Type-Guided Worst-Case Input Generation

DI WANG, Carnegie Mellon University, USA
JAN HOFFMANN, Carnegie Mellon University, USA

This paper presents a novel technique for type-guided worst-case input generation for functional programs. The technique builds on automatic amortized resource analysis (AARA), a type-based technique for deriving symbolic bounds on the resource usage of functions. Worst-case input generation is performed by an algorithm that takes as input a function, its resource-annotated type derivation in AARA, and a skeleton that describes the shape and size of the input that is to be generated. If successful, the algorithm fills in integers, booleans, and data structures to produce a value of the shape given by the skeleton. The soundness theorem states that the generated value exhibits the highest cost among all arguments of the functions that have the shape of the skeleton. This cost corresponds exactly to the worst-case bound that is established by the type derivation. In this way, a successful completion of the algorithm proves that the bound is tight for inputs of the given shape. Correspondingly, a relative completeness theorem is proved to show that the algorithm succeeds if and only if the derived worst-case bound is tight. The theorem is relative because it depends on a decision procedure for constraint solving. The technical development is presented for a simple first-order language with linear resource bounds. However, the technique scales to and has been implemented for Resource Aware ML, an implementation of AARA for a fragment of OCaml with higher-order functions, user-defined data types, and types for polynomial bounds. Experiments demonstrate that the technique works effectively and can derive worst-case inputs with hundreds of integers for sorting algorithms, operations on search trees, and insertions into hash tables.

CCS Concepts: • **Theory of computation → Type theory**; **Program analysis**;

Additional Key Words and Phrases: Resource bound analysis, worst-case analysis, type systems, amortized analysis, symbolic execution

ACM Reference Format:
Di Wang and Jan Hoffmann. 2019. Type-Guided Worst-Case Input Generation. *Proc. ACM Program. Lang.* 3, POPL, Article 13 (January 2019), 30 pages. https://doi.org/10.1145/3290326

1 INTRODUCTION

An important characteristic of a computer program is its resource requirements, that is, the amount of resource such as time, memory, power, etc. that the program needs to execute. Analyzing the worst-case resource usage of a program has many applications such as finding performance bottlenecks, detecting algorithmic complexity vulnerabilities, and identifying information leaks through side channels.

Besides an analysis of the worst-case behavior, it is often desirable to obtain specific inputs such that executing the analyzed program on these inputs *exhibits* the worst-case performance. For instance, consider algorithmic complexity attacks where an adversary can construct inputs that result in unexpected space or time usage that can break or slow down critical software systems.

Authors' addresses: Di Wang, Carnegie Mellon University, USA; Jan Hoffmann, Carnegie Mellon University, USA.

2475-1421/2019/1-ART13
https://doi.org/10.1145/3290326

As emphasized in DARPA's STAC program [Website 2015], worst-case inputs are instrumental for programmers to understand what could trigger the unexpected behavior and fix the problem to improve performance. To give a concrete example, the PHP community noticed a Denial-of-Service vulnerability [Website 2011] that has been fixed [Website 2012b] after an analysis found that it was based on hash collisions [Website 2012a].

Despite of their usefulness, manual construction of worst-case inputs can be cumbersome, because (i) programs can be complex, large, and rely on unfamiliar or unavailable library code, (ii) the worst-case inputs do *not* seem to follow any universal pattern, e.g., a worst-case quicksort requires specific ordering [McIlroy 1999] while a worst-case hash table requires maximal number of collisions [Crosby and Wallach 2003], and (iii) even if a candidate input is present, it can be still difficult to *prove* the input *does* exhibit the worst-case resource usage.

As a result, *automatic methods for worst-case input generation* are highly desirable and have received a lot of attention. On the one hand, there is a large field of fuzz testing [Forrester and Miller 2000; Godefroid et al. 2008] and symbolic execution [Godefroid et al. 2005; Sen et al. 2005]. Combinations of these methods have been recently studied for *dynamic* worst-case analysis [Burnim et al. 2009; Noller et al. 2018; Petsios et al. 2017]. These dynamic approaches are quite universal in the sense that they can be applied to arbitrary programs implemented in a widely used programming language such as Java, but they usually do *not* formally guarantee that the resulting input exposes the worst resource usage. On the other hand, there is an active community that employs *static* methods such as type systems [Hoffmann et al. 2017; Jost et al. 2010] and abstract interpretation [Albert et al. 2011; Gulwani 2009] to compute upper bounds on the worst-case resource usage. These static analyses provide sound resource bounds, but they do *not* generate a concrete witness to show the derived resource bound is tight.

In this paper, we develop a novel *type-guided worst-case input generation* algorithm for a purely functional fragment of *Resource Aware ML* (RaML) [Hoffmann et al. 2017], a resource-aware version of a subset of the functional programming language OCaml that features higher-order functions and user-defined data structures. Based on *automatic amortized resource analysis* (AARA) [Hofmann and Jost 2003], RaML infers concrete multivariate-polynomial upper bounds, parametrized with a resource metric, as functions of sizes of the inputs. Our algorithm takes in a RaML function f of type $A \to B$ along with its resource-annotated typing derivation and a first-order input *skeleton* of type A, which specifies the shape of the input (e.g., the length of a list),[1] and then either produces a concretization of the skeleton (e.g., a concrete list with the specified length), which is *guaranteed* to expose the worst-case resource usage of the function f, or reports a generation failure. Our algorithm also enjoys *relative completeness*, in the sense that if the inferred bound in RaML is tight for an input skeleton (i.e., there does exist a concretization of the skeleton that exhibits the resource usage *exactly* as the inferred bound), our generation algorithm always succeeds.[2]

From the perspective of automatic resource analysis, our work also mitigates a longstanding issue with current techniques for worst-case resource bound analysis. Existing analysis techniques [Albert et al. 2015; Brockschmidt et al. 2014; Carbonneaux et al. 2017; Gulwani et al. 2009; Hofmann and Jost 2003; Kincaid et al. 2017; Sinn et al. 2014] are sound and the derived bounds are thus always upper bounds on the worst-case behavior. However, there does not exist any guarantee on the tightness of the result. That includes the constant factors in the bounds as well as the asymptotic behavior. As a result, users often find it difficult to interpret the result of the analysis. With this view, our result can be seen as a way of automatically proving that a bound derived by RaML is tight for inputs of a given shape or size. From the relative completeness result follows also the

[1] We focus on *first-order* inputs in the sense that we do not consider the generation of an unknown function in this paper.
[2] In fact, our generation algorithm is complete *modulo constraint solving*. See §5 for details.

other direction: If we use an oracle for satisfiability and are not able to generate a worst-case input then the derived bound is not tight for the inputs described by the given skeleton.

A key challenge in the development of the worst-case input generation is to ensure *soundness*—for a given input skeleton, the generation result must expose the worst resource usage among all possible concretizations of the skeleton. It is intractable to compare the generation result with all other concretizations, because it usually requires exploration of the space of all concretizations, the number of which could be infinite, or enumeration of all the execution paths in the program, the number of which could be exponential in the size of the input. To address this challenge, we need to develop a mechanism to generate a worst-case input *without* exploring the complete space of candidate concretizations.

The other challenge is to exploit *compositionality* during the input generation—in order to scale the worst-case input generation to large input skeletons, it is usually more efficient to generate a worst-case input by *composing* its generated subparts. For example, to generate a worst-case input for a recursive function, it seems natural to generate a worst-case input for each recursive call, and then combine them to generate a worst-case input for the function body. However, combining the results from the recursive calls can be nontrivial: different calls can involve the same fragment of the input and the recursively generated results might not be compatible.

To address the first challenge, we define *symbolic input skeletons* and develop a generation algorithm based on *symbolic execution*, which searches the space of all execution paths of a program and collects *path constraints* that *suffices* for a concretization of the input skeleton to trigger the worst-case resource usage. The major novelty of our generation algorithm is that it is *type-guided*—it makes use of the typing derivation derived by RaML to guide the search as well as prune the search space. RaML's type system is based on *amortized analysis*, in the sense that it specifies the potential functions before and after the evaluation of a subexpression to account for resource usage. Because RaML derives upper bounds on resource usage, these potentials are conservative and allow for *potential waste*. If such waste occurs then the corresponding path can*not* coincide with the derived upper bound. Our type-guided generation algorithm utilizes the resource-annotated typing derivation to detect potential waste as early as possible to prune partial executions that cannot be extended to expose the resource usage indicated by the derived worst-case bound.

To address the second challenge, we propose the novel concept of *compositional input generation* and devise two search heuristics based on the concept. First, we describe *uniform execution*, which corresponds to programs that have worst-case inputs that always execute the same branch of each conditional expression. Second, we introduce *skeleton similarity*, which corresponds to recursive functions that have worst-case inputs that execute the same path in the function body for all calls to itself with inputs of the same shape. Note that skeleton similarity is more general than uniform execution and includes for instance alternating shapes in recursive calls.

We evaluate our type-guided worst-case input generation algorithm on more than 20 case studies, including time usage for sorting algorithms, operations in search trees, etc., memory usage for list operations, and customized resource metrics such as the number of collisions for hash tables. The experiments show that our algorithm is able to derive nontrivial worst-case inputs, as well as scale to large input skeletons in some of the case studies, e.g., sorting algorithms with hundreds of integers.

Contributions. Our work makes four main contributions.

- We develop a novel resource-parametric type-guided worst-case input generation algorithm for a considerable fragment of purely functional RaML.
- We prove the nontrivial soundness and relative completeness of our generation algorithm.

```
let rec lpairs l = match l with
  | [] → []
  | x1 :: xs → match xs with
    | [] → []
    | x2 :: xs' → if (x1:int) < (x2:int) then (x1, x2) :: lpairs xs' else lpairs xs'
```

Fig. 1. The function lpairs will serve as a running example in this paper.

- We propose novel concepts about compositional worst-case input generation, as well as devise and prove the correctness of two search heuristics to improve scalability.
- We implement our generation algorithm in the existing RaML system that features higher-order functions, user-defined data types, and polynomial resource bounds, and evaluate its effectiveness and efficiency on a broad suite of case studies.

2 OVERVIEW

In this section, we illustrate our type-guided worst-case input generation algorithm using a simple example. The function lpairs in Fig. 1 collects adjacent ordered pairs of integers. For example, the expression lpairs([1, 2, 3, 4]) evaluates to $[\langle 1, 2\rangle, \langle 3, 4\rangle]$ and lpairs([2, 1, 3, 4]) evaluates to $[\langle 3, 4\rangle]$. We write the type of the function as $L(\mathsf{int}) \to L(\mathsf{int} \times \mathsf{int})$, where $\to, \times$ are the standard function and product types, respectively, and $L(T)$ is the type of lists with elements of type T. We want to generate inputs for the function such that it exposes the worst *heap-space* usage. In this example, we use a slightly different memory model from OCaml's and assume each datatype constructor creates a boxed value with a header of length 2, as well as a tuple only consumes the same amount of resources as its length. Specifically, we assume a nil-node (i.e., an empty list) consumes 2 units of resource, a cons-node (i.e., a list constructed by a head element and a tail list) consumes 4 units, a pair constructor consumes 2 units. We do not consider garbage collection.

