Resource-Guided Program Synthesis

Tristan Knoth
University of California, San Diego
USA
tknoth@ucsd.edu

Nadia Polikarpova
University of California, San Diego
USA
npolikarpova@ucsd.edu

Di Wang
Carnegie Mellon University
USA
diw3@cs.cmu.edu

Jan Hoffmann
Carnegie Mellon University
USA
jhoffmann@cmu.edu

Abstract
This article presents resource-guided synthesis, a technique for synthesizing recursive programs that satisfy both a functional specification and a symbolic resource bound. The technique is type-directed and rests upon a novel type system that combines polymorphic refinement types with potential annotations of automatic amortized resource analysis. The type system enables efficient constraint-based type checking and can express precise refinement-based resource bounds. The proof of type soundness shows that synthesized programs are correct by construction. By tightly integrating program exploration and type checking, the synthesizer can leverage the user-provided resource bound to guide the search, eagerly rejecting incomplete programs that consume too many resources. An implementation in the resource-guided synthesizer ReSyn is used to evaluate the technique on a range of recursive data structure manipulations. The experiments show that ReSyn synthesizes programs that are asymptotically more efficient than those generated by a resource-agnostic synthesizer. Moreover, synthesis with ReSyn is faster than a naive combination of synthesis and resource analysis. ReSyn is also able to generate implementations that have a constant resource consumption for fixed input sizes, which can be used to mitigate side-channel attacks.

CCS Concepts • Software and its engineering → Automatic programming; • Theory of computation → Automated reasoning.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

1 Introduction
In recent years, program synthesis has emerged as a promising technique for automating low-level aspects of programming [24, 61, 66]. Synthesis technology enables users to create programs by describing desired behavior with input-output examples [18–20, 22, 47, 60, 70, 71], natural language [72], and partial or complete formal specifications [35, 39, 51, 52, 63]. If the input is a formal specification, synthesis algorithms can not only create a program but also a proof that the program meets the given specification [39, 51, 52, 63].

One of the greatest challenges in software development is to write programs that are not only correct but also efficient with respect to memory usage, execution time, or domain specific resource metrics. For this reason, automatically optimizing program performance has long been a goal of synthesis, and several existing techniques tackle this problem for low-level straight-line code [9, 49, 50, 56, 57] or add efficient synchronization to concurrent programs [11, 12, 21, 28]. However, the developed techniques are not applicable to recent advances in the synthesis of high-level looping or recursive programs manipulating custom data structures [22, 35, 39, 47, 51, 52]. These techniques lack the means to analyze and understand the resource usage of the synthesized programs. Consequently, they cannot take into account the program’s efficiency and simply return the first program that arises during the search and satisfies the functional specification.

In this work, we study the problem of synthesizing high-level recursive programs given both a functional specification of a program and a bound on its resource usage. A naive solution would be to first generate a program using conventional program synthesis and then use existing automatic static resource analyses [15, 32, 48] to check whether its resource usage satisfies the bound. Note, however, that for recursive

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
In addition to the design of Re², we prove the soundness of the type system with respect to a small-step cost semantics. In the formal development, we focus on a simple call-by-value functional language with Booleans and lists, where type refinements are restricted to linear inequalities over lengths of lists. However, we structure the formal development to emphasize that Re² can be extended with user-defined data types, more expressive refinements, or non-linear potential annotations. The proof strategy itself is a contribution of this paper. The type soundness of the logical refinement part of the system is inspired by TiML [48]. The main novelty is the soundness proof of the potential annotations using a small-step cost semantics instead of RaML’s big-step evaluation semantics.

Type-Driven Synthesis with Re² The second contribution of this paper is a resource-guided synthesis algorithm based on Re². In Sec. 4, we first develop a system of synthesis rules that prescribe how to derive well-typed programs from Re² types, and prove its soundness wrt. the Re² type system. We then show how to algorithmically derive programs using a combination of backtracking search and constraint solving. In particular this requires solving a new form of constraints we call resource constraints, which are constrained linear inequalities over unknown numeric refinement terms. To solve resource constraints, we develop a custom solver based on counter-example guided inductive synthesis [62] and SMT [17].

The ReSyn Synthesizer The third contribution of this paper is the implementation and experimental evaluation of the first resource-aware synthesizer for recursive programs. We implemented our synthesis algorithm in a tool called ReSyn, which takes as input (1) a goal type that specifies the logical refinements and resource requirements of the program, and (2) types of components (i.e. library functions that the program may call). ReSyn then synthesizes a program that provably meets the specification (assuming the soundness of components).

To evaluate the scalability of the synthesis algorithm and the quality of the synthesized programs, we compare ReSyn with baseline Synquid on a variety of general-purpose data structure operations, such as eliminating duplicates from a list or computing common elements between two lists. The evaluation (Sec. 5) shows that ReSyn is able to synthesize programs that are asymptotically more efficient than those generated by Synquid. Moreover, the tool scales better than a naive combination of synthesis and resource analysis.

2 Background and Overview

This section provides the necessary background on type-driven program synthesis (Sec. 2.1) and automatic resource analysis (Sec. 2.2). We then describe and motivate their combination in Re² and showcase novel features of the type system (Sec. 2.3). Finally, we demonstrate how Re² can be used for resource-guided synthesis (Sec. 2.4).
Type-driven program synthesis [51] is a technique for automatically generating functional programs from high-level specifications expressed as refinement types [41, 54]. For example, a programmer might describe a function that computes the common elements between two lists using the following type signature:

\[
\text{common} : \text{List} a \rightarrow \text{List} a \rightarrow \{ \nu : \text{List} a \mid \text{elems } \nu = \text{elems } l1 \cap \text{elems } l2 \}
\]

Here, the return type of \text{common} is refined with the predicate \text{elems } \nu = \text{elems } l1 \cap \text{elems } l2, which restricts the set of elements of the output list \nu to be the intersection of the sets of elements of the two arguments. Here \text{elems } is a user-defined logic-level function, also called measure [38, 68]. In addition to the synthesis goal above, the synthesizer takes as input a component library: signatures of data constructors and functions it can use. In our example, the library includes the list constructors \text{Nil} and \text{Cons} and the function

\[
\text{member} : x : a \rightarrow l : \text{List} a \rightarrow \{ \text{Bool} \mid \nu = \{ x \in \text{elems } l \}\}
\]

which determines whether a given value is in the list. Given this goal and components, the type-driven synthesizer SYNQUID [51] produces an implementation of \text{common} in Fig. 1.

The Synthesis Mechanism  Type-driven synthesis works by systematically exploring the space of programs that can be built from the component library and validating candidate programs against the goal type using a variant of liquid type inference [54]. To validate a program against a refinement type, liquid type inference generates a system of subtyping constraints over refinement types. The subtyping constraints are then reduced to implications between refinement predicates. For example, checking \text{common } xs l2 in line 3 of Fig. 1 against the goal type reduces to validating the following implication:

\[
(\text{elems } l1 = \{ x \mid \text{elems } x : a \}) \land (x \notin \text{elems } l2) \land (\text{elems } \nu = \text{elems } xs \land \text{elems } l2) \implies \text{elems } \nu = \text{elems } l1 \cap \text{elems } l2
\]

Since this formula belongs to a decidable theory of uninterpreted functions and arrays, its validity can be checked by an SMT solver [17]. In general, the generated implications may contain unknown predicates. In this case, type inference reduces to a system of constrained horn clauses [6], which can be solved via predicate abstraction.

2.1 Type-Driven Program Synthesis

For example, a programmer might describe a function that reduces to a system of

\[
\nu = \text{elems } l \land \text{elems } x : a \land (x \notin \text{elems } l2) \land (\text{elems } \nu = \text{elems } xs \land \text{elems } l2) \implies \text{elems } \nu = \text{elems } l1 \cap \text{elems } l2
\]

which determines whether a given value is in the list. Given this goal and components, the type-driven synthesizer SYNQUID [51] produces an implementation of \text{common} in Fig. 1.

The Synthesis Mechanism  Type-driven synthesis works by systematically exploring the space of programs that can be built from the component library and validating candidate programs against the goal type using a variant of liquid type inference [54]. To validate a program against a refinement type, liquid type inference generates a system of subtyping constraints over refinement types. The subtyping constraints are then reduced to implications between refinement predicates. For example, checking \text{common } xs l2 in line 3 of Fig. 1 against the goal type reduces to validating the following implication:

\[
(\text{elems } l1 = \{ x \mid \text{elems } x : a \}) \land (x \notin \text{elems } l2) \land (\text{elems } \nu = \text{elems } xs \land \text{elems } l2) \implies \text{elems } \nu = \text{elems } l1 \cap \text{elems } l2
\]

Since this formula belongs to a decidable theory of uninterpreted functions and arrays, its validity can be checked by an SMT solver [17]. In general, the generated implications may contain unknown predicates. In this case, type inference reduces to a system of constrained horn clauses [6], which can be solved via predicate abstraction.

