Synthesis with Asymptotic Resource Bounds

Qinheping Hu(✉), John Cyphert, Loris D'Antoni, and Thomas Reps

University of Wisconsin-Madison, Madison, USA
{qhu28,jcyphert,ldantoni,treps}@wisc.edu

Abstract. We present a method for synthesizing recursive functions that satisfy both a functional specification and an asymptotic resource bound. Prior methods for synthesis with a resource metric require the user to specify a *concrete* expression exactly describing resource usage, whereas our method uses big-O notation to specify the *asymptotic* resource usage. Our method can synthesize programs with complex resource bounds, such as a sort function that has complexity $O(n \log(n))$.

Our synthesis procedure uses a type system that is able to assign an asymptotic complexity to terms, and can track recurrence relations of functions. These typing rules are justified by theorems used in analysis of algorithms, such as the Master Theorem and the Akra-Bazzi method. We implemented our method as an extension of prior type-based synthesis work. Our tool, SynPlexity, was able to synthesize complex divide-and-conquer programs that cannot be synthesized by prior solvers.

1 Introduction

Program synthesis is the task of automatically finding programs that meet a given behavioral specification, such as input-output examples or complete formal specifications. Most of the work on program synthesis has been devoted to qualitative synthesis, i.e., finding *some* correct solution. However, programmers often want more than just a correct solution—they may want the program that is smallest, most likely, or most efficient. While there are some techniques for adding a quantitative *syntactic* objective in program synthesis [12]—e.g., finding a smallest solution, or a most likely solution with respect to some distribution—little attention has been devoted to quantitative *semantic* objectives—e.g., synthesizing a program that has a certain asymptotic complexity.

Recently, Knoth et al. [16] studied the problem of resource-guided program synthesis, where the goal is to synthesize programs with limited resource usage. Their approach, which combines refinement-type-directed synthesis [18] and automatic amortized resource analysis (AARA) [9], is restricted to *concrete* resource bounds, where the user must specify the *exact* resource usage of the synthesized program as a *linear* expression. This limitation has two drawbacks: (i) the user must have insights about the coefficients to put in the supplied

A. Silva and K. R. M. Leino (Eds.) CAV 2021, LNCS 12759, pp. 783–807, 2021.
https://doi.org/10.1007/978-3-030-81685-8_37

bound—which means that the user has to provide details about the complexity of code that does not yet exist; (ii) the limitation to linear bounds means that the user cannot specify resource bounds that involve logarithms, such as $O(\log n)$ and $O(n \log n)$, common in problems based on divide and conquer.

In this paper, we introduce SynPlexity, a type-system paired with a type-directed synthesis technique that addresses these issues. In SynPlexity, the user provides as input a refinement type that describes both the functionality and the *asymptotic* (big-O) resource usage of a program. For example, a user might ask SynPlexity to synthesize an implementation of a sorting function with resource usage $O(n \log n)$, where n is the length of the input list. As in prior work, SynPlexity also takes as input a set of auxiliary functions that the synthesized program can use. SynPlexity then uses a type-directed synthesis algorithm to search for a program that has the desired functionality, and satisfies the asymptotic resource bound. SynPlexity's synthesis algorithm uses a new type system that can reason about the asymptotic complexity of functions. To achieve this goal, this type system uses two ideas.

1. The type system uses *recurrence relations* instead of concrete resource potentials [9] to reason about the asymptotic complexity of functions. For example, the recurrence relation $T(u) \leq 2T(\lfloor \frac{u}{2} \rfloor) + O(u)$ denotes that on an input of size u, the function will perform at most two recursive calls on inputs of size at most $\lfloor \frac{u}{2} \rfloor$, and will use at most $O(u)$ resources outside of the recursive calls.[1] For a given recurrence relation, our type system uses refinement types to guarantee that a function typed with this recurrence relation performs the correct number of recursive calls on parameters of the appropriate sizes.
2. These typing rules are justified by classic theorems from the field of analysis of algorithms, such as the Master Theorem [5], the Akra-Bazzi method [1], or C-finite-sequence analysis [13].

Guéneau et al. observed that reasoning with O-notation can be tricky, and exhibited a collection of plausible-sounding, but flawed, inductive proofs [8, §2]. We avoid this pitfall via SynPlexity's type system, which establishes whether a term satisfies a given recurrence relation. SynPlexity uses theorems that connect the form of a recurrence relation—e.g., the number of recursive calls, and the argument sizes in the subproblems—to its asymptotic complexity. In particular, the SynPlexity type system does not encode inductive proofs of the kind that Guéneau et al. show can go astray.

SynPlexity can synthesize functions with complexities that cannot be handled by existing type-directed tools [16,18], and compares favorably with existing tools on their benchmarks. Furthermore, for some domains, SynPlexity's type system allows us to discover auxiliary functions automatically (e.g., the split function of a merge sort), instead of requiring the user to provide them.

[1] The recurrence relation above is one possible instantiation of the Master Theorem [5, §4.5 and §4.6]; it can also be instantiated as $T(u) \leq 2T(\lceil \frac{u}{2} \rceil) + O(u)$. The type system makes use of certain templates for instantiating the algorithm-analysis theorems that we use. The use of templates means that the type system does not use all possible instantiations, but all instantiations used in the type system are valid ones.

Contributions. The contributions of our work are as follows:

- A type system that uses refinement types to check whether a program satisfies a recurrence relation over a specified resource (Sect. 3).
- A type-directed algorithm that uses our type system to synthesize functions with given resource bounds (Sect. 4, Sect. 5).
- SynPlexity, an implementation of our algorithm that, unlike prior tools, can synthesize programs with desired asymptotic complexities (Sect. 6).

Complete proofs and details of the type system can be found in the technical report [11].

2 Overview

In this section, we illustrate the main components of our algorithm through an example. Consider the problem of synthesizing a function `prod` that implements the multiplication of two natural numbers, x and y. We want an efficient solution whose time complexity is $O(\log x)$ with respect to the value of the first argument x. In Subsect. 2.1, we show how existing type-directed synthesizers solve this problem in the absence of a complexity-bound constraint. In Subsect. 2.2, we illustrate how to specify asymptotic bounds in type-directed synthesis problems. In Subsect. 2.3, we show how the tracking of recurrence relations can be used to establish complexity bounds as well as guide the synthesis search.

2.1 Type-Directed Synthesis

We first review one of the state-of-the-art type-directed synthesizers, Synquid, through the aforementioned example—i.e., synthesizing a program `prod` that computes the product of two natural numbers. In Synquid, the specification is given as a refinement type that describes the desired behavior of the synthesized function. We specify the behavior of `prod` using the following refinement-type:

$$\texttt{prod} :: \texttt{x:}\{\texttt{Int} \mid v \geq 0\} \rightarrow \texttt{y:}\{\texttt{Int} \mid v \geq 0\} \rightarrow \{\texttt{Int} \mid v = \texttt{x} * \texttt{y}\}.$$

Here the types of the inputs `x` and `y`, as well as the return type of `prod` are refined with predicates. The refinement $\{\texttt{Int} \mid v \geq 0\}$ declares `x` and `y` to be non-negative, and the refinement $\{\texttt{Int} \mid v = \texttt{x}*\texttt{y}\}$ of the return type declares the output value to be an integer that is equal to the product of the inputs `x` and `y`. In addition to the specification, the synthesizer receives as input some signatures of auxiliary functions it can use. The specifications of auxiliary functions are also given as refinement types. In our example, we have the following functions:

$$\texttt{even} :: \texttt{x:Int} \rightarrow \{\texttt{Bool} \mid \texttt{x mod } 2 = 0\} \qquad \texttt{dec} :: \texttt{x:Int} \rightarrow \{\texttt{Int} \mid v = \texttt{x} - 1\}$$

$$\texttt{double} :: \texttt{x:Int} \rightarrow \{\texttt{Int} \mid v = \texttt{x} + \texttt{x}\} \qquad \texttt{div2} :: \texttt{x:Int} \rightarrow \{\texttt{Int} \mid v = \lfloor \frac{\texttt{x}}{2} \rfloor\}$$

$$\texttt{plus} :: \texttt{x:Int} \rightarrow \texttt{y:Int} \rightarrow \{\texttt{Int} \mid v = \texttt{x} + \texttt{y}\}$$

With the above specification and auxiliary functions, SYNQUID will output the implementation of `prod` shown in Eq. (1).

$$\texttt{prod} = \lambda \texttt{x}.\lambda y.\ \texttt{if x==0 then x else plus y (prod (dec x) y)} \quad (1)$$

SYNQUID uses a sophisticated type system to guarantee that the synthesized term has the desired type. Furthermore, SYNQUID uses its type system to prune the search space by only enumerating terms that can possibly be typed, and thus meet the specification. Terms are enumerated in a top-down fashion, and appropriate specifications are propagated to sub-terms. As an example, let us see how SYNQUID synthesizes the function body—an `if-then-else` term—in Eq. (1), which is of refinement type $\{\texttt{Int} \mid v = \texttt{x} * \texttt{y}\}$. SYNQUID will first enumerate an integer term for the `then` branch—a variable term `x`. Then, with the `then` branch fixed, the condition guard must be refined by some predicate φ under which the `then` branch (the term `x` refined by $v = \texttt{x}$) fulfills the goal type $\{\texttt{Int} \mid v = \texttt{x} * \texttt{y}\}$, i.e., $\forall \texttt{x}, \texttt{y} \geq 0. \varphi \wedge v = \texttt{x} \implies v = \texttt{x} * \texttt{y}$. With this constraint, SYNQUID identifies the term $\texttt{x} == 0$ as the condition. Finally, SYNQUID propagates the negation of the condition to the `else` branch—the `else` branch should be a term of type $\{\texttt{Int} \mid v = \texttt{x} * \texttt{y}\}$ with the path condition $\texttt{x} \neq 0$—and enumerates the term `plus y (prod (dec x) y)` as the else branch, which has the desired type.

