraints, which in turn depends on the specific benchmark and the algorithm (ours vs baseline) producing the constraints. Our synthesis algorithm yields more general and more succinct constraints, to which the optimizations are more often applicable. Of the 9 cases where SYNDUCE succeeds and the baseline method times out, 7 are due to the inapplicability of these (simple) optimizations. SYNDUCE outperforms the baseline algorithm with all optimizations turned off for both. The complete ablation results are given in Table 3 of Appendix B.3.

8 Related Work

Synthesizing recursive programs is a challenging task, and several automated techniques have tackled the problem with different specifications of the problem and different approaches to the solution.

Finitization, for example by bounding the depth of unbounded inputs or the number of recursive calls or loop iterations, is a straightforward way of dealing with unboundedness in synthesis [36,4] and verification [9]. In [31,32], high-level synthesis techniques use domain specific knowledge to finitize input programs. Quantifier instantiation, i.e. replacing quantified terms with ground terms, is commonly used in theorem proving and verification, and has also been useful in synthesis [30]. Our proposed algorithm can be viewed in the spirit of quantifier instantiation, with the major difference that (universally) quantified terms are replaced with other (universally) quantified terms which are still over an unbounded domain, yet with fewer degrees of freedom in unboundedness.

Synthesis through Program Transformation. Our precise problem statement is inspired by the transformation system developed by Burstall and Darlington [6]. They set to automate the task of transforming an initial program specified as a set of first-order recursion equations into a more efficient program, by altering the recursive structure. Their approach is based on transformation rules and semi-automatic. They use specific rules, e.g. *associativity* of a data operation, to perform the transformations and such rules do not generalize well. We defer the reasoning about the operations on the data to an SMT solver, and therefore need not rely on such rules. Techniques based on program transformation have been applied to the synthesis of special classes of recursive programs before [12,14]. For example, the work in [1] focuses on tail recursion and a lot of attention has been given to producing divide-and-conquer recursions in the way of automated parallelization [7,2,22].

Synthesizing Recursive Functional Programs. Inductive techniques were developed to construct recursive programs from input/output examples [34], and this approach has been extended in more recent work [15,16]. The latter two are examples of an analytical approach to program synthesis in which programs are constructed from the analysis of examples. Other recent approaches are search-based methods. Escher [3] synthesizes recursive functions from user-provided components by interactively asking for more examples from the user. λ^2 [10] synthesizes data structure transformations from input/output examples using higher-order functions.

Tools like λ^2 and Escher can be complementary to Synduce in a more general context of recursion synthesis. The user can try to synthesize an implementation of a recursive function over a *simple* data type using λ^2 or Escher using input/output examples with a higher chance of success. This then serves as the reference implementation input to Synduce which can aim for a more sophisticated implementation over a more complex recursive datatype.

Myth [26], Myth2 [11] and SynQuid [27] use type information to direct the search for a program satisfying a specification. In Myth, this specification

is a set of input/output examples. Myth2 generalizes this approach by treating examples as limited types. The specification for Synquid is a polymorphic refinement type, and the tool synthesizes an implementation of the given type using components provided by the user. Type-based approaches work well within the expressivity of refinement-types as specifications, but refinement types cannot express constraints for all desired synthesis tasks. Our specification is strictly stronger than both input/output examples and refinement types.

In SyntRec [13], reusable templates are used to facilitate the synthesis of algebraic data type (ADT) transformations. The reusable templates are meant to lessen the burden of the user in specifying the search space of the programs to be synthesized every time. The recursion skeletons in our framework are effectively (reusable) polymorphic recursion templates. The user can be provided with a library of common recursive datatypes with representation functions mapping between these types, and useful recursion skeletons on these datatypes. SyntRec [13] synthesizes ADT transformations from a functional specification. In contrast, our tool takes this transformation as input (the representation function) and synthesizes a function from ADT to a base type.

Leon [17] , a deductive verification and synthesis framework, can synthesize recursive functions from first-order specifications with recursive predicates. In Section 7, we commented on a comparison of Leon against Synduce.

Higher-Order Recursion Schemes. We use recursion schemes as a model for our programs, but our contribution has very little to do with the original work introducing this model. Higher-order recursion schemes have been introduced for model checking functional programs [18,20,19,29]. Pattern matching recursion schemes, introduced in [25], provide a model for functional programs that manipulate ADTs. We use them as an accurate description of a class of functions on ADTs and the notion of reduction associated with them as a crisp way of formulating symbolic evaluation.

9 Discussion and Future Work

We have demonstrated that partial bounding of quantifiers can be a powerful tool for the synthesis of recursive programs. Circumventing the unnecessary bounding of some quantifiers leads to simpler instances of recursion-free synthesis subtasks that can be handled by the current tools. Moreover, our counterexample generalization also yields simpler terms for bounding the quantifiers that have to be bounded. This is the result of our focus being on a class of recursive functions that perform structural recursion (i.e. recursion that deconstructs its inputs). This, together with our specific problem setup, takes the guesswork out of counterexample generalization and provides the means for a *constructive* counterexample generalization scheme which is demonstrably effective.

The reliance on structural recursion, therefore, limits the class of reference implementations and recursion skeletons that can define an acceptable synthesis instance in our framework. Another limitation tied to the input model is that

the output of the recursive functions has to belong to the base (non-recursive) types to accommodate the reduction of the problem to one that can be solved by a backend solver. Consequently, the unknowns in a target recursion scheme have to all be functions from base types to base types.

In our problem setup, the recursion strategy (given by the recursion skeleton) is an integral part of the specification since it is used to communicate programmer intent. Expecting a *complete* recursion skeleton may be viewed as another limitation of our technique. For example, the `mts` (maximal tail sum) function can be computed as function on a list maintaining only one integer value (i.e. the current value of the maximum tail sum), yet, to implement `mts` in a divide-and-conquer strategy, another computation, the sum of the elements of the list, has to be performed alongside this one. It would be great if the user can ask for a divide-and-conquer recursion strategy without having to know that the additional computation of sum is required as well.

Ideally, the user should be permitted to provide an *incomplete* recursion skeleton which sufficiently communicates the intent and leave the recursion skeleton to be completed automatically by the synthesis procedure. This is a tricky problem. There are not only many recursion strategies to choose from, but each choice also leads to unboundedly many ways to organize the computation on data. This adds yet another dimension of unboundedness to the synthesis problem beyond the two already tackled in this paper. Note that in other recursion synthesis work such as [3,13,27,11], new operations on data are not synthesized, and in contrast drawn from an existing pool of operations. Therefore, this particular problem does not apply in those contexts.