Cons x xs → if ¬(member x l2)
then common xs l2
else Cons x (common xs l2)

Figure 1. Synthesized program that computes common elements between two lists

2.2 Automatic Amortized Resource Analysis

To reason about the resource usage of programs we take inspiration from automatic amortized resource analysis (AARA) [31, 33, 34, 37]. AARA is a state-of-the-art technique for automatically deriving symbolic resource bounds on functional programs, and is implemented for a subset of OCaml in Resource Aware ML (RaML) [30, 33]. For example, RaML is able to automatically derive the worst-case bound \(2m + n \cdot m\) on the

\[
\text{SCons} x \rightarrow \text{match l2 with SNil} \rightarrow \text{Nil}
\]

\[
\text{SCons} y ys \rightarrow
\]

\[
\text{if } x < y \text{ then } \text{common }' x l2
\]

\[
\text{else if } y < x \text{ then } \text{common }' l1 ys
\]

\[
\text{else } \text{Cons } x (\text{common }' x ys)
\]

Figure 2. A more efficient version of the program in Fig. 1 for sorted lists

Synthesis and Program Efficiency  The program in Fig. 1 is correct, but not particularly efficient: it runs roughly in time \(n \cdot m\), where \(m\) is the length of \(l1\) and \(n\) is the length of \(l2\), since it calls the \text{member} function (a linear scan) for every element of \(l1\). The programmer might realize that keeping the input lists sorted would enable computing common elements in linear time by scanning the two lists in parallel. To communicate this intent to the synthesizer, they can define the type of (strictly) sorted lists by augmenting a traditional list definition with a simple refinement:

\[
\text{data SList a where SNil :: SList a}
\]

\[
\text{SCons} :: x : a \rightarrow xs : \text{SList} a \rightarrow \text{SList} a
\]

This definition says that a sorted list is either empty, or is constructed from a head element \(x\) and a tail list \(xs\), as long as \(xs\) is sorted and all its elements are larger than \(x\). Given an updated synthesis goal (where \(selems\) is a version of \text{elems} for \text{SList})

\[
\text{common'} : l1 : \text{SList} a \rightarrow \{ \nu : \text{List} a \mid \text{lems } \nu = \text{lems } l1 \cap \text{lems } l2 \}
\]

and a component library that includes \text{List}, \text{SList}, and \(<\) (but not \text{member}!), SYNQUID can synthesize an efficient program shown in in Fig. 2.

However, if the programmer leaves the function \text{member} in the library, SYNQUID will synthesize the inefficient implementation in Fig. 1. In general, SYNQUID explores candidate programs in the order of size and returns the first one that satisfies the goal refinement type. This can lead to suboptimal solutions, especially as the component library grows larger and allows for many functionally correct programs. To avoid inefficient solutions, the synthesizer has to be aware of the resource usage of the candidate programs.
number of recursive calls for the function `common` and \(m+n\) for `common'\(^3\).

**Potential Annotations** AARA is inspired by the potential method for manually analyzing the worst-case cost of a sequence of operations [64]. It uses annotated types to introduce potential functions that map program states to non-negative numbers. To derive a bound, we have to statically ensure that the potential at every program state is sufficient to cover the cost of the next transition and the potential of the following state. In this way, we ensure that the initial potential is an upper bound on the total cost.

The key to making this approach effective is to closely integrate the potential functions with data structures [34, 37]. For instance, in RaML, the type `L(int)` stands for a list that contains one unit of potential for every element. This type defines the potential function \(\phi(l : L(int)) = 1 \cdot |l|\). The potential can be used to pay for a recursive call (or, in general, cover resource usage) or to assign potential to other data structures.

**Bound Inference** Potential annotations can be derived automatically by starting with a symbolic type derivation that contains fresh variables for the potential annotations of each type, and applying syntax directed type rules that impose local constraints on the annotations. The integration of data structures and potential ensures that these constraints are linear even for polynomial potential annotations.

### 2.3 Bounding Resources with \(Re^2\)

To reason about resource usage in type-driven synthesis, we integrate AARA’s potential annotations and refinement types into a novel type system that we call \(Re^2\). In \(Re^2\), a refinement type can be annotated with a potential term \(\phi\) of numeric sort, which is drawn from the same logic as refinements. Intuitively, the type \(R^\phi\) denotes values of refinement type \(R\) with \(\phi\) units of potential. In the rest of this section we illustrate features of \(Re^2\) on a series of examples, and delay formal treatment to Sec. 3.

With potential annotations, users can specify that `common` must run in time at most \(m+n\), by giving it the following type signature:

\[
\text{common' :: l1 : SList a}^1 \rightarrow \text{l2 : SList a}^1 \\
\quad \rightarrow \{v : \text{List a} | \text{elems v = selems l1 \cap selems l2}\}
\]

This type assigns one unit of potential to every element of the arguments `l1` and `l2`, and hence only allows making one recursive call per element of each list. Whenever resource annotations are omitted, the potential is implicitly zero: for example, the elements of the result carry no potential.

Our type checker uses the following reasoning to argue that this potential is sufficient to cover the efficient implementation in Fig. 2. Consider the recursive call in line 4, which has a cost of one. Pattern-matching `l1` against `SCons x xs` transfers

\[\text{append :: xs : List a}^1 \rightarrow \text{ys : List a}^1 \rightarrow \{\text{List a} | \text{len v = len xs + len ys}\}\]

\[\text{triple :: l : List Int}^2 \rightarrow \{\text{List n} | \text{len v = 3*(len l)}\}\]

\[\text{tripleSlow :: l : List Int}^3 \rightarrow \{\text{List n} | \text{len v = 3*(len l)}\}\]

\[\text{tripleSlow = \lambda l . append (append l l) l}\]

**Figure 3.** Append three copies of a list. The type of `append` specifies that it returns a list whose length is the sum of the lengths of its arguments. It also requires one unit of potential on each element of the first list. Moreover, `append` has a polymorphic type and can be applied to lists with different element types, which is crucial for type-checking `tripleSlow`.

the potential from `l1` to the binders, resulting in types `x : a` and `xs : SList (a x <\nu>\nu)`.

\[\text{member :: x : a} \rightarrow \lambda : \text{List a} \rightarrow \{\text{Bool} | v = (x \text{ in elems } \lambda)\}\]

(requiring a unit of potential per element of \(\lambda\)), the guard in line 2 consumes all the potential stored in \(l2\); hence the occurrence of `l2` in line 3 has the type `List a`, which is not a subtype of `List a`.

**Dependent Potential Annotations** In combination with logical refinements and parametric polymorphism, this simple extension to the SynQviD type system turns out to be surprisingly powerful. Unlike in RaML, potential annotations in \(Re^2\) can be dependent, i.e. mention program variables and the special variable \(v\). Dependent annotations can encode fine-grained bounds, which are out of reach for RaML. As one example, consider function `range a b` that builds a list of all integers between `a` and `b`; we can express that it takes at most `b−a` steps by giving the argument `b` a type \(\{\text{Int} | v \geq a\}^{b−a}\). As another example, consider insertion into a sorted list `insert x xs`; we can express that it takes at most as many steps as there are elements in `xs` that are smaller than `x`, by giving `xs` the type `SList a^{\text{let}x<\nu>\nu\text{leq}x\text{.1.0}}` (i.e. only assigning potential to elements that are smaller than `x`). These fine-grained bounds are checked completely automatically in our system, by reduction to constraints in SMT-decidable theories.

**Polymorphism** Another source of expressiveness in \(Re^2\) is parametric polymorphism: since potential annotations are

\[\text{append :: xs : List a}^1 \rightarrow \text{ys : List a}^1 \\
\quad \rightarrow \{\text{List a} | \text{len v = len xs + len ys}\}\]

\[\text{triple :: l : List Int}^2 \rightarrow \{\text{List n} | \text{len v = 3*(len l)}\}\]

\[\text{tripleSlow :: l : List Int}^3 \rightarrow \{\text{List n} | \text{len v = 3*(len l)}\}\]

\[\text{tripleSlow = \lambda l . append (append l l) l}\]

**Figure 3.** Append three copies of a list. The type of `append` specifies that it returns a list whose length is the sum of the lengths of its arguments. It also requires one unit of potential on each element of the first list. Moreover, `append` has a polymorphic type and can be applied to lists with different element types, which is crucial for type-checking `tripleSlow`.

the potential from `l1` to the binders, resulting in types `x : a` and `xs : SList (a x <\nu>\nu)`.

\[\text{member :: x : a} \rightarrow \lambda : \text{List a} \rightarrow \{\text{Bool} | v = (x \text{ in elems } \lambda)\}\]

(requiring a unit of potential per element of \(\lambda\)), the guard in line 2 consumes all the potential stored in \(l2\); hence the occurrence of `l2` in line 3 has the type `List a`, which is not a subtype of `List a`.