The program in Eq. (1) is correct, but inefficient. Let us count each call to an auxiliary function as one step; and let $T(x)$ denote the number of steps in which the program runs with input x. The implementation in Eq. (1) runs in $\Theta(x)$ steps because $T(x)$ satisfies the recurrence $T(x) = T(x-1)+2$, implying $T(x) \in \Theta(x)$. Because, SYNQUID does not provide a way to specify resource bounds, such as $O(\log x)$; one cannot ask SYNQUID to find a more efficient implementation.

2.2 Adding Resource Bounds

In our tool, SYNPLEXITY, one can specify a synthesis problem with an asymptotic resource bound, and can ask SYNPLEXITY to find an $O(\log x)$ implementation of `prod`. To express this intent, the user needs to specify (1) the asymptotic resource-usage bound the synthesized program should satisfy, (2) the cost of each provided auxiliary function, and (3) the size of the input to the program.

Asymptotic Resource Bound. We extend refinement types with resource annotations. The annotated refinement types are of the form $\langle \tau; \alpha \rangle$ where τ is a regular refinement type, and α is a resource annotation. The following example asks the synthesizer to find a solution with the resource-usage bound $O(\log u)$:

$$\texttt{prod} :: \langle \texttt{x:}\{\texttt{Int} \mid v \geq 0\} \to \texttt{y:}\{\texttt{Int} \mid v \geq 0\} \to \{\texttt{Int} \mid v = \texttt{x} * \texttt{y}\}, O(\log u) \rangle$$

Cost of Auxiliary Functions. The auxiliary functions supplied by the user serve as the API in terms of which the synthesized program is programmed. Thus, the resource usage of the synthesized program is the sum of the costs of all auxiliary calls made during execution. We allow users to assign a polynomial cost $O(u^a)$,

for some constant a, or a constant cost $O(1)$ to each auxiliary function. Here, u is a free variable that represents the size of the problem on which the auxiliary function is called.

In the `prod` example, all auxiliary functions are assigned constant cost, e.g., we give `even` the signature $\texttt{even} :: \langle \texttt{x:Int} \to \{\texttt{Bool} \mid \texttt{x mod } 2 = 0\}, O(1)\rangle$.

Size of Problems. The user needs to specify a size function, $\texttt{size:}\tau \to \texttt{Int}$, that maps inputs to their sizes, e.g., when synthesizing the sorting function for an input of type `list`, the size function can be $\lambda \texttt{l}.|l|$—the length of the input list. In the `prod` example, the size function is $\texttt{size} = \lambda \texttt{x}.\lambda \texttt{y}.\texttt{x}$.

2.3 Checking Recurrence Relations

We extend SYNQUID's refinement-type system with resource annotations, so that the extended type system enforces the resource usage of terms. The idea of the type system is to check if the given function satisfies some recurrence relation. If so, it can infer that the function also satisfies the corresponding resource bound. For example, according to the Master Theorem [3], if a function f satisfies the recurrence relation $T(u) \leq T(\lfloor \frac{u}{2} \rfloor) + O(1)$ where u is the size of the input, then the resource usage of f is bounded by $O(\log u)$. Checking if a function satisfies a given recurrence relation can be performed by checking if the function contains appropriate recursive calls—e.g., if a function contains one recursive call to a sub-problem of half size, and consumes only a constant amount of resources in its body, then it satisfies $T(u) \leq T(\lfloor \frac{u}{2} \rfloor) + O(1)$.

The following rule is an example of how we connect recurrence annotations and resource bounds.

$$\frac{x : \tau_1, f : \tau_1 \to \tau_2, \Gamma \vdash t :: \langle \tau_2; ([1, \lfloor \frac{u}{2} \rfloor], O(1))\rangle}{\Gamma \vdash (\texttt{fix } f.\ \lambda x.t) :: \langle \tau_1 \to \tau_2; O(\log u)\rangle}$$

The rule instantiates the Master Theorem example above. Note that, the annotation $([1, \lfloor \frac{u}{2} \rfloor], O(1))$ states that the function body contains up to one recursive call to a problem of size $\lfloor \frac{u}{2} \rfloor$, and the resource usage in the body of t (aside from calls to f itself) is bounded by $O(1)$. The rule states that if the function body t of type τ_2 contains one recursive call to a sub-problem of size $\lfloor \frac{u}{2} \rfloor$, then the function will be bounded by $O(\log u)$.

The implementation of `prod` shown in Eq. (2) runs in $O(\log x)$ steps.

```
prod = λx. λy.if x == 0 then x else                              (2)
       if even x then  double (prod (div2 x) y)
                 else  plus y (double (prod (div2 x) y))
```

To check that, SYNPLEXITY's type system counts the number of recursive calls along any path of the function. There are three paths (two nested if-then-else terms) in the program, and at most one recursive call along each path. Also, one can check that the problem size of each recursive call is no more than $\lfloor \frac{x}{2} \rfloor$.

$$
\begin{array}{lll}
 & \text{Term} & t ::= e \mid b \\
 & \text{E-term} & e ::= x \mid c \mid \texttt{true} \mid \texttt{false} \mid x\, e_1 \ldots e_n \\
\text{I-term} \left\{ \right. & \text{Branching term} & b ::= \texttt{if}\ e\ \texttt{then}\ t\ \texttt{else}\ t \\
 & & \quad \mid \texttt{match}\ e\ \texttt{with} \mid_i \texttt{C}_i\ (x_i^1 \ldots x_i^n) \mapsto t_i \\
 & \text{Function term} & f ::= \texttt{fix f}.\lambda x_1 \ldots \lambda x_n.t
\end{array}
$$

Fig. 1. SynPlexity syntax.

$$
\begin{array}{lrl}
\text{Logical expr.} & \varphi, \phi, \psi ::= & x \mid \texttt{m}(\psi) \mid \top \mid \bot \mid c \mid \psi\ \texttt{mod}\ \psi \mid \psi \wedge \psi \mid \psi \vee \psi \\
 & & \mid \neg\psi \mid \psi = \psi \mid \psi * \psi \mid \psi / \psi \mid \psi + \psi \mid \psi - \psi \\
\text{Ordinary type} & B ::= & \texttt{Bool} \mid \texttt{Int} \mid D \\
\text{Refinement type} & \tau ::= & \{B \mid \varphi\} \mid x_1 : \tau_1 \to \ldots \to x_n : \tau_n \to y : \tau \\
\text{Annotated type} & \gamma ::= & \langle \tau; \alpha \rangle \\
\text{Recurrence ann.} & \alpha ::= & (\ [c_1, \phi_1]_{\texttt{f}}, \ldots, [c_n, \phi_n]_{\texttt{f}}; O(\psi)\ ) \\
\text{Environment} & \Gamma ::= & \cdot \mid x : \gamma; \Gamma \mid \varphi; \Gamma \mid \texttt{recFun} := x; \Gamma \mid \texttt{args} := x_1 \ldots x_n; \Gamma
\end{array}
$$

Fig. 2. SynPlexity types.

For example, the recursive call `prod (div2 x) y` calls to a problem with size `div2 x`, which is consistent with $[1, \lfloor \frac{u}{2} \rfloor]$, and u is `x` because `size x y` $=$ `x`. In addition, the condition that the resource usage of the body is bounded by $O(1)$ is satisfied because only auxiliary functions with constant cost are called.

3 The SynPlexity Type System

In this section, we present our type system. First, we give the surface language and the types, which extend the Synquid liquid-types framework with resource annotations (Subsect. 3.1). Then, we show the semantics of our language (Subsect. 3.2). Finally, we present SynPlexity's type system (Subsect. 3.3), which our synthesis algorithm uses to synthesize programs with desired resource bounds.

3.1 Syntax and Types

Syntax. Consider the language shown in Fig. 1. In the language, we distinguish between two kinds of terms: *elimination terms* (E-terms) and *introduction terms* (I-terms). E-terms consist of variable terms, constant values c, and application terms. Condition guards and match scrutinies can only be E-terms. I-terms are branching terms and function terms. The key property of I-terms is that if the type of any I-term is known, the types of its sub-terms are also known (which is not the case for E-terms).

Types. Our language of types, presented in Fig. 2, extends the one of Synquid [18] with *recurrence annotations*, which are used to track recurrence relations on functions. To simplify the presentation, we ignore some of the features of the type system of Synquid [18] that do not affect our algorithm. In particular,

we do not discuss polymorphic types and the enumerating strategy that ensures that only terminating programs are synthesized. However, our implementation is built on top of SYNQUID, and supports both of those features.

Logical expressions are built from variables, constants, arithmetic operators, and other user-defined logical functions. Logical expressions in our type system can be used as refinements φ, size expressions ϕ, or bound expressions ψ. Refinements φ are logical predicates used to refine ordinary types in refinement types $\{B \mid \varphi\}$. We usually use a reserved symbol v as the free variable in φ, and let v represents the inhabitants, i.e., inhabitants of the type $\{B \mid \varphi\}$ are valuations of v that satisfy φ. For example, the type $\{\texttt{Int} \mid v \texttt{ mod } 2 = 0\}$ represents the even integers. Size expressions and bound expressions are used in recurrence annotations, and are explained later.

Ordinary types includes primitive types and user-defined algebraic datatypes D. Datatype constructors `C` are functions of type $\tau_1 \to \ldots \to \tau_n \to D$. For example, the datatype `List(Int)` has two constructors: `Cons` : `Int` $\to$ `List(Int)` $\to$ `List(Int)`, and `Nil` : `List(Int)`. Refinement types are ordinary types refined with some predicates ψ, or arrow types. Note that, unlike SYNQUID's type system, SYNPLEXITY's type system does not support higher-order functions[2]—i.e., arguments of functions have to be non-arrow types. All occurrences of τ_i and τ in arrow types $x_1 : \tau_1 \to \ldots \to x_n : \tau_n \to y : \tau$ have to be ordinary types or refined ordinary types. We will discuss this limitation in Sect. 7.