Resource Bound Analysis. First of all, we use RaML to compute an upper bound on the worst-case heap space usage, as well as the corresponding typing derivation that our input generation algorithm demands. RaML derives a linear bound $(2 + 3M)$ for the function lpairs, where M is the number of cons-nodes of the argument, i.e., the *length* of the input list.

The resource analysis in RaML is based on the potential method of amortized analysis [Tarjan 1985]. The intuition is to introduce potential functions that depend on data structures, and the potential at a program point should be sufficient to pay for the cost of the next evaluation step as well as the potential at the next program point. In RaML, a set of fixed potential functions is fixed for every data type [Hoffmann et al. 2011, 2017; Hoffmann and Hofmann 2010; Hofmann and Jost 2003]. Types of inductive data structures are annotated with nonnegative rational numbers $p \in \mathbb{Q}_0^+$. For example, $L^p(A)$ is an annotated list type where A is another annotated type. The potential of a value a is then defined with respect to its annotated type. If $a = [a_1, \cdots, a_n]$ is a list of values of type A, its potential $\Phi(a : L^p(A))$ is defined as $\sum_{i=1}^{n}(p + \Phi(a_i : A))$, or equivalently, $n \cdot p + \sum_{i=1}^{n} \Phi(a_i : A)$. The function types are also annotated and have the form $A_1 \xrightarrow{q/q'} A_2$ where A_1 and A_2 are annotated argument and result types, and $q, q' \in \mathbb{Q}_0^+$ stand for the constant potential before a call to the function and after the call, respectively. For the function lpairs in Fig. 1, RaML derives a resource-annotated type $L^3(\mathsf{int}) \xrightarrow{2/0} L^0(\mathsf{int} \times \mathsf{int})$.

A type with positive potential on the result type like in the type $L^5(\mathsf{int}) \xrightarrow{3/1} L^2(\mathsf{int} \times \mathsf{int})$ is needed to type an application of lpairs in a composed function like $f(\mathsf{lpairs}(l))$ if f has type $L^2(\mathsf{int} \times \mathsf{int}) \xrightarrow{1/0} A$ for some type A. In general, the type of a function can be described with variables for the potential annotations and linear constraints that describe their relations.

The typing rules of RaML's type system manipulates the coefficients q associated with data types to ensure that the correct potential is assigned to new data structures or used to pay for resource usage. RaML's resource-annotated typing judgment has the form $\Gamma \left|\frac{q}{q'}\right. e : A$ where e is an expression, $q, q' \in \mathbb{Q}_0^+$ stand for constant potential before and after the evaluation of the expression, respectively, Γ is a resource-annotated typing context that maps program variables to annotated types, and A is a resource-annotated result type. Intuitively, if the initial potential is *at least* the amount specified by Γ, then it is sufficient to evaluate e to a value and the leftover potential after the evaluation is *at least* the amount specified by A. For the program in Fig. 1, two examples of typing judgements are

$$x_1 : \mathsf{int}, x_2 : \mathsf{int} \left|\frac{2}{0}\right. \langle x_1, x_2 \rangle : \mathsf{int} \times \mathsf{int} \qquad \text{and} \qquad xs' : L^3(\mathsf{int}) \left|\frac{2}{0}\right. \mathsf{lpairs}\ xs' : L^0(\mathsf{int} \times \mathsf{int}).$$

The first typing judgment indicates the evaluation of the pair construction needs 2 units of potential because the resource metric specifies the pair construction consumes 2 units of heap space. The second typing judgment indicates that if xs' is a list of length N, then the potential $(2 + 3N)$ suffices for the evaluation of the expression $\mathsf{lpairs}\ xs'$.

Worst-Case Input Generation. Before describing the input generation algorithm, we informally analyze the worst-case heap-space usage of the program in Fig. 1. Because all memory operations are constructions of the result list of pairs, and the total number of adjacent pairs that can be constructed is $\lfloor \frac{M}{2} \rfloor$ where M is the length of the input list, we deduce that the heap space usage is at most $2 + (2 + 4) \cdot \lfloor \frac{M}{2} \rfloor$: the first 2 pays for the nil-node, the second 2 pays for the pair, and the 4 is used to pay for a cons-node. It is the exact usage when all available pairs are ordered—hence the resource bound derived by RaML $(2 + 3M)$ is tight if M is even.

To generate a worst-case input for a program, the user needs to specify an input *skeleton*. For the function lpairs, a skeleton can be represented as a list of *indeterminate* integers. For example, $[\mathsf{int}^1, \mathsf{int}^2, \mathsf{int}^3, \mathsf{int}^4]$ is a skeleton of an integer list of length four. A basic approach for worst-case input generation is to evaluate the program on the input skeleton *symbolically*: search all possible execution paths and record path constraints.

We write symbolic executions of an expression e under a *skeleton environment* γ that maps program variables to skeletons as judgments of the form $\gamma \vdash e \Rightarrow \langle \phi, S \rangle$, where ϕ is the path constraint of this execution, and S is a value that might contain indeterminates, representing the evaluation result of e. For example, the symbolic execution of the conditional expression can be formalized as two rules:

SE-Cond-True

$$\frac{\gamma \vdash e_1 \Rightarrow \langle \phi, S \rangle}{\gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Rightarrow \langle ([\gamma]e) \wedge \phi, S \rangle}$$

SE-Cond-False

$$\frac{\gamma \vdash e_2 \Rightarrow \langle \phi, S \rangle}{\gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Rightarrow \langle \neg([\gamma]e) \wedge \phi, S \rangle}$$

where $[\gamma]e$ transforms e to a symbolic constraint under the environment γ, e.g., if $e = (x_1 < x_2)$ and $\gamma(x_1) = \mathsf{int}^1, \gamma(x_2) = \mathsf{int}^2$, then $[\gamma]e = (\mathsf{int}^1 < \mathsf{int}^2)$. After collecting all possible execution paths from a symbolic execution of the program, the basic input generation algorithm picks a worst-case execution path with the largest resource usage with respect to the resource metric, as well as a satisfiable path constraint. For the function lpairs, an example of worst-case execution paths is

$$l \mapsto [\mathsf{int}^1, \mathsf{int}^2, \mathsf{int}^3, \mathsf{int}^4] \vdash \mathsf{lpairs}\ l \Rightarrow \langle (\mathsf{int}^1 < \mathsf{int}^2) \wedge (\mathsf{int}^3 < \mathsf{int}^4), [\langle \mathsf{int}^1, \mathsf{int}^2 \rangle, \langle \mathsf{int}^3, \mathsf{int}^4 \rangle] \rangle \quad (1)$$

Finally, an SMT solver can be invoked to find a model for the path constraint. For the execution path (1), one model is $\{\mathsf{int}^1 \mapsto 0, \mathsf{int}^2 \mapsto 1, \mathsf{int}^3 \mapsto 0, \mathsf{int}^4 \mapsto 1\}$, which corresponds to a concrete input list $[0, 1, 0, 1]$ that indeed triggers the worst heap space usage.

The major novelty of our worst-case input generation algorithm is to make use of the resource-annotated typing derivation during the symbolic execution. RaML's type system is an *affine* type system, which means that each resource in the typing context can be used *at most once*. Potential

waste happens when some resources in the context are never used but carry positive potential. Our input generation algorithm is designed to find an execution path with imposed linearity, i.e., without potential waste. During the symbolic execution, the algorithm relies on the typing derivation to check if there is any potential waste. If such waste is detected then partial executions that involve the respective path can be pruned from the search. For example, the typing judgment of the conditional expression in the function lpairs is

$$x_1 : \mathsf{int}, x_2 : \mathsf{int}, xs' : L^3(\mathsf{int}) \,\big|^{8}_{0}\, \text{if } (x_1 < x_2) \text{ then } (\langle x_1, x_2\rangle :: \mathsf{lpairs}\ xs') \text{ else } (\mathsf{lpairs}\ xs') : L^0(\mathsf{int}\times\mathsf{int}) \tag{2}$$

and the typing judgments of two branches of the conditional expression are

$$x_1 : \mathsf{int}, x_2 : \mathsf{int}, xs' : L^3(\mathsf{int}) \quad \big|^{8}_{0} \quad \langle x_1, x_2\rangle :: \mathsf{lpairs}\ xs' : L^0(\mathsf{int}\times\mathsf{int}) \tag{3}$$

$$xs' : L^3(\mathsf{int}) \quad \big|^{2}_{0} \quad \mathsf{lpairs}\ xs' : L^0(\mathsf{int}\times\mathsf{int}) \tag{4}$$

Suppose the input skeleton is $[\mathsf{int}^1, \mathsf{int}^2, \mathsf{int}^3, \mathsf{int}^4]$. When our algorithm evaluates the conditional expression for the first time, the symbolic environment γ is

$$x_1 \mapsto \mathsf{int}^1, x_2 \mapsto \mathsf{int}^2, xs' \mapsto [\mathsf{int}^3, \mathsf{int}^4]$$

and the potential at the program point with respect to the typing judgment (2) is $8+0+0+3\cdot 2 = 14$. The then-branch needs $8+0+0+3\cdot 2 = 14$ units of potential to proceed with respect to (3), and the else-branch needs only $2+3\cdot 2 = 8$ units with respect to (4). Hence our algorithm detects potential waste in the else-branch and decides to only explore the then-branch. By this means our algorithm is able to prune the search space to contain only one execution path as (1), and know that this path is the only one that can expose the worst-case resource as given by the initial potential. More generally, every time our algorithm finds an execution path without potential loss, the associated path constraint *suffices* for the input skeleton to trigger the worst-case resource usage.

Let us try another input skeleton for the function lpairs: a singleton list $[\mathsf{int}^1]$. Note that because the length of the list is odd, the resource bound derived by RaML is not tight. The typing judgment of the inner match expression is

$$x_1 : \mathsf{int}, xs : L^3(\mathsf{int}) \,\big|^{5}_{0}\, \mathsf{match}\ xs\ \mathsf{with}\ [] \to [] \mid \cdots : L^0(\mathsf{int}\times\mathsf{int}) \tag{5}$$

and the typing judgment of the nil-case of this match expression is

$$\cdot \,\big|^{2}_{0}\, [] : L^0(\mathsf{int}\times\mathsf{int}) \tag{6}$$

When our input generation algorithm evaluates the inner match expression, the symbolic environment γ is

$$x_1 \mapsto \mathsf{int}^1, xs \mapsto []$$

and the potential at the program point is $5 + 3\cdot 0 = 5$, with respect to the typing judgment (5). Because xs is mapped to [], the nil-case of the match expression is evaluated in the next step. However, the nil-case needs only 2 units of potential to proceed with respect to (6), hence this execution path contains potential waste. For this input skeleton, our algorithm reports a generation failure, which suggests the resource bound is not tight when the input is a singleton list.