**Dependent Potential Annotations** In combination with logical refinements and parametric polymorphism, this simple extension to the SynQviD’s type system turns out to be surprisingly powerful. Unlike in RaML, potential annotations in \(Re^2\) can be dependent, i.e. mention program variables and the special variable \(v\). Dependent annotations can encode fine-grained bounds, which are out of reach for RaML. As one example, consider function `range a b` that builds a list of all integers between `a` and `b`; we can express that it takes at most `b−a` steps by giving the argument `b` a type \(\{\text{Int} | v \geq a\}^{b−a}\). As another example, consider insertion into a sorted list `insert x xs`; we can express that it takes at most as many steps as there are elements in `xs` that are smaller than `x`, by giving `xs` the type `SList a^{\text{let}x<\nu>\nu\text{leq}x\text{.1.0}}` (i.e. only assigning potential to elements that are smaller than `x`). These fine-grained bounds are checked completely automatically in our system, by reduction to constraints in SMT-decidable theories.

**Polymorphism** Another source of expressiveness in \(Re^2\) is parametric polymorphism: since potential annotations are
attached to types, type polymorphism gives us resource polymorphism for free. Consider two functions in Fig. 3, \texttt{triplet} and \texttt{tripl slow}, which implement two different ways to append a list \( \ell \) to two copies of itself. Both of them make use of a component function append, whose type indicates that it makes a linear traversal of its first argument. Intuitively, \texttt{triplet} is more efficient than \texttt{tripl slow} because in the former both calls to append traverse a list of length \( n \), whereas in the latter the outer call traverses a list of length \( 2 \cdot n \). This difference is reflected in the signatures of the two functions: \texttt{tripl slow} requires three units of potential per list element, while \texttt{triplet} only requires two.

Checking that \texttt{tripl slow} satisfies this bound is somewhat nontrivial because the two applications of append must have different types: the outer application must return \( \text{List} \ \text{Int} \), while the inner application must return \( \text{List} \ \text{Int}^1 \) (i.e. carry enough potential to be traversed by append). RaML’s monomorphic type system is unable to assign a single general type to append, which can be used at both call sites. So the function has been reanalyzed at every (monomorphic) call site. \texttt{Re}², on the other hand, handles this example out of the box, since the type variable \( \text{a} \) in the type of append can be instantiated with \( \text{Int} \) for the outer occurrence and with \( \text{Int}^1 \) for the inner occurrence, yielding the type

\[
\text{xs} : \text{List} \ \text{Int}^2 \rightarrow \text{ys} : \text{List} \ \text{Int}^1 \rightarrow (\text{List} \ \text{Int}^1 | \ldots)
\]

As a final example, consider the standard \texttt{map} function:

\[
\text{map} : (a ightarrow b) \rightarrow \text{List} \ a \rightarrow \text{List} \ b
\]

Although this type has no potential annotations, it implicitly tells us something about the resource behavior of \texttt{map}: namely, that \texttt{map} applies a function to each list element at most once. This is because \( a \) can be instantiated with a type with an arbitrary amount of potential, and the only way to pay for this potential is with a list element (which also has type \( a \)).

2.4 Resource-Guided Synthesis with Re\textsc{Syn}

We have extended \textsc{Synquid} with support for Re\textsuperscript{2} types in a new program synthesizer Re\textsc{Syn}. Given a resource-annotated signature for common \textquoteleft common\textquoteright from Sec. 2.3 and a component library that includes member, Re\textsc{Syn} is able to synthesize the efficient implementation in Fig. 2. The key to efficient synthesis is type-checking each program candidate incrementally as it is being constructed, and discarding an ill-typed program prefix as early as possible. For example, while enumerating candidates for the function common \textquoteleft common\textquoteright, we can safely discard the inefficient version from Fig. 1 even before constructing the second branch of the conditional (because the first branch together with the guard use up too many resources). Hence, as we explain in more detail in Sec. 4, a key technical challenge in Re\textsc{Syn} has been a tight integration of resources into \textsc{Synquid}’s round-trip type checking mechanism, which aggressively propagates type information top-down from the goal and solves constraints incrementally as they arise.

**Termination Checking** In addition to making the synthesizer resource-aware, Re\textsuperscript{2} types also subsume and generalize \textsc{Synquid}’s termination checking mechanism. To avoid generating diverging functions, \textsc{Synquid} uses a simple termination metric (the tuple of function’s arguments), and checks that this metric decreases at every recursive call. Using this metric, \textsc{Synquid} is not able to synthesize the function range from Sec. 2.3, because it requires a recursive call that decreases the difference between the arguments, \( b - a \). In contrast, Re\textsc{Syn} need not reason explicitly about termination, since potential annotations already encode an upper bound on the number of recursive calls. Moreover, the flexibility of these annotations enables Re\textsc{Syn} to synthesize programs that require nontrivial termination metrics, such as range.

3 The Re\textsuperscript{2} Type System

In this section, we define a subset of Re\textsuperscript{2} as a formal calculus to prove type soundness. This subset includes Booleans that are refined by their values, and lists that are refined by their lengths. The programs in Sec. 1 and Sec. 2 use \textsc{Synquid}’s surface syntax. The gap from the surface language to the core calculus involves inductive types and refinement-level measures. The restriction to this subset in the technical development is only for brevity and proofs carry over to all the features of \textsc{Synquid}.

**Syntax** Fig. 4 presents the grammar of terms in Re\textsuperscript{2} via abstract binding trees [29]. The core language is basically the standard lambda calculus augmented with Booleans and lists. A value \( v \in \text{Val} \) is either a boolean constant, a list of values, or a function. Expressions in Re\textsuperscript{2} are in a-normal-form \textsuperscript{[55]}, which means that syntactic forms occurring in non-tail position allow only atoms \( \bar{a} \in \text{Atom} \), i.e., variables and values; this restriction simplifies typing rules for applications, as we explain below. We identify a subset \textsf{SimpAtom} of \text{Atom} that contains atoms interpretable in the refinement logic. Intuitively, the value of an \( a \in \textsf{SimpAtom} \) should be either a Boolean or a list. The syntactic form impossible is introduced as a placeholder for unreachable code, e.g., the else-branch of a conditional whose predicate is always true.

The syntactic form tick(\( c, e_0 \)) is used to specify resource usage, and it is intended to cost \( c \in \mathbb{Z} \) units of resource and then reduce to \( e_0 \). If the cost \( c \) is negative, then \( -c \) units of resource will become available in the system. tick terms support flexible user-defined cost metrics: for example, to count recursive calls, the programmer may wrap every such call in tick(\( 1, \cdot \)); to keep track of memory consumption, they might wrap every data constructor in tick(\( c, \cdot \)), where \( c \) is the amount of memory that constructor allocates.

**Operational Semantics** The resource usage of a program is determined by a small-step operational cost semantics. The semantics is a standard one augmented with a resource parameter. A step in the evaluation judgment has the form

\[
\text{tick}(c, e) \rightarrow \text{tick}(c - r, e_0)
\]
The multi-step evaluation relation \( \rightarrow^* \) is the reflexive transitive closure of \( \rightarrow \). The judgment \( \langle e, q \rangle \rightarrow^* \langle e', q' \rangle \) expresses that with \( q \) units of available resources, \( e \) evaluates to \( e' \) without running out of resources and \( q' \) resources are left. Intuitively, the high-water mark resource usage of an evaluation of \( e \) to \( e' \) is the minimal \( q \) such that \( \langle e, q \rangle \rightarrow^* \langle e', q' \rangle \). For monotone resources like time, the cost is the sum of costs of all the evaluated tick expressions. In general, this net cost is invariant, that is, \( p - p' = q - q' \) if \( \langle e, p \rangle \rightarrow^n \langle e', p' \rangle \) and \( \langle e, q \rangle \rightarrow^m \langle e', q' \rangle \), where \( \rightarrow^* \) is the relation obtained by self-composing \( \rightarrow \) for \( n \) times.

**Refinements** We now combine Synquid's type system with AARA to reason about resource usage. Fig. 5 shows the syntax of the Re\(^2\) type system. Refinements \( \psi \) are distinct from program terms and classified by sorts \( \Delta \). Re\(^2\)'s sorts include Booleans \( \mathbb{B} \), natural numbers \( \mathbb{N} \), and uninterpreted symbols \( \delta_a \). Refinements can be logical formulas and linear expressions, which may reference program variables. Logical refinements \( \psi \) have sort \( \mathbb{B} \), while potential annotations \( \phi \) have sort \( \mathbb{N} \). Re\(^2\) interprets a variable of Boolean type as its value, list type as its length, and type variable \( a \) as an uninterpreted symbol with a corresponding sort \( \delta_a \). We use the following interpretation \( I(\cdot) \) to reflect interpretable atoms \( a \in \text{SimpAtom} \) in the refinement logic:

\[
\begin{align*}
I(x) &= x \\
I(\text{true}) &= \top \\
I(\text{false}) &= \bot \\
I(\text{cons}(a_i)) &= I(a_i) + 1
\end{align*}
\]

**Types** We classify types into four categories. Base types \( B \) include Booleans, lists and type variables. Type variables \( a \) are annotated with a *multiplicity* \( m \in \mathbb{Z}^*_+ \cup \{\infty\} \), which denotes an upper bound on the number of usages of a variable like in bounded linear logic [23]. For example, \( L(2 \cdot a) \) denotes a universal list whose elements can be used at most twice.