We use `recFun` to denote the name of the function for which we are performing type-checking, and `args` to denote the tuple of arguments to `recFun`. For example, in the function `prod` shown in Eq. (1), `recFun=prod` and `args=x y`. An environment Γ is a sequence of variable bindings $x : \gamma$, path conditions φ, and assignments for variables `recFun` and `args`.

Recurrence Annotations. Annotated types are refinement types annotated with recurrence annotations. A recurrence annotation is a pair $([c_1, \phi_1]_{\texttt{f}}, \ldots, [c_n, \phi_n]_{\texttt{f}}; O(\psi))$ consisting of (1) a set of recursive-call costs of the form $[c_i, \phi_i]_{\texttt{f}}$, and (2) a resource-usage bound of the form $O(\psi)$. Intuitively, a recurrence annotation tracks the number c_i of recursive calls to `f` of size ϕ_i in the first element $[c_1, \phi_1]_{\texttt{f}}, \ldots, [c_n, \phi_n]_{\texttt{f}}$ of the pair, as well as the asymptotic resource usage of the *body* of the function (the second element $O(\psi)$). Using these quantities, we can compute a recurrence relation describing the resource usage of the function `recFun`. For example, the recurrence annotation $([1, u-1]_{\texttt{f}}, [1, u-2]_{\texttt{f}}; O(1))$ corresponds to the recurrence relation $T_{\texttt{f}}(u) \leq T_{\texttt{f}}(u-1) + T_{\texttt{f}}(u-2) + O(1)$.

A *recursive-call cost* $[c, \phi]_{\texttt{f}}$ associated with a function `f` denotes that the body of `f` can contain up to c recursive calls to subproblems that have sizes up to the one specified by size expression ϕ. A size expression, ϕ, is a polynomial over a reserved variable symbol u that represents the size of the top-level problem. In our paper, a *problem* with respect to a function $g :: x_1 : \tau_1 \to \ldots \to x_n : \tau_n \to y : \tau$ is a tuple of terms $e_1 \ldots e_n$, to which g can be applied—i.e., e_i has type τ_i for all

[2] However, the type system can be extended to support restricted higher-order functions (Sect. 5).

i from 1 to n. For the problems of function g, the size of each problem is defined by a *size function* $\texttt{size}_g$—a user-defined logical function that has type $\tau_1 \to \ldots \to \tau_n \to \texttt{Int}$; i.e., it takes a problem of g as input and outputs a non-negative integer. In the body of g, we say that a recursive-call term $g\ e_1 \ldots e_n$ *satisfies* a size expression ϕ if for all $x_1, \ldots, x_n$, $\texttt{size}_g\ [\![e_1]\!] \ldots [\![e_n]\!] \leq [(\texttt{size}_g\ x_1 \ldots x_n)/u]\phi$, where the x_i's are the arguments of g and the $[\![e_i]\!]$'s are the evaluations of e_i on input $x_1 \ldots x_n$. (See Sect. 3.2 for the formal definition of $[\![\cdot]\!]$.) Note that one annotation can contain multiple recursive-call costs, which allows the function to make recursive calls to sub-problems with different sizes. We often abbreviate $\langle \tau, (O(1)) \rangle$ as τ and omit `f` in recursive-call costs if it is clear from context.

A resource bound $O(\psi)$ of a non-arrow type specifies the bound of the resource usage strictly within the top-level-function body. A resource bound in a signature of an auxiliary function f specifies the resource usage of f. *Bound expressions* ψ in $O(\psi)$ are of the form $u^a \log^b u + c$ where a, b, and c are all non-negative constants, and u represents the size of the top-level problem.

Example 1. In the function `prod` (Eq. (2)), the recursive-call term `prod (div2 x) y` satisfies the recursive-call cost $[1, \lfloor \frac{u}{2} \rfloor]$, because $\texttt{size}_{\texttt{prod}} = \lambda z.\lambda w.z$, and

$$\texttt{size}_{\texttt{prod}}\ [\![\texttt{(div2 x)}]\!]\ [\![\texttt{y}]\!] = [\![\texttt{div2 x}]\!] = \lfloor \frac{\texttt{x}}{2} \rfloor = [(\texttt{size}_{\texttt{prod}}\ \texttt{x}\ \texttt{y})/u] \lfloor \frac{u}{2} \rfloor.$$

3.2 Semantics and Cost Model

We introduce the *concrete*-cost semantics of our language here. The semantics serves two goals: (1) it defines the evaluation of terms (i.e., how to obtain values), which can be used to compute the sizes of problems in application expressions, and (2) it defines the resource usages of terms.

Besides the syntax shown in Fig. 1, implementations of auxiliary functions can contain calls to a tick function $\texttt{tick}(c, t)$, which specifies that c units of a resource are used, and the overall value is the value of t. Note that in our synthesis language, we are not actually synthesizing programs with `tick` functions. We assume that `tick` functions are only called in the implementations of auxiliary functions. In the concrete-cost semantics, a configuration $\langle t, C \rangle$ consists of a term t and a nonnegative integer C denoting the resource usage so far. The evaluation judgment $\langle t, C \rangle \hookrightarrow \langle t', C + C_\Delta \rangle$ states that a term t can be evaluated in one step to a term (or a value) t', with resource usage C_Δ. We write $\langle t, C \rangle \hookrightarrow^* \langle t', C + C_\Delta \rangle$ to indicate the reduction from t to t' in zero or more steps. All of the evaluation judgments are standard. Here we show the judgment of the tick function, where resource usage happens.

$$\frac{}{\langle \texttt{tick}(c, t), C \rangle \hookrightarrow \langle t, C + c \rangle}\ \textsc{Sem-Tick}$$

For a term t, $[\![t]\!]$ denotes the evaluation result of t, i.e., $\langle t, \cdot \rangle \hookrightarrow^* \langle [\![t]\!], \cdot \rangle$.

Example 2. Consider the following function that doubles its input.

```
fix double.λx.if x = 0 then 0 else tick(1,2 + double(x-1)).
```

Let $t_{\texttt{body}}$ denote the function body `if x=0 then 0 else tick(1,2+double(x-1))`. The result of evaluating `double` on input 5 is 10, with resource usage 5.

$$
\begin{aligned}
&\langle(\texttt{fix double}.\lambda\texttt{x}.t_{\texttt{body}})5, 0\rangle \\
\hookrightarrow\ &\langle\texttt{if 5=0 then 0 else tick(1,2+double(4))}, 0\rangle \\
\hookrightarrow\ &\langle\texttt{if false then 0 else tick(1,2+ double(4))}, 0\rangle \\
\hookrightarrow\ &\langle\texttt{tick(1,2+double(4))}, 0\rangle \hookrightarrow \langle\texttt{2+double(4)}, 1\rangle \\
\hookrightarrow\ &\langle\texttt{2+(fix double}.\lambda x.t_{\texttt{body}})4, 1\rangle \hookrightarrow^* \langle\texttt{4+double(3)}, 2\rangle \hookrightarrow^* \langle\texttt{10+double(0)}, 5\rangle \\
\hookrightarrow\ &\langle\texttt{10+(if 0=0 then 0 else tick(1,2+double(0-1)))}, 5\rangle \\
\hookrightarrow\ &\langle\texttt{10+(if true then 0 else tick(1,2+double(0-1)))}, 0\rangle \hookrightarrow \langle\texttt{10+0}, 5\rangle
\end{aligned}
$$

With the standard concrete semantics, the complexity of a function f is characterized by its resource usage when the function is evaluated on inputs of a given size.

Definition 1 (Complexity). *Given a function* $\texttt{fix}\ f.\lambda\overline{y}.t$ *of type* $: \tau_1 \to \tau_2$, *with size function* $\texttt{size}_f : \tau_1 \to \mathbb{N}$, *and suppose that for any possible input* $\overline{x}$, *the configuration* $\langle(\texttt{fix}\ f.\lambda\overline{y}.t)\overline{x}, 0\rangle$ *can be reduced to* $\langle v, C_{\overline{x}}\rangle$ *for some value* v. *Then, if* $T_f : \mathbb{N} \to \mathbb{N}$ *is a function such that, for all,* $\mathrm{u} \geq 0$, $T_f(\mathrm{u}) = \sup_{\overline{x}\ s.t.\ \texttt{size}_f(\overline{x})=\mathrm{u}} C_{\overline{x}}$, *we say that* T_f *is* **the complexity function of** f.

Note that Definition 1 assumes that the top-level term $(\texttt{fix}\ f.\lambda\overline{y}.t)\overline{x}$ can be reduced to some value. Thus, Definition 1 only applies to terminating programs.

Definition 2 (Big-O notation). *Given two integer functions* f *and* g, *we say that* f *dominates* g, *i.e.,* $g \in O(f)$, *if* $\exists c, M \geq 0.\ \forall x \geq c.\ g(x) \leq Mf(x)$.

In the rest of the paper, we use T_f to denote the complexity function of the function f, and we say the complexity of f is *bounded* by a function g if $T_f \in O(g)$. As an example, the complexity of the `double` function shown in Example 2 is $T_{\texttt{double}}(u) := u$, and hence $T_{\texttt{double}}(u) \in O(u)$.

