Recursion Synthesis with Unrealizability Witnesses

Azadeh Farzan
Department of Computer Science
University of Toronto
Toronto, Canada
azadeh@cs.toronto.edu

Danya Lette
Department of Computer Science
University of Toronto
Toronto, Canada
danya@cs.toronto.edu

Victor Nicolet
Department of Computer Science
University of Toronto
Toronto, Canada
victorn@cs.toronto.edu

Abstract

We propose SE^{2}GIS, a novel inductive recursion synthesis approach with the ability to both synthesize code and declare a problem unsolvable. SE^{2}GIS combines a symbolic variant of counterexample-guided inductive synthesis (CEGIS) with a new dual inductive procedure, which focuses on proving a synthesis problem unsolvable rather than finding a solution for it. A vital component of this procedure is a new algorithm that produces a witness, a set of concrete assignments to relevant variables, as a proof that the synthesis instance is not solvable. Witnesses in the dual inductive procedure play the same role that solutions do in classic CEGIS; that is, they ensure progress. Given a reference function, invariants on the input recursive data types, and a target family of recursive functions, SE^{2}GIS synthesizes an implementation in this family that is equivalent to the reference implementation, or declares the problem unsolvable and produces a witness for it. We demonstrate that SE^{2}GIS is effective in both cases; that is, for interesting data types with complex invariants, it can synthesize non-trivial recursive functions or output witnesses that contain useful feedback for the user.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; **Automatic programming**; *Software verification*; • **Theory of computation** → **Invariants**; *Program specifications*; *Abstraction*; *Program schemes.*

Keywords: Program Synthesis, Functional Programming, Invariants, Unrealizability, Recursion, Abstraction

ACM Reference Format:
Azadeh Farzan, Danya Lette, and Victor Nicolet. 2022. Recursion Synthesis with Unrealizability Witnesses. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523726

PLDI '22, June 13–17, 2022, San Diego, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523726

1 Introduction

Recursive program synthesis has received a lot of attention in recent years [1, 10–12, 27, 32, 34]. The specific setup of these synthesis problems and their solution strategies vary greatly.

In this paper, we address the problem of synthesizing a recursive function whose behaviour is equivalent to a given (reference) implementation. We assume that the programmer has access to such an implementation of $f : \tau \longrightarrow D$ on a recursive data type τ, and now wishes to have an equivalent implementation $g : \theta \longrightarrow D$ on a new recursive data type θ. This can be motivated, for example, by the fact that a more efficient computation can be performed on θ. We propose a synthesis algorithm that can automatically synthesize g so that the programmer need not implement it from scratch. More precisely, the recursion synthesis problem is defined by the following components:

- a *reference function* $f : \tau \longrightarrow D$,
- a *representation function* $r : \theta \longrightarrow \tau$ that maps objects of type θ to objects of type τ,
- a *type invariant* $I_\tau : \tau \longrightarrow Bool$ for τ, and
- a *type invariant* $I_\theta : \theta \longrightarrow Bool$ for θ.

The following specification then defines the synthesis problem for a family of recursive functions $\mathcal{G}$:

$$\exists g \in \mathcal{G}\ \forall x : \theta\ \cdot\ I_\theta(x) \wedge I_\tau(r(x)) \Rightarrow g(x) = (f \circ r)(x) \quad (1)$$

Intuitively, $\mathcal{G}$ is used to communicate the specific recursive solution intended by the user; more precisely, it encodes high level stipulations such as traversal strategies and time complexity budgets[1]. The goal is to either *synthesize a solution* for this specification, or *produce a witness for its unrealizability.* A witness is useful feedback for the user on why the problem cannot be solved, and can be used to root cause the unrealizability of the synthesis goal. This gives our synthesis technique the means to have a meaningful interaction with the user in revising problematic specifications. Most synthesis techniques focus on solutions exclusively, although recently there has been some interest [15, 16, 23] in addressing the unrealizability problem for synthesis. These efforts, however, have been limited to non-recursive code.

Dependable solver support for synthesis is only available for a limited family of non-recursive base types. We present a novel inductive synthesis algorithm for solving

[1] In Section 3, this notion is fully formalized, but the details are not required for the high level exposition here.

the recursion synthesis problem that uses existing (standard) solvers for non-recursive functions at its core. In the spirit of counterexample-guided inductive synthesis (CEGIS)[37], our technique solves the recursion synthesis problem by iterating through a series of *non-recursive approximations* of the original specification (Equation 1). In sharp contrast to CEGIS, these non-recursive approximations are not strict *under-approximations* of Equation 1. This requires a paradigm shift in the mechanisms used for revising the approximate specifications.

1.1 Partial Bounding

As a very simple example, let us assume that τ and θ are both of type *List* (non-empty cons-lists). In addition, there is a type invariant on θ asserting that lists are sorted in increasing order. The representation function r is simply the identity function. Consider a reference function $min : List \rightarrow Int$ that computes the smallest element of a non-empty list. The goal is to synthesize a function $min_s : List \rightarrow Int$ that computes the smallest element of a *sorted* list in *constant* time. $\downarrow$ is shorthand for a binary operator that returns the minimum of its two operands. One can immediately observe that the implementation of *min* (for general lists) is already a valid solution for min_s. This is precisely why the user needs to express their intent for a constant time solution using the recursion skeleton for min_s illustrated here. It is parametric on two unknown functions b_1 and b_2, but only admits constant-time solutions due to the lack of any recursive calls on the list.

$List = Elt(A) \mid Cons(A, List)$
$min(Elt(a)) \rightarrow a$ $min(Cons(a,\ell)) \rightarrow a \downarrow min(\ell)$
$min_s(Elt(a)) \rightarrow b_1(a)$ $min_s(Cons(a,\ell)) \rightarrow b_2(a)$

The unknown functions b_1 and b_2 are constrained by the following specification (an instantiation of Equation 1):

$$\exists b_1, b_2 \cdot \forall \ell : list \cdot \ sorted(\ell) \Rightarrow min_s(\ell) = min(\ell) \quad (2)$$

A symbolic CEGIS-style routine would instantiate ℓ as lists of size one, two, or more, containing symbolic elements, in order to transform the above specification into a recursion-free specification. Recently, we introduced *partial bounding* [10] as an alternative technique that can significantly improve recursion synthesis. The thesis of partial bounding is that it is not strictly necessary to *bound* every instance of *recursion* (e.g. instances of ℓ above) in the specification to obtain a recursion-free specification but, rather, this bounding can be done *parsimoniously*. With partial bounding, some recursive calls with recursively typed inputs are encapsulated by appropriately typed variables that stand in for the results of those calls. For example, if every instance of ℓ appears as $min(\ell)$ or $min_s(\ell)$, then each can be simply replaced by a variable v of type integer (i.e. the return type of both functions). This will transform the recursive constraints from Equation 2 into recursion-free ones, if we temporarily overlook the invariant $sorted(\ell)$:

$$\boxed{\begin{array}{l} \exists b_1, b_2\ \forall a : int, \ell : list \\ b_1(a) = a \\ b_2(a) = a \downarrow min(\ell) \end{array}} \xrightarrow{\text{Partial Bounding}} \boxed{\begin{array}{l} \exists b_1, b_2\ \forall a, v : int \\ b_1(a) = a \\ b_2(a) = a \downarrow v \end{array}}$$

The CEGIS-style algorithm of [10] relies on an invariant that the non-recursive approximations are always strict under-approximations of the original specification. The problem setup in [10] is a limited instantiation of the problem posed in this paper. In particular, the type invariants (e.g. $sorted(\ell)$) are not taken into account by [10]. Additionally, unrealizability outcomes do not exist in the technique of [10].

In the absence of the fact that ℓ is sorted, the constraints illustrated above are *unrealizable*, since no such function b_2 exists for *arbitrary* lists. The invariant $sorted(\ell)$ limits the valid choices for ℓ in the recursive constraints to sorted lists only. Yet, in the recursion-free constraints, having eliminated ℓ, one needs a semantically equivalent invariant constraining the participating symbols (e.g. a and v). In this case, from the fact that ℓ is sorted, one can *infer* that $a \leq min(\ell)$; that is, the first element of a sorted list is smaller than or equal to the minimum element of the tail of the same list. Therefore, the appropriate (realizable) version of the second constraint becomes $a \leq v \Rightarrow b_2(a) = a \downarrow v$. But, how can an inductive synthesis algorithm be guided to infer invariants of this type?

The key point is that $a \leq min(\ell)$ is not just about the sortedness of ℓ. It is a non-trivial fact about how the function *min* behaves on a sorted list $Cons(a, \ell)$. Such facts need to be inferred from the top invariant through an elaborate process. If one starts by approximating the invariant $a \leq v$ with a general placeholder such as *true*, then the approximate recursion-free specification is no longer a strict under-approximation of the original specification; observe that the original specification is realizable while the approximation $true \Rightarrow b_2(a) = a \downarrow v$ is not. Therefore, one requires the means to revise *unrealizable* (approximate) specifications, which are conspicuously absent in CEGIS-style algorithms.

1.2 Revising Unrealizable Approximations

Consider a dual (to CEGIS) inductive algorithm $\mathcal{A}$ that, by default, assumes that the high level synthesis specification Ψ is unrealizable and aims to generate a *witness* to this unrealizability. $\mathcal{A}$ uses a sequence $\psi_0, \psi_1, \ldots$ of approximate specifications in place of Ψ, where each approximation ψ_i is unrealizable. At each round i, $\mathcal{A}$ produces an unrealizability witness w_i for ψ_i, with the hope that w_i also certifies the unrealizability of Ψ. If not, w_i is used to revise ψ_i in the next round to ψ_{i+1}. The focus of $\mathcal{A}$ is on unrealizability; it shrinks the set of possible witnesses in each round until it finds a witness to the unrealizability of Ψ. A witness w_i in $\mathcal{A}$ plays the same role that a counterexample does in CEGIS, and it is as essential.

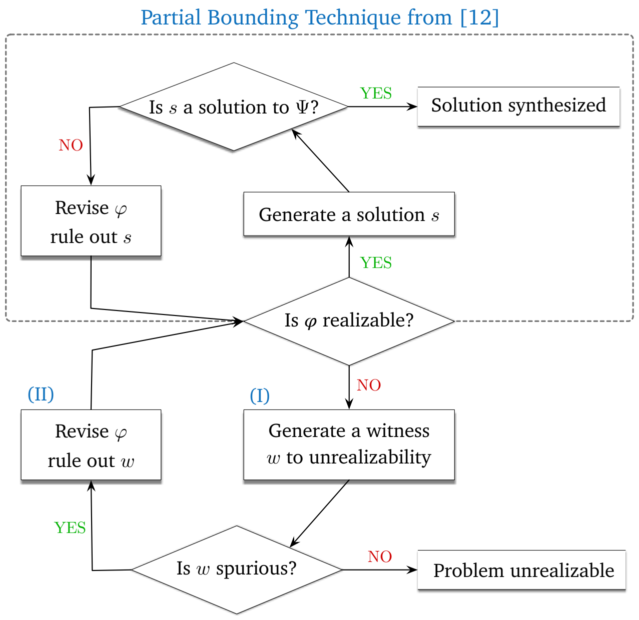

Partial Bounding Technique from [12]
Is s a solution to Ψ?
YES
Solution synthesized
NO
Revise φ rule out s
Generate a solution s
YES
Is φ realizable?
NO
(II)
(I)
Revise φ rule out w
Generate a witness w to unrealizability
YES
Is w spurious?
NO
Problem unrealizable

Figure 1. Overview of SE^{2}GIS.