Refinement types are *subset types* and *dependent arrow types*. The inhabitants of the subset type \( \{B \mid \psi\} \) are values of type \( B \) that satisfy the refinement \( \psi \). The refinement \( \psi \) is a logical predicate over program variables and a special value variable \( v \), which does not appear in the program and stands for the inhabitant itself. For example, \( \{\text{bool} \mid v\} \) is a type of true, and \( \{L(\text{bool}) \mid v \leq 5\} \) represents Boolean lists of length at most 5. Dependent arrow types \( x:T_x \rightarrow T \) are function types whose return type may reference the formal argument \( x \). As type variables, these function types are also annotated with a multiplicity \( m \in \mathbb{Z}^*_+ \cup \{\infty\} \) restricting the number of times the function may be applied.

To apply the potential method of amortized analysis [65], we need to define potentials with respect to the data structures in the program. We introduce resource-annotated types as a refinement type augmented with a potential annotation, written \( R^\phi \). Intuitively, \( R^\phi \) assigns \( \phi \) units of potential to values of the refinement type \( R \). The potential annotation \( \phi \) may also reference the value variable \( v \). For example, \( L(\text{bool})^\langle5\cdot v\rangle \) describes Boolean lists \( \ell \) with \( 5|\ell| \) units of potential where \( |\ell| \) is the length of \( \ell \). The same potential can be expressed by assigning 5 units of potential to every element using the type \( L(\text{bool})^5 \).

Type schemas represent (possibly) polymorphic types. Note that the type quantifier \( \forall \) can only appear outermost in a type.

Similar to Synquid, we introduce a notion of *scalar* types, which are resource-annotated base types refined by logical constraints. Intuitively, interpretable atoms are scalars and Re\(^2\) only allows the refinement-level logic to reason about values of scalar types. We will abbreviate \( 1 \cdot a = a, \{B \mid \tau\} \) as \( B, \infty \cdot (x:T_x \rightarrow T) = x:T_x \rightarrow T, \) and \( R^0 \) as \( R \).

**Typing Rules** In Re\(^2\), the typing context \( \Gamma \) is a sequence of variable bindings \( x:S, \) type variables \( a, \) path conditions \( \psi \), and free potentials \( \phi \). Our type system consists of five judgments: sorting, well-formedness, subtyping, sharing, and typing. We omit sorting and well-formedness rules and include them in the technical report [49]. The sorting judgment \( \Gamma \vdash \psi \in \Delta \) states that a refinement \( \psi \) has a sort \( \Delta \) under a context \( \Gamma \). A type \( S \) is said to be well-formed under a context \( \Gamma \), written \( \Gamma \vdash S \), if every referenced variable in it is in the correct scope.

Fig. 6 presents selected typing rules for Re\(^2\). The typing judgment \( \Gamma \vdash e : S \) states that the expression \( e \) has type \( S \) in context \( \Gamma \). The intuitive meaning is that if there is at least the amount resources as indicated by the potential in the context \( \Gamma \) then this suffices to evaluate \( e \) to a value \( v \), and after the evaluation there are at least as many resources available as indicated by the potential in \( S \). The auxiliary typing judgment \( \Gamma \vdash a : B \) assigns base types to interpretable atoms. Atomic
typing is useful in the rule (T-SimpAtom), which uses the interpretation $I(\cdot)$ to derive a most precise refinement type for interpretable atoms.

The subtyping judgment $\Gamma \vdash T_1 <: T_2$ is defined in a standard way, with the extra requirement that the potential $T_1$ should be greater than or equal to that in $T_2$. Subtyping is often used to "forget" some program variables in the type to ensure the result type does not reference any locally introduced variable, e.g., the result type of let$(e_1,e_2,e_3)$ cannot have $x$ in it and the result type of mat$(a_0,e_1,x_3,x_4,e_2)$ cannot reference $x_3$ or $x_4$.

To reason about logical refinements, we introduce validity checking, written $\Gamma \models \psi$, to state that a logical refinement $\psi$ is always true under any instance of the context $\Gamma$. The validity checking relation is established upon a denotational semantics for refinements. Validity checking in Re² is decidable because it can be reduced to Presburger arithmetic. The full development of validity checking is included in the technical report [40].

We reason about inductive invariants for lists in rule (T-MatL), using interpretation $I(\cdot)$. In our formalization, lists are refined by their length thus the invariants are: (i) nil has length 0, and (ii) the length of cons$(\cdot,a_1)$ is the length of $a_1$ plus one. The type system can be easily enriched with more refinements and data types (e.g., the elements of a list are the union of its head and those of its tail) by updating the interpretation $I(\cdot)$ as well as the premises of rule (T-MatL).

Finally, notable are the two typing rules for applications: (T-App) and (T-App-SimpAtom). In the former case, the function return type $T$ does not mention $x$, and hence can be directly used as the type of the application (this is the case e.g. for all
Resources The rule (T-CONSUME-P) states that an expression tick(c,e₀) is only well-typed in a context that contains a free potential term c. To transform the context into this form, we can use the rule (S-TRANSFER) to transfer potential within the context between variable types and free potential terms, as long as we can prove that the total amount of potential remains the same. For example, the combination of (S-TRANSFER) and (S-RELAX) allows us to derive both x : bool₁ + x : bool₁ and x : bool₁ + tick(1,x) : bool (but not x : bool₁ + tick(1,x) : bool₁).

The typing rules of Re² form an affine type system [69]. To use a program variable multiple times, we have to introduce explicit sharing to ensure that the program cannot gain potential. The sharing judgment Γ ⊢ S Y₁ S₁ | S₂ means that in the context Γ, the potential indicated by S is apportioned into two parts to be associated with S₁ and S₂. We extend this notion to context sharing, written Γ Y₁ Γ₁ | Γ₂, which states that Γ₁, Γ₂ has the same sequence of bindings as Γ, but the potentials of type bindings in Γ are shared point-wise, and the free potentials in the Γ are also split. A special context sharing Γ Y Γ₁ | Γ is used in the typing rules (T-Abs) and (T-Fix) for functions. The self-sharing indicates that the function can only reference potential-free free variables in the context. This is also used to ensure that the program cannot gain more potential through free variables by applying the same function multiple times.

Restricting functions to be defined under potential-free contexts is undesirable in some situations. For example, a curried function of type x : Tₓ → y : Tᵧ → T might require nonzero units of potential on its first argument x, which is not allowed by rule (T-Abs) or (T-Fix) on the inner function type y : Tᵧ → T. We introduce another rule (T-Abs-Lin) to relax the restriction. The rule associates a multiplicity m with the function type, which denotes the number of times that the function could be applied. Instead of context self-sharing, we require the potential in the context to be enough for m function applications. Note that in RxSyn's surface syntax used in the Sec. 2, every curried function type implicitly has multiplicity 1 on the inner function: x : Tₓ → 1-(y : Tᵧ → T).

Example Recall the function triplate from Fig. 3, which can be written as follows in Re² core syntax:

\[
\text{triple} \equiv \ell : L(\text{bool}^2) \rightarrow \{L(\text{bool}) | v = 3 \times \ell \}
\]

\[
\text{triple} = \lambda(\ell, \text{let}(\text{app}(\text{app}(\text{app}(\ell, \ell'), \ell'), \ell'))
\]

Next, we illustrate how Re² uses the signature of append:

\[
\text{append} \equiv \forall a. \forall x : L(a^3) \rightarrow 1-(y : L(a) \rightarrow \{L(a) | v = xs + ys\})
\]

to justify the resource bound 2|\ell| on triplate. Suppose Γ is a typing context that contains the signature of append. The argument \ell is used three times, so we need to use sharing relations to apportion the potential of \ell. We have Γ ⊢ L(\text{bool}^2) \uplus L(\text{bool}) \uplus L(\text{bool}^1) \uplus L(\text{bool}) \uplus L(\text{bool}^1), and we assign L(\text{bool}^1), L(\text{bool}^1), and L(\text{bool}^1) to the three occurrences of \ell respectively in the order they appear in the program. To reason about e₁ = app(app(append,\ell),\ell), we instantiate append with \alpha → bool₀, inferring its type as

\[
x s : L(\text{bool}^1) \rightarrow 1-(y : L(\text{bool}^0) \rightarrow \{L(\text{bool}^0) | v = xs + ys\})
\]

and by (T-APP-SIMPATOM) we derive the following:

\[
\Gamma, \ell : L(\text{bool}^1) \vdash e₁ : \{L(\text{bool}^0) | v = \ell + \ell\}
\]

We then can typecheck e₂ = app(app(append,\ell),\ell') with the same instantiation of append:

\[
\Gamma, \ell : L(\text{bool}^1), \ell' : T₁ \vdash e₂ : \{L(\text{bool}^0) | v = xs + (xs + xs)\}
\]

(where T₁ is the type of e₁). Finally, by subtyping and the following valid judgment in the refinement logic

\[
\Gamma, \ell : L(\text{bool}^0), v : L(\text{bool}^0) \vdash v = \ell + (\ell + \ell) \implies v = 3 \times \ell,
\]

we conclude Γ ⊢ triplate : L(\text{bool}^2) → \{L(\text{bool}) | v = 3 \times \ell\}.