Compositional Input Generation. Our type-guided worst-case input generation algorithm provides new opportunities to develop search heuristics. In this paper, we focus on heuristics that exploit *compositionality*. Intuitively, compositional generation produces a worst-case input for a function by first generating subparts of the input that are used in function calls and then combining them. Because in the function body, different function calls can involve the same fragment of the input, it is more reasonable to generate *path constraints* that suffice for an input skeleton to trigger the worst-case resource usage, by combining *path constraints* on subparts of the input generated from the function calls. Then the major obstacle to compositionality is the exponential number of

```
let rec wc_lpairs l = match l with
  | [] → (⊤, [])
  | x1 :: xs → match xs with
    | [] → (⊥, [])
    | x2 :: xs' → let (ϕ, ret) = wc_lpairs xs' in ((x1 < x2) ∧ ϕ, (x1, x2) :: ret)
```

Fig. 2. Pseudo-code of a compositional input generation procedure for the function lpairs in Fig. 1

combinations of branch choices of conditional expressions. To reduce the number of combinations that the algorithm needs to investigate, we propose two different heuristics.

The first heuristic, named *uniform execution*, is based on the observation that many programs have worst-case inputs that trigger the evaluation of the same branch of each conditional expression. For example, the function lpairs in Fig. 1 always evaluates the then-branch of the conditional expression to expose its worst-case heap space usage. Therefore, this heuristic enumerates the combinations of branch choices of conditional expressions in the code and then runs the type-guided symbolic execution to check whether it has potential waste. Because the number of conditional expressions in the code is independent of the size of the input, the heuristic can scale to large inputs. We can use the heuristic for the function lpairs, to derive an input generation procedure for the function that computes a sufficient constraint for worst-case inputs from an input skeleton. Fig. 2 presents the pseudo-code of this procedure, takes in a symbolic input and returns a path constraint as well as a symbolic result. The symbols $\top$ and $\bot$ stands for true and false, respectively.

The second heuristic, named *skeleton similarity*, is based on the observation that a recursive function usually has worst-case inputs such that for all the calls to this function with the same *shape* of inputs, it executes the same path in the function body. For example, Fig. 3 shows a modified version of the function lpairs in Fig. 1. The function lpairs_alt takes an extra boolean argument d to pick either an ordered pair or a reversely ordered pair. Then this function collects adjacent pairs of integers, and these pairs should be ordered and reversely ordered alternatively. The uniform-execution heuristic does not work here—although the first two branches of the conditional expression do not waste potential, both of them should be executed on a worst-case input because inside these branches the boolean argument d is inverted. Instead, the function lpairs_alt has worst-case inputs for skeletons of even lengths, such that if the length of the argument list is a multiple of four, the function evaluates the second branch, and otherwise, it evaluates the first branch. For example, if the argument list has four elements, a worst-case input is $\langle \text{false}, [1, 0, 0, 1] \rangle$, and if the argument list has two elements, a worst-case input is $\langle \text{true}, [0, 1] \rangle$. Operationally, this heuristic records satisfiable execution paths for different shapes of the inputs of the recursive function. If it encounters a call to the function with an input skeleton of the shape it has already explored then it tries the recorded execution path first.

```
let rec lpairs_alt d l = match l with
  | [] → []
  | x1 :: xs → match xs with
    | [] → []
    | x2 :: xs' →
      if d && (x1:int) < (x2:int) then (x1, x2) :: lpairs_alt (not d) xs'
      else if (not d) && (x1:int) > (x2:int) then (x1, x2) :: lpairs_alt (not d) xs'
      else lpairs_alt d xs'
```

Fig. 3. A modified version of the function lpairs in Fig. 1

3 SETTING THE STAGE: RESOURCE AWARE ML

In this section, we introduce a purely functional first-order fragment of RaML that includes booleans, integers, pairs, lists, binary trees, recursion, and pattern match. We then present a resource-aware type system with linear potential for upper bounds. We will use this language to define and formalize our type-guided worst-case input generation algorithm in §5. The restriction to this fragment in the technical development is only for brevity. Our results carry over to the full purely functional fragment of RaML, which includes multivariate polynomial potential functions, user-defined types, and higher-order functions [Hoffmann et al. 2017]. The reason is that the technical development is, in principle, independent of the shape of potential functions. Our worst-case input generation tool has also been implemented for this larger fragment (see §7.1).

Syntax. The expressions are in *share-let-normal-form* [Hoffmann et al. 2011], which means that syntactic forms allow only variables rather than arbitrary terms whenever possible, without loss of expressivity. Fig. 4 presents the grammar of expressions via abstract binding trees [Harper 2016]. The syntactic form $\mathsf{op}_\diamond(x_1, x_2)$ represents expressions that perform primitive binary operations $\diamond$ on booleans and integers. The syntactic form $\mathsf{share}(x, x_1.x_2.e)$ has to be used to introduce multiple occurrences of a variable x in an expression. We skip the standard notions of integer constants $n \in \mathbb{Z}$, variable identifiers $x \in \mathsf{VID}$, and function identifiers $f \in \mathsf{FID}$.

Simple Types. The language has a usual ML-like type system, where well-typed expressions are assigned with a *simple type* without resource annotations. As defined in Fig. 4, simple types are data types A and first-order types F. A set of semantic values is assigned to each data type A in an obvious way, written $[\![A]\!]$. For example, $[\![T(\mathsf{int} \times \mathsf{int})]\!]$ is the set of finite binary trees, each node of which contains a pair of integers. First-order types F are types of functions. For example, the type of the function lpairs in Fig. 1 is $L(\mathsf{int}) \to L(\mathsf{int} \times \mathsf{int})$.

A *typing context* Γ is a finite partial mapping from variable identifiers to data types. A *signature* Σ is a finite partial mapping from function identifiers to first-order types. The typing judgment $\Sigma; \Gamma \vdash e : A$ states that the expression e has type A under the signature Σ and context Γ. The typing rules are standard and in fact, a subset of the resource-aware typing rules in Fig. 6 by omitting the resource annotations. Then a *program* consists of a signature Σ and a family $\{\lambda x^f.e^f\}_{f \in \mathsf{dom}(\Sigma)}$ of top-level function definitions with a distinguished variable identifier as the formal parameter, such that $\Sigma; x^f : A \vdash e^f : B$ if $\Sigma(f) = A \to B$.

Big-Step Operational Cost Semantics. The resource usage of a program is determined by a big-step operational cost semantics. The cost is parametric in the resource metric and can measure every quantity whose usage in a single evaluation step can be bounded by a constant. The semantics is formulated with respect to an environment as usual. A *value* $v \in \mathsf{Val}$ is either a null value null, a boolean constant $b \in \{\mathsf{true}, \mathsf{false}\}$, an integer constant $n \in \mathbb{Z}$, or a pair of values $\langle v_1, v_2 \rangle$. It is convenient to identify tuples like $\langle v_1, v_2, v_3 \rangle$ with the pair $\langle v_1, \langle v_2, v_3 \rangle \rangle$. An *environment*

$$
\begin{array}{rcl}
e & ::= & \langle\rangle \mid \mathsf{true} \mid \mathsf{false} \mid n \mid x \mid \mathsf{op}_\diamond(x_1, x_2) \mid \mathsf{app}(f, x) \mid \mathsf{let}(e_1, x.e_2) \mid \mathsf{pair}(x_1, x_2) \\
& \mid & \mathsf{matp}(x, x_1.x_2.e) \mid \mathsf{nil} \mid \mathsf{cons}(x_h, x_t) \mid \mathsf{matl}(x, e_1, x_h.x_t.e_2) \mid \mathsf{leaf} \mid \mathsf{node}(x_0, x_1, x_2) \\
& \mid & \mathsf{matt}(x, e_1, x_0.x_1.x_2.e_2) \mid \mathsf{if}(x, e_1, e_2) \mid \mathsf{share}(x, x_1.x_2.e) \\
\diamond & \in & \{+, -, \times, \mathsf{div}, \mathsf{mod}, =, \neq, <, >, \wedge, \vee\} \\
A & ::= & \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid A_1 \times A_2 \mid L(A) \mid T(A) \\
F & ::= & A_1 \to A_2
\end{array}
$$

Fig. 4. Syntax of the language

$V : \mathsf{VID} \rightharpoonup \mathsf{Val}$ is a finite partial mapping from variables to values. The operational evaluation judgment has the form $V \vdash \frac{q}{q'}\, e \Downarrow v$ where $q, q' \in \mathbb{Q}_0^+$ are nonnegative rational numbers. The intuitive meaning is that under the environment V and q units of available resource, e evaluates to the value v without running out of resource and q' units of resource are available after the evaluation. Then the evaluation consumes $\delta = q - q'$ units of resource. Fig. 5 show the evaluation rules of the big-step semantics where K is a resource metric that maps syntactic forms to nonnegative rational numbers.[3] For example, to compute heap space usage, we specify $K^{\mathsf{nil}} = 2$, $K^{\mathsf{cons}} = 4$, $K^{\mathsf{pair}} = 2$, and other syntactic forms are assigned with a zero cost. The evaluation is deterministic in the sense that there is at most one combination of q', v such that $V \vdash \frac{q}{q'}\, e \Downarrow v$ for a given expression e, an environment V, and q units of initial resource. If v is a value, A is a type, and $a \in [\![A]\!]$ is a semantic value of type A, we write $\models v \mapsto a : A$ to mean that v defines a. We also write $\models v : A$ to indicate that there exists a semantic value $a \in [\![A]\!]$ satisfying $\models v \mapsto a : A$. We write $\models V : \Gamma$, if $\models V(x) : \Gamma(x)$ for every $x \in \mathsf{dom}(\Gamma)$.