Recently, some progress has been made in producing unrealizability witnesses in the context of grammar-based synthesis [15, 16], where the root cause of unrealizability is the lack of expressivity in the grammar. This makes these routines unsuitable for the specific usage we require here. As a major contribution of this paper, we propose a class of unrealizability witnesses called *functional unrealizability witnesses*, and an algorithm for generating them (see Section 6). These witnesses are used to revise the approximate specifications effectively to guarantee the progress of an algorithm in the style of $\mathcal{A}$.

Similar to CEGIS, where a solution to an approximate specification may not be a solution to the original specification, a witness w_i to the unrealizability of an approximate specification may not be a witness to the unrealizability of the original specification. CEGIS uses a *verify* step to check the solutions to approximate specifications. Similarly, $\mathcal{A}$ requires a step to check whether each w_i is a real or *spurious* witness; i.e. not a witness to the unrealizability of the original specification. A spurious witness triggers another revision round in $\mathcal{A}$.

1.3 SE^{2}GIS

We propose an algorithm called SE^{2}GIS that combines the two inductive algorithms – the partial bounding inductive synthesis scheme of [10] and the dual algorithm $\mathcal{A}$ explained in Section 1.2 – into one coherent inductive synthesis routine that solves the recursive specification in Equation 1.

Figure 1 illustrates the overall idea. The recursive specification of Equation 1 is approximated by a sequence of non-recursive specifications $\varphi_0, \varphi_1, \ldots$. Independent of the realizability/unrealizability of Equation 1, each φ_i may be realizable or unrealizable. The top loop is an instance of the *partial bounding* symbolic algorithm presented in [10], which controls the set of symbolic input-outputs. The bottom loop is our new dual inductive algorithm and the most significant contribution of this paper. This loop is activated whenever the approximate specification φ is unrealizable, which is whenever the (recursion-free) approximation of the $I_\theta(x) \wedge I_\tau(r(x))$ part of the specification is too weak. This loop controls the approximations of $I_\theta(x) \wedge I_\tau(r(x))$ parametric on the current set of symbolic input-outputs that are set by the top loop.

The two loops work together to form an inductive synthesis algorithm in the following sense. While φ is realizable, the top loop makes progress in revising φ to be closer to the original specification. If φ becomes unrealizable, then the bottom loop revises φ to be closer to the original specification. SE^{2}GIS may alternate between the two loops as many times as necessary until either a solution or an unrealizability witness is found. We present soundness and progress properties for the novel bottom loop, and for SE^{2}GIS as the combination of both loops.

We have implemented SE^{2}GIS in a tool called Synduce and evaluated it on 140 benchmarks. We present experimental results that demonstrate that SE^{2}GIS is substantially better at performing recursion synthesis than symbolic CEGIS, and that our proposed *functional unrealizability* solver is effective independent of the SE^{2}GIS setup.

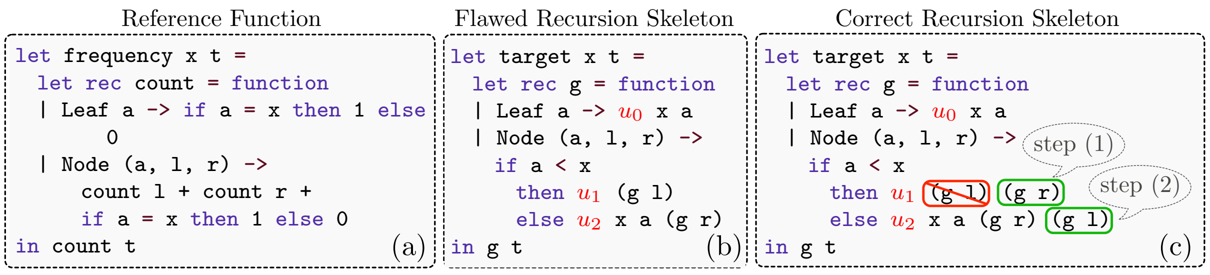
In summary the contributions of this paper are:

- A new inductive synthesis algorithm for recursion synthesis that (to our knowledge) is the first that uses unrealizability of approximate specifications for progress and can output an unrealizability witness.
- A new and interesting class of unrealizability root causes and an effective algorithm for generating them.
- An implementation and evaluation that demonstrates that the new ideas proposed in this paper substantially advance the marker on recursive program synthesis.

2 Motivating Example

Figure 2(a) implements a function, `frequency`, that computes the number of times an input parameter `x` appears in a tree `t` (that permits duplicates). The tree has integer-labelled nodes and leaves, and the recursive function `count` (in the body of the function `frequency`) recursively inspects each node and increments the count if the label is equal to `x`.

Suppose the programmer decides to use binary search trees (which permit duplicates) instead of arbitrary trees and, therefore, wishes to port the `frequency` function to this new data type. The programmer can conjecture that a more efficient implementation of `frequency` may exist for binary search trees. They use the recursion skeleton in Figure 2(b) to communicate this conjecture to Synduce. The (non-recursive) functions u_0, u_1 and u_2 are *unknown*, and code must be synthesized for them such that `target`



```
let frequency x t =
  let rec count = function
  | Leaf a -> if a = x then 1 else
      0
  | Node (a, l, r) ->
      count l + count r +
      if a = x then 1 else 0
in count t
```

```
let target x t =
  let rec g = function
  | Leaf a -> u0 x a
  | Node (a, l, r) ->
    if a < x
    then u1 (g l)
    else u2 x a (g r)
in g t
```

```
let target x t =
  let rec g = function
  | Leaf a -> u0 x a
  | Node (a, l, r) ->
    if a < x
    then u1 ~~(g l)~~ (g r)
    else u2 x a (g r) (g l)
in g t
```

Figure 2. Synthesizing the frequency function on binary search trees.

becomes equivalent to `frequency` on inputs that are binary search trees.

This skeleton is not clever; it distinguishes a base case, includes the (generally understood) insight of comparing the label of the node to the input parameter `x`, and vaguely tries to be *efficient* by not recursing on the entire tree in each case. It is in fact completely wrong: the recursive calls `g(l)` and `g(r)` are both misplaced. Note that, without such restrictions, `frequency` itself is a valid choice on binary search trees; therefore, the skeleton plays an essential role.

Since the recursion skeleton is wrong, the synthesis instance is *unrealizable*. Synduce correctly outputs that it is unrealizable and generates a witness for this in less than a second; i.e. two (sets of) inputs to the program that demonstrate that a solution to the synthesis problem does not exist.

The witness pinpoints u_1 as the problem: both inputs (in the witness) have the same value for `l` (and therefore `g(l)`), but expect different outcomes for the return value of u_1. No such function u_1 can exist, which is the precise root cause of the unrealizability. Figure 2(c) illustrates how the programmer repairs the skeleton with the guidance of the tool. First, they replace the argument `g(l)` of u_1 by `g(r)` (step (1) in the figure). The problem remains unrealizable and Synduce returns a witness that points to u_2 this time, and to the fact that `g(l)` is missing as an argument. The programmer then adds `g(l)` to the list u_2's arguments[2]. After this step, the skeleton is correct, and it is optimal in the following sense: removing any recursive call to g results in an unrealizable instance.

After the skeleton is repaired, Synduce synthesizes the following solution for the unknown functions in less than one second.

```
let u0 x a = if a = x then 1 else 0
let u1 z = z
let u2 x a y z = if a = x then 1 + y + z else y + z
```

Unrealizability witnesses play two distinct roles in this example: (1) as discussed, they can guide the user through the repair of the wrong skeleton, and (2) once the skeleton is repaired, they play a vital role in the discovery of the solution by guiding the inference of the required invariants. For instance, ignoring the invariant, the specification leads to the following constraint for u_1:

$$\mathtt{a} < \mathtt{x} \Rightarrow u_1(\mathtt{g(r)}) = \mathtt{count(r)} + \mathtt{count(l)}. \quad (3)$$

SE^{2}GIS eliminates all the instances of recursion by observing that `g(r)` = `count(r)` and replacing both terms with a fresh variable v_r, and by replacing `count(l)` with another fresh variable v_l, both of type integer, resulting in:

$$\mathtt{a} < \mathtt{x} \Rightarrow u_1(v_r) = v_r + v_l. \quad (4)$$

This is unrealizable for a similar reason to the example in Section 1.2. The witness to its unrealizability is a pair of values for (v_r, v_l): $(1, 1)$ and $(1, 2)$. It is easy to observe that there exists no function u_1 that takes 1 as an input and returns 2 in one instance and 3 in another. However, since the original specification is realizable, this cannot be a witness to the unrealizability of Equation 3. In particular, observe that, under the assumption $\mathtt{a} < \mathtt{x}$, neither 1 nor 2 is a valid value for $v_l = \mathtt{count(l)}$, since $\mathtt{count(l)} = 0$. The unrealizability of Equation 4 is precisely due to this missing invariant. The spurious witnesses help the bottom loop in Figure 1 infer the (nontrivial) fact that, under the condition $a < x$, we have $\mathtt{count(l)} = 0$. The realizable constraint then becomes:

$$(a < x \wedge v_l = 0) \Rightarrow u_1(v_r) = v_r + v_l$$

After learning this *invariant*, a solution is synthesized. Therefore, the solution is synthesized after one round of the top loop (performing the partial bounding) and one round of the bottom loop of Figure 1.

In the bottom loop, while producing the pair of witnesses to unrealizability of the approximate specification φ, the backend solver can make the job of the synthesis tool harder if it produces an invalid value for v_r, for example -1 that cannot correspond to the number of occurrences of a value. At the high level, we understand that `count(r)` is always non-negative. This information, however, is missing in the tool and can be another source of unrealizability. In cases like this, Synduce, using the same mechanism it does for the missing type invariants, can infer these essential missing invariants about the reference function to help the refinement process move ahead. In contrast to our method, the technique in [10] requires the user to provide such invariants in advance.

[2]Note that swapping `g(r)` for `g(l)` in this step would lead to another unrealizability witness. Step (2) is taking the short cut here for brevity.

Synduce manages to synthesize the problem instance of Figure 2 without bounding any input instances. In cases like this, once a solution is synthesized, the solution is fully verified. This is in sharp contrast to the bounded verification step employed by most tools that target recursion (or looping). In cases where Synduce performs partial bounding, even though it cannot claim that the result is fully verified, there is still more confidence in the correctness of the solutions produced because, for the unbounded inputs, we have a guarantee of correctness for all instances. A symbolic algorithm that bounds all inputs lacks this feature and takes a much longer time to synthesize a solution to this problem (88 seconds compared to one second).

3 Background

The notation introduced in this section is used for formalizing the result of applying recursive functions to symbolic inputs. We assume that all recursion is representable as pattern-matching recursive schemes [31], which gives us some well-formedness guarantees.