Auxiliary functions. We allow users to supply signatures for auxiliary functions, instead of implementations. It is an obligation on users that such signatures be sensible; in particular, when the user gives the signature $\langle\tau_1 \to \{B \mid \varphi(v, \overline{y})\}, O(\psi(u))\rangle$ for auxiliary function f, the user asserts that there exists some implementation $\texttt{fix}\ f.\lambda\overline{y}.t$ of f, such that: 1) for any input $\overline{x}$, the output of f on $\overline{x}$ satisfies φ, i.e., $\varphi([\![(\texttt{fix}\ f.\lambda\overline{y}.t)\overline{x}]\!], \overline{x})$ is valid; and 2) for any input $\overline{x}$, the complexity of f is bounded by $\psi(u)$, i.e., $T_f(u) \in O(\psi(u))$. Signatures always over-approximate their implementations, as illustrated by the following example.

Example 3. The signature `doubleRelaxed` $:: \langle\texttt{x:Int} \to \{\texttt{Int} \mid v \leq 3 * x\}, O(u^2)\rangle$ describes an auxiliary function that computes *no more* than the input times 3, and has quadratic resource usage. Note that the function `double` shown in Example 2 can be an implementation of this signature because $[\![\texttt{double}(\overline{x})]\!] = 2 * x \leq 3 * x$, and the complexity function $T_{\texttt{double}}(u) = u$ is in $O(u^2)$.

3.3 Typing Rules

The typing rules of SynPlexity are inspired by bidirectional type checking [17] and type checking with cost sharing [16]. Recall that we use `recFun` to denote the name of the function for which we are performing type-checking, and `args` to denote the tuple of arguments to `recFun`.

An environment Γ is a sequence of variable bindings of the form $x : \gamma$, path conditions φ, and assignments of the form $x = \varphi$ for `recFun` and the components of `args`. SynPlexity's typing rules use three judgments: 1) $\Gamma \vdash t :: \gamma$ states that t has type γ, 2) $\Gamma \vdash \gamma_1 <: \gamma_2$ states that γ_2 is a subtype of γ_1, and 3) $\Gamma \vdash \gamma \curlyvee \gamma_1 | \gamma_2$ states that γ_1 and γ_2 share the costs in γ

Subtyping. The subtyping relations between refinement types are relatively standard and can be found in the technical report [11]. The subtyping relations between annotated types allow us to compare resource consumption of recurrence annotations. The following is the rule for comparing recursive-call costs.

$$\frac{c' > c \quad \Gamma \models \forall u.\ \phi' \geq \phi}{\Gamma \vdash [c, \phi] <: [c', \phi']} <:\text{-Rec}$$

For example, if one branch of some branching term has type $\langle \tau, ([1, \lfloor \frac{u}{3} \rfloor], O(\psi)) \rangle$, it can be over-approximated by a super type $\langle \tau, ([1, \lfloor \frac{u}{2} \rfloor], O(\psi)) \rangle$. The idea is that the resource usage of an application calling to a problem of size $\lfloor \frac{u}{2} \rfloor$ will be larger than the resource usage of the application calling to a smaller problem of size $\lfloor \frac{u}{3} \rfloor$ (assuming all resource usages are monotonic).

Subtyping rules also allow the type system to compare branches with a different number of recursive calls. For example, base cases of recursive procedures have no recursive calls, and thus have types of the form $\langle \tau, ([], O(\psi)) \rangle$. With subtyping, these types can be over-approximated by types of the form $\langle \tau, ([c, \phi], O(\psi)) \rangle$.

Cost Sharing. When a term has more than one sub-term in the same path, e.g., the condition guard and the `then` branch are in the same path in an `ite` term, the recursive-call costs of the term will be shared among its sub-terms. The sharing operator $\alpha \curlyvee \alpha_1 | \alpha_2$ partitions the recursive-call costs of α into α_1 and α_2—i.e., the sum of the costs in α_1 and α_2 equals the cost in α. The following is the sharing rule for a single recursive-call cost:

$$\frac{c_1, c_2 \geq 0 \qquad c_1 + c_2 \leq c}{\Gamma \vdash [c, \phi] \curlyvee [c_1, \phi] \mid [c_2, \phi]} \text{S-Pot}$$

Other sharing rules can be found in the technical report [11]. The idea is that a single cost c can be shared as two costs c_1 and c_2 such that their sum is no more than c. An annotation can be shared as two parts if every recursive cost $[c_i, \phi_i]$ in it can be shared as two parts $[c_i^1, \phi_1]$ and $[c_i^2, \phi_2]$. Finally, annotations can also be shared as more than two parts.

Table 1. Annotations that can be used to instantiate the rule T-Abs.

	Bound (B)	Recurrence relation	*Annotation* (A)
Master Theorem	$O(\log u)$	$T(u) \leq T(\lfloor \frac{u}{d} \rfloor) + O(1),\ d \geq 2$	$([1, \lfloor \frac{u}{d} \rfloor]; O(1)),\ d \geq 2$
	$O(u \log u)$	$T(u) \leq dT(\lfloor \frac{u}{d} \rfloor) + O(u),\ d \geq 2$	$([d, \lfloor \frac{u}{d} \rfloor]; O(u)),\ d \geq 2$
Akra–Bazzi	$O(u \log u)$	$T(u) \leq T(\lceil \frac{u}{2} \rceil) + T(\lfloor \frac{u}{2} \rfloor) + O(u)$	$([1, \lceil \frac{u}{2} \rceil], [1, \lfloor \frac{u}{2} \rfloor]; O(u))$
C-Finite Seq.	$O(u)$	$T(u) \leq T(u-d) + O(1),\ d \geq 1$	$([1, u-d]; O(1)),\ d \geq 1$
	$O(u^2)$	$T(u) \leq T(u-d) + O(u),\ d \geq 1$	$([1, u-d]; O(u)),\ d \geq 1$

Example 4. There are multiple ways to share the recurrence annotation $([1, \lfloor \frac{u}{2} \rfloor], [1, \lceil \frac{u}{2} \rceil]; O(u))$:

$$\Gamma \vdash ([1, \lfloor \frac{u}{2} \rfloor], [1, \lceil \frac{u}{2} \rceil]; O(u)) \curlyvee ([1, \lfloor \frac{u}{2} \rfloor], [1, \lceil \frac{u}{2} \rceil]; O(u)) \mid ([\,], O(u)),$$

where one annotation contains both recursive-call costs $[1, \lfloor \frac{u}{2} \rfloor], [1, \lfloor \frac{u}{2} \rfloor]$; and the other contains no recursive-call cost. And

$$\Gamma \vdash ([1, \lfloor \frac{u}{2} \rfloor], [1, \lceil \frac{u}{2} \rceil]; O(u)) \curlyvee ([1, \lfloor \frac{u}{2} \rfloor]; O(u)) \mid ([1, \lceil \frac{u}{2} \rceil]; O(u)),$$

where each annotation contains one recursive-call cost.

Function Terms. The rule T-Abs shown below is really a rule-schema that is parameterized in terms of an annotation (A) for a function body t, and a resource bound (B) for the function term. If the function body t has some recurrence relation described by the annotation A, then the function `f` will satisfy the resource-usage bound B. Some example patterns are shown in Table 1.[3]

$$\frac{\begin{array}{c}\Gamma' = [\texttt{recFun} \leftarrow \texttt{f}][\texttt{args} \leftarrow x_1 \ldots x_n]\Gamma \\ \gamma_f = \langle x_1 : \tau_1 \rightarrow \ldots \rightarrow x_n : \tau_n \rightarrow \tau, (B)\rangle \\ \Gamma'; x_1 : \langle \tau_1, O(1)\rangle; \ldots; x_n : \langle \tau_n, O(1)\rangle; \texttt{f} : \gamma_{\texttt{f}} \vdash t :: \langle \tau, (A)\rangle\end{array}}{\Gamma \vdash \texttt{fix f}.\lambda x_1 \ldots \lambda x_n.t :: \langle x_1 : \tau_1 \rightarrow \ldots \rightarrow x_n : \tau_n \rightarrow \tau, (B)\rangle}\ \text{T-Abs}$$

For example, if the annotation of the function body is $([1, \lfloor \frac{u}{2} \rfloor]; O(1))$, then the resource bound in the function type will be $O(\log u)$, i.e., the resource usage of `f` is bounded by $O(\log(\texttt{size}_{\texttt{f}}\ x_1 \ldots x_n))$.

At the same time, the rule stores the name `f` of the recursive function into `recFun`, and its arguments as a tuple into `args`.

Example 5. We use a function $\texttt{fix bar}.\lambda x.\texttt{if}\ x = 1\ \texttt{then}\ 1\ \texttt{else}\ 1 + \texttt{bar}(\texttt{div2}\ x)$ to illustrate the first pattern in Table 1. The body of `bar` has the annotated type $([1, \lfloor \frac{u}{2} \rfloor]; O(1))$ because (i) there exists only one recursive call to a sub-problem whose size is half of the top-level problem size u, and (ii) the resource usage inside the body is constant (with the assumption that all auxiliary functions

[3] The patterns shown in Table 1 are those we used in the implementation. Patterns capturing other recurrence relations can be added to the type system if needed.

have constant resource usage). This type appears in row 1, column 4 of Table 1. Consequently, the recurrence relation of `bar` is $T(u) \leq T(\lfloor \frac{u}{2} \rfloor) + O(1)$ (row 1, column 3), where $T(u)$ is the resource usage of `bar` on problems with size u. Finally, according to the Master Theorem, the resource usage of `bar` is bounded by $O(\log u)$ (row 1, column 2).