Finally, our method currently does not take into account invariants over recursive data types, e.g. an invariant that specifies that a tree is a binary search tree. Some properties of the datatypes can be encoded through the representation function, e.g. the associativity of the concatenation operator in the category of list parallelization benchmarks. Incorporating the more general invariants in future work will broaden the expressivity of the framework in handling more interesting problems.

References

1. Abrahamsson, O., Myreen, M.O.: Automatically Introducing Tail Recursion in CakeML. In: Trends in Functional Programming. pp. 118–134. Lecture Notes in Computer Science, Springer International Publishing (2018)
2. Ahmad, M.B.S., Cheung, A.: Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In: Proceedings of the 2018 International Conference on Management of Data. SIGMOD '18, ACM (2018)
3. Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive Program Synthesis. In: Computer Aided Verification. pp. 934–950. Lecture Notes in Computer Science, Springer (2013)
4. Alur, R., Bodik, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: 2013 Formal Methods in Computer-Aided Design. pp. 1–8. IEEE (2013)
5. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovi, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Computer Aided Verification. pp. 171–177. Lecture Notes in Computer Science, Springer (2011)
6. Burstall, R.M., Darlington, J.: A Transformation System for Developing Recursive Programs. Journal of the ACM **24**(1), 44–67 (1977)
7. Farzan, A., Nicolet, V.: Synthesis of Divide and Conquer Parallelism for Loops. In: Proceedings of the 38th ACM Conference on Programming Language Design and Implementation. PLDI '17 (2017)
8. Fedyukovich, G., Ahmad, M.B.S., Bodik, R.: Gradual synthesis for static parallelization of single-pass array-processing programs. In: Proceedings of the 38th ACM Conference on Programming Language Design and Implementation. PLDI 2017 (2017)
9. Feldman, Y.M.Y., Padon, O., Immerman, N., Sagiv, M., Shoham, S.: Bounded Quantifier Instantiation for Checking Inductive Invariants. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 76–95. Lecture Notes in Computer Science (2017)
10. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: Proceedings of the 36th ACM Conference on Programming Language Design and Implementation. PLDI '15 (2015)
11. Frankle, J., Osera, P.M., Walker, D., Zdancewic, S.: Example-directed Synthesis: A Type-theoretic Interpretation. In: Proceedings of the 43rd ACM Symposium on Principles of Programming Languages. POPL '16 (2016)
12. Hamilton, G.W., Jones, N.D.: Distillation with labelled transition systems. In: Proceedings of the ACM 2012 Workshop on Partial Evaluation and Program Manipulation. pp. 15–24. PEPM '12, ACM (2012)
13. Inala, J.P., Polikarpova, N., Qiu, X., Lerner, B.S., Solar-Lezama, A.: Synthesis of Recursive ADT Transformations from Reusable Templates. In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science (2017)
14. Itzhaky, S., Singh, R., Solar-Lezama, A., Yessenov, K., Lu, Y., Leiserson, C., Chowdhury, R.: Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In: Proceedings of the 2016 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 145–164. ACM (2016)
15. Katayama, S.: An analytical inductive functional programming system that avoids unintended programs. In: Proceedings of the 2012 Workshop on Partial Evaluation and Program Manipulation. PEPM '12 (2012)

16. Kitzelmann, E., Schmid, U.: Inductive Synthesis of Functional Programs: An Explanation Based Generalization Approach. Journal of Machine Learning Research **7**(15), 429–454 (2006)
17. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: Proceedings of the 2013 International Conference on Object Oriented Programming Systems Languages & Applications. OOPSLA '13 (2013)
18. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: Proceedings of the 36th ACM Symposium on Principles of Programming Languages. POPL '09 (2009)
19. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Proceedings of the 32nd ACM Conference on Programming Language Design and Implementation. pp. 222–233. PLDI '11 (2011)
20. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order multi-parameter tree transducers and recursion schemes for program verification. In: Proceedings of the 37th ACM Symposium on Principles of Programming Languages. POPL '10 (2010)
21. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rmy, D., Vouillon, J.: The OCaml system release 4.11: Documentation and user's manual p. 823
22. Morihata, A., Matsuzaki, K.: Automatic Parallelization of Recursive Functions Using Quantifier Elimination. In: Proceedings of the 10th International Conference on Functional and Logic Programming. FLOPS'10 (2010)
23. Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: The Third Homomorphism Theorem on Trees: Downward & Upward Lead to Divide-and-conquer. In: Proceedings of the 36th ACM Symposium on Principles of Programming Languages. POPL '09 (2009)
24. de Moura, L., Bjrner, N.: Z3: An Efficient SMT Solver. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Lecture Notes in Computer Science, Springer (2008)
25. Ong, C.H.L., Ramsay, S.J.: Verifying higher-order functional programs with pattern-matching algebraic data types. In: Proceedings of the 38th ACM Symposium on Principles of Programming Languages. POPL '11 (2011)
26. Osera, P.M., Zdancewic, S.: Type-and-example-directed Program Synthesis. In: Proceedings of the 36th ACM Conference on Programming Language Design and Implementation. PLDI '15 (2015)
27. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program Synthesis from Polymorphic Refinement Types. In: Proceedings of the 37th ACM Conference on Programming Language Design and Implementation. PLDI '16 (2016)
28. Raghothaman, M., Reynolds, A., Udupa, A.: The SyGuS Language Standard Version 2.0 p. 22
29. Ramsay, S.J., Neatherway, R.P., Ong, C.H.L.: A type-directed abstraction refinement approach to higher-order model checking. In: Proceedings of the 41st ACM Symposium on Principles of Programming Languages. POPL '14 (2014)
30. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.: Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In: Computer Aided Verification. pp. 198–216. Lecture Notes in Computer Science (2015)
31. Solar-Lezama, A., Arnold, G., Tancau, L., Bodik, R., Saraswat, V., Seshia, S.: Sketching stencils. In: Proceedings of the 28th ACM Conference on Programming Language Design and Implementation. PLDI '07 (2007)
32. Solar-Lezama, A., Jones, C.G., Bodik, R.: Sketching concurrent data structures. In: Proceedings of the 29th ACM Conference on Programming Language Design and Implementation. PLDI '08 (2008)

33. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 404–415. ASPLOS XII (2006)
34. Summers, P.D.: A Methodology for LISP Program Construction from Examples. Journal of the ACM **24**(1), 161–175 (1977)
35. Victor, N.: SYNDUCE, https://github.com/victornicolet/Synduce
36. Yang, W., Fedyukovich, G., Gupta, A.: Lemma Synthesis for Automating Induction over Algebraic Data Types. In: Principles and Practice of Constraint Programming. pp. 600–617. Lecture Notes in Computer Science (2019)

A Proofs

In this section we give additional proofs and justification for the results presented in this paper. Note that we always assume that terms and recursion schemes are well-typed. In this section, we also assume that the terms in T have unique variables. That is, a variable in a term of T appears only once in the term, and it appears only in this term. This invariant is guaranteed by the pointwise expansion introducing fresh variables every time it is called.