Recursion Skeletons. Our problem finds a solution to Equation 1 within a family of recursion functions $\mathcal{G}$. This family of functions can be specified as a recursion skeleton:

Definition 3.1 (Recursion Skeleton)**.** Let $\mathcal{U}$ be a finite set of unknown functions from scalar types to scalar types. A recursion skeleton $\mathcal{G}[\mathcal{U}]$ is a family of recursive functions parameterized by the unknown functions $\mathcal{U}$, such that replacing the unknowns $\mathcal{U}$ by some implementation in $\mathcal{G}[\mathcal{U}]$ results in a fully determined recursive function.

We model the family of recursive functions by recursion skeletons in order to distinguish a set of unknown scalar functions, which are the unknowns for which we need an implementation. An implementation of the function in $\mathcal{U}$ is all that is needed to make the recursive function $\mathcal{G}[\mathcal{U}]$ fully defined.

Terms. We make use of a set of *symbols* that are partitioned into *terminal symbols* Σ and an infinite set of typed *variables* $\mathcal{V}$. We also reserve a distinguished set of symbols $\{\circ_i\}_{i\in\mathbb{N}}$, the "holes", representing placeholders to manipulate expressions and construct precise substitution functions. Terms are defined by the grammar $S \rightarrow x \mid S(S)$ where x is a symbol, and $S(S)$ is a function application. *Concrete terms* $T(\Sigma)$ are the terms 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. The relation $\geq$ over symbolic terms is a **partial order** where $t \geq t'$ iff there exists a substitution $\sigma : FV(t) \rightarrow T(FV(t') \cup \Sigma)$ such that $t' = \sigma t$. Single variables are maximal elements according to this partial order and concrete terms (of any depth) are minimal.

Types. We use capital letters A, B, C, and D to refer to base types, which are scalar types (*Int*, *Bool*, *Char*, ...) or tuples of scalar types (e.g. $Int \times Int$). The set of variables of base type is denoted $\mathcal{V}_B$.

We write $x : \tau$ to denote that x is of type τ. The universal quantification with x ranging over all the values of type τ is written $\forall x : \tau$. The set of variables of type τ in $\mathcal{V}$ is denoted $\mathcal{V}_\tau$. For a finite set of variables $V = \{x_1 : \tau_1, x_2 : \tau_2, \ldots\}$ we write the quantification $\forall x_1 : \tau_1, x_2 : \tau_2, \ldots$ as $\forall \vec{x} \in V$.

Given the distinction between base types and recursive types, we can differentiate **bounded terms**, which are symbolic terms where all free variables are of base type, from **unbounded terms**, where free variables can be of any type, including recursive types. An unbounded term t is a finite symbolic term where infinitely many bounded terms are expansions of t.

4 Synthesizing Recursive Functions

In this section, we first present a formal definition of the problem posed in Section 1 as Equation (1). We then present the formal definition of recursion-free approximations used in our inductive synthesis algorithm. Finally, we give an overview of our solution based on the formal version that can be used as a road map for Sections 5 and 7.

As a first observation, remark that one can account for the invariant I_τ of the source type τ using the representation function $r : \theta \rightarrow \tau$ and the invariant I_θ of the destination type θ. In Equation (1), the quantification is over all possible values of type θ, and not values of type τ. Any constraint induced by I_τ can be incorporated into a modified representation function r' and a type invariant I'_θ. The new specification, without I_τ, would be equivalent to the old specification iff $\forall x : \theta. I'_\theta(x) \Leftrightarrow I_\theta(x) \wedge I_\tau(r'(x))$. For example, if I_τ states that a list is sorted and all its elements are positive, then the original representation function can be composed with any list sorting function, and I'_θ ensures that individual elements are positive.

The second observation is that the family of functions $\mathcal{G}$ can be effectively and elegantly captured using a *recursion skeleton* (see Definition 3.1). Using these observations, the formal synthesis problem addressed in this paper is then:

Definition 4.1 (Recursion Synthesis Problem)**.** Given a *reference function* $f : \tau \rightarrow D$, a *representation function* $r : \theta \rightarrow \tau$, a *family of target recursive functions* $\mathcal{G}[\mathcal{U}] : \theta \rightarrow D$ parameterized by a set of *unknowns* $\mathcal{U}$ and a *type invariant* $I_\theta : \theta \rightarrow Bool$, the recursion synthesis problem consists in finding an implementation of $\mathcal{U}$ such that:

$$\Psi \equiv \forall x : \theta \cdot I_\theta(x) \Rightarrow \mathcal{G}[\mathcal{U}](x) = (f \circ r)(x)$$

Two additional assumptions are made about the problem instances: (1) recursive functions are terminating and (2) all recursion is structural. Our technique relies on symbolic evaluation of bounded and unbounded terms, and these conditions ensure that it always terminates and yields a term.

4.1 Recursion-Free Approximation

The synthesis problem of Ψ (from Definition 4.1) boils down to the synthesis of solutions for a set of unknowns $\mathcal{U}$ associated with an infinite set of programs $\mathcal{L}_{\mathcal{U}}$.

As discussed in Section 1, Ψ is approximated by a sequence of recursion-free approximations. These approximations and Ψ share the same set of unknowns. The point is that it is viable to synthesize solutions for these unknowns given the approximate recursion-free specifications using existing solvers, but the same is not viable given Ψ.

System of Guarded Functional Equations. Our recursion-free approximations are defined by a set of guarded equations. These mirror the structure of Ψ but they contain only free variables of base (non-recursive) types.

Definition 4.2 (System of Guarded Functional Equations)**.** A system of guarded functional equations (SGE) $\mathcal{E}$ is a finite set of constraints of the form $\{p_i \Rightarrow l_i = r_i\}_{1\leq i\leq n}$ where $n \geq 0$, and for $1 \leq i \leq n$, p_i and r_i are terms in $T(\Sigma, \mathcal{V}_B)$ and l_i is a term in $T(\Sigma \cup \mathcal{U}, \mathcal{V}_B)$.

A system of functional equations $\mathcal{E}$ defines the following synthesis problem:

$$\exists \mathcal{U}. \forall \vec{x} \in FV(\mathcal{E}) \cdot \bigwedge_{1\leq i\leq n} p_i \Rightarrow l_i = r_i$$

When the types of the variables in $FV(\mathcal{E})$ are sorts of a theory supported by Satisfiability Modulo Theory (SMT) and/or SyGuS (syntax-guided synthesis [2]) solvers, and a context-free grammar is given for each function in $\mathcal{U}$, then the synthesis problem of a system of guarded functional equations can be solved by one of these tools. In our inductive synthesis loop, each approximation of Ψ is an SGE.

Approximation of Ψ. A recursion-free SGE is constructed by systematically eliminating recursive variables and functions from the specification Ψ. Our process for *recursion elimination* is the same as the one introduced in [10], which we formalize here by defining a function that performs this elimination. In this paper, we are also interested in the inverse translation that would reintroduce unbounded terms.

Definition 4.3 (Recursion Elimination)**.** Let $\mathcal{V}_{elim}$ be a distinguished set of variables of type D. Let α a bijection between $\mathcal{V}_\theta$ and $\mathcal{V}_{elim}$. *Recursion elimination* is the function $[\![.]\!]_{elim}$ on terms $T(\Sigma, \mathcal{V})$ defined recursively by:

$$\begin{array}{lcl} [\![(f \circ r)(x)]\!]_{elim} & = & \alpha(x) \text{ } if \text{ } x \in \mathcal{V}_\theta \\ [\![\mathcal{G}[\mathcal{U}](x)]\!]_{elim} & = & \alpha(x) \text{ } if \text{ } x \in \mathcal{V}_\theta \\ [\![x]\!]_{elim} & = & x \text{ } if \text{ } x \in \mathcal{V} \\ [\![g(t_1, t_2, \dots)]\!]_{elim} & = & g([\![t_1]\!]_{elim}, [\![t_2]\!]_{elim}, \dots) \end{array}$$

We say that a term t is *canonical*[3] iff symbolically evaluating the reference function and the target on t results in an expression whose recursion elimination contains no recursively typed variables. That is, $FV([\![(f \circ r)(t)]\!]_{elim}) \subset \mathcal{V}_B$ and $FV([\![(\mathcal{G}[\mathcal{U}])(t)]\!]_{elim}) \subset \mathcal{V}_B$.

Example 4.4. Recall the example from the introduction, where $f = min$ and $\mathcal{G}[b_1, b_2] = min_s$ (and r is identity). Let a_1, a_2 be two integer variables and l a variable of type *List*. Then $t_1 = Elt(a_1)$ is trivially a canonical term: $[\![min(Elt(a_1))]\!]_{elim} = [\![a_1]\!]_{elim} = a_1$ and $[\![min_s(Elt(a_1))]\!]_{elim} = b_1(a_1)$ do not contain recursively typed variables. In general, bounded terms are canonical terms. More interestingly, $t_2 = Cons(a_2, l)$ is a canonical term. Let $v_l = \alpha(l)$, then:

$$[\![min(t_1)]\!]_{elim} = a_2 \downarrow [\![f(l)]\!]_{elim} = a_2 \downarrow v_l$$
$$[\![min_s(t_1)]\!]_{elim} = b_2(a_2)$$

are two terms free of recursively typed variables. ⌟

The map α is a bijection and therefore recursion elimination can be inverted: $[\![.]\!]^{-1}_{elim}$ replaces every scalar variable $x \in \mathcal{V}_{elim}$ with a recursive call $(f \circ r)(\alpha^{-1}(x))$. Remark that we always choose to replace $x \in \mathcal{V}_{elim}$ with $(f \circ r)(\alpha^{-1}(x))$ rather than $\mathcal{G}[\mathcal{U}](\alpha^{-1}(x))$.

The equation $[\![(f \circ r)(t) = \mathcal{G}[\mathcal{U}](t)]\!]_{elim}$ is recursion free for any canonical term t. However, there is no guarantee that $[\![I_\theta(t)]\!]_{elim}$ is recursion-free, since the recursive functions that appear in I_θ are not the ones eliminated by $[\![.]\!]_{elim}$ (which are $f \circ r$ or $\mathcal{G}[\mathcal{U}]$). We could eliminate recursion from the constraint $I_\theta(t) \Rightarrow \mathcal{G}[\mathcal{U}](t) = (f \circ r)(t)$ in straightforward way by choosing a bounded term t instead of a canonical one. This, however, would mean that our algorithm could not take advantage of partial bounding, which has a significant impact on tractability [10]. Therefore, our solution offers a way to leave t partially bounded and aims for a recursion-free *strengthening* of $I_\theta(t)$.