Soundness The type soundness for Re² is based on progress and preservation. The progress theorem states that if we derive a bound \eta for an expression e with the type system and p ≥ q resources are available, then (e,p) can make a step if e is not a value. In this way, progress shows that resource bounds are indeed bounds on the high-water mark of the resource usage since states (e,p) in the small step semantics can be stuck based on resource usage if, for instance, p = 0 and e = tick(1,e').

Theorem 1 (Progress). If \eta + e : S and p ≥ q, then either e ∈ Val or there exist e' and p' such that (e',p') → (e,p').

\[\square\]

The preservation theorem accounts for resource consumption by relating the left over resources after a computation to the type judgment of the new term.

Theorem 2 (Preservation). If \eta + e : S, p ≥ q and (e,p) → (e',p'), then p' + e' : S.

\[\square\]

The proof of preservation makes use of the following crucial substitution lemma.

Lemma 1 (Substitution). If Γ₁,x : \{B | \psi\}^δ, Γ' + e : S, Γ₂ + t : \{B | \psi\}^δ, t ∈ Val and ⊢ Γ Y Γ₁ | Γ₂, then Γ,[I(ℓ)/x]Γ' + [t/x]e : [I(ℓ)/x]S.

\[\square\]

Proof. By induction on Γ₁,x : \{B | \psi\}^δ, Γ' + e : S.
Since we found the purely syntactic soundness statement about results of computations (they are well-typed values) somewhat unsatisfactory, we also introduced a denotational notion of consistency. For example, a list of values \( v = [v_1,\ldots,v_n] \) is consistent with \( q \vdash \ell : L(\{\text{bool} \mid \neg v\})^\ast + 5 \) if \( q \geq n + 5 \) and each value \( v_i \) of the list is false. We then show that well-typed values are consistent with their typing judgment.

**Lemma 2** (Consistency). If \( q \vdash v : S \), then \( v \) satisfies the conditions indicated by \( S \) and \( q \) is greater than or equal to the potential stored in \( v \) with respect to \( S \).

As a result, we derive the following theorem.

**Theorem 3** (Soundness). If \( q \vdash e : S \) and \( p \geq q \) the either

- \( (e,p) \rightarrow^* (v,p') \) and \( v \) is consistent with \( p' \vdash v : S \) or
- for every \( n \) there is \( (e',p') \) such that \( (e,p) \rightarrow^* (e',p') \).

Complete proofs can be found in the technical report [40].

**Inductive Datatypes and Measures** We can generalize our development of list types for inductive types \( \mu X.C : T \times X^k \), where \( C \) is the constructor name, \( T \) is the element type that does not contain \( X \), and \( X^k \) is the \( k \)-element product type \( X \times X \times \cdots \times X \). The introduction rules and elimination rules are almost the same as (T-Ni), (T-Cons) and (T-MatL), respectively, except that we need to capture inductive invariants for each constructor \( C \) in the rules correspondingly. In SYNQUID, these invariants are specified by inductive measures that map values to refinements. We can introduce new sorting rules for inductive types to embed values as their related measures in the refinement logic.

**Constant Resource** Our type system infers upper bounds on resource usage. Recently, AARA has been generalized to verify constant-resource behavior [46]. A program is said to be constant-resource if its executions on inputs of the same size consume the same amount of resource. We can adapt the technique in [46] to \( \mathbb{R}^2 \) by (i) changing the subtyping rules to keep potentials invariant (i.e. replacing \( \geq \) with \( = \) in (Sub-TVAr), (Sub-Arrow), (Sub-Pot)), and (ii) changing the rule (Simp-Atom-Var) to require \( \phi = 0 \). Based on the modified type system, our synthesis algorithm can also synthesize constant-time implementations (see Sec. 5.2 for more details).

## 4 Type-Driven Synthesis with \( \mathbb{R}^2 \)

In this section, we first show how to turn the type checking rules of \( \mathbb{R}^2 \) into synthesis rules, and then leverage these rules to develop a synthesis algorithm.

### 4.1 Synthesis Rules

**Extended Syntax** To express synthesis rules, we extend \( \mathbb{R}^2 \) with a new syntactic form \( \hat{e} \) for expression templates. As shown in Fig. 7, templates are expressions that can contain holes \( \phi \) in certain positions. The flat let form \( \text{lets}(D,\hat{e}) \), where \( D \) is a sequence of bindings, is a shortcut for a nest of let-expressions

\[
D \coloneqq \cdot | D; x \leftarrow e \\
\hat{e} \coloneqq e \circ \phi \mid \text{app}(x,\phi) \mid \text{if}(x,\phi,\psi) \mid \text{matl}(x,\phi,x_h,x_t,\phi) \mid \text{let}(D,\hat{e})
\]

**Figure 7.** Extended syntax

let\((x_1,d_1,\ldots,\text{let}(x_n,d_n,\hat{e}))\); we write fold\((\text{let}(D,e))\) to convert a flat let (without holes) back to the original syntax. We also extend the language of types with an unknown type \( \nu \), which is used to build partially defined goal types, as explained below.

**Synthesis for A-Normal-Form** Our synthesis relation consists of two mutually recursive judgments: the synthesis judgment \( \Gamma \vdash \hat{e} : T \leadsto e \) intuitively means that the template \( \hat{e} \) can be completed into an expression \( e \) such that \( \Gamma \vdash e : S \); the purpose of the auxiliary atomic synthesis judgment is explained below. Selected rules for both judgments are given in Fig. 8; the full technical development can be found in the technical report [40].

The synthesis rule (Syn-Gen) handles polymorphic goal types. The rules (Syn-Fix) and (Syn-Ans) handle arrow types and derive either a fixpoint term or an abstraction. The rule (Syn-Imn) derives impossible in an inconsistent context (which may arise e.g. in a dead branch of a pattern match). The rest of the rules handle the common case when the goal type \( T \) is scalar and the context is consistent; in this case the target expression can be either a conditional, a match, or an E-term [51], i.e. a term made of variables, applications, and constructors. Special care must be taken to ensure that these expressions are in a-normal-form: generally, a-normalizing an expression requires introducing fresh variables and let-bindings for them. To retain completeness, our synthesis rules need to do the same: intuitively, in addition to an expression \( e \), a rule might also need to produce a sequence of let-bindings \( D \) that define fresh variables in \( e \). To this end, we introduce the atomic synthesis judgment \( \Gamma \vdash \hat{e} : T \leadsto^\dagger \text{lets}(D,a) \), which synthesizes normalized E-terms, where \( a \) is an atom and each definition in \( D \) is an application or a constructor in a-normal-form.

As an example, consider the rule (Syn-Conn) for synthesizing conditionals: ideally, we would like to synthesize a guard \( e \) of type bool, and then synthesize the two branches under the assumptions that \( e \) evaluates to true and false, respectively. Recall, however, that the guard must be atomic; hence, to synthesize a well-formed conditional, we use atomic synthesis to produce a guard \( \text{let}(D,x) \). Now to get a well-scoped program we must place the whole conditional inside the bindings \( D \); to that end, the second premise of (Syn-Conn) uses a nontrivial template \( \text{let}(D,\text{if}(x,\phi,\psi)) \). The rules (Fill-Let) and (Fill-Cond) handle this template by integrating it into the typing context and exposing the hole; along the way (Fill-Let) takes care of context sharing, which accounts for the potential consumed by the definitions in \( D \). Synthesis of matches works similarly using (Syn-MatL) and (Fill-MatL).

**Atomic Synthesis** The first four rules of atomic synthesis generate a simple atom if its type matches the goal; the rest of
the rules deal with the hardest part: normalized applications. Consider the rule (ASYN-App): given a goal type $T$ for the application $app(e_1,e_2)$, we need to construct goal types for $e_1$ and $e_2$, to avoid enumerating them blindly. Following SYNQUId’s round-trip type checking idea, we use the type $\_? \rightarrow T$ as the goal for $e_1$ (i.e. a function from unknown type to $T$). The subtyping rules for $\_?$ are such that $\Gamma \vdash (y:T_1 \rightarrow T_2) < (\_? \rightarrow T)$ holds if $T_2$ and $T$ agree in shape and those refinements that do not mention $y$; hence this goal type filters out those functions $e_1$ that cannot fulfill the desired goal type $T$, independently of the choice of $e_2$. One difference with SYNQUId is that the goal type for $e_1$ is linear, reflecting that we intend to use $e_1$ only once and allowing it to capture positive potential.

Similarly to the conditional case explained above, the synthesized left-hand side of the application, $e_1$, has the form $\text{lets}(D_1,x)$, and the argument $e_2$ must be synthesized inside the bindings $D_1$. These bindings are processed by (AFILL-LET), and the actual argument synthesis happens in either (AFILL-App) or (AFILL-SimpAtom), depending on whether the argument type is a scalar. The former corresponds to a higher-order application: here $T_1$ is an arrow type, and hence the argument cannot occur in the function’s return type; in this case, synthesizing an expression of type $T_1$ must yield an abstraction or fixpoint (since $T_1$ is an arrow), both of which are atoms. The latter corresponds to a first-order application: here the return type $T’$ can mention $y$, so after synthesizing an argument of type $T_y$, we still need to check whether the resulting application $\text{lets}(D,app(x,a))$ has the right type $T$. Note how both (AFILL-App) or (AFILL-SimpAtom) return normalized E-terms by generating a fresh variable and binding it to an application.