Resource-Aware Type System. To apply the potential method of amortized analysis [Tarjan 1985], one has to establish a mapping from program points to potentials. The potential at a program point should suffice for the cost of any possible evaluation step as well as the potential at the next program point. Potential functions are usually defined with respect to data structures used in the program. To assign *linear* potentials to data structures, inductive data types (i.e., lists and binary trees) are annotated with a nonnegative rational number $p \in \mathbb{Q}_0^+$ [Hofmann and Jost 2003]. The intuitive meaning is that every internal constructor in the inductive data structure is assigned with p units of potential. The following grammar defines the *resource-annotated* data types A.

$$A ::= \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid A_1 \times A_2 \mid L^p(A) \mid T^p(A) \text{ where } p \in \mathbb{Q}_0^+$$

Formally, the *potential* $\Phi(a : A)$ of a semantic value $a \in [\![A]\!]$, where A is a resource-annotated data type, is defined as follows.[4] For a binary tree $t \in [\![T(A)]\!]$, we write $\mathsf{elems}(t)$ for its elements in pre-order.

$$\begin{aligned}
\Phi(a : A) &= 0 && \text{if } A \in \{\mathsf{unit}, \mathsf{bool}, \mathsf{int}\}\\
\Phi(a : A_1 \times A_2) &= \Phi(a_1 : A_1) + \Phi(a_2 : A_2) && \text{if } a = \langle a_1, a_2\rangle\\
\Phi(l : L^p(B)) &= n \cdot p + \textstyle\sum_{i=1}^{n} \Phi(a_i : B) && \text{if } l = [a_1, \cdots, a_n]\\
\Phi(t : T^p(B)) &= n \cdot p + \textstyle\sum_{i=1}^{n} \Phi(a_i : B) && \text{if } \mathsf{elems}(t) = [a_1, \cdots, a_n]
\end{aligned}$$

Let $v \in \mathsf{Val}$ be a value such that $\models v \mapsto a : A$, then the *potential* $\Phi(v : A)$ of v is defined as $\Phi(v : A) \stackrel{\mathsf{def}}{=} \Phi(a : A)$. Further, let V be an environment and Γ be a resource-annotated typing context that maps variables to resource-annotated data types such that $\models V : \Gamma$, then the *potential* of Γ under V is defined as $\Phi_V(\Gamma) \stackrel{\mathsf{def}}{=} \sum_{x \in \mathsf{dom}(\Gamma)} \Phi(V(x) : \Gamma(x))$.

Example 3.1. *Let an environment be* $V = \{l \mapsto \langle 0, \langle 1, \langle 0, \langle 1, \mathsf{null}\rangle\rangle\rangle\rangle\}$ *and a resource-annotated typing context be* $\Gamma = \{l : L^3(\mathsf{int})\}$. *Then* $\models V(l) \mapsto [0, 1, 0, 1] : L(\mathsf{int})$. *The potential of the typing context* Γ *under* V *is computed as* $\Phi_V(\Gamma) = \Phi(V(l) : L^3(\mathsf{int})) = \Phi([0, 1, 0, 1] : L^3(\mathsf{int})) = 4 \times 3 = 12$.

The *resource-annotated* first-order types are then defined with respect to the following grammar. The intuitive meaning is that q and q' are constant potentials before a call to the function and after

[3] The resource usage can also be negative, which means the evaluation releases some resources, e.g., memory could become available during evaluation [Hoffmann et al. 2011, 2017].

[4] The potential of trees depends on the elements but *not* on the structure of the tree. We inherit this design choice from RaML. It keeps the type rules simple and ensures compositionality because the potential is invariant under tree transformations.

$\boxed{V \vert\!\frac{q}{q'}\, e \Downarrow v}$ e evaluates to v with q' units of resource left over under V and q units of resource

(E-Triv)
$$\frac{}{V \vert\!\frac{q+K^{\text{unit}}}{q}\, \langle\rangle \Downarrow \text{null}}$$

(E-Bool)
$$\frac{b \in \{\text{true}, \text{false}\}}{V \vert\!\frac{q+K^{\text{bool}}}{q}\, b \Downarrow b}$$

(E-Int)
$$\frac{n \in \mathbb{Z}}{V \vert\!\frac{q+K^{\text{int}}}{q}\, n \Downarrow n}$$

(E-Var)
$$\frac{x \in \text{dom}(V)}{V \vert\!\frac{q+K^{\text{var}}}{q}\, x \Downarrow V(x)}$$

(E-Op)
$$\frac{x_1, x_2 \in \text{dom}(V) \quad v = V(x_1) \diamond V(x_2)}{V \vert\!\frac{q+K^{\text{op}}}{q}\, \text{op}_\diamond(x_1, x_2) \Downarrow v}$$

(E-App)
$$\frac{V(x) = v \quad V[x^f \mapsto v] \vert\!\frac{q}{q'}\, e^f \Downarrow v'}{V \vert\!\frac{q+K^{\text{app}}}{q'}\, \text{app}(f, x) \Downarrow v'}$$

(E-Let)
$$\frac{V \vert\!\frac{q}{q_1}\, e_1 \Downarrow v_1 \quad V[x \mapsto v_1] \vert\!\frac{q_1}{q'}\, e_2 \Downarrow v_2}{V \vert\!\frac{q+K^{\text{let}}}{q'}\, \text{let}(e_1, x.e_2) \Downarrow v_2}$$

(E-Pair)
$$\frac{x_1, x_2 \in \text{dom}(V) \quad v = \langle V(x_1), V(x_2)\rangle}{V \vert\!\frac{q+K^{\text{pair}}}{q}\, \text{pair}(x_1, x_2) \Downarrow v}$$

(E-MatP)
$$\frac{V(x) = \langle v_1, v_2\rangle \quad V[x_1 \mapsto v_1, x_2 \mapsto v_2] \vert\!\frac{q}{q'}\, e \Downarrow v}{V \vert\!\frac{q+K^{\text{matP}}}{q'}\, \text{matp}(x, x_1.x_2.e) \Downarrow v}$$

(E-Cons)
$$\frac{x_h, x_t \in \text{dom}(V) \quad v = \langle V(x_h), V(x_t)\rangle}{V \vert\!\frac{q+K^{\text{cons}}}{q}\, \text{cons}(x_h, x_t) \Downarrow v}$$

(E-Nil)
$$\frac{}{V \vert\!\frac{q+K^{\text{nil}}}{q}\, \text{nil} \Downarrow \text{null}}$$

(E-MatL-Nil)
$$\frac{V(x) = \text{null} \quad V \vert\!\frac{q}{q'}\, e_1 \Downarrow v}{V \vert\!\frac{q+K^{\text{matLN}}}{q'}\, \text{matl}(x, e_1, x_h.x_t.e_2) \Downarrow v}$$

(E-MatL-Cons)
$$\frac{V(x) = \langle v_h, v_t\rangle \quad V[x_h \mapsto v_h, x_t \mapsto v_t] \vert\!\frac{q}{q'}\, e_2 \Downarrow v}{V \vert\!\frac{q+K^{\text{matLC}}}{q'}\, \text{matl}(x, e_1, x_h.x_t.e_2) \Downarrow v}$$

(E-Node)
$$\frac{x_0, x_1, x_2 \in \text{dom}(V) \quad v = \langle V(x_0), V(x_1), V(x_2)\rangle}{V \vert\!\frac{q+K^{\text{node}}}{q}\, \text{node}(x_0, x_1, x_2) \Downarrow v}$$

(E-MatT-Leaf)
$$\frac{V(x) = \text{null} \quad V \vert\!\frac{q}{q'}\, e_1 \Downarrow v}{V \vert\!\frac{q+K^{\text{matTL}}}{q'}\, \text{matt}(x, e_1, x_0.x_1.x_2.e_2) \Downarrow v}$$

(E-MatT-Node)
$$\frac{V(x) = \langle v_0, v_1, v_2\rangle \quad V[x_0 \mapsto v_0, x_1 \mapsto v_1, x_2 \mapsto v_2] \vert\!\frac{q}{q'}\, e_2 \Downarrow v}{V \vert\!\frac{q+K^{\text{matTN}}}{q'}\, \text{matt}(x, e_1, x_0.x_1.x_2.e_2) \Downarrow v}$$

(E-Cond-True)
$$\frac{V(x) = \text{true} \quad V \vert\!\frac{q}{q'}\, e_1 \Downarrow v}{V \vert\!\frac{q+K^{\text{condT}}}{q'}\, \text{if}(x, e_1, e_2) \Downarrow v}$$

(E-Leaf)
$$\frac{}{V \vert\!\frac{q+K^{\text{leaf}}}{q}\, \text{leaf} \Downarrow \text{null}}$$

(E-Cond-False)
$$\frac{V(x) = \text{false} \quad V \vert\!\frac{q}{q'}\, e_2 \Downarrow v}{V \vert\!\frac{q+K^{\text{condF}}}{q'}\, \text{if}(x, e_1, e_2) \Downarrow v}$$

(E-Share)
$$\frac{V(x) = v \quad V[x_1 \mapsto v, x_2 \mapsto v] \vert\!\frac{q}{q'}\, e \Downarrow v'}{V \vert\!\frac{q}{q'}\, \text{share}(x, x_1.x_2.e) \Downarrow v'}$$

Fig. 5. Evaluation rules of the big-step operational cost semantics

it, respectively.

$$F ::= A_1 \xrightarrow{q/q'} A_2 \text{ where } q, q' \in \mathbb{Q}_0^+$$

The *resource-annotated* typing judgment has the form $\Sigma; \Gamma \vert\!\frac{q}{q'}\, e : A$, where Σ is a finite partial mapping from function identifiers to *nonempty sets of* resource-annotated first-order types, Γ is a resource-annotated typing context, A is a resource-annotated data type, and $q, q' \in \mathbb{Q}_0^+$ are nonnegative numbers. The intuitive meaning is that if there are *at least* $q + \Phi(\Gamma)$ units of potential, then it suffices to evaluate e to a value v satisfying that there are *at least* $q' + \Phi(v : A)$ units of

$\boxed{\Sigma; \Gamma \left|\frac{q}{q'}\right. e : A}$ e has type A under Σ and Γ, and q, q' are constant pre- and post-potential

$$\text{(A-Unit)}\ \frac{}{\cdot \left|\frac{K^{\text{unit}}}{0}\right. \langle\rangle : \text{unit}} \qquad \text{(A-Bool)}\ \frac{b \in \{\text{true}, \text{false}\}}{\cdot \left|\frac{K^{\text{bool}}}{0}\right. b : \text{bool}} \qquad \text{(A-Int)}\ \frac{n \in \mathbb{Z}}{\cdot \left|\frac{K^{\text{int}}}{0}\right. n : \text{int}} \qquad \text{(A-Var)}\ \frac{}{x : A \left|\frac{K^{\text{var}}}{0}\right. x : A} \qquad \text{(A-Op)}\ \frac{}{x_1 : \diamond_{\text{arg}_1}, x_2 : \diamond_{\text{arg}_2} \left|\frac{K^{\text{op}}}{0}\right. \text{op}_\diamond(x_1, x_2) : \diamond_{\text{res}}}$$

$$\text{(A-App)}\ \frac{A_1 \xrightarrow{q/q'} A_2 \in \Sigma(f)}{x : A_1 \left|\frac{q + K^{\text{app}}}{q'}\right. \text{app}(f, x) : A_2} \qquad \text{(A-Let)}\ \frac{\Gamma_1 \left|\frac{q}{q_1}\right. e_1 : A_1 \quad \Gamma_2, x : A_1 \left|\frac{q_1}{q'}\right. e_2 : A_2}{\Gamma_1, \Gamma_2 \left|\frac{q + K^{\text{let}}}{q'}\right. \text{let}(e_1, x.e_2) : A_2} \qquad \text{(A-Pair)}\ \frac{}{x_1 : A_1, x_2 : A_2 \left|\frac{K^{\text{pair}}}{0}\right. \text{pair}(x_1, x_2) : A_1 \times A_2}$$