A.1 Additional Notation

We use some additional notation in the proofs. Throughout this paper, we are specifically interested in the expansions of terms of recursive type. Given a term t, we denote the free variables of t that are of recursive type by $FV_r(t)$. Given a term t and a substitution function $\theta : FV(t) \to T(\Sigma, \mathcal{V})$, we denote (as in the main body of the paper) by θt the term resulting from substituting all occurrences of variables in t by the image in the substitution function. Given a variable $x \in FV(T)$, $\theta(x)$ denotes the image of x by the substitution function θ.

Termination and Structural Recursion Note that in the paper, we assume the symbolic reduction to be terminating on all symbolic inputs. This implies that if a recursion scheme $(\Sigma, \mathcal{N}, \mathcal{R}, \Lambda)$ has a rule $F\ x_1\ \ldots x_m\ p \to t$ in $\mathcal{R}$, and p is of type θ, then there for each constructor κ of the type θ, there is a rule of the form $F\ x_1\ \ldots x_m\ \kappa(\bar{t}) \to t$ where $\bar{t}$ is a tuple of symbolic terms. We also assume that recursion is purely structural: a rule with head non-terminal F that is not pattern-matching cannot have a recursive call to F (an applicative subterm with head F) in the right-hand side of the rule. As a result, any non-terminal symbol used for non-pattern-matching rules always disappears from symbolic reduction results (i.e. the rule can always be applied).

A.2 Proof of Proposition 4

Let us first remind the proposition that identifies why a term u_0 is not maximally reducible:

Proposition 4. *Let $u_0 \in T(\Sigma, \mathcal{V})$ and $g = (\Sigma, \mathcal{N}, \mathcal{R}, \Lambda)$ a recursion scheme. u_0 is not maximally reducible by g iff there exists a subterm of $[\![u_0]\!]_g$ of the form $s = F\ t_1\ \ldots t_n\ x$, where $F \in \mathcal{N}$ and $F \neq \Lambda$, the terms $t_1 \ldots t_n$ are applicative terms, and $x \in FV(u_0)$.*

Proof. Let u_0 be a symbolic term in $T(\Sigma, \mathcal{V})$ that is not a MR-term by g. We assume g has type $\alpha \to D$ where α is some recursive type. By definition of maximally reducible terms, there are applicative terms of the form $F\ t_1 \ldots t_n\ t_{n+1}$ in $[\![u]\!]_g$, otherwise $[\![u]\!]_g$ would be a symbolic term, and therefore u_0 an MR-term.

Let us show that one of these applicative terms $s = F\ t_1 \ldots t_n\ t_{n+1}$ is such that $F \neq \Lambda$ and $t_{n+1} \in FV(u_0)$.

Assume all terms s are such that $F = \Lambda$. Since $\Lambda : \alpha \to D$, there can only be one argument in the applicative term, so $n = 1$ and $s = \Lambda\ t_1$. If t_1 is a variable, then s is not a term that causes u_0 to not be maximally reducible. Since g is a total function, t_1 cannot be a symbolic term: the top symbol would necessarily be a constructor of α, and there would be some rule $\Lambda\ p \to t \in \mathcal{R}$ to reduce the term, which contradicts the assumption that s is a subterm of a reduced term. Therefore, t_1 is an application of type α, using a different non-terminal symbol. We have necessarily a term of the form $s = F\ t_1 \dots t_n\ t_{n+1}$ where $F \neq \Lambda$. Suppose t_{n+1} is not a variable. Assume that t_{n+1} is not a maximally reducible term obeying Prop.4. Then either (1) $t_{n+1} = \Lambda x$ for some x or (2) $t_{n+1} = \kappa(\bar{t})$ for some constructor κ and tuple of terms $\bar{t}$. (1) is impossible, a well-formed PMRS cannot recurse on terms of base type and Λx is of base type. (2) is impossible, this would contradict the fact that the term if fully reduced: there is at least one rule pattern-matching the constructor κ. Therefore, either t_{n+1} is a term obeying Prop.4 (induction) or t_{n+1} is a variable. Since reduction cannot introduce fresh variables, if t_{n+1} is a variable then $t_{n+1} \in FV(u_0)$.

A.3 Properties of Expand and ExpandOnce

First, we define precisely a *local boundary* of a term.

Definition 6. *S is a local boundary of t iff:*

- *S is an antichain,*
- *for every bounded term t_0 such that $t \succeq t_0$, there exists $t' \in S$ such that $t' \succeq t_0$.*
- *for every $t_1, t_2 \in S$, $\{t' | t_1 \succeq t'\} \cap \{t' | t_2 \succeq t'\} = \emptyset$.*

Proposition 6. *The pointwise expansion of t at variable x yields a local boundary of t.*

Proof. Let $C[\ ]$ the context such that $t = C[x]$. The pointwise expansion of t at x is the set of terms $\hat{T_x} = \{C[\kappa_1(\overline{x_1})], \dots, C[\kappa_n(\overline{x_n})]\}$ where $\kappa_1, \dots, \kappa_n$ are the n distinct constructors of the type β of x, and $\overline{x_i}$ are n distinct tuples of distinct variables whose size is the arity of the corresponding constructor, and the type of each variable is the expected type of the constructor argument. We denote by $\hat{x} = \{\kappa_1(\overline{x_1}), \dots, \kappa_n(\overline{x_n})\}$ the expansions of x.

First, note that $\hat{T_x}$ is an antichain: there is no substitution of variables that can substitute an expansion of x for another expansion of x.

Any term in $\hat{T_x}$ is trivially an expansion of t, as the definition name suggests. For each $1 \leq i \leq n$ the substitution function witnessing this is θ_i such that $\theta_i(x) = \kappa(\overline{x_i})$ and $\theta_i(y) = y$ for $y \neq x$. We show that any (well-typed) bounded expansion of t is either a term of $\hat{T_x}$ or an expansion of a term of $\hat{T_x}$. Let a bounded term t' be an expansion of t and $\theta : FV(t) \to T(\sigma, \mathcal{V} \setminus FV(t))$ the substitution function witnessing it. Since t' is a well-typed term, $\theta(x)$ is a term where the root is one of the constructors $\kappa_1, \dots, \kappa_n$ (this is the only possible construction that yields terms of type β). Remark that $\theta(x) \neq x$, since x is

assumed to be of type β which is a recursive type, and therefore x cannot appear in t'. So x is expanded, and there exists $x' \in \hat{x}$ such that $x' \succeq \theta(x)$. Let i be the index such that $\hat{x}_i = \kappa_i(\overline{x_i}) \succeq \theta(x)$, and θ_i be the substitution witnessing this expansion ordering. Let θ' defined by $\theta'(z) = \theta_i(z)$ for $z \in \overline{x_i}$ and $\theta'(y) = \theta(y)$ for $y \notin \overline{x_i}$. Then $\theta'\hat{T_{xi}} = t'$, i.e t' is an expansion of $\hat{t_{xi}}$.