Cost Metrics In the context of synthesis we cannot rely on programmer-written tick terms to model cost. Instead in our formalization we use a simple cost metric where each function application consumes one unit of resource; hence every application generated by (AFILL-App) or (AFILL-SimpAtom) is wrapped in tick($1$, $\cdot$). Our implementation provides more flexibility and allows the programmer to annotate any arrow type with a non-negative cost $c$ to denote that applying a function of this type should incur cost $c$.

Soundness The synthesis rules always produce a well-typed expression (proof can be found in the technical report [40]).

**Theorem 4** (Soundness of Synthesis). If $\Gamma \vdash o : S \rightarrow e$ then $\Gamma \vdash e : S$.

### 4.2 Synthesis Algorithm

In this section we discuss how to turn the declarative synthesis rules of Sec. 4.1 into a synthesis algorithm, which takes as input a goal type $S$, a context $\Gamma$, and a bound $k$ on the program depth, and either returns a program $e$ of depth at most $k$
The core algorithm follows the recipe from prior work on
Selected cases for translating typing constraints
Γ
ing constraints yields valid subtyping, sharing, and transfer
constraints, is a map
L
Γ
unknown refinement terms
Typing constraints efficiently, we use the following constraint-based approach.
ϕ
is used to instantiate a type variable, we need to guess both
context
Γ
is shared as
Γ
in a synthesis derivation stem from the following premises
sources of non-determinism to constraint solving.

1. Resource-Guided Program Synthesis
2. type-driven synthesis

The main new challenge then is to
solve a system of resource constraints of the form
ψ
≢0
, where
ψ
is now a known formula of the refinement logic. Since potential annotations in Re²
are restricted to linear terms over program variables, we can replace each unknown term
U
Γ
in
ϕ
with a linear template
Σx∈C
i
x
, where each
C
i
is an unknown integer coefficient and
X
is the set of all variables in
Γ
such that
x
∈
N
. After normalization, the system of resource constraints is reduced to the following
doubly-quantified system of linear inequalities:
\[ \exists C_i, \forall x \bigwedge_{r \in R} r(C_i, x) \]
where each clause
r
is of the form
ψ(x) = \sum f(C_i) x \geq 0
, where
ψ
is a known formula over the program variables
x
, and each
f
is a linear function over unknown integer coefficients
C_i
.

Note a crucial difference between these constraints and those generated by RaML: since RaML’s potential annotations are not dependent—i.e. non-μ
r
cannot mention program variables
x
—its resource constraints reduce to plain linear inequalities:
\[ \exists C_i, \sum C_i x \geq c \]
where
C_i
is a known constant, which can be handled by an LP solver. In our case, the challenge stems both from the double quantification and the fact that individual clauses
r
are bounded
by formulas
ψ
, which are often nontrivial.

For example, synthesizing the function range from Sec. 2 gives rise to the following (simplified) resource constraints:
\[ \exists C_0 \ldots C_3, \forall a, b, v
\]
\[ \neg(a \geq b) \land v = b \implies (C_0 + 1) \cdot a + C_1 \cdot b + (C_2 - 1) \cdot v + C_3 \geq 0 \]
\[ \neg(a \geq b) \land v = b \implies C_0 \cdot a + C_1 \cdot b + C_2 \cdot v + C_3 \geq 0 \]
where a solution only exists if the bounds are taken into account. One solution is
[\{ C_0 \mapsto -1, C_1 \mapsto 0, C_2 \mapsto 1, C_3 \mapsto 0 \}]
which stands for the potential term
v - a
.

\footnote{Our implementation uses SyRig’s default greatest-fixpoint Horn solver, which technically renders this technique incomplete, however we observed that it works well in practice.}

Figure 9. Selected cases for translating typing constraints to validity constraints.

such that
Γ \vdash e : S
, or determines that no such program exists.
The core algorithm follows the recipe from prior work on
type-driven synthesis and performs a fairly standard
goal-directed backtracking proof search with
Γ \vdash o = e \leadsto S
as the top-level goal. In the rest of this section, we explain
how to make such proof search feasible by reducing the core
sources of non-determinism to constraint solving.

Typing constraints The main sources of non-determinism in a synthesis derivation stem from the following premises of synthesis and typing rules: (1) whenever a given context
Γ
is shared as
Γ \vdash \Gamma \vdash \Gamma_1 \vdash \Gamma_2
, we need to guess how to apportion potential annotations in
Γ
; (2) whenever potential in a given context
Γ
is transferred, we need to guess potential annotations in
Γ’
such that
Φ(Γ) = Φ(Γ’)
; and finally (3) whenever
( B | \psi φ
is used to instantiate a type variable, we need to guess both
ϕ
and
ψ
. All three amount to inference of unknown refinement
terms of either Boolean or numeric sort. To infer these terms
efficiently, we use the following constraint-based approach.
First, we build a symbolic synthesis derivation, which may contain
unknown refinement terms
U
Γ
, and collect all subtyping,
sharing, and transfer premises from the derivation into a
system of typing constraints. Here
Δ
records the desired sort
of the unknown refinement term, and
Γ
records the context in
which it must be well-formed. A solution to a system of typing
constraints, is a map
L : U \rightarrow \psi
such that for every unknown
U
Γ
, \Gamma \vdash L(U) \in \Delta
and substituting
L(U)
for
U
within the typing
constraints yields valid subtyping, sharing, and transfer judgments.

Constraint Solving To solve typing constraints, the algorithm first transforms them into validity constraints of one of two forms:
Γ \vdash \psi \equiv \psi’
or
Γ \vdash \psi \equiv 0
; the interesting cases of this translation are shown in Fig. 9. Then, using the definition of validity (in the technical report [40]), we further reduce these into a system of:
1. Horn constraints of the form
\psi_1 \land \ldots \land \psi_n \implies \psi_0
, and
2. resource constraints of the form
\psi_1 \land \ldots \land \psi_n \implies \phi \geq 0.

Here any
ψ_i
can be either a Boolean unknown
U^B
or a known refinement term, and
ϕ
is a sum of zero or more numeric unknowns
U^N
and a known (linear) refinement term. While prior
work has shown how to efficiently solve Horn constraints using
predicative abstraction [51, 54], resource constraints present
a new challenge, since they contain unknown terms of both Boolean and numeric sorts. In the interest of efficiency, our
synthesis algorithm does not attempt to solve for both Boolean
and numeric terms at the same time. Instead, it uses existing
techniques to find a solution for the Horn constraints, and
then plugs this solution into the resource constraints. Note that this approach does not sacrifice completeness, as long as the Horn solver returns the least-fixpoint (i.e. strongest)
solution for each
U^B
, since Boolean unknowns only appear
negatively in resource constraints.

Resource Constraints The main new challenge then is to
solve a system of resource constraints of the form
ψ \equiv \phi \geq 0
, where
ψ
is now a known formula of the refinement logic. Since potential annotations in Re²
are restricted to linear terms over program variables, we can replace each unknown term
U
Γ
in
ϕ
with a linear template
Σx∈C
i
x
, where each
C
i
is an unknown integer coefficient and
X
is the set of all variables in
Γ
such that
x
∈
N
. After normalization, the system of resource constraints is reduced to the following
doubly-quantified system of linear inequalities:
The algorithm first queries the SMT solver for a counter-example \( e \in \text{SMT}(\exists \bar{x}. \neg R(C, \bar{x})) \).

\[
e \leftarrow \text{SMT}(\exists \bar{x}. \neg R(C, \bar{x}))
\]

if \( e = \bot \) then

\[
\Rightarrow \text{No counter-example}
\]

return \((C, \bar{e})\)

else

\[
E' \leftarrow E \cup e
\]

\[
R' \leftarrow \{ r \in R \mid \neg r(C, e) \}
\]

\[
C' \leftarrow \text{SMT}(\exists C_i, \land e \in E'. R'(C_i, e))
\]

if \( C' = \bot \) then return \( \bot \)

\[
\Rightarrow \text{No solution}
\]

else \( \text{Solve}(R, C \cup C', E') \)

**Incremental Solving** Constraints of this form can be solved using counter-example guided synthesis (CEGIS) \([62]\), which is, however, relatively expensive. We observe that in the context of synthesis we have to repeatedly solve similar systems of resource constraints because a program candidate is type-checked incrementally as it is being constructed, which corresponds to an incrementally growing set of clauses \( R \). Moreover, we observe that as new clauses are added, only a few existing coefficients \( C_i \) are typically invalidated, so we can avoid solving for all the coefficients from scratch. To this end, we develop an incremental version of the CEGIS algorithm, shown in Algorithm 1.