$$\text{(A-MatP)}\ \frac{\Gamma, x_1 : A_1, x_2 : A_2 \left|\frac{q}{q'}\right. e : A}{\Gamma, x : A_1 \times A_2 \left|\frac{q + K^{\text{matP}}}{q'}\right. \text{matp}(x, x_1.x_2.e) : A} \qquad \text{(A-Nil)}\ \frac{}{\cdot \left|\frac{K^{\text{nil}}}{0}\right. \text{nil} : L^p(A)} \qquad \text{(A-Cons)}\ \frac{}{x_h : A, x_t : L^p(A) \left|\frac{p + K^{\text{cons}}}{0}\right. \text{cons}(x_h, x_t) : L^p(A)}$$

$$\text{(A-MatL)}\ \frac{\Gamma \left|\frac{q - K^{\text{matLN}}}{q'}\right. e_1 : A' \quad \Gamma, x_h : A, x_t : L^p(A) \left|\frac{q + p - K^{\text{matLC}}}{q'}\right. e_2 : A'}{\Gamma, x : L^p(A) \left|\frac{q}{q'}\right. \text{matl}(x, e_1, x_h.x_t.e_2) : A'} \qquad \text{(A-Cond)}\ \frac{\Gamma \left|\frac{q - K^{\text{condT}}}{q'}\right. e_1 : A \quad \Gamma \left|\frac{q - K^{\text{condF}}}{q'}\right. e_2 : A}{\Gamma, x : \text{bool} \left|\frac{q}{q'}\right. \text{if}(x, e_1, e_2) : A}$$

$$\text{(A-Share)}\ \frac{\Gamma, x_1 : A_1, x_2 : A_2 \left|\frac{q}{q'}\right. e : A' \quad \curlyvee(A \mid A_1, A_2)}{\Gamma, x : A \left|\frac{q}{q'}\right. \text{share}(x, x_1.x_2.e) : A'}$$

$$\text{(A-Leaf)}\ \frac{}{\cdot \left|\frac{K^{\text{leaf}}}{0}\right. \text{leaf} : T^p(A)} \qquad \text{(A-Node)}\ \frac{}{x_0 : A, x_1 : T^p(A), x_2 : T^p(A) \left|\frac{p + K^{\text{node}}}{0}\right. \text{node}(x_0, x_1, x_2) : T^p(A)}$$

$$\text{(A-MatT)}\ \frac{\Gamma \left|\frac{q - K^{\text{matTL}}}{q'}\right. e_1 : A' \quad \Gamma, x_0 : A, x_1 : T^p(A), x_2 : T^p(A) \left|\frac{q + p - K^{\text{matTN}}}{q'}\right. e_2 : A'}{\Gamma, x : T^p(A) \left|\frac{q}{q'}\right. \text{matt}(x, e_1, x_0.x_1.x_2.e_2) : A'} \qquad \text{(A-Weakening)}\ \frac{\Gamma \left|\frac{q}{q'}\right. e : A'}{\Gamma, x : A \left|\frac{q}{q'}\right. e : A'}$$

$$\text{(A-Relax)}\ \frac{\Gamma \left|\frac{p}{p'}\right. e : A \quad q \geq p \quad q - p \geq q' - p'}{\Gamma \left|\frac{q}{q'}\right. e : A} \qquad \text{(A-Subtype)}\ \frac{\Gamma \left|\frac{q}{q'}\right. e : A \quad A <: B}{\Gamma \left|\frac{q}{q'}\right. e : B} \qquad \text{(A-Supertype)}\ \frac{\Gamma, x : B \left|\frac{q}{q'}\right. e : C \quad A <: B}{\Gamma, x : A \left|\frac{q}{q'}\right. e : C}$$

Fig. 6. Typing rules of the resource-aware type system

potential leftover after the evaluation.[5] Then a *resource-annotated* program consists of a resource-annotated signature Σ and a family $\{\lambda x^f.e^f\}_{f \in \text{dom}(\Sigma)}$ of function definitions such that $\Sigma; x^f : A \left|\frac{q}{q'}\right. e^f : B$ for every $A \xrightarrow{q/q'} B \in \Sigma(f)$.

The resource-aware typing rules, in fact, form an *affine* linear type system. It ensures that every variable is used *at most* once by allowing exchange and weakening [Walker 2002]. The rules can be organized into syntax-directed and structural rules. Fig. 6 lists the typing rules. We assume a fixed global signature Σ that we omit from the typing rules. While the share expressions make "copies" of a variable, the *sharing relation* $\curlyvee(A \mid A_1, A_2)$ ensures that the program cannot gain more potential

[5] Both the pre- and post-evaluation potentials are needed because resources might be *non*-monotone for the same reason in footnote 3. Although we consider monotone resources in this paper, we keep this design to be consistent with RaML.

by making copies—it apportions the potential indicated by A into two parts to be associated with A_1 and A_2. Formally, this relation is defined as follows.

$$\frac{A \in \{\mathsf{unit}, \mathsf{bool}, \mathsf{int}\}}{\curlyvee(A \mid A, A)} \qquad \frac{\curlyvee(A \mid A_1, A_2) \quad \curlyvee(B \mid B_1, B_2)}{\curlyvee(A \times B \mid A_1 \times B_1, A_2 \times B_2)} \qquad \frac{\curlyvee(A \mid A_1, A_2) \quad p = p_1 + p_2}{\curlyvee(L^p(A) \mid L^{p_1}(A_1), L^{p_2}(A_2))} \qquad \frac{\curlyvee(A \mid A_1, A_2) \quad p = p_1 + p_2}{\curlyvee(T^p(A) \mid T^{p_1}(A_1), T^{p_2}(A_2))}$$

The structural rules (A-Weakening),(A-Relax),(A-Subtype),(A-Supertype) can be applied to every expression. The *sub-typing relation* $A <: B$ indicates that A and B are structurally identical, and for every semantic value a, the potential $\Phi(a : A)$ is greater or equal than the potential $\Phi(a : B)$. Formally, this relation is defined as follows.

$$\frac{A \in \{\mathsf{unit}, \mathsf{bool}, \mathsf{int}\}}{A <: A} \qquad \frac{A_1 <: A_2 \qquad B_1 <: B_2}{A_1 \times B_1 <: A_2 \times B_2} \qquad \frac{A_1 <: A_2 \qquad p_1 \geq p_2}{L^{p_1}(A_1) <: L^{p_2}(A_2)} \qquad \frac{A_1 <: A_2 \qquad p_1 \geq p_2}{T^{p_1}(A_1) <: T^{p_2}(A_2)}$$

Example 3.2. *Recall the program in Fig. 1. An example of a resource-annotated derivation with the heap space metric established by only using syntax-directed rules is as follows. In this typing derivation, every variable is used* exactly *once, which indicates that the annotated potential function for this expression is* tight*—just enough to pay for all the resource usage to complete the evaluation under any environment V such that $\models V : \{y : \mathsf{int} \times \mathsf{int}, xs' : L^3(\mathsf{int})\}$.*

$$\frac{\dfrac{L^3(\mathsf{int}) \xrightarrow{2/0} L^0(\mathsf{int} \times \mathsf{int}) \in \Sigma(\mathsf{lpairs})}{xs' : L^3(\mathsf{int}) \left|^{2}_{0}\right. \mathsf{app}(\mathsf{lpairs}, xs') : L^0(\mathsf{int} \times \mathsf{int})} \qquad \frac{}{y : \mathsf{int} \times \mathsf{int}, ys : L^0(\mathsf{int} \times \mathsf{int}) \left|^{4}_{0}\right. \mathsf{cons}(y, ys) : L^0(\mathsf{int} \times \mathsf{int})}}{y : \mathsf{int} \times \mathsf{int}, xs' : L^3(\mathsf{int}) \left|^{6}_{0}\right. \mathsf{let}(\mathsf{app}(\mathsf{lpairs}, xs'), ys.\mathsf{cons}(y, ys)) : L^0(\mathsf{int} \times \mathsf{int})}$$

Following is an example of derivations involving structural rules. The rule (A-Relax) in the derivation indicates a potential waste of 6 units—hence the annotated potential function for this expression is not tight*. Note the rule (A-Weakening) in the derivation does not indicate potential waste, because the variables x_{12}, x_{22} only carry zero potential.*

$$\frac{\cdots \qquad \dfrac{\dfrac{\dfrac{L^3(\mathsf{int}) \xrightarrow{2/0} L^0(\mathsf{int} \times \mathsf{int}) \in \Sigma(\mathsf{lpairs})}{xs' : L^3(\mathsf{int}) \left|^{2}_{0}\right. \mathsf{app}(\mathsf{lpairs}, xs') : L^0(\mathsf{int} \times \mathsf{int})}}{x_{12} : \mathsf{int}, x_{22} : \mathsf{int}, xs' : L^3(\mathsf{int}) \left|^{2}_{0}\right. \mathsf{app}(\mathsf{lpairs}, xs') : L^0(\mathsf{int} \times \mathsf{int})}\ \textit{A-Weakening} \qquad 8 \geq 2}{x_{12} : \mathsf{int}, x_{22} : \mathsf{int}, xs' : L^3(\mathsf{int}) \left|^{8}_{0}\right. \mathsf{app}(\mathsf{lpairs}, xs') : L^0(\mathsf{int} \times \mathsf{int})}\ \textit{A-Relax}}{b : \mathsf{bool}, x_{12} : \mathsf{int}, x_{22} : \mathsf{int}, xs' : L^3(\mathsf{int}) \left|^{8}_{0}\right. \mathsf{if}(b, \cdots, \mathsf{app}(\mathsf{lpairs}, xs')) : L^0(\mathsf{int} \times \mathsf{int})}$$

Soundness. A crucial characterization of a type system is its soundness with respect to an operational semantics. For resource-aware type systems, soundness theorems state the derived potential functions at the program points are always sufficient to complete the evaluation [Hoffmann et al. 2011, 2017; Hofmann and Jost 2003]. We formalize the soundness theorem of the semantics and the type system as follows.