Let $t_1, t_2 \in \hat{T_x}$ $(t_1 \neq t_2)$ and t'_1 some expansion of t_1, t'_2 some expansion of t_2. Since t_1, t_2 are two different terms of $\hat{T_x}$, there is some $\kappa_i(\overline{x_i})$ and $\kappa_j(\overline{x_j})$ $(1 \leq i, j \leq n, i \neq j)$ such that $t_1 = C[\kappa_i(\overline{x_i})]$ and $t_2 = \kappa_j(\overline{x_j})$. So there exists contexts C'_1 and C'_2, and tuples of terms $\overline{s_1}$ and $\overline{s_2}$ such that $t'_1 = C'_1[\kappa_i(\overline{s_1})]$ and $t'_2 = C'_2[\kappa_j(\overline{s_2})]$. Therefore, t'_1 and t'_2 cannot be equal, they differ at least in the subterm rooted at x in the original term t.

Another small lemma guarantees that expanding a term in a local boundary preserves the local boundary:

Lemma 1. *Let t be a symbolic term. Assume S is a local boundary of t, and let S' be a local boundary of some $t_0 \in S$. Then $(S \setminus t_0) \cup S'$ is a local boundary of t.*

Proof. The proof follows from the transitivity of $\succeq$. Let t_b be a bounded term s.t. $t \succeq t_b$. Since S is a local boundary of t, there is some t' in S s.t. $t' \succeq t_b$. If $t' \neq t_0$, then t' is already in $(S \setminus t_0) \cup S'$, so t_b is an expansion of a term in that new boundary. If $t' = t_0$, by definition of S' the local boundary of t_0 (and $t_0 \succeq t_b$) there is a term t_s in S' such that $t_s \succeq t_b$. So $(S \setminus t_0) \cup S'$ is a local boundary of t. Since $S' \subseteq \{t' | t_0 \succeq t\}$ and S' is a boundary, the third condition of the local boundary is also satisfied.

A.4 Proof of Proposition 5

Proposition 5. *Let t be some symbolic term and T', U' be the results of the call to Expand(t). Then $T' \cup U'$ is a boundary of the set $\{t' | t \succeq t'\}$.*

Proof. The proof follows directly from Proposition 6 and Lemma 1. Let u_0 be an unbounded term and $T', U' = Expand(u_0)$. We want to prove that $T' \cup U'$ is a local boundary of u_0. In the Expand algorithm, at the start of the loop, $T' \cup U' = \{u_0\}$ is trivially a local boundary of u_0. Lemma 1 and Prop. 6 guarantee that at any step of the loop in Expand, the invariant that $T' \cup U'$ is a boundary of u_0 is maintained.

Proposition 5 guarantees, by construction of T and U from Expand, that $T \cup U$ is a boundary during all rounds of the algorithm. We can also state that $T \cup U$ is a set of terms such that the expansions of any two terms in the set do not intersect (a consequence of the third condition of Definition 6).

A.5 Parsimony

Let us define what it means for a term to be *parsimonious*.

Definition 7. *A term t is parsimonious iff there is no $t_0 \succeq t$ that is a MR-term.*

Intuitively, when a term is parsimonious, there is no simpler term that could be used to construct a well-formed set of constraints. With Prop. 4 and Proposition 5, we can prove the following proposition.

Proposition 7. *Let u_c be a parsimonious term picked from U at some step of the algorithm, and $T', U' = \mathsf{Expand}(u_c)$. Then $T' \cup U'$ is a set of parsimonious terms.*

Proof. Let u_c be a generalization of a counterexample from the algorithm and T, U the boundary at the step of the algorithm where u_c is selected ($u_C \in U$). Note that u_c is not a MR-term. Let $T', U' = \mathsf{Expand}(u_c)$. We assume that u_c is parsimonious. At each step of the Expand algorithm, new terms are obtained using the ExpandOnce subroutine. By Propositions 6 and 5, we have the invariant that in Expand, $T' \cup U'$ is a local boundary (Definition 6) of u_c, and until the loop exits $T' = \emptyset$. The invariant that there is no MR-term t_0 such that $t_0 \succeq t$ for some term in U is also maintained; if there was one, it must be a term u such that $u_c \succeq u$, which is necessarily the result of some ExpandOnce call, and this term would have been placed in T'. Therefore, all the u_0 terms picked by the loop are not MR-terms.

Now suppose that after n iterations of the loop in Expand, we obtain a non-empty set of MR-terms T. Let $t_0 \in T$.

First, note that if there is a MR-term t_0' such that $t_0' \succeq t_0$, then necessarily $u_C \succeq t_0'$. This is because (1) we assumed that there is no MR-term t_0' s.t. $t_0' \succeq u_C$ and (2) u_C is picked from the boundary T, U in the algorithm, and there is no term t' such that $t' \succeq t_0$ but t' is not an expansion of u_C or $t' \succeq u_C$. If t_0' and t_0 and both are MR-terms, there is at least one variable x in $FV_r(t_0')$ that has been replaced by a term t_x in t_0. That is, if θ is the substitution function such that $\theta t_0' = t_0$, then $\theta(x) = t_x$ where $t_x \neq x$. We can prove that no such x should exist in t_0' if it is maximally reducible. It x has been expanded in t_0, then it should be expanded in t_0' as well.

Definition 8 (Universe of boundaries). *The universe of boundaries $\mathcal{B}$ is the set of all pairs of sets T, U such that $T \cup U$ is a boundary.*

Definition 9. *A refinement round of the algorithm is uniquely identified by the boundary $T, U \in \mathcal{B}$, the counterexample x_C returned by* Verify*. We denote by (T, U, x_C) such a refinement round.*

The following proposition states that our algorithm is parsimonious: when a term is added to T, there is no other terms that is less expanded that could have been added to T.

Theorem 2. *[Parsimony] Let us assume (T, U) is a boundary that our algorithm reaches in some round, then (T, U) is optimal in the following two senses:*

- *for every $t \in T \cup U$ there is no MR-term t' such that $t' \succeq t$.*

– *there is no non-empty subset* T' *of* T *and set* U' *such that* $(T \setminus T') \cup U'$ *is a boundary and* $\mathcal{E}(T \setminus T') \precsim \mathcal{E}(T)$.