Branching Terms. In rule T-If, the condition has type `Bool` with refinement φ_e. Two branches have different types—the `then` branch follows the path condition φ_e, and the refinement φ of the branch term, while the `else` branch follows the path condition $\neg\varphi_e$. By having both branches share the same recurrence annotation, T-If can introduce some imprecision. In particular, if the branches belong to different complexity classes, the annotation of the conditional term will be the upper bound of both branches.

$$\frac{\Gamma \vdash \alpha \curlyvee \alpha_1 | \alpha_2 \quad \Gamma \vdash e :: \langle \{\texttt{Bool} \mid \varphi_e\}, \alpha_1 \rangle \quad \Gamma, \varphi_e \vdash t_1 :: \langle \{B \mid \varphi\}, \alpha_2 \rangle \quad \Gamma, \neg\varphi_e \vdash t_2 :: \langle \{B \mid \varphi\}, \alpha_2 \rangle}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ t_1\ \texttt{else}\ t_2 :: \langle \{B \mid \varphi\}, \alpha \rangle}\ \text{T-If}$$

The rule T-Match is slightly different: (1) there can be more than two branches, (2) all branches have the same type $\langle \tau, \alpha_2 \rangle$, and (3) variables in each case $\texttt{C}_i\ (x_i^1 \ldots x_i^n)$ are introduced in the corresponding branch.

$$\frac{\Gamma \vdash \alpha \curlyvee \alpha_1 | \alpha_2 \quad \Gamma \vdash e :: \langle \tau_s, \alpha_1 \rangle \quad \texttt{C}_i = \tau_1 \to \ldots \to \tau_n \to \tau_s \quad \Gamma; x_i^1 : \tau_1; \ldots; x_i^n : \tau_n \vdash t_i :: \langle \tau, \alpha_2 \rangle}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ |_i\ \texttt{C}_i\ (x_i^1 \ldots x_i^n) \mapsto t_i :: \langle \tau, \alpha \rangle}\ \text{T-Match}$$

E-terms. The typing rules for E-terms are shown in Fig. 3. The two rules for application terms are the key rules of our type system. Let us first look at the E-RecApp rule for recursive-call terms. Recall that the recursive-call annotation tracks the number of recursive calls and the sizes of sub-problems. If the term $\texttt{f}\ e_1 \ldots e_n$ is a recursive call—i.e., $\Gamma(\texttt{recFun}) = \texttt{f}$—the number of recursive calls in one of the recursive-call costs will increase by one—i.e., $[c_k, \phi_k]$ in the premise becomes $[c_k + 1, \phi_k]$ in the conclusion. Also, we want to make sure that the size of the subproblem this application term is called on satisfies the size expression ϕ_k. If each callee term is refined by the predicate φ_i, i.e., $\Gamma \vdash e_i :: \langle \{B_i \mid \varphi_i\}, \alpha_i \rangle$, then the fact that the size of the problem $e_1 \ldots e_n$ satisfies ϕ_k can be implied by the validity of the predicate $\bigwedge_{i=1}^{m} [y_i/v]\varphi_i \Rightarrow (\texttt{size}\ y_1 \ldots y_m \leq [\texttt{size}\ \Gamma(\texttt{args})/u]\phi_k)$. We introduce validity checking, written $\Gamma \models \varphi$, to state that a predicate expression φ is always true under any instance of the environment Γ.

Example 6. Recall Eq. (2). According to the rule T-RecApp, the recursive call `prod (div2 x) y` has type $\langle \{\texttt{Int} \mid v = \lfloor \frac{\texttt{x}}{2} \rfloor * \texttt{y}\}, ([1, \frac{u}{2}]); O(1) \rangle$. Note that the first argument `(div2 x)` has type $\{\text{Int} \mid v = \lfloor \frac{\texttt{x}}{2} \rfloor\}$, the second argument `y` has

$\boxed{\Gamma \vdash e :: \gamma}$

$$\frac{\Gamma \vdash e :: \gamma' \qquad \Gamma \vdash \gamma' <: \gamma}{\Gamma \vdash e :: \gamma}\ \text{E-SubType} \qquad \frac{\Gamma(x) = \gamma}{\Gamma \vdash x :: \gamma}\ \text{E-Var}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{g} : \langle x_1 : \tau_1 \to \ldots \to x_m : \tau_m \to \{B \mid \varphi\}, (O(\psi_{\mathtt{g}}))\rangle \\ \Gamma(\mathtt{recFun}) \neq \mathtt{g} \qquad \Gamma \vdash ([c_1, \phi_1], \ldots, \ldots, [c_n, \phi_n]; O(\psi)) \curlyvee \alpha_1 | \ldots | \alpha_m \\ \forall 1 \leq i \leq m \qquad \Gamma \vdash e_i :: \langle \{B_i \mid \varphi_i\}, \alpha_i\rangle \qquad \Gamma \vdash \{B_i \mid \varphi_i\} <: \tau_i \\ \Gamma \models \bigwedge_{i=1} [y_i/v]\varphi_i \Rightarrow ([\mathtt{size}_{\mathtt{g}}\ y_1 \ldots y_m/u]\psi_{\mathtt{g}} \in O([\mathtt{size}\ \Gamma(\mathtt{args})/u]\psi)) \\ \tau = \{B \mid [z_i/x_i]\varphi \wedge \bigwedge_{i=1} [z_i/v]\varphi_i\}\ z_i \notin FV(\varphi), z_i \notin FV(\varphi_i)\end{array}}{\Gamma \vdash \mathtt{g} e_1 \ldots e_m :: \langle \tau, ([c_1, \phi_1], \ldots, [c_n, \phi_n]; O(\psi))\rangle}\ \text{E-App}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{f} : \langle x_1 : \tau_1 \to \ldots \to x_m : \tau_m \to \{B \mid \varphi\}, \alpha\rangle \qquad \Gamma(\mathtt{recFun}) = \mathtt{f} \\ \Gamma \vdash ([c_1, \phi_1], \ldots, [c_k, \phi_k], \ldots, [c_n, \phi_n]; O(\psi)) \curlyvee \alpha_1 | \ldots | \alpha_m \\ \forall 1 \leq i \leq m \qquad \Gamma \vdash e_i :: \langle \{B_i \mid \varphi_i\}, \alpha_i\rangle \qquad \Gamma \vdash \{B_i \mid \varphi_i\} <: \tau_i \\ \Gamma \models \bigwedge_{i=1} [y_i/v]\varphi_i \Rightarrow (\mathtt{size}\ y_1 \ldots y_m \leq [\mathtt{size}\ \Gamma(\mathtt{args})/u]\phi_k) \\ \tau = \{B \mid [z_i/x_i]\varphi \wedge \bigwedge_{i=1} [z_i/v]\varphi_i\}\ z_i \notin FV(\varphi), z_i \notin FV(\varphi_i)\end{array}}{\Gamma \vdash \mathtt{f} e_1 \ldots e_m :: \langle \tau, ([c_1, \phi_1], \ldots [c_k + 1, \phi_k], \ldots, [c_n, \phi_n]; O(\psi))\rangle}\ \text{E-RecApp}$$

Fig. 3. Typing rules of E-terms

type $\{\text{Int} \mid v = \mathtt{y}\}$, the size function is $\mathtt{size}_{\mathtt{prod}} = \lambda z.\lambda w.z$, and the arguments in the context are $\Gamma(\mathtt{args}) = \mathtt{x}\ \mathtt{y}$. Therefore, the following predicate is valid:

$$[y_1/v](v = \lfloor \frac{x}{2} \rfloor) \wedge [y_2/v](v = y) \Rightarrow \mathtt{size}_{\mathtt{prod}}\ y_1\ y_2 = [\mathtt{size}_{\mathtt{prod}}\ \Gamma(\mathtt{args}/u)]\lfloor \frac{u}{2} \rfloor$$
$$\Leftrightarrow\ (y_1 = \lfloor \frac{x}{2} \rfloor) \wedge (y_2 = y) \Rightarrow y_1 = \lfloor \frac{x}{2} \rfloor.$$

The rule E-App states that callees have types τ_i, and the resource usage does not exceed the bound $O(\psi)$ in the annotation. Similar to the E-RecApp rule, the size of the problem $\mathtt{g}$ calls to is $[\mathtt{size}_{\mathtt{g}}\ y_1 \ldots y_m/u]$ with the premise $\bigwedge_{i=1}^{m} [y_i/v]\varphi_i$. The validation checking $\bigwedge_{i=1}^{m} [y_i/v]\varphi_i \Rightarrow \big([\mathtt{size}_{\mathtt{g}}\ y_1 \ldots y_m/u]\psi_{\mathtt{g}} \in O([\mathtt{size}\ \Gamma(\mathtt{args})/u]\psi)\big)$ in the rule states that for any instance of Γ, the size of the problem in the application term is in the big-O class $O([\mathtt{size}\ \Gamma(\mathtt{args})/u]\psi)$. Note that the membership of big-O classes can be encoded as an $\exists\forall$ query. The query is non-linear, and hence undecidable in general. However, we observed in our experiments that for many benchmarks the query stays linear. Furthermore, even when the query is non-linear, existing SMT solvers are capable of handling many such checks in practice.

3.4 Soundness

We assume that the resource-usage function ψ and the complexities T of each function are all nonnegative and monotonic integer functions—both the input and the output are integers. We show soundness of the type system with respect to the resource model. The soundness theorem states that if we derive a bound $O(\psi)$ for a function `f`, then the complexity of `f` is bounded by ψ.

Theorem 1 (Soundness of type checking). *Given a function* $\texttt{fix}\ f.\lambda x_1 \ldots \lambda x_n.t$ *and an environment* Γ*, if* $\Gamma \vdash \texttt{fix}\ f.\lambda x_1 \ldots \lambda x_n.t :: \langle \tau, O(\psi) \rangle$*, then the complexity of* f *is bounded by* ψ*.*

Our type system is incomplete with respect to resource usage. That is, there are functions in our programming language that are actually in a complexity class $O(p(x))$, but cannot be typed in our type system. The main reason why our type system is incomplete is that it ignores condition guards when building recurrence relations, and over-approximates `if-then-else` terms by choosing the largest complexity among all the paths including even unreachable ones.

4 The *SynPlexity* Synthesis Algorithm

In this section, we present the SynPlexity synthesis algorithm, which uses annotated types to guide the search of terms of given types.