Example 4.5. Recall Example 4.4. The term $t_2 = Cons(a_2, l)$ is canonical. However, to eliminate recursion from the term $sorted(t_2) = a_2 \leq head(l) \wedge sorted(l)$, one must infer (new) properties involving *sorted* and *head*. ⌟

Our approximation is constructed with parameters T, a set of unbounded *canonical* terms, and $\mathcal{P}$, a set of recursion-free terms that we call *guards*:

Definition 4.6 (Approximation of Ψ)**.** Given a set of terms $T = \{t_i\}_{1\leq i\leq n}$, and a set of guards $\mathcal{P} = \{p_i\}_{1\leq i\leq n}$, such that $\forall \vec{x} \in FV(t_i) \cdot I_\theta(t_i) \Rightarrow [\![p_i]\!]^{-1}_{elim}$, the approximation of Ψ is

$$\mathcal{E}(T, \mathcal{P}) = \{p_i \Rightarrow l_i = r_i\}_{1\leq i\leq n}$$

where $l_i = [\![\mathcal{G}[\mathcal{U}](t_i)]\!]_{elim}$ and $r_i = [\![(f \circ r)(t_i)]\!]_{elim}$.

Observe that each t_i in the definition corresponds to one p_i. So, given an approximation $\mathcal{E}(T, \mathcal{P})$, each term $t \in T$ has a unique **corresponding predicate** in $\mathcal{P}$. We also require that the terms of T have **no shared free variables**: for $i \neq j$, t_i and t_j have no free variables in common.

[3]Canonical terms are referred to as *maximally reducible* in [10].

This approximate specification is a recursion-free guarded system of functional equations (Definition 4.2) which can be solved by off-the-shelf synthesis solvers. The two parameters of the approximation, T and $\mathcal{P}$, determine the precision of the approximation. The larger T is, the more likely it is for a solution of $\mathcal{E}(T, \mathcal{P})$ to be a solution of Ψ (analogous to increasing the set of input-output examples in CEGIS). The stronger the predicates in $\mathcal{P}$ are, the more likely it is for a witness of unrealizability of $\mathcal{E}(T, \mathcal{P})$ to be a witness of Ψ. With this insight in mind, we describe our synthesis algorithm.

Example 4.7. Recall Example 4.4. Let $T = \{t_1, t_2\}$ a set of canonical terms. Let $\mathcal{P} = \{p_1, p_2\}$ where $p_1 = p_2 = \top$. Then the approximation is:

$$\mathcal{E}(T, \mathcal{P}) = \{\top \Rightarrow b_1(a_1) = a_1, \top \Rightarrow b_2(a_2) = a_2 \downarrow v_l\}$$

Note that this approximate specification is unrealizable due to its second constraint and the fact that $p_2 = \top$. Another valid choice for p_2 is $a_2 \leq v_l$, since $sorted(Cons(a_2, l)) \Rightarrow a_2 \leq min(l)$, as we argued in Section 1. With this choice, $b_1 = b_2 = \lambda x.x$ is a solution. ⌟

4.2 Symbolic SE²GIS with Partial Bounding

With our approximate specification formally defined, let us recall the overview of SE²GIS from Figure 1 to make its key steps more concrete and provide a roadmap to the rest of the technical presentation in this paper.

In Figure 1, the approximate specification φ is a system of guarded functional equations $\mathcal{E}(T, \mathcal{P})$. The top loop is the *refinement loop* from [10] that updates T to make the solution space of $\mathcal{E}(T, \mathcal{P})$ smaller. The set T is strictly increasing through refinement rounds of this loop. The bottom loop updates $\mathcal{P}$ to make the set of unrealizability witnesses for $\mathcal{E}(T, \mathcal{P})$ smaller. We refer to this loop as the *coarsening* loop, in the sense that it is the dual of the standard *refinement* loop. The guards $\mathcal{P}$ are strictly strengthened across rounds of coarsening.

Figure 3 illustrates how a run of SE²GIS makes progress across multiple refinement and coarsening rounds. Solid circles identify realizable approximations and hollow ones stand for unrealizable ones. Initially, SE²GIS starts with a minimal set of initial terms T_0 and the trivial set of guards $\mathcal{P}_0 = \{true\}$. In each round, if $\mathcal{E}(T, \mathcal{P})$ is realizable and yet does not yield a solution to Ψ, then T is augmented with new (canonical) terms. This step also ensures that the new canonical terms have no free variables in common with the previous ones, as required for the construction of $\mathcal{E}(T, \mathcal{P})$.

If $\mathcal{E}(T, \mathcal{P})$ is unrealizable and yet does not yield an unrealizability witness for Ψ, then $\mathcal{P}$ is strengthened by the *coarsening* loop. An update of $\mathcal{P}_0$ to $\mathcal{P}_1$ strengthens the constraints imposed on the approximate specification to rule out unrealizability witnesses that do not satisfy the type invariants or some invariant of the reference function f.

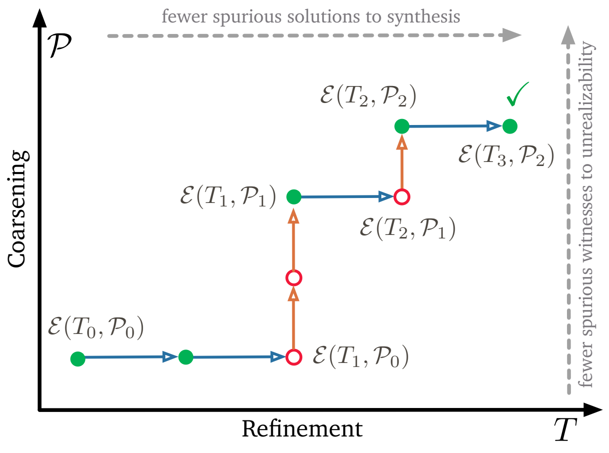

fewer spurious solutions to synthesis
$\mathcal{P}$
Coarsening
$\mathcal{E}(T_2, \mathcal{P}_2)$
✓
$\mathcal{E}(T_3, \mathcal{P}_2)$
$\mathcal{E}(T_1, \mathcal{P}_1)$
$\mathcal{E}(T_2, \mathcal{P}_1)$
$\mathcal{E}(T_0, \mathcal{P}_0)$
$\mathcal{E}(T_1, \mathcal{P}_0)$
fewer spurious witnesses to unrealizability
Refinement
T

Figure 3. Symbolic SE²GIS with Partial Bounding

The *Coarsening* step relies on two subroutines: one that generates an unrealizability witness for $\mathcal{E}(T, \mathcal{P})$ and one that checks if this witness is *spurious*, i.e., if it corresponds to a witness to the unrealizability of Ψ. In Section 5, we formalize the concept of unrealizability witnesses and categorize them as *valid* and *spurious*. In Sections 6, we present a decision procedure for producing a family of unrealizability witnesses for an unrealizable SGE $\mathcal{E}(T, \mathcal{P})$. In Section 7, we present an algorithm for strengthening the guards $\mathcal{P}$ based on spurious unrealizability witnesses. Combined, they guarantee that our proposed *coarsening* loop has the same soundness and progress properties as the *refinement* loop while enjoying the benefits of the *partial bounding* technique. We then show that SE²GIS – that is, the combination of the two loops – retains the benefits of *partial bounding* and has the same soundness and progress properties as the individual loops.

5 Unrealizability Witnesses

Program synthesis techniques are mostly focused on synthesizing a solution and often fail or diverge if the synthesis problem is unrealizable. SE²GIS actively generates unrealizable synthesis subproblems and relies on the unrealizability witnesses for them to make progress in the high level synthesis goal. In this section, we formally define these witnesses for systems of guarded functional equations (SGEs).

Valid and Spurious Witnesses. Recall that all variables in an SGE have base type, i.e., they may be scalars or tuples of scalars. A valuation for these variables is a *model* m, which is a map from $Dom(m) \subset \mathcal{V}_B$ to values of the appropriate type. A model can be used to evaluate a term: given a term t such that $FV(t) \subseteq Dom(m)$, $[\![t]\!]_m$ is the value of the term t where all its free variables have been assigned their value in the model m. Unrealizability witnesses of SGEs can be formally defined based on such models:

Definition 5.1 (Unrealizability Witness of an SGE)**.** An unrealizability witness of a system of guarded functional equations $\mathcal{E}$ of size n is a finite set M of models such that: $\forall \mathcal{U} \cdot \exists i \in [1, n] \cdot \exists m \in M \cdot [\![p_i]\!]_m \wedge [\![l_i]\!]_m \neq [\![r_i]\!]_m$.

If M is an unrealizability witness for an SGE $\mathcal{E}$, then $\mathcal{E}$ has no solutions. However, there exist SGEs that have no solutions and yet no (finite) unrealizability witness exists for them.

Let the SGE $\mathcal{E}(T, \mathcal{P})$ be an approximation of Ψ. The unrealizability of $\mathcal{E}(T, \mathcal{P})$ does not necessarily imply the unrealizability of Ψ. Let M be an unrealizability witness for $\mathcal{E}(T, \mathcal{P})$. We need a way to determine if M also *witnesses* the unrealizability of Ψ. The difficulty is that the models in M are valuations of base-type variables, whereas a witness to the unrealizability of Ψ must be a set of terms of (recursive) type θ, since Ψ is universally quantified over θ. The following definition, inspired by recursion elimination (Definition 4.3), suggests how M can be transformed into a potential witness for unrealizability of Ψ.

Definition 5.2 (Inverse of a model)**.** Let m be a model for some variables in $\mathcal{V}_B \cup \mathcal{V}_{elim}$. For $x \in Dom(m)$ define:

$$m^{-1}(x) \equiv \begin{cases} (f \circ r)(\alpha^{-1}(x)) = [\![x]\!]_m & x \in \mathcal{V}_{elim} \\ x = [\![x]\!]_m & otherwise \end{cases}$$

The inverse m^{-1} maps a variable to an equality constraint, depending on the nature of the variable. For example, the model $m = [a_2 \leftarrow 0, \alpha(l) \leftarrow 1]$ inverted yields the equality constraints $a_2 = 0$ and $(f \circ r)(l) = 1$.

We say a term t is *compatible with* a model m iff there is some assignment of t's free variables that is compatible with the values of m given the definition of $f \circ r$. More formally, consider a model m and an unbounded term t, where all variables that are assigned in the model pertain to t. That is, $\forall v \in Dom(m) \cdot (v \in FV(t) \vee \alpha^{-1}(v) \in FV(t))$. Define the relation $\ltimes$ as

$$t \ltimes m \equiv \bigwedge_{x \in Dom(m)} m^{-1}(x).$$

We say t is *compatible with* m iff $t \ltimes m$ is satisfiable.

This relation between (recursively typed) terms and models is the key in distinguishing between *valid* and *spurious* unrealizability witnesses for SGEs, which are, respectively, witnesses that *do* and *do not* correspond to an unrealizability witness for the high level specification Ψ.

Definition 5.3 (Spurious Witness)**.** Let the SGE $\mathcal{E}(T, \mathcal{P})$ be an approximation of Ψ and let the set of models M be an unrealizability witness for $\mathcal{E}(T, \mathcal{P})$. We call M *spurious* iff there is a model m in M such that $\forall x : \theta \cdot \forall \vec{z} \in FV(x) \cdot x \ltimes m \Rightarrow \neg I_\theta(x)$.

The key to designing a decision procedure for *spuriousness* is the observation that the quantification of $\forall x : \theta$ in Definition 5.3 is not really necessary. It suffices to limit the quantifier to the set of terms T that defines $\mathcal{E}$. This gives us a straightforward way of checking if a witness is spurious.