This theorem states that at any round of our algorithm, (1) the terms in T cannot be replaced by any less expanded term, and (2) removing any subset of terms in T that preserves the boundary invariant does not preserve the set of solutions of the approximation.

Proof. Let us assume that $(T, U) \in \mathcal{B}$ is a boundary at some round of the refinement loop. The algorithm maintains at any round that for any term $t \in T \cup U$, there is no MR-term t' such that $t' \succeq t$. The proof follows directly from induction on the steps of the algorithm, and Proposition 7 proves the induction step holds (the induction hypothesis being that u_c is a parsimonious term).

Note that if $T, U \in \mathcal{B}$, by definition of a boundary, no term in t_0 in T or U can be removed such that $T \cup U \setminus \{t_0\}$ is still a boundary. If a term is removed from the boundary, then the guarantee that all terms are either expansions or expanded from a term in the boundary is not preserved.

Now suppose that there is a set of MR-terms $T' \subseteq T$, $T' \neq \emptyset$ and a set of non maximally reducible terms U' such that $(T \setminus T') \cup U'$ is a boundary. Each term in T has been introduced at some round of the algorithm. Let t' be some term in $T'(\subset T)$. Let $r = (T_r, U_r, x_c)$ the round at which t' has been introduced, and $T'_r, U'_r = \mathsf{Expand}(u_c)$ where $u_c \in U$ is the generalization of x_c, i.e. $u_c \succeq x_c$. By definition, $t' \in T'_r$. We know through Proposition 6 that T'_r is a local boundary of u_C(Definition 6). If t' is removed in T'_r, $T'_r \cup U'_r$ is not a local boundary anymore. At the level of T, the reparation implied by having removed t' from T requires removing T'_r within T and U'_r within U, and adding u_c to U. A consequence of the necessary reparation of the boundary is that the counterexample x_c is not an expansion of some term in $T \setminus T'$. Additionally, any set of terms that have been added by expanding terms in U'_r also need to be removed. As a result the spurious solutions of round r and the rounds made possible by the refinement happening at round r are spurious solutions of $\mathcal{E}(T \setminus T')$. Therefore, $\mathcal{E}(T \setminus T')$ has solutions that are not solutions of $\mathcal{E}(T)$.

B Extras

B.1 Case Studies

In this section we provide more information about the different categories of benchmarks chosen in this paper. We give the recursive types, the representation functions and the recursion skeletons that are common to every benchmark in each of the categories.

Note that in a given category there might be some variation in the definitions. Because the function need to be total, the recursive types need to represent all possible inputs of the function. For example, a traditional definition of lists is the recursive type `a list =Nil | Cons of a * a list`, but one might need to use the recursive type defining lists with at least one element, which can be

defined by `a list =Elt of a | Cons of a * a list`. For each category we give the definitions with empty as base case.

Tail Recursive. In this set of benchmarks the input function is a non tail-recursive function on lists. The type of lists is `a list =Nil | Cons of a * a list`. The input function is a function of the form

```
let rec spec =
    function
    | Nil -> baseCase
    | Cons(hd, tl) -> accum hd (spec tl)
```

with a given definition for `baseCase` and `accum`. The recursion skeleton is the following function:

```
let rec rskel = function x -> aux baseCaseTr x
and aux s =
    function
    | Nil -> s
    | Cons(hd, tl) -> rksel (accumTr s hd) tl
```

where `baseCaseTr` and `accumTr` are the unknowns to be synthesized. In this recursion skeleton, all recursive calls (to `aux`) are in tail position. However, the transformation from the specification is not trivial.

Combine. The benchmarks in combine use both tail-recursive and non-tail recursive implementations and combine them into a single implementation.

From Trees to Paths. The trees considered in this set of benchmarks is the set of labelled binary trees, defined by the recursive type `a btree =Empty | Node of a * a btree * a btree`. The paths for this particular type of trees is `c path =Top | Zip of sel * c * c btree * c path` where sel is a simple sum type `sel =Left | Right`. The representation function from (binary tree) paths to binary trees is:

```
let rec repr =
    function
    | Top -> Empty
    | Zip(w, val, child, z) -> h val child z w
and aux h val child z =
    function
    | Left -> Node(val, child, repr z)
    | Right -> Node(val, repr z, child)
```

The target recursion scheme for this set of benchmarks is the following function with unknowns `s0` and `joinl, joinr`:

```
let rec rskel =
    function
    | Top -> s0
    | Zip(Left, a, child, z) -> joinl (spec child) (rskel z)
    | Zip(Right, a, child, z) -> joinr (spec child) (rskel z)
```

Tree Flattening. A plane tree is a tree where each node contains a list of children `a ptree =PNil | PNode of a * (a ptree)list`. The following representation function translates plane trees to labelled binary trees:

```
let rec repr =
    function
    | PNil -> Empty
    | PNode(a, l) -> Node(a, Empty, f l)
and f =
    function
    | Nil -> Empty
    | Cons(hd, tl) -> Node(0, repr hd, f tl)
```

In this example a default value 0 is placed in some nodes in order to encode the geometry of the original tree. In the instance where no appropriate value exists, one can use an option type and place some elements in the leaves, and none in the geometry encoding nodes. The recursion skeleton requires synthesizing two unknown functions `j1, j2` and two base case values `s1, s2`. It is specified as follows:

```
let rec rskel =
    function
    | PNil -> s1
    | PNode(a, children) -> j1 a (aux children)
and aux l =
    function
    | Nil -> s2
    | Cons(hd, tl) -> j2 (rskel hd) (aux tl)
```

There are variations of this recursion skeleton where an accumulator can be used instead of this direct-style recursion.

Change Tree Traversal. Some functions on trees are more easily written with a specific traversal. For example, computing the maximal in-order prefix sum of a tree can be trivially written with an in-prefix traversal of the tree, and similarly a post-order prefix sum can be computed with a post-order traversal. Our tool can automatically synthesize the equivalent implementation corresponding to the divide-and-conquer tree homomorphism.

Parallelizing Functions on Lists. Parallelizing a single-pass function on lists amounts to transforming a function on lists to a homomorphism on lists built with the concatenation operator. These lists can be specified with the recursive type `a clist =CNil | Single of a | Concat of a clist * a clist`. That is, a list is either empty, a single element or a concatenation of two lists. The represent function translate concatenation lists into traditional lists:

```
let rec repr  =
    function
    | CNil -> Nil
    | Single(a) -> Cons(a, Nil)
```

```
    | Concat(x, y) -> dec y x
and dec l1 =
    function
    | CNil -> repr l1
    | Single(a) -> Cons(a, repr l1)
    | Concat(x, y) -> dec (Concat(l1, y)) x
```

An interesting feature of this formulation is that the *associativity* of the concatenation operator is encoded by the representation function. The concatenation lists `Concat(Single(a), Concat(Single(b), Single(c)))` and `Concat(Concat(Single(a), Single(b)), Single(c))` will be translated to the same list `Cons(a,Cons(b, Cons(c, Nil)))`. When solving the problem, the solver has no explicit knowledge of this property, yet the synthesized result will reflect the associativity of the concatenation operator.