4.1 Overview of the Synthesis Algorithm

The algorithm takes as input a goal type $\texttt{f} : \langle \tau, O(\psi) \rangle$, an environment Γ that includes type information of auxiliary functions, and the `size` functions for `f` and all auxiliary functions. The goal is to find a function term of type $\langle \tau, O(\psi) \rangle$.

The algorithm uses the rules of the SynPlexity type system to decompose goal types into sub-goals, and then applies itself recursively on the sub-goals to synthesize sub-terms. Concretely, given a goal γ, the algorithm tries all the typing rules, where the type in the conclusion matches γ, to construct sub-goals: for each sub-term t in the conclusion, there must be a judgment $\Gamma \vdash t :: \gamma'$ in the premise; thus, we construct the sub-goal γ'—the desired type of t. For each I-term rule, the type of each sub-term is always known, and thus a fixed set of sub-goals is generated. For each E-term rule, the algorithm enumerates E-terms up to a certain depth (the depth can be given as a parameter or it can automatically increase throughout the search). If the algorithm fails to solve some sub-goal using some E-term rule, it backtracks to an earlier choice point, and tries another rule.

Because the top-level goal is always a function type, the algorithm always starts by applying the rule T-Abs, which matches the resource bound $O(\psi)$ using Table 1 to infer a possible recurrence annotation for the type of the function body. Also T-Abs constructs a sub-goal type for the function body. In the rest of this section, we assume that goals are not function types.

Algorithm 1: GenerateE(Γ, γ, d)

Input : Context Γ, goal type $\gamma = \langle \{B \mid \varphi\}, \alpha \rangle$, depth bound d

1 **for** $t \leftarrow$ EnumerateE(Γ, d, B) **do**

2 **if** CheckE(t, Γ, γ) **then return** t

3 **return** $\bot$

Synthesizing E-Terms. The algorithm for synthesizing E-terms is shown in Algorithm 1. It enumerates each E-term t—with depth up to d—that satisfies the base type B in the goal $\gamma := \langle \{B \mid \varphi\}, ([c_1, \phi_1]..[c_n, \phi_n]; O(\psi)) \rangle$ from the context Γ. For each such E-term t, the algorithm checks whether t satisfies the goal type with a subroutine CheckE, which operates as follows.

When t is a variable term, CheckE checks the refined type of t against the goal. When t is an application term, CheckE first checks if the total number of recursive calls in the term t exceeds the bound $\sum_i c_i$, and if it does, the term t is rejected. Otherwise, CheckE checks the sizes of sub-problems of recursive calls in t. Formally, to check if a recursive application term $f(t_1, .., t_m)$ is consistent with some $[c_k, \phi_k]$, the algorithm queries the validity of the following predicate

$$(\bigwedge_{i=1}^{m} [y_i/v]\varphi_i \Rightarrow (\mathtt{size}_f(y_1\ ..\ y_m) = [\mathtt{size}_f(\Gamma(\mathtt{args}))/v]\phi_k)),$$

where the y_i's are fresh variables, and the φ_i's are the refinements of terms t_i's. If the sizes of sub-problems are not consistent with the recursive-call costs $[c_1, \phi_1]..[c_n, \phi_n]$, the term t is rejected. Note that one recursive call can possibly satisfy more than one $[c_k, \phi_k]$. The algorithm enumerates all possible matches. Finally, CheckE checks the refined type of t against the goal.

Checking the validity of auxiliary application terms is similar. CheckE needs to establish that the following predicate holds, which asserts that the resource usage of an auxiliary function does not exceed the bound $O(\psi)$.

$$\bigwedge_{i=1}^{m} [y_i/v]\varphi_i \Rightarrow \left([\mathtt{size}_{\mathtt{g}}\ y_1..y_m/v]\psi_{\mathtt{g}} \in O([\mathtt{size}\ \Gamma(\mathtt{args})/v]\psi)\right).$$

Recall that the above query is undecidable in general, and is checked with best effort by an SMT solver in SynPlexity.

Synthesizing I-Terms. Algorithm 2 shows the algorithm for synthesizing I-Terms. GenerateI first tries to synthesize an E-term for the goal γ (line (1)).

If there is no E-term that satisfies the goal, and the match bound m is greater than 0, GenerateI chooses to apply the rule T-Match lines (2)–(8). First, it enumerates candidate scrutinees s, which are E-terms of some data type. Then it generates `match` *patterns* according to the type of s (line (3)), updates the goal with a new recursive-call cost (line (4)), and generates case terms t_i for each pattern *pattern*$[i]$ (lines (5)–(7)). The subroutine UpdateCost is used to

subtract the recursive-call cost usage from the cost in γ. Finally, if all case terms are found, the algorithm constructs the corresponding `match`-term and returns it.

If there is no `match`-term satisfying the goal, GenerateI applies the rule T-If to synthesize a term of the form **if** *cond* **then** t_T **else** t_F, and performs three steps to construct sub-goals for sub-terms *cond*, t_T, and t_F: (1) it enumerates the condition guard *cond* (line (10)) of type `bool`; (2) it updates the cost in the goal γ (line (11)); and (3) it propagates sub-goals to the two branches t_T and t_F with *cond* and $\neg$*cond* as the path condition (lines (12) and (13)), respectively. Finally, if both t_T and t_F are found, the algorithm constructs the corresponding `if`-term and returns it as a solution (line (14)).

Optimization. Algorithm 2 discussed above is based on bidirectional type-guided synthesis with liquid types (Synquid [18]). Therefore, *liquid abduction* and *match abduction*, two optimizations used in Synquid, can also be used in SynPlexity. These two techniques allow one to synthesize the branches of `if`- and `match`-terms, and then use logical abduction to infer the weakest assumption under which the branch fulfills the goal type.

Algorithm 2: GenerateI(Γ, γ, d, m).

```
   Input : Context Γ, goal type γ, depth bound d, match bound m
 1 if t ← GenerateE(Γ, γ, d) then return t
 2 if m > 0 then for s ← EnumerateE(Γ, d, dataType) do
 3     patterns ← GeneratePatterns(Γ, TypeOf(s))
 4     γ' ← UpdateCost(s, γ)
 5     for i ∈ [1, Size(patterns)] do
 6         t_i ← GenerateI(UpdateContext(Γ, s == patterns[i]), γ', d, m − 1)
 7         if t_i == ⊥ then return ⊥
 8     return Match s with |_i patterns[i] → t_i
 9
10 for cond ← EnumerateE(Γ, d, Bool) do
11     γ' ← UpdateCost(s, γ)
12     t_T ← GenerateI(UpdateContext(Γ, cond), γ', d, m)
13     t_F ← GenerateI(UpdateContext(Γ, ¬cond), γ', d, m)
14     if t_T ≠ ⊥ ∧ t_F ≠ ⊥ then return If cond then t_T else t_F
15 return ⊥
```

Example 7. We illustrate in Fig. 4 how the algorithm synthesizes the $O(\log x)$ implementation of `prod` presented in Eq. (2). We omit the type contexts in the example. We will use "`??`" to denote intermediate terms being synthesized (i.e., holes in the program). At the beginning, the type of $??_1$ (i.e., the term we are synthesizing) is an arrow type with resource bound $O(\log u)$ specified by the input goal. In this example, SynPlexity applies to the arrow type the rule

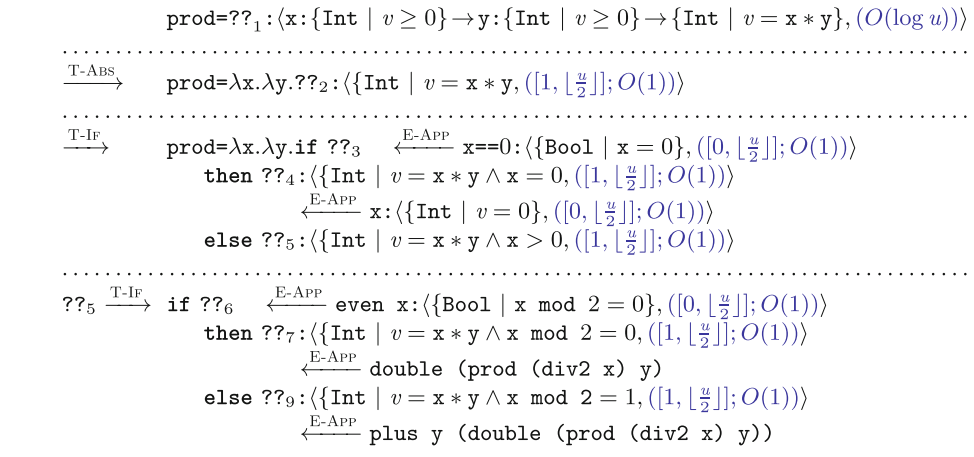

Fig. 4. Trace of the synthesis of an $O(\log x)$ implementation of `prod`.

T-Abs, parameterized according to the first rule in Table 1. This step produces the sub-problem of synthesizing the function body $??_2$, whose annotation is $([1, \lfloor \frac{u}{2} \rfloor]; O(1))$—which means that $??_2$ should contain at most one recursive call to sub-problems with size $\lfloor \frac{u}{2} \rfloor$.