Theorem 3.3. *If $\models V : \Gamma, V \vdash e \Downarrow v, \Sigma; \Gamma \left|^{q}_{q'}\right. e : A$, then for all $p, r \in \mathbb{Q}_0^+$ such that $p = q + \Phi_V(\Gamma) + r$, there exists $p' \in \mathbb{Q}_0^+$ satisfying $V \left|^{p}_{p'}\right. e \Downarrow v$ and $p' \geq q' + \Phi(v : A) + r$.*

4 PROBLEM STATEMENT

To formalize the problem of worst-case input generation, we introduce input *skeletons*. Skeletons can contain *indeterminate* booleans, integers, as well as unknown structures of inductive data types. The following grammar defines these skeletons $S \in \mathsf{Skel}$.

$$\begin{aligned} S \quad ::= \quad & \mathsf{null} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{bool}^i \mid n \mid \mathsf{int}^i \mid \langle S_1, S_2 \rangle \\ \mid \quad & \textsc{nil} \mid \textsc{cons}(S_h, S_t) \mid \textsc{listof}(S_1, \cdots, S_n) \\ \mid \quad & \textsc{leaf} \mid \textsc{node}(S_0, S_1, S_2) \mid \textsc{treeof}(S_1, \cdots, S_n) \end{aligned}$$

$\boxed{\sigma \vdash S : A}$ Skeleton S has type A under σ

$$\frac{}{\sigma \vdash \mathsf{null} : \mathsf{unit}} \qquad \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\sigma \vdash b : \mathsf{bool}} \qquad \frac{}{\sigma \vdash \mathsf{bool}^i : \mathsf{bool}} \qquad \frac{n \in \mathbb{Z}}{\sigma \vdash n : \mathsf{int}} \qquad \frac{}{\sigma \vdash \mathsf{int}^i : \mathsf{int}}$$

$$\frac{\sigma \vdash S_1 : A_1 \qquad \sigma \vdash S_2 : A_2}{\sigma \vdash \langle S_1, S_2 \rangle : A_1 \times A_2} \qquad \frac{\sigma \vdash \sigma(\ell) : A}{\sigma \vdash \ell : A} \qquad \frac{\forall i \in \{1, \cdots, n\} : \sigma \vdash S_i : A}{\sigma \vdash \textsc{listof}(S_1, \cdots, S_n) : L(A)} \qquad \frac{\forall i \in \{1, \cdots, n\} : \sigma \vdash S_i : A}{\sigma \vdash \textsc{treeof}(S_1, \cdots, S_n) : T(A)}$$

$$\frac{}{\sigma \vdash \textsc{nil} : L(A)} \qquad \frac{\sigma \vdash S_h : A \qquad \sigma \vdash S_t : L(A)}{\sigma \vdash \textsc{cons}(S_h, S_t) : L(A)} \qquad \frac{}{\sigma \vdash \textsc{leaf} : T(A)} \qquad \frac{\sigma \vdash S_0 : A \qquad \sigma \vdash S_1 : T(A) \qquad \sigma \vdash S_2 : T(A)}{\sigma \vdash \textsc{node}(S_0, S_1, S_2) : T(A)}$$

Fig. 7. Typing rules for skeletons

bool^i is a boolean indeterminate with index i. int^i is a integer indeterminate with index i. NIL, $\textsc{cons}(S_h, S_t)$ are list constructors. $\textsc{listof}(S_1, \cdots, S_n)$ is a a list indeterminate with its elements in order. LEAF, $\textsc{node}(S_0, S_1, S_2)$ are binary tree constructors. $\textsc{treeof}(S_1, \cdots, S_n)$ is a binary tree indeterminate with its elements in pre-order.

To allow sharing of unknown data structures in the input skeleton, we introduce *pointers* $\ell \in \mathsf{Loc}$ as skeletons. Then a *skeleton environment* $\gamma : \mathsf{VID} \rightharpoonup \mathsf{Skel}$ is a finite partial mapping from variables to skeletons, and a *skeleton heap* $\sigma : \mathsf{Loc} \rightharpoonup \mathsf{Skel}$ is finite partial mapping from pointers to skeletons. Fig. 7 defines the typing rules for skeletons under a skeleton heap σ, written $\sigma \vdash S : A$. We also write $\sigma \vdash \gamma : \Gamma$, where Γ is a typing context, if $\sigma \vdash \gamma(x) : \Gamma(x)$ for every $x \in \mathrm{dom}(\Gamma)$. In this paper, we assume all the data structure skeletons (i.e., list and binary tree constructors) are saved in the skeleton heap, and the skeleton environment records primitive skeletons (i.e., booleans, integers, and pairs) as well as pointers to data structures.

Given a program, the worst-case input generation is aimed to find a *concretization* of a specified input skeleton, which exposes the worst-case resource usage of the program with respect to the operational cost semantics. A concretization consists of a *model* M to resolve boolean and integer indeterminates, and a *heap* H to resolve unknown structures of inductive data types like lists and binary trees. Formally, a model M is a finite partial mapping from boolean and integer indeterminates to constants, and a heap H is a finite partial mapping from pointers to values. Under a *model* M and a *heap* H, the concretization v of a skeleton S, written $M; H \vdash S \rightsquigarrow v$ is formalized in Fig. 8. We write $M; H \vdash \gamma \rightsquigarrow V$, if $M; H \vdash \gamma(x) \rightsquigarrow V(x)$ for every $x \in \mathrm{dom}(\gamma)$. Because the skeleton environment γ only records primitive skeletons and pointers, the judgment $M; H \vdash \gamma \rightsquigarrow V$ is deterministic. We also write $M \vdash \sigma \sqsubseteq H$, if $M; H \vdash \sigma(\ell) \rightsquigarrow H(\ell)$ for every $\ell \in \mathrm{dom}(\sigma)$. We use the "refinement" operator $\sqsubseteq$ because a skeleton heap might correspond to different concrete heaps.

The general worst-case input generation program can be formalized as follows.

> Given a program with signature Σ and a function f of type $\Sigma(f) = A \to B$, for a specified input skeleton γ, σ such that $\sigma \vdash \gamma : \{x^f : A\}$ (i.e., $\sigma \vdash \gamma(x^f) : A$) and a resource metric, generate a concretization M, H such that $M \vdash \sigma \sqsubseteq H$, $M; H \vdash \gamma \rightsquigarrow V$, $V \left|\frac{q}{q'}\right. e^f \Downarrow v$, and the resource consumption $\delta = q - q'$ is greater or equal to the resource consumption of all possible concretizations of the same input skeleton.

Example 4.1. *Recall the function lpairs in Fig. 1. The type of lpairs is* $L(\mathsf{int}) \to L(\mathsf{int} \times \mathsf{int})$. *The formal parameter of lpairs is* l. *Let* $\gamma = \{l \mapsto \ell\}$ *and* $\sigma = \{\ell \mapsto \textsc{listof}(\mathsf{int}^1, \mathsf{int}^2, \mathsf{int}^3, \mathsf{int}^4)\}$ *be an input skeleton that represents an integer list of length four. A solution to the worst-case input generation for the heap space usage of the function lpairs is* $M = \{\mathsf{int}^1 \mapsto 0, \mathsf{int}^2 \mapsto 1, \mathsf{int}^3 \mapsto 0, \mathsf{int}^4 \mapsto 1\}$, *and* $H = \{\ell \mapsto \langle 0, \langle 1, \langle 0, \langle 1, \mathsf{null}\rangle\rangle\rangle\rangle\}$. *Then* $V(l)$ *represents the list* $[0, 1, 0, 1]$.

$\boxed{M; H \vdash S \rightsquigarrow v}$ Skeleton S is concretized to v under model M and heap H

$$\frac{}{M; H \vdash \text{null} \rightsquigarrow \text{null}} \qquad \frac{}{M; H \vdash \text{bool}^i \rightsquigarrow M(\text{bool}^i)} \qquad \frac{}{M; H \vdash \text{int}^i \rightsquigarrow M(\text{int}^i)} \qquad \frac{}{M; H \vdash \ell \rightsquigarrow H(\ell)}$$

$$\frac{M; H \vdash S_1 \rightsquigarrow v_1 \qquad M; H \vdash S_2 \rightsquigarrow v_2}{M; H \vdash \langle S_1, S_2 \rangle \rightsquigarrow \langle v_1, v_2 \rangle} \qquad \frac{b \in \{\text{true}, \text{false}\}}{M; H \vdash b \rightsquigarrow b} \qquad \frac{n \in \mathbb{Z}}{M; H \vdash n \rightsquigarrow n} \qquad \frac{}{M; H \vdash \textsc{listof}(\cdot) \rightsquigarrow \text{null}}$$

$$\frac{M; H \vdash S_1 \rightsquigarrow v_h \qquad M; H \vdash \textsc{listof}(S_2, \cdots, S_n) \rightsquigarrow v_t}{M; H \vdash \textsc{listof}(S_1, \cdots, S_n) \rightsquigarrow \langle v_h, v_t \rangle} \qquad \frac{}{M; H \vdash \textsc{treeof}(\cdot) \rightsquigarrow \text{null}} \qquad \frac{}{M; H \vdash \textsc{leaf} \rightsquigarrow \text{null}}$$

$$\frac{M; H \vdash S_1 \rightsquigarrow v_0 \qquad M; H \vdash \textsc{treeof}(S_2, \cdots, S_m) \rightsquigarrow v_1 \qquad M; H \vdash \textsc{treeof}(S_{m+1}, \cdots, S_n) \rightsquigarrow v_2}{M; H \vdash \textsc{treeof}(S_1, \cdots, S_n) \rightsquigarrow \langle v_0, v_1, v_2 \rangle}$$

$$\frac{}{M; H \vdash \textsc{nil} \rightsquigarrow \text{null}} \qquad \frac{M; H \vdash S_h \rightsquigarrow v_h \qquad M; H \vdash S_t \rightsquigarrow v_t}{M; H \vdash \textsc{cons}(S_h, S_t) \rightsquigarrow \langle v_h, v_t \rangle} \qquad \frac{M; H \vdash S_0 \rightsquigarrow v_0 \qquad M; H \vdash S_1 \rightsquigarrow v_1 \qquad M; H \vdash S_2 \rightsquigarrow v_2}{M; H \vdash \textsc{node}(S_0, S_1, S_2) \rightsquigarrow \langle v_0, v_1, v_2 \rangle}$$

Fig. 8. Concretization rules for skeletons

We also consider a restricted version of the general problem: If we know an upper bound on the resource usage, we want to generate an input with the same resource usage as the bound indicates.

> Given a program with resource-annotated signature Σ and a function f of type $A \xrightarrow{q/q'} B \in \Sigma(f)$, for a specified input skeleton γ, σ such that $\sigma \vdash \gamma : \{x^f : A\}$, find a concretization M, H satisfying that $M \vdash \sigma \sqsubseteq H$, $M; H \vdash \gamma \rightsquigarrow V$, $V \left|\frac{p}{p'}\right. e^f \Downarrow v$, and $p - p' = (q + \Phi_V(x^f : A)) - (q' + \Phi(v : B))$.

Intuitively, because Thm. 3.3 guarantees the soundness of the upper bound, every input that exposes the exact resource consumption as the upper bound is indeed a worst-case input of its shape. Later we will prove that the solution to the restricted worst-case input generation problem is always a solution to the general one (see §5).

Remark 4.2. *This formalization might seem too restricted at a first glance. However, we find the problem still interesting for two reasons: (i) RaML is quite precise and tight in practice [Hoffmann et al. 2017], and our experiments also show the derived bounds are indeed the resource usage of the worst-case inputs (see §7), and (ii) it is straightforward to modify our algorithm to generate d-*bounded worst-case inputs*, which allow at most d units of potential waste in the execution (see §5).*

5 TYPE-GUIDED WORST-CASE INPUT GENERATION ALGORITHM

In this section, we present our worst-case input generation algorithm and prove its soundness as well as relative completeness.