The target recursion skeleton is a list homomorphism with unknowns `s0`, `f0` and `join`:

```
let rec hom =
    function
    | CNil -> s0
    | Single a -> f0 a
    | Concat(x, y) -> join (hom x) (hom y)
```

Note that the calls `hom x` and `hom y` can be performed in parallel.

Lists to Trees* This set of benchmarks does not appear in the main body of the paper. Table 2 lists three benchmarks that we added to the set to illustrate an example where the same solution is synthesized for three different represent functions. In this category of benchmarks, the reference function is a function from lists to scalar type. For example, one can write the following function for a given `init` constant and `accum` function:

```
let rec spec =
    function
    | Nil -> init
    | Cons(hd, tl) -> accum hd (spec tl)
```

The recursion skeleton that needs to be synthesized is a homomorphic on labelled binary trees:

```
let rec treehom =
    function
    | Empty -> s0
    | Node (a, l, r) -> join a (treehom l) (treehom r)
```

The user has a choice of how to implement the representation function that flattens a labelled binary tree into a list. One can use a pre-order, in-order or post-order traversal to do so. For example, the following is a valid represent function:

```
let rec repr =
    function
```

```
    | Empty -> Nil
    | Node(a,l,r) -> dec Cons(a, repr r) l
and dec li =
    function
    | Empty -> li
    | Node(a, ll, lr) -> dec Cons(a, dec li lr) ll
```

Remark that the traversal used by the representation function does not matter if the function to be synthesized is permutation invariant (the accumulator is commutative). However, the choice of the representation function has an impact on whether Synduce will be able to leave some terms unbounded. The benchmark *search (v3)* requires one fewer iteration of the refinement loop than the other benchmarks because Synduce is able to leave one branch of the input trees unbounded. The representation function, in these cases, is an in-order traversal. Remark also that the representation function choice has no impact on the number of refinement steps taken by the symbolic CEGIS algorithm.

B.2 Evaluation of the Concrete CEGIS Algorithm

In the paper, we compare our approach with a baseline implementation. This baseline implementation is a symbolic counterexample guided inductive synthesis algorithm.

We compare our two previous implementations to another solution which can be described as a *concrete* CEGIS algorithm, in contrast to the *symbolic* CEGIS algorithm used as the naive implementation. Table 2 summarizes the experimental results for the same set of benchmarks as the main body of the paper. The *sum of even elements* benchmark is an example where concrete CEGIS is behaving particularly badly. The counterexamples sequence effectively explores only the unbounded integer domain of the problem, but never the unbounded recursive dimension. Each program generated is a case distinction on the possible valuation of lists of size at most two.

It also illustrates the difficulty of expressing such problems with input/output examples. The *mps* example, depending on the target recursion scheme, requires 7 to more than 15 input/output counterexamples.

Verification Time In Table 2, we give for each implementation the percentage of the total running time spent verifying solutions. This number can give an idea of the syntactic complexity of the solutions; when more time is spent in the verification oracle than the synthesis oracle, it is a good indication that the solution is relatively simple. The synthesis solver finds a solution in little time, and in comparison the verification takes more time. Remark that despite the simplicity of the recursive types, the number of symbolic terms of a given depth is large. We use the partial bounding of the terms in Synduce during the verification, effectively cutting the size of the set of constraints that needs to be verified.

Table 2. Extended Experimental Results. Three algorithm are compared: the selective bounding CEGIS in SYNDUCE, symbolic CEGIS and concrete CEGIS. Benchmarks are grouped by categories introduced in Section 7.1. # indicates the number of refinement rounds. ver.% indicates the percentage of total time spent verifying solutions. T_{last} is the elapsed time before the last call to the SyGuS solver in the last refinement step before timeout ('.' indicates that there was no previous round). All times are in seconds. A '-' indicates timeout(> 10 min). Experiments are run on a laptop with 16G memory and an i7-8750H 6-core CPU at 2.20GHz running Ubuntu 19.10.