Proposition 5.4. *Let $\mathcal{E}(T, \mathcal{P})$ an approximation and M an unrealizability witness for it. Then the witness M is spurious iff $\exists m \in M \cdot \forall t \in T \cdot t \ltimes m \Rightarrow \neg I_\theta(t)$.*

Example 5.5. Let us assume in this example that the reference function f returns the length of a list, and r is identity. Let a an integer, l a list variable, $v_l = \alpha(l)$, and $t = Cons(a, l)$. Let M be the witness $\{[a \leftarrow 0, v_l \leftarrow 1], [a \leftarrow 0, v_l \leftarrow -1]\}$, a set of two models. Then t is compatible with $M(0)$ since $a = 0 \wedge (f \circ r)(l) = 1$ is satisfiable. For instance, it may be satisfied by assigning 0 to a and $Cons(2, Nil)$ to l. However, t is not compatible with $M(1)$, since there is no list of length -1. So, M would be spurious, independently of I_θ.

Now let us assume I_θ is the invariant that lists are sorted in strictly *decreasing* order, and have only *non-negative elements*. Then M is a spurious witness, regardless of $M(1)$: t is compatible with $M(0)$ but then cannot satisfy the invariant. There is no list $t = Cons(a, l)$ such that $a = 0 \wedge length(l) = 1$ where t is a list of strictly decreasing non-negative values. ⌟

Proposition 5.4 gives us a straightforward way of checking if an unrealizability witness is spurious by discharging the logical constraint to an SMT solver. Additionally, the construction of $\mathcal{E}(T, \mathcal{P})$ (Definition 4.6) guarantees that the terms in T have no free variables in common. This further simplifies the spuriousness check. For each model m in M, there can only be one term in T that matches the domain of m and therefore could be compatible with it. So, for each model m in M, one can select the unique term t in T that matches the domain of m. Given t and m, one can check that $t \ltimes m \Rightarrow \neg I_\theta(t)$ using an SMT solver with induction support. If this formula is valid, then M is a spurious witness.

Definition 5.6. [S-Certificate] For a spurious unrealizability witness M, the pair (m, t), where $m \in M$ and $t \in T$ matches the domain of m, is a *certificate of spuriousness* (s-certificate) of M iff $\forall \vec{z} \in FV(t).t \ltimes m \Rightarrow \neg I_\theta(t)$.

Example 5.7. Recall Example 4.7. The unrealizable approximate specification, with $T = \{Elt(a_1), Cons(a_2, l)\}$ and $p_1 = p_2 = \top$, admits a witness $M = \{[a_2 \leftarrow 1, v_l \leftarrow 0], [a_2 \leftarrow 1, v_l \leftarrow 1]\}$.

This witness is spurious. The term $Cons(a_2, l)$ from T matches the domains of the models in M. The resulting compatibility $Cons(a_2, l) \ltimes [a_2 \leftarrow 1, v_l \leftarrow 0]$ means that the minimum of the tail of the list l is 0 and its first element is 1. This contradicts the invariant that the list is sorted in increasing order, and therefore, implies its negation: $\neg sorted(Cons(a_2, l))$. Therefore, $([a_2 \leftarrow 1, v_l \leftarrow 0], Cons(a_2, l))$ is an s-certificate. As mentioned before, for a model $[a_2 \leftarrow 1, v_l \leftarrow 0]$, there is always exactly one compatible term from T, and the reader may observe in this example that $Elt(a_1)$ is not compatible. ⌟

The existence of an s-certificate for a spurious witness M is guaranteed by Proposition 5.4. Intuitively, s-certificates play the same role in the dual loop that counterexamples do in the classical CEGIS loop. In Section 7, we will show how an s-certificate is used to strengthen the predicates in $\mathcal{P}$.

6 Functional Unrealizability

So far, we have established a way of categorizing unrealizability witnesses for SGEs into *valid* and *spurious*. But, where do these witnesses come from? Checking the unrealizability of recursion-free specifications like SGEs is, in general, undecidable [7]. There are *approximate* techniques [15, 16] to prove unrealizability in the context of syntax-guided synthesis. However, they target cases where a grammar for the unknowns exists and the limitations of this grammar is the root cause of unrealizability.

We propose an alternative approach that forgoes the above limitations at the cost of other limitations. Our technique is not specific to syntax-guided synthesis (and does not rely on a grammar). Instead, it focuses on a subset of possibilities for unrealizability. Specifically, it considers synthesis problems where the nonexistence of any solutions stems from the fact that the components of the solution must be *functions*.

Consider a constraint of the form $h(x_1, \ldots x_n) = x_0$ for some function h and terms $x_0, \ldots, x_n$. Consider two different evaluations of the x_i terms $v_0, \ldots v_n$ and $v'_0, \ldots, v'_n$ such that we have $v_1 = v'_1, \ldots, v_n = v'_n$ and $v_0 \neq v'_0$. These evaluations suggest that, on equal inputs, h must produce different outputs, which violates the definition of h as a *function*. A *pair of models* forms an unrealizability witness, if the instantiation of one or two equations from the SGE produces two constraints of the above form. This idea is the essence of *functional unrealizability*.

Definition 6.1 (Functional Unrealizability)**.** We say an SGE is *functionally unrealizable* iff there exists a pair of models (m, m') and two equations $p_i \Rightarrow l_i = r_i$ and $p_j \Rightarrow l_j = r_j$ (including the $i = j$ case) such that the following is unsatisfiable:

$$\llbracket p_i \rrbracket_m \Rightarrow \llbracket l_i \rrbracket_m = \llbracket r_i \rrbracket_m \wedge \llbracket p_j \rrbracket_{m'} \Rightarrow \llbracket l_j \rrbracket_{m'} = \llbracket r_j \rrbracket_{m'}$$

Note that these form a strict subset of all unrealizable SGEs. More generally, one may ask if the SGE (as a synthesis specification with one alternation of quantifiers) is realizable. The boolean query for this can be discharged to an SMT solver that handles the "exists forall" fragment [6], as long as the underlying theory admits model based projection [21]. In our context, we are interested in cases that may (at least lightly) step outside these clean theories, but more importantly, we are not purely interested in the boolean answer to the query. We need the *pair* of models (m, m') to make progress in the coarsening loop. Z3 [9, 13] can produce proofs for unsatisfiable $\exists\forall$ queries, but they are verbose, and it is unclear if one can extract a witness (m, m') from the proof, since they are not actively targeting the subclass of interest. Instead, we propose a lightweight algorithm that targets the limited class directly and seems to work very well in practice when the goal is the generation of (m, m'). In [30], we discuss two small examples that show where our technique fails and Z3 succeeds, as well as the converse.

For example, the pair of models $[x \leftarrow -3, y \leftarrow 2]$ and $[x \leftarrow -1, y \leftarrow 2]$ witness the unrealizability of the equation $h_1(max(x, 0)) + h_2(y) = max(x + y, 0)$. (This corresponds to a case where $i = j$ in Definition 6.1.) To see why, define h' as $h'(a, b) = h_1(a) + h_2(b)$, and the pair of models witness that h' cannot be a well-defined function. Below, we formally define the generic form of the syntactic manipulation we call *framing* that transforms a term with unknown functions into a single function application over subterms.

Proposition 6.2. *Any term e in $T(\Sigma \cup \mathcal{U}, \mathcal{V})$ can be framed as a pair of a term F with $c \geq 0$ holes and no variables ($F \in T(\Sigma \cup \mathcal{U} \cup \{\circ_i\}_{1 \leq i \leq c})$) and a tuple of c terms $t_1, \ldots, t_c$ ($\in T(\Sigma, \mathcal{V})$) such that $e = F[t_1/\circ_1][\ldots][t_c/\circ_c]$.*

$F(t_1, \ldots, t_c)$ denotes the substitution of the indexed holes by the terms, i.e., short for $F[t_1/\circ_1][\ldots][t_c/\circ_c]$. This proposition makes the concept of the **frame** of a term well-defined. A frame $(F, (t_1, \ldots t_c))$ is **maximal** if for any other frame $(F', (t'_1, \ldots t'_c))$, we have $F \geq F'$. Maximal frames are the ones we use for our syntactic manipulation.

In an SGE, we can obtain unrealizability witnesses from pairs of *different* constraints, as long as they share the same frame. Suppose that, in our previous example, we also had the constraint $h_1(0) + h_2(z) = z$, which can be framed as $h'(0, z) = z$. The pair of models $[z \leftarrow 2]$ and $[x \leftarrow -3, y \leftarrow 2]$ (for the earlier constraint) also form a unrealizability witness. The new constraint gives us $h'(0, 2) = 2$ whereas the previous one gives us $h'(0, 2) = -1$. If the earlier constraint had been framed as $h''(x, y) = h_1(max(x, 0)) + h_2(y)$, capturing only x instead of $max(x, 0)$, then we would not be able to produce a witness of unrealizability for the pair, since $h'' \neq h'$.

Consider an SGE $\mathcal{E}$ of size n. The left-hand side l_i of every equation can be framed as $F_i(t_{i,1}, \ldots, t_{i,c_i})$, and therefore, every constraint can then be transformed to $p_i \Rightarrow F_i(t_{i,1}, \ldots, t_{i,c_i}) = r_i$. Observe that p_i, $t_{i,1}, \ldots, t_{i,c_i}$ and r_i contain only variables and no unknowns and, in contrast, F_i contains no variables and all the unknowns. After framing the left-hand side of all equations in an SGE, we define *witnesses to functional unrealizability*:

Definition 6.3 (Witness)**.** Let $\mathcal{E}$ be the system of functional equations $\{p_i \Rightarrow F_i(t_{i,1}, \ldots, t_{i,c_i}) = r_i\}_{1 \leq i \leq n}$ with unknowns $\mathcal{U}$. A witness to the functional unrealizability of $\mathcal{E}$ is a pair of models (m_i, m_j) $(1 \leq i, j \leq n)$ such that:

- $F_i = F_j$ (and therefore $c_i = c_j$)
- $\llbracket p_i \rrbracket_{m_i}$ and $\llbracket p_j \rrbracket_{m_j}$ (are true).
- $\llbracket r_i \rrbracket_{m_i} \neq \llbracket r_j \rrbracket_{m_j}$ and $\forall k \in [1, c_i] . \llbracket t_{i,k} \rrbracket_{m_i} = \llbracket t_{j,k} \rrbracket_{m_j}$.