The goal of the algorithm is to find a solution \( C : C_i \to \mathbb{Z} \) that maps unknown coefficients to integers such that \( \forall x. R(C, \bar{x}) \) holds (we write \( R(C, \bar{x}) \) as a shorthand for \( \land_{r \in R} r(C, \bar{x}) \)). The algorithm takes as input a set of clauses \( R \) (which includes both old and new clauses), the current solution \( C \) (new coefficients \( C_i \) are mapped to 0), and the current set of examples \( E \), where an example \( e \in E \) is a partial assignment to universally-quantified variables \( e : X \to \mathbb{N} \).

The algorithm first queries the SMT solver for a counter-example \( e \) to the current solution. If no such counter-example exists, the solution is still valid (this happens surprisingly often, since many resource constraints are trivial). Otherwise, the current solution needs to be updated. To this end, a traditional CEGIS algorithm would query the SMT solver with the following synthesis constraint: \( \exists C_i, \land e \in E'. R'(C_i, e) \), which enforces that all clauses are satisfied on the extended set of examples. Instead, our incremental algorithm picks out only those clauses \( R' \) that are actually violated by the new counter-example; since in our setting \( R' \) is typically small, this optimization significantly reduces the size of the synthesis constraint and synthesis times for programs with dependent annotations (as we demonstrate in Sec. 5).

**4.3 Implementation**

We implemented the resource-guided synthesis algorithm in ReSyn, which extends SYNQUID with support for resource-annotated types and a type-based resource constraint solver. Note that while our formalization is restricted to Booleans and length-indexed lists, our implementation supports the full expressiveness of SYNQUID’s types: types include integers and user-defined algebraic datatypes, and refinement formulas support sets and can mention arbitrary user-defined measures. More importantly, resource terms in ReSyn can mention integer variables and use subtraction, multiplication, conditional expressions, and numeric measures; finally, multiplicities on type variables can be dependent (mention variables). These changes have the following implications: (1) resource terms are not syntactically guaranteed to be non-negative, so we emit additional well-formedness constraints to enforce this; (2) resource terms are not syntactically restricted to be linear; our implementation is incomplete, and simply rejects the program if a nonlinear term arises; (3) subtyping and sharing constraints with conditional resource terms are decomposed into unconditional ones by moving the guard to the context; so the search space for all numeric unknowns remains unconditional; (4) to handle measure applications in resource constraints, we replace them with fresh integer variables, and avoid spurious counter-examples by explicitly instantiating the congruence axiom with all applications in the constraint.

**5 Evaluation**

We evaluated ReSyn using the following criteria:

**Relative performance**: How do ReSyn’s synthesis times compare to SYNQUID’s? How much does the additional burden of solving resource constraints affect its performance?

**Efficacy of resource analysis**: Can ReSyn discover more efficient programs than SYNQUID?

**Value of round-trip type checking**: Does round-trip type checking afforded by the tight integration of resource analysis into SYNQUID effective at pruning the search space? How does it compare to the naive combination of synthesis and resource analysis?

**Value of incremental solving**: To what extent does incremental solving of resource constraints improve ReSyn’s performance?

### 5.1 Relative Performance

To evaluate ReSyn’s performance relative to SYNQUID, we selected 43 problems from SYNQUID’s original suite, annotated them with resource bounds, and re-synthesized them with ReSyn. The rest of the original 64 benchmarks require non-linear bounds, and thus are out of scope of Re\(^2\). The details of this experiment are shown in Tab. 1, which compares ReSyn’s synthesis times against SYNQUID’s on these linear-bounded benchmarks.

Unsurprisingly, due to the additional constraint-solving, ReSyn generally performs worse than SYNQUID: the median synthesis time is about 2.5x higher. Note, however, that in return it provides provable guarantees about the performance...
Table 1. Comparison of ReSyn and SynQuid. For each benchmark, we report the set of provided Components; cumulative size of synthesized Code (in AST nodes) for all goals; as well as running times (in seconds) for ReSyn (Time) and SynQuid (TimeNR).

<table>
<thead>
<tr>
<th>Group</th>
<th>Description</th>
<th>Components</th>
<th>Code</th>
<th>Time</th>
<th>TimeNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>is empty</td>
<td>true, false</td>
<td>16</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>member</td>
<td>true, false, :=, #</td>
<td>41</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>duplicate each element</td>
<td></td>
<td>39</td>
<td>0.5</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>replicate</td>
<td>0, inc, dec, ≤, #</td>
<td>31</td>
<td>2.9</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>append two lists</td>
<td></td>
<td>38</td>
<td>1.5</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>take first n elements</td>
<td>0, inc, dec, ≤, #</td>
<td>34</td>
<td>2.4</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>drop first n elements</td>
<td>0, inc, dec, ≤, #</td>
<td>30</td>
<td>20.4</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>concat list of lists</td>
<td>append</td>
<td>49</td>
<td>3.3</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>delete value</td>
<td>:=, #</td>
<td>49</td>
<td>0.8</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>zip</td>
<td></td>
<td>32</td>
<td>0.4</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>zip with i-th element</td>
<td>0, inc, dec, ≤, #</td>
<td>35</td>
<td>0.3</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>index of element</td>
<td>0, inc, dec, :=, #</td>
<td>43</td>
<td>0.5</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>insert at end</td>
<td></td>
<td>42</td>
<td>0.4</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>balanced split</td>
<td>fst, snd, abs</td>
<td>64</td>
<td>9.6</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>reverse</td>
<td>insert at end</td>
<td>35</td>
<td>0.4</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>insert (sorted)</td>
<td>≤, #</td>
<td>57</td>
<td>2.0</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>extract minimum</td>
<td>≤, ≤</td>
<td>71</td>
<td>18.1</td>
<td>8.3</td>
</tr>
<tr>
<td></td>
<td>fold</td>
<td></td>
<td>43</td>
<td>1.8</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>length using fold</td>
<td>0, inc, dec</td>
<td>39</td>
<td>0.3</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>append using fold</td>
<td></td>
<td>42</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>map</td>
<td></td>
<td>27</td>
<td>0.3</td>
<td>0.2</td>
</tr>
<tr>
<td>Unique list</td>
<td>insert</td>
<td>:=, #</td>
<td>49</td>
<td>0.8</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>delete</td>
<td>:=, #</td>
<td>45</td>
<td>0.5</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>compress</td>
<td>:=, #</td>
<td>64</td>
<td>5.0</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td>integer range</td>
<td>0, inc, dec, ≤, #</td>
<td>46</td>
<td>88.4</td>
<td>5.1</td>
</tr>
<tr>
<td></td>
<td>partition</td>
<td>≤</td>
<td>71</td>
<td>13.0</td>
<td>5.5</td>
</tr>
<tr>
<td>Sorted list</td>
<td>insert</td>
<td>&lt;</td>
<td>64</td>
<td>1.6</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>delete</td>
<td>&lt;</td>
<td>52</td>
<td>0.5</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>intersect</td>
<td>&lt;</td>
<td>71</td>
<td>17.0</td>
<td>0.8</td>
</tr>
<tr>
<td>Tree</td>
<td>node count</td>
<td>0, 1, *</td>
<td>34</td>
<td>3.8</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>preorder</td>
<td>append</td>
<td>45</td>
<td>3.0</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>to list</td>
<td>append</td>
<td>45</td>
<td>3.0</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>member</td>
<td>false, not, or, :=</td>
<td>63</td>
<td>2.2</td>
<td>0.4</td>
</tr>
<tr>
<td>BST</td>
<td>insert</td>
<td>≤, #</td>
<td>90</td>
<td>4.5</td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td>delete</td>
<td>≤, #</td>
<td>103</td>
<td>26.8</td>
<td>9.3</td>
</tr>
<tr>
<td></td>
<td>BST sort</td>
<td>≤, #</td>
<td>191</td>
<td>9.0</td>
<td>4.3</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>insert</td>
<td>≤, #</td>
<td>90</td>
<td>3.2</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>member</td>
<td>false, not, or, ≥, ≤, #</td>
<td>78</td>
<td>2.3</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>1-element constructor</td>
<td>≤, #</td>
<td>44</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>2-element constructor</td>
<td>≤, #</td>
<td>91</td>
<td>0.7</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>3-element constructor</td>
<td>≤, #</td>
<td>274</td>
<td>21.4</td>
<td>4.0</td>
</tr>
</tbody>
</table>

of generated code. ReSyn was able to discover a more efficient implementation for only one of the original SynQuid benchmarks (compress, discussed below). In general, these benchmarks contain only the minimal set of components required to produce a valid implementation, which makes it hard for SynQuid to find a non-optimal version. Four of the benchmarks in Tab. 1 use advanced features of Re²: for example, any function using natural numbers to index or construct a data structure requires dependent potential annotations.

5.2 Case Studies

The value of resource-guided synthesis becomes clear when the library of components grows. To confirm this intuition, we assembled a suite of 16 case studies shown in Tab. 2, each exemplifying some feature of ReSyn.