Benchmark	SYNDUCE			Symbolic CEGIS			Concrete CEGIS		
	time	ver. %	#	time	ver. %	#	time	ver. %	#
sum	**0.03**	*47.7*	2	0.04	*54.0*	3	0.05	*56.6*	4
max	**0.33**	*99.6*	1	0.34	*93.1*	2	1.41	*23.5*	4
max 2	**0.25**	*99.5*	1	0.34	*93.7*	2	2.82	*11.0*	6
min	**0.23**	*99.4*	1	0.33	*92.7*	2	6.31	*5.5*	7
min-max	**0.85**	*64.6*	3	73.16	*0.8*	3	-	*∼0.*	12
max weighted path	0.09	*61.7*	3	**0.07**	*60.4*	3	0.09	*64.4*	5
sorted in-order	**0.01**	*88.3*	1	43.97	*0.1*	4	88.02	*0.1*	9
pre-order poly.	**16.09**	*0.1*	2	-	*∼0.*	4	-	*∼0.*	5
`mips`	**0.29**	*12.4*	2	-	*∼0.*	4	-	*∼0.*	8
in-order mts	**0.41**	*8.0*	2	-	*∼0.*	4	-	*∼0.*	8
post-order mps	**132.14**	*0.9*	4	-	*∼0.*	6	-	*∼0.*	13
sum	0.07	*72.5*	2	**0.06**	*69.7*	3	2.77	*99.0*	5
height	**0.90**	*8.9*	1	1.24	*4.1*	5	3.52	*48.7*	5
max weighted path	0.15	*74.6*	2	**0.12**	*73.4*	3	0.14	*73.1*	6
max w. path (hom)	**0.01**	*74.8*	1	1.69	*6.0*	5	2.00	*5.1*	11
leftmost odd	**0.01**	*76.3*	1	-	*∼0.*	4	-	*∼0.*	4
`mips`	**1.57**	*25.1*	3	-	*∼0.*	7	-	*0.2*	15
sum	**0.02**	*48.5*	2	0.03	*47.8*	3	0.03	*52.4*	3
mts	**5.88**	*0.4*	2	575.30	*∼0.*	3	-	*∼0.*	11
mps	**1.69**	*1.3*	2	4.82	*0.4*	3	9.47	*0.5*	7
mts + sum	**9.95**	*0.2*	2	112.57	*∼0.*	3	19.77	*0.2*	6
sum + mts + mps	**0.26**	*18.3*	3	-	*∼0.*	3	-	*∼0.*	7
sum	**0.07**	*25.4*	3	0.08	*14.6*	2	0.07	*16.1*	2
product	**0.07**	*16.5*	2	0.17	*6.8*	2	0.15	*11.4*	3
max of heads	0.21	*16.4*	2	**0.18**	*18.4*	3	0.42	*7.8*	5
max of lasts	0.21	*13.7*	2	0.33	*6.9*	3	**0.13**	*19.8*	4
max sibling sum	5.26	*0.3*	2	**2.72**	*1.0*	3	31.11	*0.1*	6
sum	**0.08**	*83.6*	1	0.31	*93.5*	3	0.31	*89.3*	3
sum of even elts.	**0.10**	*70.5*	1	0.39	*89.9*	3	-	*0.5*	170
length	**0.07**	*87.7*	1	0.22	*90.2*	4	-	*∼0.*	3
last	**0.01**	*50.0*	1	0.03	*40.7*	2	0.04	*37.2*	2
product	**0.07**	*86.6*	1	0.31	*93.3*	3	0.31	*88.8*	3
polynomial	**0.07**	*43.2*	1	0.71	*86.1*	5	0.71	*87.4*	5
hamming	**0.10**	*77.7*	1	0.46	*92.8*	3	53.50	*0.9*	6
min	**0.02**	*62.9*	1	0.08	*72.8*	2	0.14	*50.9*	3
is sorted	3.45	*0.5*	2	**3.20**	*1.2*	4	6.01	*0.8*	6
linear search	**0.08**	*81.4*	1	0.35	*93.8*	3	0.44	*83.7*	2
line of sight	**0.86**	*9.6*	2	213.41	*0.1*	7	-	*∼0.*	8
mts	**0.10**	*68.1*	1	4.80	*98.4*	4	103.48	*5.3*	11
mps	**0.09**	*71.6*	1	4.73	*98.5*	4	259.12	*2.3*	11
mts and mps combined	**0.38**	*42.0*	2	210.84	*4.8*	6	-	*∼0.*	9
mss	**4.82**	*12.5*	3	-	*∼0.*	6	-	*∼0.*	7
count max elements	**138.20**	*∼0.*	1	-	*∼0.*	3	-	*∼0.*	7
search	**5.53**	*0.6*	3	8.99	*0.3*	4	6.93	*0.4*	4
search (v2)	**5.53**	*0.6*	3	6.94	*0.4*	4	8.97	*0.3*	4
search (v3)	**6.13**	*0.2*	2	6.93	*0.4*	4	6.86	*0.4*	4

B.3 Optimizations for Synthesizing Solutions of Recursion-Free Systems of Equations

In order to make finding a solution easier, we apply the following optimizations to the systems of equations, independently of the context in which the system of equations has been generated.

Syntactic Definitions. The system of equations might contain equations that are syntactic definitions. Those are the equations of the form $\xi\ x_1 \dots\ x_n = t$ where ξ is an unknown to be synthesized, $x_1, \dots, x_n$ are n *distinct* variables ($n \geq 0$) and t is some symbolic term. One can easily notice that this is an equation that defines the function ξ unequivocally as $\xi(x_1, \dots, x_n) = t$. If a syntactic definition for ξ is found, the unknown ξ is replaced by its definition in the rest of the equations.

Separating Tuple Components. When the output type of an unknown is a tuple, each component of the tuple can be separated. We transform an unknown $\xi : A \to D_1 \times \ldots \times D_n$ to a set of unknowns $\xi_i : A \to D_i$ for each $1 \leq i \leq n$. An equation between tuples is also translated to a set of equations between the components. This optimization significantly reduces the complexity of the synthesis task when used in combination with the next optimization.

Independent Subsystems. Smaller equations systems with fewer unknowns are easier to solve. If a subset of unknowns $X \subset \Xi$ appears only in a subset of equations $E' \subset E$, and no other unknown appears in E', then the equation system can be split into the equations E' with unknowns X, and the equations $E \setminus E'$ with unknowns $\Xi \setminus X$. A system of equations can be split recursively into a set of subsystems that can be solved independently.

Initialization A better initialization strategy could save some early calls that are bound to be unsuccessful. If an equation system does not constrain one of the unknowns, then there is very little chance that an arbitrary solution for this unknown will be correct.

Ideally, an initial set of terms S should guarantee that each unknown appears at least in one equation of the approximation. For a given recursion skeleton $\mathcal{S}[\Xi]$, such a set S can be produced by expanding a variable of type θ into a set of terms, and then recursively expand those terms until the set satisfies the following condition: for every unknown ξ, there is a term $t \in S$ such that ξ appears in $\mathcal{S}[\Xi](t)$.

S as described above may contain terms that are not maximally reducible. Our algorithm initializes T, U by calling Expand to expand every term in S to at least one maximally reducible term.

Proposition 8. *Let S be the set described above, and T a set of terms such that for all $t_0 \in S$ there exists a term t in T that is an expansion of t_0 (i.e $t_0 \succeq t$). Then for each $\xi \in \Xi$ there is at least one equation in $\mathcal{E}(T)$ that constrains ξ.*

By starting from such a set S and expanding the terms in it, it is guaranteed that at any step of the algorithm, the approximation contains at least one equation for each unknown.

Evaluation of the optimizations In Table 3,we evaluate the different optimizations with the symbolic CEGIS algorithm, and the selective bounding algorithm implemented in SYNDUCE. The optimizations proposed improve the running times of the algorithm overall.

- *Initialization* The column #i indicates the number of rounds required without the optimized initialization, and the column `-ini` the synthesis time. Note that this optimization is not applicable to the symbolic CEGIS algorithm, since is it tied to searching terms using and the Expand function which performs partial bounding. In general, the synthesis time is marginally improved by this optimization. Using the optimization saves at least one round of refinement.
- *Separating Tuple Components and Independent Subsystems* The column `-sys` indicate when the optimization is tuned off. The use of the separating tuple components and splitting into subsystems is, in some examples, making running time worse. However, there is no instance where a benchmark would time out because the optimization is turned on. Without the optimization, 9 out 46 benchmarks time out in SYNDUCE, and 2 out of 37 passing benchmark time out in symbolic CEGIS.
- *Syntactic Definitions* The columns `- stx` indicate the synthesis times when the optimization is turned off. Without the optimization, 5 benchmarks out of 46 time out in SYNDUCE, and 10 out of 37 passing benchmarks time out in symbolic CEGIS.