5.1 Formulation

We formulate our algorithm as a set of rules. The intended purpose of these rules is to search for an execution path with a path constraint sufficient for the input skeleton to expose the worst-case resource usage. The worst-case input generation judgments are of the form $\Sigma; \Gamma; \gamma; \sigma \left|\frac{q}{q'}\right. e : A \Rightarrow \langle \phi, S, \sigma' \rangle$ where γ, σ form an input skeleton such that $\sigma \vdash \gamma : \Gamma$, $\phi \in \mathcal{L}[\text{bool}^i, \text{int}^i]$ is a formula in some theory of booleans and integers with a decision procedure, and S is a skeleton that is intended to have type A under the skeleton heap σ'. In the rules, we restrict the result skeleton S to be either primitive skeletons or pointers to data structures in σ'. The intuitive meaning is that under the environment V that is a concretization of the skeleton environment γ with the skeleton heap σ'

$\boxed{\Sigma; \Gamma; \gamma; \sigma \left|\frac{q}{q'}\right. e : A \Rightarrow \langle \phi, S, \sigma' \rangle}$ Under γ, σ, a worst-case path for e returns S, σ' with constraint ϕ

(WC-Unit)
$$\frac{}{\cdot; \gamma; \sigma \left|\frac{K^{\text{unit}}}{0}\right. \langle\rangle : \text{unit} \Rightarrow \langle \top, \text{null}, \sigma \rangle}$$

(WC-Bool)
$$\frac{b \in \{\text{true}, \text{false}\}}{\cdot; \gamma; \sigma \left|\frac{K^{\text{bool}}}{0}\right. b : \text{bool} \Rightarrow \langle \top, b, \sigma \rangle}$$

(WC-Int)
$$\frac{n \in \mathbb{Z}}{\cdot; \gamma; \sigma \left|\frac{K^{\text{int}}}{0}\right. n : \text{int} \Rightarrow \langle \top, n, \sigma \rangle}$$

(WC-Var)
$$\frac{x \in \text{dom}(\gamma)}{x : A; \gamma; \sigma \left|\frac{K^{\text{var}}}{0}\right. x : A \Rightarrow \langle \top, \gamma(x), \sigma \rangle}$$

(WC-Op)
$$\frac{x_1, x_2 \in \text{dom}(\gamma) \qquad S = \gamma(x_1) \diamond \gamma(x_2)}{x_1 : \diamond_{\text{arg}_1}, x_2 : \diamond_{\text{arg}_2}; \gamma; \sigma \left|\frac{K^{\text{op}}}{0}\right. \text{op}_\diamond(x_1, x_2) : \diamond_{\text{res}} \Rightarrow \langle \top, S, \sigma \rangle}$$

(WC-App)
$$\frac{\gamma(x) = S \qquad A_1 \xrightarrow{q/q'} A_2 \in \Sigma(f) \qquad x^f : A_1; \gamma[x^f \mapsto S]; \sigma \left|\frac{q}{q'}\right. e^f : A_2 \Rightarrow \langle \phi, S', \sigma' \rangle}{x : A_1; \gamma; \sigma \left|\frac{q+K^{\text{app}}}{q'}\right. \text{app}(f, x) : A_2 \Rightarrow \langle \phi, S', \sigma' \rangle}$$

(WC-Let)
$$\frac{\Gamma_1; \gamma; \sigma \left|\frac{q}{q_1}\right. e_1 : A_1 \Rightarrow \langle \phi_1, S_1, \sigma_1 \rangle \qquad \Gamma_2, x : A_1; \gamma[x \mapsto S_1]; \sigma_1 \left|\frac{q_1}{q'}\right. e_2 : A_2 \Rightarrow \langle \phi_2, S_2, \sigma_2 \rangle}{\Gamma_1, \Gamma_2; \gamma; \sigma \left|\frac{q+K^{\text{let}}}{q'}\right. \text{let}(e_1, x.e_2) : A_2 \Rightarrow \langle \phi_1 \wedge \phi_2, S_2, \sigma_2 \rangle}$$

(WC-Pair)
$$\frac{A = A_1 \times A_2 \qquad x_1, x_2 \in \text{dom}(\gamma) \qquad S = \langle \gamma(x_1), \gamma(x_2) \rangle}{x_1 : A_1, x_2 : A_2; \gamma; \sigma \left|\frac{K^{\text{pair}}}{0}\right. \text{pair}(x_1, x_2) : A \Rightarrow \langle \top, S, \sigma \rangle}$$

(WC-MatP)
$$\frac{\gamma(x) = \langle S_1, S_2 \rangle \qquad \gamma_o = \gamma[x_1 \mapsto S_1, x_2 \mapsto S_2] \qquad \Gamma, x_1 : A_1, x_2 : A_2; \gamma_o; \sigma \left|\frac{q}{q'}\right. e : A \Rightarrow \langle \phi, S, \sigma' \rangle}{\Gamma, x : A_1 \times A_2; \gamma; \sigma \left|\frac{q+K^{\text{matP}}}{q'}\right. \text{matp}(x, x_1.x_2.e) : A \Rightarrow \langle \phi, S, \sigma' \rangle}$$

(WC-Nil)
$$\frac{\ell \notin \text{dom}(\sigma)}{\cdot; \gamma; \sigma \left|\frac{K^{\text{nil}}}{0}\right. \text{nil} : L^p(A) \Rightarrow \langle \top, \ell, \sigma[\ell \mapsto \text{NIL}] \rangle}$$

(WC-MatL-Cons)
$$\frac{\gamma(x) = \ell \qquad \sigma(\ell) = \text{CONS}(S_h, S_t) \qquad \gamma' = \gamma[x_h \mapsto S_h, x_t \mapsto S_t] \qquad \Gamma, x_h : A, x_t : L^p(A); \gamma'; \sigma \left|\frac{q+p-K^{\text{matLC}}}{q'}\right. e_2 : A' \Rightarrow \langle \phi, S, \sigma' \rangle}{\Gamma, x : L^p(A); \gamma; \sigma \left|\frac{q}{q'}\right. \text{matl}(x, e_1, x_h.x_t.e_2) : A' \Rightarrow \langle \phi, S, \sigma' \rangle}$$

(WC-Cons)
$$\frac{x_h, x_t \in \text{dom}(\gamma) \qquad R = \text{CONS}(\gamma(x_h), \gamma(x_t)) \qquad \ell \notin \text{dom}(\sigma) \qquad \sigma' = \sigma[\ell \mapsto R] \qquad \Gamma = x_h : A, x_t : L^p(A)}{\Gamma; \gamma; \sigma \left|\frac{p+K^{\text{cons}}}{0}\right. \text{cons}(x_h, x_t) : L^p(A) \Rightarrow \langle \top, \ell, \sigma' \rangle}$$

(WC-MatL-Nil)
$$\frac{e = \text{matl}(x, e_1, x_h.x_t.e_2) \qquad \gamma(x) = \ell \qquad \sigma(\ell) = \text{NIL} \qquad \Gamma; \gamma; \sigma \left|\frac{q-K^{\text{matLN}}}{q'}\right. e_1 : A' \Rightarrow \langle \phi, S, \sigma' \rangle}{\Gamma, x : L^p(A); \gamma; \sigma \left|\frac{q}{q'}\right. e : A' \Rightarrow \langle \phi, S, \sigma' \rangle}$$

(WC-MatL-List-Empty)
$$\frac{\gamma(x) = \ell \qquad \sigma(\ell) = \text{LISTOF}(\cdot) \qquad \Gamma; \gamma; \sigma[\ell \mapsto \text{NIL}] \left|\frac{q-K^{\text{matLN}}}{q'}\right. e_1 : A' \Rightarrow \langle \phi, S, \sigma' \rangle}{\Gamma, x : L^p(A); \gamma; \sigma \left|\frac{q}{q'}\right. \text{matl}(x, e_1, x_h.x_t.e_2) : A' \Rightarrow \langle \phi, S, \sigma' \rangle}$$

(WC-MatL-List-NonEmpty)
$$\frac{\gamma(x) = \ell \qquad \sigma(\ell) = \text{LISTOF}(S_1, \cdots, S_n) \qquad \ell_t \notin \text{dom}(\sigma) \qquad S_t = \text{LISTOF}(S_2, \cdots, S_n) \qquad \Gamma, x_h : A, x_t : L^p(A); \gamma[x_h \mapsto S_1, x_t \mapsto \ell_t]; \sigma[\ell \mapsto \text{CONS}(S_1, \ell_t), \ell_t \mapsto S_t] \left|\frac{q+p-K^{\text{matLC}}}{q'}\right. e_2 : A' \Rightarrow \langle \phi, S, \sigma' \rangle}{\Gamma, x : L^p(A); \gamma; \sigma \left|\frac{q}{q'}\right. \text{matl}(x, e_1, x_h.x_t.e_2) : A' \Rightarrow \langle \phi, S, \sigma' \rangle}$$

Fig. 9. Rules of the type-guided worst-case input generation algorithm (I)

and satisfies the constraint ϕ, it furthermore takes $q + \Phi_V(\Gamma)$ units of resource to evaluate e to a value v , which is the corresponding concretization of S and there are *exactly* $q' + \Phi(v : A)$ units of resource left over. These rules essentially formulate a type-guided symbolic execution of the expression e. Figs. 9 and 10 present the syntax-directed rules. We assume a fixed global signature Σ.