Algorithm 1: For Generating Witness M to Functional Unrealizability of SGE $\mathcal{E}$.

```
   Input: $\mathcal{E} = \{p_i \Rightarrow l_i = r_i\}_{1 \le i \le n}$
1  $M \leftarrow \emptyset$;
2  forall $1 \le j \le i \le n$ do
3      $F_i, (t_{i,1}, \dots, t_{i,c_i}) \leftarrow$ FRAME($l_i$);
4      $F_j, (t_{j,1}, \dots, t_{j,c_j}) \leftarrow$ FRAME($l_j$);
5      if $F_i = F_j$ then
6          $p'_j, r'_j, t'_{j,1}, \dots, t'_{j,c_j} \leftarrow$
             RENAME($p_j, r_j, t_{j,1}, \dots, t_{j,c_j}$);
7          $\mu \leftarrow$ SOLVE($p_i \land p'_j \land r_i \neq r'_j \bigwedge_{1 \le k \le c_i} t_{i,k} = t'_{j,k}$);
8          if $\mu$ is a satisyfing assignment then
9              $M \leftarrow M \cup \{$PROJ($\mu, FV(l_i, r_i)$),
                 PROJ($\mu, FV(l_j, r_j)$)$\}$;
10 return $M$
```

It is straightforward to see that a witness to the functional unrealizability of an SGE is an unrealizability witness in the more general sense (Definition 5.1). Remark that, Definition 6.3 only considers *maximal* frames. This is because one can show that this can be done without loss of generality: if functional unrealizability can be derived from constraints with two arbitrary frames F and F', then it can be derived using maximal frames. The extended version of this paper [30] includes a proof for this fact.

Generating a Witness to Functional Realizability. Algorithm 1 outlines our procedure for generating the functional unrealizability witnesses of Definition 6.3. The algorithm relies on FRAME, that returns a maximal frame, and SOLVE, which is implemented by an SMT query.

Algorithm 1 inspects every pair of constraint indices i, j in the input SGE, including pairs where $i = j$. If the frames F_i and F_j match, the variables of constraint j are given fresh names in order to ensure that variables in each constraint are distinct (even in the case $i = j$). The procedure SOLVE then solves the formula that corresponds to the constraints of Definition 6.3. If that formula has a satisfying assignment, then a new witness has been found. The PROJ function projects the model on the variables of each constraint, resulting in two models: one that assigns values to the free variables of the constraint $p_i \Rightarrow l_i = r_i$ and another for the free variables of $p_j \Rightarrow l_j = r_j$.

Under the assumption that SOLVE is a decision procedure, Algorithm 1 becomes a decision procedure for Definition 6.3. It is important to note that, theoretically, Definition 6.3 may not compute all pairs (m, m') from Defintion 6.1; the extended version of this paper [30] provides an example.

7 Invariant Inference

If the unrealizability witness M is spurious, the set of guards $\mathcal{P}$ in the approximate specification $\mathcal{E}(T, \mathcal{P})$ have to be strengthened. As discussed in Section 5, a spurious unrealizability witness M yields a set C of s-certificates (Definition 5.6). In this section, we discuss how s-certificates are used in the coarsening loop of SE^{2}GIS.

7.1 Classification of S-Certificates

First, the set C of s-certificates is partitioned into two types of s-certificates. Intuitively, the first class captures the cases where the spuriousness is caused by the return values for a function symbol being strictly more limited than otherwise indicated by its return type. The second class captures the cases where the model is spurious due a violation of the type invariant for one of the input values.

Definition 7.1 (s-certificate classification)**.** An s-certificate (m, t) is called:

- an unsatisfiable certificate if $\forall \vec{z} \in FV(t) \cdot \neg(t \ltimes m)$.
- a mistyped certificate if $\exists \vec{z} \in FV(t).t \ltimes m$ and $\forall \vec{z} \in FV(t) \cdot (t \ltimes m \Rightarrow \neg I_\theta(t))$.

Example 7.2. Let $t_2 = Cons(a_2, l), t_3 = Cons(a_3, Cons(a_4, l'))$. Recall Example 5.5, where $f \circ r = length$ gives the length of a list. The spurious witness $m = [a_2 \rightarrow 0, v_l \rightarrow -1]$ for term t_2 yields the s-certificate $c_1 = ([a_2 \rightarrow 0, v_l \rightarrow -1], t_2)$. This is an **unsatisfiable certificate** because there is no valuation of l that satisfies $length(l) = -1$. So, t_2 is not compatible with m.

In the setup of the example in Section 1.1 (last seen in Example 5.7), $f \circ r = min$ returns the smallest item in a list; the type invariant *sorted* asserts that lists are sorted in increasing order. The s-certificate $c_2 = ([a_2 \rightarrow 1, v_l \rightarrow -1], t_2)$ is a **mistyped certificate** because any valuation of a_2 and l that satisfies $a_2 = 1 \land min(l) = -1$ does not satisfy *sorted*; similarly for the mistyped certificate $c_3 = ([a_3 \rightarrow 1, a_4 \rightarrow 0, v_{l'} \rightarrow 2], t_3)$, since a_4 should be greater than a_3. ⌟

The guards $\mathcal{P}$ in the approximate specification $\mathcal{E}(T, \mathcal{P})$ tentatively approximate both types of missing *invariants*, and the classification signals which type of invariant has to be strengthened in the next round. A mistyped certificate triggers the coarsening step presented in Section 7.2.1 that learns a stronger recursion-free approximation of the type invariant I_θ; an unsatisfiable certificate triggers the coarsening step presented in Section 7.2.2 that learns a new invariant about the reference function.

It is straightforward to see that the above partitioning is well-defined: each s-certificate belongs to one of the two classes and the classes are disjoint. We classify the certificates of C by encoding the conditions of Definition 7.1 for each $c \in C$ into an SMT query that is passed to a black-box solver; [30] includes an extended example.

7.2 Learning Invariants

To strengthen the set of guards $\mathcal{P}$, we generate a new set of predicates and strengthen each new predicate's relevant guards in $\mathcal{P}$ by adding the new predicate as a conjunct of the existing one. These new predicates are learned from examples: the *negative* examples are extracted directly from s-certificates and the *positive* examples are generated from incorrect candidates during the learning process.

Algorithm 2 presents the learning routine. It calls on subroutines Negative and Verify, which have different implementations for the two classes of s-certificates. For Synthesize, any synthesis-by-example tool that admits positive and negative examples can be used. The algorithm starts with a fixed set of negative examples extracted from the s-certificates by Negative and iteratively adds positive examples obtained from a failed verification by Verify. The learning algorithm converges when Verify succeeds. In the following, we fully instantiate Algorithm 2 for s-certificates of each type.

Algorithm 2: InferInvariant(c)

```
   Input: $c$ is an s-certificate
1  $pred\ (x_0, ..., x_k) \leftarrow \bot$;
2  $positive \leftarrow \emptyset$ ;
3  $negative \leftarrow \{$ Negative $(c)\}$ ;
4  while $\neg$ Verify($pred$) do
5      $c' \leftarrow$ a counterexample to Verify($pred$) ;
6      $positive \leftarrow positive \cup \{c'\}$ ;
7      $pred \leftarrow$ Synthesize($positive, negative$);
8  return $pred$
```

7.2.1 Learning from Mistyped Certificates.

A mistyped certificate signals that the spurious unrealizability witness cannot correspond to an actual recursive input that satisfies I_θ. Our goal is to strengthen the guards $\mathcal{P}$ according to I_θ to exclude this witness. We construct $\mathcal{P}'$ by accumulating the contributions of each mistyped certificate c in C by calling InferInvariant(c). Let $c = (m, t)$ be a mistyped certificate where $Dom(m) = x_0, ..., x_k$ and p_i is the guard of the equation that is relevant to c in the current approximation.

Negative. The extraction of negative examples is straightforward in this case. The model, being an evaluation, is the negative example; for instance $m = [a \leftarrow 1, v_l \leftarrow -1]$.

Verify. The goal is to strengthen p_i such that the negative example is excluded, but the new guard should be *supported* by the type invariant I_θ. Recall from Definition 4.6 that all $p_i \in \mathcal{P}$ have to satisfy the constraint $\forall \vec{z} \in FV(t) \cdot I_\theta(t) \Rightarrow [\![p_i]\!]^{-1}_{elim}$. Therefore, Verify performs this exact check as an SMT query. When the check fails, its negation — an existential formula $\exists \vec{z} \in FV(t) \cdot \ldots$ — is satisfiable. We can map such a $\vec{z}$ to a unique model m' by directly taking the scalar-type variables of $\vec{z}$ and by applying $(f \circ r)$ to the inductively-typed variables of $\vec{z}$. This m' is what Verify produces as a counterexample and is subsequently added to the synthesis constraints as a positive example.

To illustrate, suppose that we are calling InferInvariant on the s-certificate c_2 of Example 7.2. Our aim is to guess a predicate $pred$ that meets the conditions $\neg pred(1, -1)$ and Verify($pred$). The Verify($pred$) subroutine checks whether $\forall a_2, l \cdot sorted(Cons(a_2, l)) \Rightarrow pred(a_2, min(l))$ holds and, if not, produces a counterexample.

Initially, $pred(a_2, l) = \bot$, which does not satisfy Verify. This incorrect guess may yield the positive example $Cons(1, Elt(2))$. As a result, the next guess for $pred$ must be so that $pred(1, 2)$ holds. If we then guess $pred(a_2, v_l) = a_2 < v_l$, Verify holds and p_2 is subsequently updated to $p_2 \wedge a_2 < v_l$.

7.2.2 Learning from Unsatisfiable Certificates.

In the case of an unsatisfiable certificate, the new learned predicate is a useful invariant of the reference implementation f. First, let us make a helpful observation.

Lemma 7.3 (Unsatisfiable Model). *An s-certificate (m, t) arising from a witness to the functional unrealizability of an approximation of Ψ is an unsatisfiable certificate if and only if*

$$\exists v \in \mathcal{V}_{elim} \cap Dom(m) \cdot \forall t : \theta \cdot (f \circ r)(t) \neq [\![v]\!]_m$$

The satisfiability of a unsatisfiable certificate (m, t) corresponds directly to the question of whether there is an elimination variable whose value under m is not in the image of $f \circ r$. Intuitively, the predicate we would like to learn captures an invariant of the image of $f \circ r$ and adding it to $\mathcal{P}$ amounts to a restriction of $\mathcal{V}_{elim}$ to the image of $f \circ r$. The predicate's domain is then simply one (fresh) variable x which ranges over the return type of f.

Negative. A negative example is any value from m (for an elimination variable) that lies outside the image of $(f \circ r)$. Lemma 7.3 guarantees that each unsatisfiable certificate includes at least one such value, but there may be several such choices. For example, if $m = [a_2 \leftarrow 0, v_l \leftarrow -1], t_2 = Cons(a_2, l)$, and $(f \circ r)$ is *length*, we add the negative example -1 to our constraints.

Verify. Verify($pred$) simply checks that $pred$ is an invariant of the image of $(f \circ r)$, that is, $\forall t : \theta \cdot pred((f \circ r)(t))$ is checked by querying an SMT solver. When this check fails, there must be some term t such that $\neg pred((f \circ r)(t))$. Then, Verify returns $c' = (f \circ r)(t)$ to be added to the set of positive example.

7.3 Correctness of SE^{2}GIS

Algorithm 2 has a weak progress guarantee which ensures that an unrealizability witness from any coarsening round will not appear in the next round.