The last column of each algorithm reports the synthesis time when all optimizations are turned off. In SYNDUCE, this results in 18 benchmarks out of 46 to time out. In the symbolic CEGIS algorithm, 10 out of 37 passing benchmarks time out. We did not evaluate the effect of using the optimizations in the concrete CEGIS algorithm. The only optimization that is applicable is the tuple component separation and independent subsystems. There is no syntactic definitions in the concrete algorithm, since all constraints are expressed over concrete inputs.

Table 3. Optimization Evaluation Results: We evaluate SYNDUCE and symbolic CEGIS with some optimizations turned off. `- ini` indicates that optimized initialized is off, `- sys` indicates tuple components and independent subsystems is turned off, and `stx` that syntactic definitions are turned off. `on` (and `off`) indicate that all optimizations are turned on (resp. off). Benchmarks are grouped by categories introduced in Section 7.1. # indicates the number of refinement rounds for the given algorithm, and #i indicate the number of refinement rounds without the optimized initialization. All times are in seconds. A '-' indicates timeout(> 10 min). Experiments are run on a laptop with 16G memory and an i7-8750H 6-core CPU at 2.20GHz running Ubuntu 19.10.

Benchmark	SYNDUCE							Symbolic CEGIS				
	#	on	#i	-ini	-sys	-stx	off	#	on	-sys	-stx	off
sum	2	**0.03**	3	0.04	0.03	0.05	15.07	3	**0.04**	0.04	3.76	3.80
max	1	0.33	2	0.35	0.33	**0.30**	0.54	2	**0.34**	0.34	1.23	1.22
max 2	1	**0.25**	2	0.25	0.25	0.30	0.36	2	**0.34**	0.34	1.37	1.36
min	1	**0.23**	2	0.24	0.23	0.29	0.54	2	0.33	**0.32**	2.06	2.07
min-max	3	**0.85**	4	0.91	-	0.87	15.96	3	**73.16**	-	-	-
max weighted path	3	**0.09**	3	0.09	0.09	0.11	1.01	3	**0.07**	0.07	28.22	28.69
sorted in-order	1	**0.01**	1	0.01	0.01	42.14	41.52	4	**43.97**	44.10	44.04	43.98
pre-order poly.	2	16.09	3	15.98	-	**16.03**	-	4	**-**	-	-	-
`mips`	2	**0.29**	3	0.31	-	0.33	-	4	**-**	-	-	-
in-order mts	2	**0.41**	3	0.43	-	0.48	-	4	**-**	-	-	-
post-order mps	4	132.14	5	**131.49**	-	133.88	-	6	**-**	-	-	-
sum	2	**0.07**	3	0.08	0.09	0.11	1.16	3	**0.06**	0.06	54.25	53.75
height	1	**0.90**	2	0.95	58.06	0.94	58.19	5	**1.24**	18.04	1.26	18.39
max weighted path	2	0.15	3	**0.14**	0.20	0.20	0.20	3	**0.12**	0.12	0.12	0.12
max w. path (hom)	1	**0.01**	4	0.22	0.01	0.03	0.55	5	1.69	1.42	1.71	**1.42**
leftmost odd	1	**0.01**	2	-	0.01	-	-	4	**-**	-	-	-
`mips`	3	**1.57**	4	1.59	-	2.17	-	7	**-**	-	-	-
sum	2	**0.02**	3	0.03	0.03	0.48	0.48	3	**0.03**	0.03	0.37	0.45
mts	2	5.88	3	**5.84**	-	5.93	-	3	575.30	115.58	565.67	**118.10**
mps	2	1.69	3	1.65	**0.31**	-	-	3	4.82	**0.34**	-	-
mts + sum	2	9.95	3	10.07	**4.72**	-	-	3	112.57	**5.42**	-	-
sum + mts + mps	3	**0.26**	4	0.28	-	0.32	-	3	**-**	-	-	-
sum	3	**0.07**	2	0.08	0.07	0.07	0.07	2	0.08	0.07	0.08	**0.07**
product	2	**0.07**	3	0.34	0.07	0.07	0.07	2	0.17	0.16	0.17	**0.16**
max of heads	2	0.21	3	**0.12**	0.23	0.47	0.49	3	**0.18**	0.20	0.45	0.46
max of lasts	2	**0.21**	3	0.27	0.23	0.71	0.68	3	**0.33**	0.34	0.54	0.50
max sibling sum	2	**5.26**	3	5.31	5.46	39.42	37.64	3	**2.72**	2.84	18.63	19.22
sum	1	0.08	2	0.08	**0.07**	0.08	0.41	3	0.31	**0.30**	2.53	2.84
sum of even elts.	1	**0.10**	2	0.11	0.10	-	-	3	**0.39**	0.39	-	-
length	1	**0.07**	2	0.08	0.07	0.08	1.63	4	**0.22**	0.23	-	-
last	1	**0.01**	2	0.03	0.01	0.02	0.02	2	**0.03**	0.03	0.04	0.03
product	1	**0.07**	2	0.08	0.07	0.09	0.19	3	**0.31**	0.31	1.68	1.68
polynomial	1	**0.07**	2	0.08	7.71	0.08	-	5	**0.71**	14.17	-	-
hamming	1	**0.10**	2	0.11	0.10	3.12	-	3	**0.46**	0.46	-	-
min	1	**0.02**	2	0.03	0.02	0.03	0.04	2	**0.08**	0.11	0.15	0.17
is sorted	2	3.45	3	3.34	3.41	**3.44**	3.46	4	3.20	3.12	3.13	**3.12**
linear search	1	**0.08**	2	0.09	0.08	-	-	3	**0.35**	0.36	-	-
line of sight	2	0.86	3	0.90	6.93	**0.84**	4.12	7	213.41	**7.67**	-	-
mts	1	**0.10**	2	0.11	1.51	0.12	36.35	4	**4.80**	6.18	-	-
mps	1	**0.09**	2	0.11	1.36	0.11	-	4	**4.73**	6.04	-	-
mts and mps combined	2	**0.38**	3	0.40	-	0.44	-	6	**210.84**	-	-	-
mss	3	**4.82**	4	4.87	-	5.05	-	6	**-**	-	-	-
count max elements	1	138.20	2	**136.71**	-	139.43	-	3	**-**	-	-	-
search	3	5.53	4	5.53	5.54	5.53	**4.14**	4	**8.99**	9.01	12.47	12.43
search (v2)	3	5.53	4	5.53	5.52	5.54	**4.15**	4	6.94	**6.91**	9.05	9.21
search (v3)	2	**6.13**	3	6.18	6.19	6.21	6.61	4	**6.93**	6.93	9.04	9.02