**Optimization** The first six benchmarks showcase ReSyn’s ability to generate faster code than SynQuid (the cost metric in each case is the number of recursive calls). Benchmark 1 is triple from Sec. 2.3, where both SynQuid and ReSyn generate the same efficient solution; benchmark 2 is slight modification of this example: it uses a component append‘, which traverses its second argument (unlike append, which traverses its first). In this case, ReSyn generates the efficient solution, associating the two calls to append‘ to the left, while SynQuid still generates the same—now inefficient—solution, associating these calls to the right. In benchmark 3 ReSyn makes the optimal choice of accumulator to avoid a quadratic-time implementation. Benchmark 4 is compress from Tab. 1: the task is to remove adjacent duplicated from a list. Here SynQuid makes an unnecessary recursive call, resulting in a solution that is slightly shorter but runs in exponential time! In other cases, ReSyn drastically changes the structure of the program to find an optimal implementation. Benchmark 5 is common from Sec. 2.1, where ReSyn must find an implementation that does not call member. Benchmark 6 works similarly, but computes the difference between two lists instead of their intersection. On these benchmarks, the performance disparity between ReSyn and SynQuid is much worse, as ReSyn must reject many more programs before it finds an appropriate implementation. On the other hand, these benchmarks also showcase the value of round-trip type checking: the column T-EAC reports synthesis times for a naive combination of synthesis and resource analysis, where we simply ask SynQuid to enumerate functionally correct programs until one type-checks under Re². As you can see, for benchmarks 5 and 6 this naive version times out after ten minutes.

**Dependent Potentials** Benchmarks 7–13 showcase fine-grained bounds that leverage dependent potential annotations. The first three of those synthesize a function insert that inserts an element into a sorted list. In benchmark 7 we use a simple linear bound (the length of the list), while benchmarks 8 and 9 specify a tighter bound: insert x xs can only make one recursive call per element of xs larger than x. These two examples showcase two different styles of specifying precise bounds: in 8 we define a custom measure numgt that counts list elements greater than a certain value; in 9, we instead annotate each list element with a conditional term indicating that it carries potential only if its value is larger than x. As discussed in Sec. 2, benchmark 13 (range) cannot be synthesized by SynQuid at all, because of restrictions on its termination checking mechanism, while ReSyn handles this benchmark out of the box.

For benchmarks 8–13, which make use of dependent potential annotations, we also report the synthesis times without incremental solving of resource constraints (T-Nlnc), which are up to 2x higher.
Table 2. Case Studies. For each synthesis problem, we report: the run time of ReSyn (T), SYNQUID (T-NR), naive combination of SYNQUID and resource analysis (T-EAC), ReSyn without incremental solving (T-NInc); as well as the tightest resource bound for the code generated by ReSyn (B) and by SYNQUID (B-NR). Here, SL is the type of sorted lists, and CL refers to the type of lists without adjacent duplicates. T0 is 10 min; all benchmarks count recursive calls.

<table>
<thead>
<tr>
<th>Description</th>
<th>Type Signature</th>
<th>Components</th>
<th>T</th>
<th>T-NR</th>
<th>T-EAC</th>
<th>T-NInc</th>
<th>B</th>
<th>B-NR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 triple</td>
<td>Var. x: L(a)→ [L(a)]</td>
<td>append</td>
<td>0.9</td>
<td>0.4</td>
<td>0.4</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 triple</td>
<td>Var. x: L(a)→ [L(a)]</td>
<td>append</td>
<td>2.8</td>
<td>0.4</td>
<td>1.2</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 conc list</td>
<td>Var. x: L(a)→ [L(a)]</td>
<td>append</td>
<td>3.2</td>
<td>0.9</td>
<td>1.1</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 compress</td>
<td>Var. x: L(a)→</td>
<td></td>
<td>3.8</td>
<td>3.1</td>
<td>4.1</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 common</td>
<td>Var. vs: SL(a)→ [L(a)]</td>
<td>&lt;, member</td>
<td>30.8</td>
<td>1.1</td>
<td>TO</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 list diff</td>
<td>Var. vs: SL(a)→ [L(a)]</td>
<td>&lt;, member</td>
<td>173.5</td>
<td>1.3</td>
<td>TO</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 insert</td>
<td>Var. x→ x: SL(a)→ [SL(a)]</td>
<td>&lt;</td>
<td>1.3</td>
<td>0.4</td>
<td>-</td>
<td>n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 insert'</td>
<td>Var. x→ x: SL(a)→ [SL(a)]</td>
<td>&lt;</td>
<td>49.6</td>
<td>0.7</td>
<td>102.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 insert''</td>
<td>Var. x→ x: SL(a)→ [SL(a)]</td>
<td>&lt;</td>
<td>7</td>
<td>0.7</td>
<td>13</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 replicate</td>
<td>Var. n: nat→ x:SL(a)→ [L(a)]</td>
<td>zero, inc, dec</td>
<td>1.4</td>
<td>0.2</td>
<td>2.7</td>
<td>n</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>11 take</td>
<td>Var. n: nat→ x:SL(a)→</td>
<td>zero, inc, dec</td>
<td>1.2</td>
<td>0.2</td>
<td>2.4</td>
<td>n</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>12 drop</td>
<td>Var. n: nat→ x:SL(a)→</td>
<td>zero, inc, dec</td>
<td>12</td>
<td>0.2</td>
<td>17</td>
<td>n</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>13 range</td>
<td>log(1.0→ hi)</td>
<td>inc, dec, ≥</td>
<td>11.8</td>
<td>0.2</td>
<td>-</td>
<td>hi→ lo</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>14 CT insert</td>
<td>Var. x→ x: SL(a)→ [SL(a)]</td>
<td>&lt;</td>
<td>2.2</td>
<td>0.6</td>
<td>0.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15 CT compare</td>
<td>Var. y: SL(a)→ [L(a)]</td>
<td>true, false, and</td>
<td>14.3</td>
<td>0.5</td>
<td>9.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 compare</td>
<td>Var. y: SL(a)→ [L(a)]</td>
<td>true, false, and</td>
<td>1</td>
<td>0.3</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Constant Resource As discussed in Sec. 3, a simple extension to Re3 enables it to verify constant-resource implementations. We showcase this feature in benchmarks 14–16. Benchmark 15 is an example from [46], which compares a public list ys with a secret list zs. By allotting potential only to ys, we guarantee that the resource consumption of the generated program is independent of the length of zs. If this requirement is relaxed (as in benchmark 16), the generated program indeed terminates early, potentially revealing the length of zs to an adversary (in case zs is the shorter of the two lists). Benchmark 14 is a constant-time version of benchmark 7 (insert), which is forced to make recursive calls so as not to reveal the length of the list.

6 Related Work

Resource Analysis Automatic static resource analysis has been extensively studied and is an active area of research. Many advanced techniques for imperative integer programs apply abstract interpretation to generate numerical invariants. The obtained size-change information forms the basis for the computation of actual bounds on loop iterations and recursion depths; using counter instrumentation [26], ranking functions [2, 4, 10, 59], recurrence relations [1, 3], and abstract interpretation itself [13, 73]. Automatic resource analysis techniques for functional programs are based on sized types [67], recurrence relations [16], term-rewriting [5], and amortized resource analysis [31, 34, 37, 58]. There exist several tools that can automatically derive loop and recursion bounds for imperative programs including SPEED [26, 27], KoAT [10], PUBS [1], Rank [4], ABC [7] and LOOPUS [59, 73]. These techniques are passive in the sense that they provide feedback about a program without actively synthesizing or repairing programs.

Domain-Specific Program Synthesis Most program synthesis techniques [18–20, 22, 55, 39, 47, 51, 52, 60, 63, 70, 71] do not explicitly take resource usage into account during synthesis. Many of them, however, leverage domain knowledge to restrict the search space to only include efficient programs [14, 25] or to encode domain-specific performance considerations as part of the functional specification [36, 44, 45].

Synthesis with Quantitative Objectives Two lines of prior work on synthesis are explicitly concerned with optimizing resource usage. One is quantitative automata-theoretic synthesis, which has been used to synthesize optimal Mealy machines [8] and place synchronization in concurrent programs [11, 12, 28]. In contrast, we focus on synthesis of high-level programs that can manipulate custom data structures, which are out of reach for automata-theoretic synthesis.

The second relevant line of work is synthesis-aided compilation [49, 50, 56, 57]. This work is limited to generating low-level straight-line code, which is an easy target for correctness validation and cost estimation. Perhaps the closest work to ours is the Synapse tool [9], which supports a richer space of programs, but requires extensive guidance from the user (in the form of meta-sketches), and relies on bounded reasoning, which can only provide correctness and optimality guarantees for a finite set of inputs. In contrast, we use type-based verification and resource analysis techniques, which enable ReSyn to handle high-level recursive programs and provide guarantees for an unbounded set of inputs.

Acknowledgments This article is based on research supported by the United States Air Force under DARPA AA Contract FA8750-18-C-0092 and DARPA STAC Contract FA8750-15-C-0082, and by the National Science Foundation under SaTC Award 1801369, SHF Award 1812876, and CAREER Award 1845514. Any opinions, findings, and conclusions contained in this document are those of the authors and do not necessarily reflect the views of the sponsoring organizations.
References