Proposition 7.4. *Let M be a spurious unrealizability witness for $\mathcal{E}(T, \mathcal{P})$ and $\mathcal{P}'$ be a strengthening of $\mathcal{P}$ resulting from Algorithm 2 with s-certificates extracted from M. Then, M is not an unrealizability witness for $\mathcal{E}(T, \mathcal{P}')$.*

It is straightforward to generalize the above proposition, through an induction argument on the number of coarsening rounds, to hold for an arbitrary number of coarsening rounds between $\mathcal{E}(T, \mathcal{P})$ and $\mathcal{E}(T, \mathcal{P}')$. In [10], similar weak progress results are presented for the refinement loop. It remains to show that any arbitrary alternation of refinement and coarsening loops satisfies a similar weak progress property.

Theorem 7.5 (Progress of SE^{2}GIS)**.** *Let $\mathcal{E}(T, \mathcal{P})$ and $\mathcal{E}(T', \mathcal{P}')$ be two approximations from two arbitrary rounds of the SE^{2}GIS algorithm, where $\mathcal{E}(T, \mathcal{P})$ appears in an earlier round. We have:*

- *If $\mathcal{E}(T, \mathcal{P})$ is unrealizable with a spurious witness M which is used in the coarsening loop, then M does not witness unrealizability of $\mathcal{E}(T', \mathcal{P}')$.*
- *If $\mathcal{E}(T, \mathcal{P})$ is realizable with a solution s which is used in the refinement loop, then s is not a solution to $\mathcal{E}(T', \mathcal{P}')$.*

The theorem is not surprising but the interaction between the two loops is not straightforward, hence the proof appears in [30]. Algorithm 2 is similarly guaranteed to return the correct result, simply by relying on the soundness of its Verify subroutine. Based on the soundness result for the refinement loop from [10], we can conclude:

Theorem 7.6 (Soundness of SE^{2}GIS)**.** *If SE^{2}GIS outputs a witness for unrealizability or a solution, then they are valid.*

8 Experimental Results

The SE^{2}GIS algorithm is implemented as part of the tool Synduce [10, 29]. Synduce is written in OCaml [24] and accepts OCaml programs as inputs. We use syntax extensions to identify the different components of the specification. The tool interfaces with solvers using the SMT-LIB standard [**?**] and makes syntax-guided synthesis queries via the SyGuS standard [35]. In our experiments, we use CVC4 [4] version 1.8 for SMT queries when support for induction is required, and otherwise Z3 [9] version 4.8.10. CVC4 is also used for syntax-guided synthesis.

The SMT calls for invariant inference, which are all implicitly queries of the form $\forall t : \theta \cdot p(t)$ for some predicate p, are implemented as parallel calls to two solver instances. The first instance attempts to prove $\forall t : \theta \cdot p(t)$ by induction. The second does a bounded check of its negation $(\exists t : \theta \cdot \neg p(t))$ by unrolling bounded symbolic terms of type θ up to a fixed depth.

8.1 Experimental Setup

The goal of our experimentation is the evaluation of our two main contributions. We investigate the following questions:
Q1: How well does SE^{2}GIS work for recursion synthesis?
Q2: How well does our unrealizability checker (and witness generator) work, independent of the SE^{2}GIS context?

Due to differences in problem setup, Synduce cannot be compared directly against any of the existing synthesis recursion tools. To evaluate the additive value of the novel ideas of the SE^{2}GIS algorithm, we built two baseline variations as described below.

Symbolic CEGIS. Much of the innovation in SE^{2}GIS has been centred around taking full advantage of *partial bounding*. To support our decision to use partial bounding, we have two baseline versions of Synduce called SEGIS and SEGIS+UC that forgo partial bounding in favour of the classic full bounding for synthesis. **SEGIS** performs a symbolic CEGIS loop using only bounded terms, in the style of the baseline of [10]. In this case, there is no need to infer invariants, since bounded versions of invariants are effectively present in the fully bounded approximate specification.

SEGIS+UC is an extension of SEGIS that has access to our unrealizability checker and witness generator. SEGIS+UC uses fully bounded approximate specifications, but can produce unrealizability outcomes. Experimentation with SEGIS+UC lets us isolate the effectiveness of our unrealizability checker in a neutral context. Moreover, comparing SEGIS+UC against SE^{2}GIS over unrealizable benchmarks isolates the impact of partial bounding for detection of unrealizability.

Benchmarks. We evaluated our implementation on a set of 140 benchmarks that cover a wide range of recursive function synthesis problems. We devised these by drawing standard examples of recursive functions from the literature and textbooks. Some of our benchmarks are variations on the benchmarks of [10], to which we have added type invariants and modified the skeletons so that invariants are required in order for the synthesis problem to be solvable.

Our benchmarks operate on 8 distinct[4] recursive data types and 18 type invariants. These include data types such as lists and trees with constraints on *ordering* (sorted lists, unimodal lists, constant lists, and binary search trees), *structure* (balanced trees, symmetric trees, perfect trees, and trees with an empty subtree), *contents* (positive elements, distinct elements, and even or odd elements), and *auxiliary data* (memoized sum, max, min, and number of children). We use 8 different *representation functions* to map these types.

Our benchmark set includes 67 different reference functions and 20 target recursion skeletons. The reference functions used are straightforward implementations of commonly used algorithms. Each reference function can instantiate a distinctly new problem when combined with different type invariants and target recursion skeletons. Target recursion skeletons represent programs that have a variety of desirable features, such as parallelism or better time complexity. A significant number of our benchmarks aim at synthesizing more *efficient traversals* of a data structure.

[4]We do not count small differences in the base constructor of the datatypes or added data fields for memoization as differences.

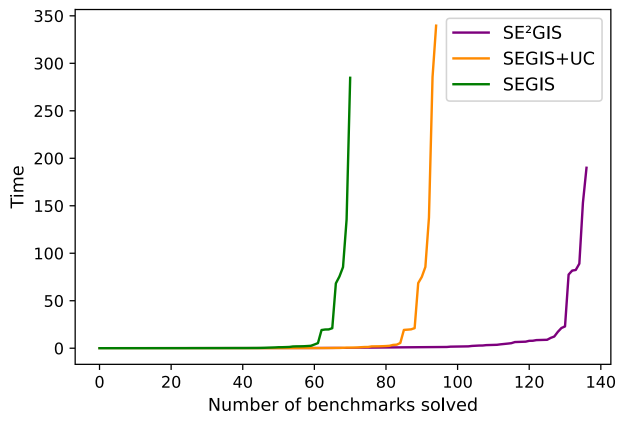


Figure 4. Comparison based on the number of solved benchmarks.

The example discussed in Section 2 is a fair representative of this set. Others involve synthesizing efficient implementations using memoized data in nodes or performing binary search over data structure satisfying interesting invariants. Our tool can also be used to synthesize *divide-and-conquer parallelism.*

We also include 45 *unrealizable benchmarks* to evaluate the efficacy of our unrealizability technique. The majority of these benchmarks are variations of the realizable benchmarks from the set, in which some parts have been modified to make it unrealizable, in the spirit of the example from Section 2.

8.2 Results

Our experiments were run on a laptop with an Intel Core i7-8750H 6-core processor and 32 GB of RAM running Ubuntu 21.04. Each benchmark is run 10 times and the resulting runtimes are averaged. The timeout is 400 seconds. An extended version of the results presented here appears in [30].

The quantile plot in Figure 4 compares SE^{2}GIS, SEGIS and SEGIS+UC based on how many benchmarks, from a total of 140, each can solve. The vertical axis is time (in seconds) taken to solve the corresponding benchmark. The set of all such times are displayed in non-decreasing order for each algorithm. The precise count of benchmarks solved by each algorithm is listed below.

	SE^{2}GIS	SEGIS+UC	SEGIS
Realizable	93	70	70
Unrealizable	44	25	0
Total	137	95	70

Other highlights from the detailed results are: (1) SE^{2}GIS solves the easier benchmarks in one alternation between the refinement and the coarsening loops, but does more alternations for the more complex benchmarks, and (2) in 70% of the cases, the inferred invariants are proved correct by induction, and the rest are checked for bounded inputs.

The scatter plot in Figure 5 compares the running times of SE^{2}GIS and SEGIS+UC for the benchmarks that do not time out in either method. For the realizable instances that were solved by both methods, SEGIS+UC is faster than SE^{2}GIS in 60% of the cases. This is due to the tension between the complexity of the required invariants and that of the solution for the unknowns. When the solution is syntactically very simple, SEGIS has a higher chance of finding it faster, mainly by pure luck, while SE^{2}GIS has to spend a lot of time inferring missing complex invariants. In contrast, partial bounding has the biggest impact when the solution is complex and the invariants are simple. In one extreme case, (see [30]), SE^{2}GIS times out because invariant inference diverges.

SE^{2}GIS and SEGIS+UC can easily complement each other in a portfolio version of Synduce, which runs both algorithms in parallel, and waits for the first result.

Unrealizability Checker. Whenever Synduce declares a problem unrealizable, it is provably unrealizable. This is in contrast to solutions to realizable instances that just pass a bounded verification check in most synthesis tools. Unsurprisingly, all the benchmarks solved by SEGIS+UC but not by SEGIS are unrealizable benchmarks. This difference is precisely the contribution of our unrealizability solver in a neutral context. SE^{2}GIS solves more unrealizability benchmarks than SEGIS+UC, which demonstrates that partial bounding additionally contributes to unrealizability outcomes as well as it does to synthesis of solutions in realizable instances. In Figure 5, over the mutually solved unrealizability instances, SE^{2}GIS is faster in 50% of cases. SEGIS+UC performs best for unrealizable instances where unrealizability is provable with a very shallow level of bounding. Otherwise, SE^{2}GIS is more reliable.

Invariants Synthesized. In 79 of the 137 benchmarks, Synduce infers invariants. The following table lists the number of benchmarks for which an invariant on the reference implementation (Section 7.2.2) or the input datatype (Section 7.2.1) is inferred, categorized by the realizability of the instances.

	Reference	Datatype	Total
Realizable	10	57	67
Unrealizable	0	12	12
Total	10	69	79

For unrealizable benchmarks, both spurious and non-spurious witnesses are generated during the process. Synduce only learns invariants when the witness to the approximate specification happens to be spurious. Therefore, the need for invariant inference in these cases partly depends on whether Synduce gets lucky with the witness draw or not. When the input datatype invariant is present and matters (84 out of 95 benchmarks), Synduce has to *synthesize* an invariant only when partial bounding is used. For bounded inputs, the given invariant of the datatype can be used directly.

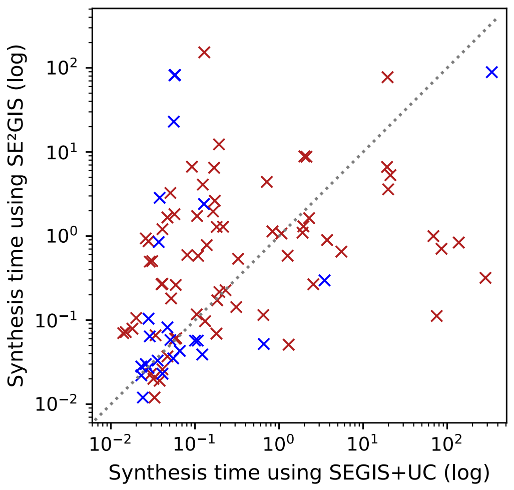


Figure 5. Comparing the running times (in seconds) of SE^{2}GIS and SEGIS+UC in logarithmic scale. Blue points are unrealizable and red points are realizable benchmarks.