Next, SynPlexity chooses to fill $??_2$ with an `if-then-else` term (by applying the T-If rules) with three sub-problems: the condition guard $??_3$, the `then` branch $??_4$ and the `else` branch $??_5$. Note that here we share the number of recursive calls $[1, \frac{u}{2}]$ as follows: 0 recursive calls in the condition guard, and 1 in the `then` branch and the `else` branch. The left arrow `E-App` shows how SynPlexity enumerates terms and checks them against the goal types of sub-problems. For example, to fill $??_4$, SynPlexity enumerates terms of type $\langle \{\texttt{Int} \mid v = \texttt{x} * \texttt{y} \wedge \texttt{x} = 0, ([1, \frac{u}{2}]; O(1))\rangle$, which are restricted to contain at most one recursive call to `prod`. In Fig. 4, SynPlexity has picked the term `x` to fill $??_4$. The refinement type of the variable term `x` is $\{\texttt{Int} \mid v = \texttt{x} \wedge \texttt{x} = 0\}$ where $\texttt{x} = 0$ is the path condition. To check that `x` also satisfies the type of $??_4$, the algorithm needs to apply rule E-SubType, and check that, for any v and `x`, $v = \texttt{x} \wedge \texttt{x} = 0$ implies $v = \texttt{x} * \texttt{y} \wedge \texttt{x} = 0$, and $[0, \lfloor \frac{u}{2} \rfloor]$ is approximated by $[1, \lfloor \frac{u}{2} \rfloor]$.

After applying another T-If rule for $??_5$, SynPlexity produces three new sub-problems $??_6$, $??_7$, and $??_8$. When enumerating terms to fill $??_7$, SynPlexity finds an application term `double (prod (div2 x) y)` that satisfies the goal $\langle \{\texttt{Int} \mid v = \texttt{x} * \texttt{y} \wedge \texttt{x mod 2} = 0, ([1, \lfloor \frac{u}{2} \rfloor]; O(1))\rangle$. To check that the size of the problem in the recursive call `prod (div2 x) y` satisfies the recursive-call cost $[1, \lfloor \frac{u}{2} \rfloor]$, the type system first checks the refinement of the callee. The refinement of the first argument `(div2 x)` is $\varphi_1 := v = \lfloor \frac{\texttt{x}}{2} \rfloor$. The refinement of the second argument `y` is $\varphi_2 := v = \texttt{y}$. Consequently, the size of the sub-problem `prod (div2 x) y` satisfies $[1, \lfloor \frac{u}{2} \rfloor]$ because $[z/v]\varphi_1 \wedge [w/v]\varphi_2 \implies \texttt{size}\ z\ w = [(\texttt{size x y})/v]\lfloor \frac{u}{2} \rfloor$, which can be simplified

to $z = \lfloor \frac{\mathtt{x}}{2} \rfloor \wedge w = \mathtt{y} \implies z = \lfloor \frac{\mathtt{x}}{2} \rfloor$. (Recall that the size function for `prod` is $\mathtt{size} := \lambda z.\lambda w.z.$)

The algorithm is sound because it only enumerates well-typed terms.

Theorem 2 (Soundness of the synthesis algorithm). *Given a goal type* $\langle \tau, O(\psi) \rangle$ *and an environment* Γ*, if a term* $\mathtt{fix}\ f.\lambda x_1..\lambda x_n.t$ *is synthesized by* SynPlexity*, then the complexity of* f *is bounded by* ψ*.*

5 Extensions to the *SynPlexity* Type System

In this section, we introduce two extensions to the SynPlexity type system.

Recurrence Relations with Correlated Sizes. The type system shown in Sect. 3 only tracks sub-problems with independent sizes. For example, consider the recurrence relation $T(u) = T(l) + T(r) + O(1)$, where the variables l and r are correlated by the constraint $l + r < u$. This relation is needed to reason about programs that manipulate binary trees or binary heaps, where l and r represent the sizes of the two children. To support such a recurrence relation, we extend SynPlexity's type system with recursive-call costs of the form $[1, l], [1, u - 1 - l]$, where l is a free variable. When correlated recurrence relations are present, the synthesis algorithm will: (1) match the first enumerated recursive-call term to $[1, l]$, and instantiate the size l with s, where s is the size of the recursive-call term (s should be smaller than the size u of the top-level function); and (2) use the size s of the recursive-call term computed in step 1 to constrain the algorithm to enumerate only recursive-call terms of sizes at most $u - 1 - s$.

Synthesis of Auxiliary Functions. Most of the existing type-directed approaches require the input to the problem to contain all needed auxiliary functions. With SynPlexity, some of the auxiliary functions needed to solve synthesis problems with resource annotations can be synthesized automatically.

For example, consider the problem `prod` described in Sect. 2. In this problem, we observe that one of the provided auxiliary functions, `div2`, strongly resembles one of the elements of the recurrence relation, $T(u) \leq T(\lfloor \frac{u}{2} \rfloor) + O(1)$, needed to synthesize a program with the desired resource usage. In particular, we know that one needs an auxiliary function that can take an input of size u and produce an output of size $\lfloor \frac{u}{2} \rfloor$. In this example, the required auxiliary function `div2` merely needs to divide the input by 2 (and round down), but in certain cases it might need a more precise refinement type than merely changing the size of the input. For example, the auxiliary function split used by merge sort needs to split the input list `xs` into two lists `v1` and `v2` that are half the length of the input *and* such that $\mathtt{elems(v1)} \uplus \mathtt{elems(v2)} = \mathtt{elems(xs)}$. However, all we know from the refinement is that the output lists must be half the length of the original list.

Although we do not know what this auxiliary function should do exactly, we can use the size constraint appearing in the recurrence relation to define part

of the refinement type we want the auxiliary function to satisfy. SynPlexity builds on this idea and incorporates an (optionally enabled) algorithm, SynAuxRef, that while trying to synthesize a solution to the top-level synthesis problem also tries in parallel to synthesize auxiliary functions that can create sub-problems with the size constraints needed in the recurrence relation. To address the problem mentioned above—i.e., that we do not know the exact refinement type the auxiliary function should satisfy—SynAuxRef enumerates *auxiliary refinements*, which are possible specifications that the auxiliary function `aux` we are trying to synthesize might satisfy.

Synthesis with Higher-Order Functions. Although SynPlexity does not support higher-order functions in general, it can solve restricted but practical problems with higher-order functions. The restriction supported introduces four assumptions on the synthesis problems. First, we assume that the resource usage of any function argument g is constant, i.e., $g : \langle \tau, O(1) \rangle$. Second, arrow-type arguments in recursive calls in the synthesized program are the same as the arrow-type arguments of the top-level function. For example, in the body of a higher-order function $\texttt{fix}\ f.\lambda g \lambda x \lambda y.t$, all recursive application terms must be of the form $f(g, \_, \_)$ where $\_$ can be any well-typed term. Third, we assume that the sizes of outputs of functions as arguments do not affect the asymptotic resource usage of the synthesized programs. Finally, arrow-type arguments cannot appear in size functions.

We extend the syntax and the type system of SynPlexity to support the restricted problems (the detail of this extension can be found in the technical report [11]). We also modify the synthesis algorithm to prune E-terms that break the second or third restriction mentioned above.

To support the second restriction (i.e., that we need to call the same function arguments in recursive calls), the synthesis algorithm first stores the function arguments of the top-level functions. Later, when a recursive call is enumerated, the synthesizer checks whether the recursive call has the same function arguments, and rejects the candidate if it does not.

To support the third restriction (i.e., that the behavior of function arguments should not affect the resource usage), the synthesis algorithm avoids enumerating nested application terms where the resource usage of the outer application depends on the value of an inner application term that calls a function argument.

6 Evaluation

In this section, we evaluate the effectiveness and performance of SynPlexity, and compare it to existing tools.[4] We implemented SynPlexity in Haskell on top of Synquid by extending its type system with recurrence annotations as presented in Sect. 3. The detailed results can be found in the technical report [11].

[4] All the experiments were performed on an Intel Core i7 4.00 GHz CPU, with 8 GB of RAM. We used version 4.8.9 of Z3. The timeout for each benchmark was 10 min.

6.1 Comparison to Prior Tools

We compared SynPlexity against two related tools: Synquid [18] and ReSyn [16], which are also based on refinement types.

Benchmarks. We considered a total of 77 synthesis problems: 56 synthesis problems from ReSyn (each benchmark specifies a concrete linear-time resource annotation), 16 synthesis problems from Synquid (which do not include resource annotations) that are not included in ReSyn, and 5 new synthesis problems involving non-linear resource annotations. In these synthesis problems, synthesis specifications and auxiliary functions are all given as refinement types. For 3 of the new benchmarks, the auxiliary function required to split the input into smaller ones is not given—i.e., the synthesizer needs to identify it automatically.

The three solvers (SynPlexity, Synquid, and ReSyn) have different features, and hence not all synthesis problems can be encoded as synthesis benchmarks for a single solver. In the rest of this section, we describe what benchmarks we considered for each tool, and how we modified the benchmarks when needed.

Synquid: Synquid does not support resource bounds, so we encoded 77 synthesis problems as Synquid benchmarks by dropping the resource annotations. Synquid returns the first program that meet the synthesis specification, and cannot provide any guarantees about the resource usage of the returned program. Synquid can solve 75 benchmarks, and takes on average 3.3s. For 10 benchmarks Synquid synthesizes a non-optimal program—i.e., there exists another program with better concrete resource usage. For example, on the ReSyn-triple-2 benchmark (where the input is a list xs), Synquid found a solution with resource usage $O(|xs|^2)$, while both SynPlexity and ReSyn can synthesize a more efficient implementation with resource usage $O(|xs|)$. The two benchmarks that Synquid failed to solve include the new benchmark SynPlexity-merge-sort'. In this benchmark, the auxiliary function required to break the input into smaller inputs is not given, without which the sizes of solutions become much larger. Therefore Synquid times out.

ReSyn: We ran ReSyn on the 56 ReSyn benchmarks with the corresponding concrete resource bounds. We could not encode 16 problems because ReSyn does not support non-linear resources bounds—e.g., the bound $\log|y|$ in the AVL-insert Synquid benchmark. ReSyn solved all 56 benchmarks with an average running time of 18.3 s.

SynPlexity: We manually added resource usages and resource bounds to existing problems to encode them for SynPlexity. For Synquid benchmarks without concrete resource bounds, we chose well-known time complexities as the bounds, e.g., we added the resource bound $O(u \log u)$ to the Sort-merge-sort problem. For the ReSyn benchmarks, we translated the concrete resource usage and resource bounds to the corresponding asymptotic ones—e.g., for the ReSyn-common' benchmark with the concrete resource bound $|ys|+|zs|$, we constructed a SynPlexity variant with the asymptotic bound $O(u)$ and a size function $\lambda ys.\lambda zs.|ys| + |zs|$. We could not encode 3 synthesis problems as SynPlexity

benchmarks: two of them involved higher-order functions that do not satisfy the assumptions introduced in Sect. 5, and the other one has an exponential resource-usage bound $O(2^u)$ (the Tree-create-balanced problem from SYNQUID).

SYNPLEXITY solved 73 benchmarks with an average running time of 8.1s. Unlike SYNQUID, SYNPLEXITY guarantees that the synthesized program satisfies the given resource bounds. After extending the implementation to support the restrictions discussed in Sect. 5, SYNPLEXITY solved 5/6 benchmarks with higher-order functions. For 10 benchmarks, SYNPLEXITY found programs that had better resource usage than those synthesized by SYNQUID. Furthermore, SYNPLEXITY can encode and solve 9 problems that RESYN could not solve because the resource bounds involve logarithms. However, SYNPLEXITY cannot encode and solve 2 benchmarks that involve higher-order functions and do not satisfy the restrictions introduced in Sect. 5. SYNPLEXITY could solve 3 problems that required synthesizing both the main function (e.g., SYNPLEXITY-merge-sort) and its auxiliary function (e.g., a function splitting a given list into two balanced partitions). No other tool could solve the SYNPLEXITY-merge-sort' benchmark.

Finding. SYNPLEXITY can express and solve 73/77 benchmarks. SYNPLEXITY has comparable performance to existing tools, and can synthesize programs with resource bounds that are not supported by prior tools.

6.2 Pruning the Search Space with Annotated Types

SYNPLEXITY uses recurrence annotations to guide the search and avoids enumerating terms that are guaranteed to not match the specified complexity. We compared the numbers of E-terms enumerated by SYNPLEXITY and SYNQUID for 56 benchmark on which both tool produced same solutions. SYNQUID always enumerated at least as many E-terms as SYNPLEXITY, and SYNPLEXITY enumerated strictly fewer E-terms for 26/56 benchmarks. For these 26 benchmarks, SYNPLEXITY can on average prune the search space by 6.2%. For example, in one case (BST-delete) SYNPLEXITY enumerated 2,059 E-terms, while SYNQUID enumerated 2,202.

Finding. On average, SYNPLEXITY reduces the size of the search space by 6.2% for approximately half of the benchmarks.

7 Related Work

Resource-Bound Analysis. Rather than determining whether a given program satisfies a specification, a synthesizer determines whether there exists a program that inhabits a given specification. The branch of verification that we draw upon for resource-based synthesis is resource-bound analysis [20].

Within the literature on automated resource-bound analysis, there are methods that extract and solve recurrence relations for imperative code [2,4,7,15].

However, these methods are unlike the type system presented in this work because they extract concrete complexity bounds as recurrence relations, and then solve the recurrences to find a concrete upper bound on resource usage. The dominant terms of the resulting concrete bounds can then be used to state a big-O complexity bound. In contrast, we want to synthesize programs with respect to a big-O complexity directly, which is more similar to the manual reasoning of [6,8]. Thus, if we were to use these techniques for our problem, the first step in our synthesis algorithm would be to pick a concrete complexity function given a big-O complexity, and then reverse the verification problem with regards to that concrete complexity. However, for any big-O complexity, there are an infinite number of functions that satisfy that complexity, which presents a significant challenge at the outset. Our design choice also has some drawbacks. As noted in [8], reasoning compositionally with big-O complexity is challenging due to the hidden quantifier structure of big-O notation. Thus, to maintain soundness our type system has to sacrifice precision and generality in some places. For example, when a function has multiple paths, our type system over-approximates by choosing the largest complexity among all the paths.

Another set of methods to generate resource bounds are type-based [9,10,14,19]. As we discussed throughout the paper, the complexities generated by these methods are concrete functions and not expressed with big-O notation, although [19] is sometimes able to pattern match a case of the Master Theorem. These type systems differ from ours in a few ways. The AARA line of research [9,10,14] is able to assign amortized complexity to programs, but is not able to generate logarithmic bounds. [19] is also able to perform amortized analysis; however, the technique is not fully automated, and instead requires the user to provide type annotations on terms, which are then checked by the type system.

Type- and Resource-Aware Synthesis. The SynPlexity implementation is built on top of Synquid [18] a type-directed synthesis tool based on refinement types and polymorphism. The work that most closely resembles ours is ReSyn [16]. As in our work, they combine the type-directed synthesizer Synquid with a type system that is able to assign complexity bounds to functional programs. The type system used in ReSyn is based on one originally used in the context of verification [10]. That work uses a sophisticated type system to assign amortized resource-usage bounds to a given program. The type system of ReSyn differs from the one presented in Sect. 3 in a few significant ways.

As highlighted earlier, ReSyn automatically infers bounds on recursive functions using amortized analysis and is restricted to linear bounds, whereas our system is able to synthesize complexities of the form $O(n^a \log^b n + c)$.

Another difference is that ReSyn synthesizes programs with a concrete complexity bound. This approach has advantages and disadvantages. For instance, it places an extra burden on the human to provide the correct bound with precise coefficient. On the other hand, the user might want an implementation that has a complexity with a small coefficient, whereas our system provides no guarantee that the complexity of an implementation will have a small coefficient in the dominant term: SynPlexity only guarantees asymptotic behavior.

ReSyn can synthesize programs with higher-order functions, which are supported only in a restricted manner by SynPlexity. To handle higher-order functions, ReSyn attaches resource units to types, which gives it *resource polymorphism*. Moreover, costs of inputs with function types can be written generally as polymorphic types (i.e., costs can be polymorphic with respect to the size of the specific input types). SynPlexity does not have *asymptotic resource polymorphism* because it cannot directly compose unknown big-O functions (i.e., the complexity of higher-order inputs). We envision that with carefully crafted restrictions on the resource annotations of higher-order functions, SynPlexity could handle synthesis problems involving such functions, e.g., assuming that the complexity of input functions is known and the refinements of input functions are precise enough. Detailed discussion about these restrictions can be found in Sect. 5 and the technical report [11]. Because big-O functions cannot be directly composed, developing a more general extension to SynPlexity that supports higher-order functions is a challenging direction for future work.

Acknowledgments. Supported, in part, by a gift from Rajiv and Ritu Batra; by multiple Facebook Research Awards; by a Microsoft Faculty Fellowship; by NSF under grants 1420866, 1763871, and 1750965; and by ONR under grants N00014-17-1-2889 and N00014-19-1-2318. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities.

References

1. Akra, M., Bazzi, L.: On the solution of linear recurrence equations. Comput. Optim. Appl. **10**, 195–210 (1998)
2. Albert, E., Arenas, P., Genaim, S., Puebla, Germán, P.: Closed-form upper bounds in static cost analysis. J. Autom. Reasoning **46**, 161–203 (2011) https://doi.org/10.1007/s10817-010-9174-1
3. Bentley, J.L., Haken, D., Saxe, J.B.: A general method for solving divide-and-conquer recurrences. ACM SIGACT News **12**(3), 36–44 (1980)
4. Breck, J., Cyphert, J., Kincaid, Z., Reps, T.: Templates and recurrences: better together. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI (2020)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3nd edn. The MIT Press, Cambridge (2009)
6. Eberl, M.: Proving divide and conquer complexities in Isabelle/HOL. J. Autom. Reasoning **58**(4), 483–508 (2016). https://doi.org/10.1007/s10817-016-9378-0
7. Flores Montoya, A.: Cost Analysis of Programs Based on the Refinement of Cost Relations. Ph.D. thesis, TU Darmstadt (2017)
8. Guéneau, A., Charguéraud, A., Pottier, F.: A fistful of dollars: formalizing asymptotic complexity claims via deductive program verification. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 533–560. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_19
9. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 357–370 (2011)

10. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource aware ml. In: The International Conference on Computer Aided Verification, CAV (2012)
11. Hu, Q., Cyphert, J., D'Antoni, L., Reps, T.: Synthesis with asymptotic resource bounds. arXiv preprint arXiv:2103.04188 (2021)
12. Hu, Q., D'Antoni, L.: Syntax-guided synthesis with quantitative syntactic objectives. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 386–403. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_21
13. Kauers, M., Paule, P.: The Concrete Tetrahedron: Symbolic Sums, Recurrence Equations, Generating Functions, Asymptotic Estimates. Springer, Cham (2011) https://doi.org/10.1007/978-3-7091-0445-3
14. Khan, D.M., Hoffmann, J.: Exponential automatic amortized resource analysis. In: International Conference on Foundations of Software Science and Computation Structures. FoSSaCs (2020)
15. Kincaid, Z., Cyphert, J., Breck, J., Reps, T.: Non-linear reasoning for invariant synthesis. In: ACM SIGPLAN Symposium on Principles of Programming Languages, POPL (2018)
16. Knoth, T., Wang, D., Polikarpova, N., Hoffmann, J.: Resource-guided program synthesis. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 253–268 (2019)
17. Pierce, B.C., Turner, D.N.: Local type inference. ACM Trans. Program. Lang. Syst. (TOPLAS) **22**(1), 1–44 (2000)
18. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. ACM SIGPLAN Notices **51**(6), 522–538 (2016)
19. Wang, P., Wang, D., Chlipala, A.: Timl: A functional language for practical complexity analysis with invariants. Proc. ACM Program. Lang. **1**(OOPSLA), 1–26 (2017)
20. Wegbreit, B.: Mechanical program analysis. In: Communications of the ACM (1975)