Limitations. We have already discussed how learning complex invariants can be the Achilles heel of SE^{2}GIS. Another point of failure for SE^{2}GIS is that the lightweight method for producing functional unrealizability witnesses can theoretically fail; a discussion on this issue appeared in Section 6. In practice, this lightweight method works remarkably well and did not fail for any of our benchmarks, but the theoretical possibility exists. It would be interesting to see if more expressive witnesses produced out of this step can help ameliorate some of the problems when learning complex invariants. Finally, the synthesis step for SGEs (done by SyGuS in our implementation) can become a bottleneck, even in unrealizable instances. In some cases, it becomes the cause of a timeout, even if the problem is unrealizable; in order to derive unrealizability, SE^{2}GIS must first complete a refinement step (see [30] for a concrete example).

9 Related Work

In this section, we focus on work related to the synthesis of recursive functions only and refer the reader to [14] for a broader survey of program synthesis techniques.

Recursive Function Synthesis. Synthesis of recursive functions dates back to inductive techniques used to synthesize recursive programs from input/output examples [38], which has recently been further extended in [17, 18]. Types have been extensively used to direct the search for a program [11, 12, 32, 34]. λ^2 [11], Myth [32], Myth2 [12] and SMyth [25] accept input/output examples as specifications, which are a good choice to specify simple recursive functions with little data manipulation. In contrast, we target more sophisticated synthesis tasks such as maximum sums or inclusion checking with non-trivial predicates. SynQuid [34] and ReSyn [20] take refinement types as specifications. Type-based approaches work very well within the expressivity of refinement-types as specifications, but refinement types cannot express constraints for all desired synthesis tasks. These techniques, and others like Escher [1], require the user to provide the components used as building blocks of recursion synthesis. In contrast, we focus on synthesizing these components when a recursive skeleton is provided. As such, the two sets of methods are complementary.

Leon [19], the older version of Synduce from [10], and Burst [27] all accept specifications that are close to ours. Neither tool handles unrealizability. Burst [27] accepts multiple forms of specifications (input/output, reference implementations, and logical specifications). However, we cannot directly encode our problem into a specification for Burst, notably because we cannot specify type invariants. Leon [19] is the technique that accepts specifications that are closest in form to ours, since one can write specifications with functional equivalence constraints. We can also encode recursion skeletons, but Leon seems to lack the mechanisms for reasoning about unknowns within a recursive function. We did not succeed in synthesizing solutions, even for simple benchmarks. The older version of Synducefrom [10] can handle some of our benchmarks, by asking the user to input the missing reference function invariants. It cannot handle the benchmarks that rely on type invariants.

On a technical front, we borrow the idea of *partial bounding* from [10]. This idea and our invariant inference routine is similar to specification strengthening in Burst [27].

Invariant Inference. Example-driven [28, 33] and formula-driven [39] invariant and lemma inference has been used in program verification. Theory exploration techniques [3, 8, 36] aim at generating a collection of lemmas pertaining to a set of possibly recursive components by eagerly proving lemmas before they are known to be needed. Our technique, on the other hand, is parsimonious and generates invariants only when they are required in order to rule out a spurious unrealizability witness.

In the 30% of the cases where Synduce fails to prove an inferred invariant correct by induction, it currently uses bounded checks to verify it. Theory exploration techniques may be useful to prove these remaining 30% of inferred invariants by supplying helper lemmas.

Unrealizability. Traditionally, program synthesis, especially syntax-guided synthesis, has been biased towards finding solutions and not proving unrealizability. Unrealizability is undecidable [7] for syntax-guided synthesis, but, recently, approximate techniques [15, 16, 26] for checking unrealizability of such instances have been proposed. There are restricted instances where unrealizability (and realizability) is decidable, notably for uninterpreted functions [23] and, more generally, finite variable logics [22]. We found the reliance of these techniques on a specific grammar to be limiting for our context. Our technique is lightweight and can be directly integrated as a preprocessing check for SyGuS inputs, an existing standard.

References

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification* (Berlin, Heidelberg, 2013) *(Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *2013 Formal Methods in Computer-Aided Design* (Portland, OR, 2013-10). 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

[3] Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. 2017. Congruence Closure with Free Variables. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2017) *(Lecture Notes in Computer Science)*, Axel Legay and Tiziana Margaria (Eds.). 214–230. https://doi.org/10.1007/978-3-662-54580-5_13

[4] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification* (Berlin, Heidelberg, 2011) *(Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

[5]]smtlib Clark Barrett, Pascal Fontaine, and Aaron Stump. [n. d.]. The SMT-LIB Standard. ([n. d.]), 104.

[6] Nikolaj Bjorner and Mikolas Janota. 2016. Playing with Quantified Satisfaction *(LPAR-20)*. 15–1. https://doi.org/10.29007/vv21

[7] Benjamin Caulfield, Markus N. Rabe, Sanjit A. Seshia, and Stavros Tripakis. 2016. *What's Decidable about Syntax-Guided Synthesis?* arXiv:1510.08393 [cs]

[8] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2013. Automating Inductive Proofs Using Theory Exploration. In *Automated Deduction – CADE-24*, Maria Paola Bonacina (Ed.). Lecture Notes in Computer Science, Vol. 7898. 392–406. https://doi.org/10.1007/978-3-642-38574-2_27

[9] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008) *(Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[10] Azadeh Farzan and Victor Nicolet. 2021. Counterexample-Guided Partial Bounding for Recursive Function Synthesis. In *Computer Aided Verification* (Virtual, USA, 2021-07-20) *(Lecture Notes in Computer Science, Vol. 1)*. 23.

[11] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2015-06-03) *(PLDI '15)*. 229–239. https://doi.org/10.1145/2737924.2737977

[12] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-Directed Synthesis: A Type-Theoretic Interpretation. 51, 1 (2016), 802–815. https://doi.org/10.1145/2914770.2837629

[13] Yeting Ge and Leonardo Moura. 2009. Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In *Proceedings of the 21st International Conference on Computer Aided Verification* (Grenoble, France, 2009-06-23) *(CAV '09)*. 306–320. https://doi.org/10.1007/978-3-642-02658-4_25

[14] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. *Program Synthesis.* Number 4.2017, 1-2 in Foundations and Trends in Programming Languages.

[15] Qinheping Hu, Jason Breck, John Cyphert, Loris D'Antoni, and Thomas Reps. 2019. Proving Unrealizability for Syntax-Guided Synthesis. In *Computer Aided Verification* (Cham, 2019-07-13) *(Lecture Notes in Computer Science)*, Isil Dillig and Serdar Tasiran (Eds.). 335–352. https://doi.org/10.1007/978-3-030-25540-4_18

[16] Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas Reps. 2020. Exact and Approximate Methods for Proving Unrealizability of Syntax-Guided Synthesis Problems. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London UK, 2020-06-11). 1128–1142. https://doi.org/10.1145/3385412.3385979

[17] Susumu Katayama. 2012. An Analytical Inductive Functional Programming System That Avoids Unintended Programs. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation* (New York, NY, USA, 2012-01-23) *(PEPM '12)*. 43–52. https://doi.org/10.1145/2103746.2103758

[18] Emanuel Kitzelmann and Ute Schmid. 2006. Inductive Synthesis of Functional Programs: An Explanation Based Generalization Approach. 7, 7 (2006), 26.

[19] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo Recursive Functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis Indiana USA, 2013-10-29). 407–426. https://doi.org/10.1145/2509136.2509555

[20] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-Guided Program Synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix AZ USA, 2019-06-08). 253–268. https://doi.org/10.1145/3314221.3314602

[21] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based Model Checking for Recursive Programs. 48, 3 (2016), 175–205. https://doi.org/10.1007/s10703-016-0249-4

[22] Paul Krogmeier and P. Madhusudan. 2021. *Learning Formulas in Finite Variable Logics.* arXiv:2111.03534 [cs]

[23] Paul Krogmeier, Umang Mathur, Adithya Murali, P. Madhusudan, and Mahesh Viswanathan. 2020. Decidable Synthesis of Programs with Uninterpreted Functions. In *Computer Aided Verification* (Cham, 2020) *(Lecture Notes in Computer Science)*, Shuvendu K. Lahiri and Chao Wang (Eds.). 634–657. https://doi.org/10.1007/978-3-030-53291-8_32

[24] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2020. The OCaml System Release 4.11: Documentation and User's Manual.

[25] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. 4 (2020), 109:1–109:29. Issue ICFP. https://doi.org/10.1145/3408991

[26] P. Madhusudan, Umang Mathur, Shambwaditya Saha, and Mahesh Viswanathan. 2018. *A Decidable Fragment of Second Order Logic With Applications to Synthesis.* https://doi.org/10.4230/LIPIcs.CSL.2018.31 arXiv:1712.05513 [cs]

[27] Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. In *Proceedings of the 49th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2022), Vol. 1. 31.

[28] Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. 2020. Data-Driven Inference of Representation Invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London UK, 2020-06-11). 1–15. https://doi.org/10.1145/3385412.3385967

[29] Victor Nicolet and Danya Lette. 2022. *Synduce.*

[30] Victor Nicolet, Danya Lette, and Azadeh Farzan. 2022. Recursion Synthesis with Unrealizability Witnesses (Extended Version).

[31] C.-H. Luke Ong and Steven J. Ramsay. 2011. Verifying Higher-Order Functional Programs with Pattern-Matching Algebraic Data Types. In

Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (New York, NY, USA, 2011-01-26) *(POPL '11)*. 587–598. https://doi.org/10.1145/1926385.1926453

[32] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2015-06-03) *(PLDI '15)*. 619–630. https://doi.org/10.1145/2737924.2738007

[33] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-Driven Precondition Inference with Learned Features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2016-06-02) *(PLDI '16)*. 42–56. https://doi.org/10.1145/2908080.2908099

[34] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2016-06-02) *(PLDI '16)*. 522–538. https://doi.org/10.1145/2908080.2908093

[35] Mukund Raghothaman, Andrew Reynolds, and Abhishek Udupa. 2019. The SyGuS Language Standard Version 2.0. (2019), 22.

[36] Eytan Singher and Shachar Itzhaky. 2021. Theory Exploration Powered by Deductive Synthesis. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Lecture Notes in Computer Science, Vol. 12760. 125–148. https://doi.org/10.1007/978-3-030-81688-9_6

[37] Armando Solar-Lezama. 2013. Program Sketching. 15, 5 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

[38] Phillip D. Summers. 1977. A Methodology for LISP Program Construction from Examples. 24, 1 (1977), 161–175. https://doi.org/10.1145/321992.322002

[39] Weikun Yang, Grigory Fedyukovich, and Aarti Gupta. 2019. Lemma Synthesis for Automating Induction over Algebraic Data Types. In *Principles and Practice of Constraint Programming* (Cham, 2019) *(Lecture Notes in Computer Science)*, Thomas Schiex and Simon de Givry (Eds.). 600–617. https://doi.org/10.1007/978-3-030-30048-7_35