$\boxed{\Sigma;\Gamma;\gamma;\sigma \left|\frac{q}{q'}\right. e : A \Rightarrow \langle \phi, S, \sigma'\rangle}$ Under γ, σ, a worst-case path for e returns S, σ' with constraint ϕ

(WC-Leaf)

$$\frac{\ell \notin \mathrm{dom}(\sigma)}{\cdot;\gamma;\sigma \left|\frac{K^{\mathrm{leaf}}}{0}\right. \mathrm{leaf} : T^p(A) \Rightarrow \langle \top, \ell, \sigma[\ell \mapsto \textsc{leaf}]\rangle}$$

(WC-MatT-Leaf)

$$\frac{\gamma(x) = \ell \qquad \sigma(\ell) = \textsc{leaf} \qquad \Gamma;\gamma;\sigma \left|\frac{q-K^{\mathrm{matTL}}}{q'}\right. e_1 : A' \Rightarrow \langle \phi, S, \sigma'\rangle}{\Gamma, x : T^p(A);\gamma;\sigma \left|\frac{q}{q'}\right. \mathrm{matt}(x, e_1, x_0.x_1.x_2.e_2) : A' \Rightarrow \langle \phi, S, \sigma'\rangle}$$

(WC-Node)

$$\frac{x_0, x_1, x_2 \in \mathrm{dom}(\gamma) \qquad R = \textsc{node}(\gamma(x_0), \gamma(x_1), \gamma(x_2)) \qquad \ell \notin \mathrm{dom}(\sigma)}{x_0 : A, x_1 : T^p(A), x_2 : T^p(A);\gamma;\sigma \left|\frac{p+K^{\mathrm{node}}}{0}\right. \mathrm{node}(x_0, x_1, x_2) : T^p(A) \Rightarrow \langle \top, \ell, \sigma[\ell \mapsto R]\rangle}$$

(WC-MatT-Node)

$$\frac{\gamma(x) = \ell \qquad \sigma(\ell) = \textsc{node}(S_0, S_1, S_2) \qquad \Gamma, x_0 : A, x_1 : T^p(A), x_2 : T^p(A);\gamma[x_0 \mapsto S_0, x_1 \mapsto S_1, x_2 \mapsto S_2];\sigma \left|\frac{q+p-K^{\mathrm{matTN}}}{q'}\right. e_2 : A' \Rightarrow \langle \phi, S, \sigma'\rangle}{\Gamma, x : T^p(A);\gamma;\sigma \left|\frac{q}{q'}\right. \mathrm{matt}(x, e_1, x_0.x_1.x_2.e_2) : A' \Rightarrow \langle \phi, S, \sigma'\rangle}$$

(WC-MatT-Tree-Empty)

$$\frac{\gamma(x) = \ell \qquad \sigma(\ell) = \textsc{treeof}(\cdot) \qquad \Gamma;\gamma;\sigma[\ell \mapsto \textsc{leaf}] \left|\frac{q-K^{\mathrm{matTL}}}{q'}\right. e_1 : A' \Rightarrow \langle \phi, S, \sigma'\rangle}{\Gamma, x : T^p(A);\gamma;\sigma \left|\frac{q}{q'}\right. \mathrm{matt}(x, e_1, x_0.x_1.x_2.e_2) : A' \Rightarrow \langle \phi, S, \sigma'\rangle}$$

(WC-MatT-Tree-NonEmpty)

$$\frac{\begin{array}{c}\gamma(x) = \ell \qquad \sigma(\ell) = \textsc{treeof}(S_1, \cdots, S_n) \qquad \ell_1, \ell_2 \notin \mathrm{dom}(\sigma) \\ R_1 = \textsc{treeof}(S_2, \cdots, S_m) \qquad R_2 = \textsc{treeof}(S_{m+1}, \cdots, S_n) \qquad \sigma_o = \sigma[\ell \mapsto \textsc{node}(S_1, \ell_1, \ell_2), \ell_1 \mapsto R_1, \ell_2 \mapsto R_2] \\ \Gamma, x_0 : A, x_1 : T^p(A), x_2 : T^p(A);\gamma[x_0 \mapsto S_1, x_1 \mapsto \ell_1, x_2 \mapsto \ell_2];\sigma_o \left|\frac{q+p-K^{\mathrm{matTN}}}{q'}\right. e_2 : A' \Rightarrow \langle \phi, S, \sigma'\rangle\end{array}}{\Gamma, x : T^p(A);\gamma;\sigma \left|\frac{q}{q'}\right. \mathrm{matt}(x, e_1, x_0.x_1.x_2.e_2) : A' \Rightarrow \langle \phi, S, \sigma'\rangle}$$

(WC-Cond-True)

$$\frac{\gamma(x) = S \qquad \Gamma;\gamma;\sigma \left|\frac{q-K^{\mathrm{condT}}}{q'}\right. e_1 : A \Rightarrow \langle \phi, S', \sigma'\rangle}{\Gamma, x : \mathrm{bool};\gamma;\sigma \left|\frac{q}{q'}\right. \mathrm{if}(x, e_1, e_2) : A \Rightarrow \langle S \wedge \phi, S', \sigma'\rangle}$$

(WC-Cond-False)

$$\frac{\gamma(x) = S \qquad \Gamma;\gamma;\sigma \left|\frac{q-K^{\mathrm{condF}}}{q'}\right. e_2 : A \Rightarrow \langle \phi, S', \sigma'\rangle}{\Gamma, x : \mathrm{bool};\gamma;\sigma \left|\frac{q}{q'}\right. \mathrm{if}(x, e_1, e_2) : A \Rightarrow \langle \neg S \wedge \phi, S', \sigma'\rangle}$$

(WC-Share)

$$\frac{\gamma(x) = S \qquad \Gamma, x_1 : A_1, x_2 : A_2;\gamma[x_1 \mapsto S, x_2 \mapsto S];\sigma \left|\frac{q}{q'}\right. e : A' \Rightarrow \langle \phi, S', \sigma'\rangle \qquad \curlyvee(A \mid A_1, A_2)}{\Gamma, x : A;\gamma;\sigma \left|\frac{q}{q'}\right. \mathrm{share}(x, x_1.x_2.e) : A' \Rightarrow \langle \phi, S', \sigma'\rangle}$$

Fig. 10. Rules of the type-guided worst-case input generation algorithm (II)

Most of these rules are *deterministic*—for a configuration of the input skeleton γ, σ and the expression e, the generation algorithm is usually able to pick a unique evaluation step. For example, for the expression $\mathrm{let}(e_1, x.e_2)$, the rule (WC-Let) first generates a candidate worst-case execution path for e_1 and then returns a path constraint ϕ_1 together with the corresponding result skeleton S_1. The rule then generates a worst-case execution path for e_2 under the same skeleton environment with the binding variable x updated with S_1. If the path constraint for e_2 is ϕ_2, the conjunction of two path constraints $\phi_1 \wedge \phi_2$ is a sufficient condition for the let-expression to expose worst-case resource usage.

The rule (WC-App) for function applications looks up the skeleton of x in the current skeleton environment, and passes it to the function body e^f to generate a candidate worst-case execution

path. We treat inductive data structures differently from the operational cost semantics in Fig. 5. For list and binary tree constructors, we create a fresh pointer and put the data structure in the inductive skeleton heap. For example, the rule (WC-Node) for the expression $\text{node}(x_0, x_1, x_2)$ first looks up the skeletons of x_0, x_1, x_2 in the current skeleton environment as S_0, S_1, S_2, respectively. Then it creates an inductive skeleton for a binary tree node as $\text{node}(S_0, S_1, S_2)$, and puts it in a fresh location of the skeleton heap.

There are three rules that exhibit nondeterminism: (WC-Cond-True), (WC-Cond-False), and (WC-MatT-Tree-NonEmpty). The first two rules are nondeterministic because the predicate of a conditional expression might not be able to resolve because the predicate might refer to indeterminate booleans and integers. For example, for the conditional expression $\text{if}(x, e_1, e_2)$, the rule (WC-Cond-True) looks up the skeleton of x in the current skeleton environment as S, and then tries to find a path constraint ϕ for e_1 to trigger worst-case behavior, and then return a path constraint $S \wedge \phi$ that indicates the expression evaluates the then-branch. The nondeterminism of the rule (WC-MatT-Tree-NonEmpty) arises because the structure of the binary tree being matched is unknown. Suppose the inductive skeleton for the tree is $\text{treeof}(S_1, \cdots, S_n)$. Because the elements are in pre-order, the element assigned to the root of this tree is S_1, and the input generation algorithm tries to partition $\{S_2, \cdots, S_n\}$ into the left and right subtrees. Suppose $R_1 = \text{treeof}(S_2, \cdots, S_m)$ and $R_2 = \text{treeof}(S_{m+1}, \cdots, S_n)$ are two inductive skeletons for the left and right subtrees, respectively. Then the algorithm records the partition in the skeleton heap and then proceeds to search path constraints for the body expression of the match-expression.

Example 5.1. *Recall the program in Fig. 1 and consider the subexpression* $\text{let}(\text{app}(\text{lpairs}, xs'), ys.\text{cons}(y, ys))$. *Let an input skeleton be* $\gamma = \{y \mapsto \langle \text{int}^1, \text{int}^2 \rangle, xs' \mapsto \ell_1\}$, $\sigma = \{\ell_1 \mapsto \text{cons}(\text{int}^3, \text{cons}(\text{int}^4, \text{nil}))\}$. *For the heap space metric, our algorithm derives the following judgment for the function call:* $xs' : L^3(\text{int}); \gamma; \sigma \left|^2_0\right. \text{app}(\text{lpairs}, xs') : L^0(\text{int} \times \text{int}) \Rightarrow \langle \text{int}^3 < \text{int}^4, \ell_3, \sigma_1 \rangle$ *where* $\sigma_1 = \sigma[\ell_2 \mapsto \text{nil}, \ell_3 \mapsto \text{cons}(\langle \text{int}^3, \text{int}^4 \rangle, \ell_2)]$. *Then for the body expression of the let-expression, our algorithm derives the following judgment by setting the binding variable ys to* ℓ_3 *in the skeleton heap* σ_1: $y : \text{int} \times \text{int}; ys : L^0(\text{int} \times \text{int}); \gamma[ys \mapsto \ell_3]; \sigma_1 \left|^4_0\right. \text{cons}(y, ys) : L^0(\text{int} \times \text{int}) \Rightarrow \langle \top, \ell_4, \sigma_2 \rangle$ *where* $\sigma_2 = \sigma_1[\ell_4 \mapsto \text{cons}(\langle \text{int}^1, \text{int}^2 \rangle, \ell_3)]$. *Thus by rule (WC-Let) we have the following:* $y : \text{int} \times \text{int}, xs'; \gamma; \sigma \left|^6_0\right. \text{let}(\text{app}(\text{lpairs}, xs'), ys.\text{cons}(y, ys)) : L^0(\text{int} \times \text{int}) \Rightarrow \langle \text{int}^3 < \text{int}^4, \ell_4, \sigma_2 \rangle$. *The list that* ℓ_4 *points to then corresponds to* $[\langle \text{int}^1, \text{int}^2 \rangle, \langle \text{int}^3, \text{int}^4 \rangle]$.

In order to formulate our input generation algorithm for structural typing rules, we define the *potential* of skeletons, written $\tilde{\Phi}_\sigma(S : A)$, as follows.

$$\begin{aligned}
\tilde{\Phi}_\sigma(S : A) &= 0 \quad \text{where } A \in \{\text{unit}, \text{bool}, \text{int}\} \\
\tilde{\Phi}_\sigma(S : A_1 \times A_2) &= \tilde{\Phi}_\sigma(S_1 : A_1) + \tilde{\Phi}_\sigma(S_2 : A_2) \quad \text{where } S = \langle S_1, S_2 \rangle \\
\tilde{\Phi}_\sigma(\ell : A) &= \tilde{\Phi}_\sigma(R : A) \quad \text{where } R = \sigma(\ell) \\
\tilde{\Phi}_\sigma(\text{nil} : L^p(A)) &= 0 \\
\tilde{\Phi}_\sigma(\text{cons}(S_h, S_t) : L^p(A)) &= p + \tilde{\Phi}_\sigma(S_h : A) + \tilde{\Phi}_\sigma(S_t : L^p(A)) \\
\tilde{\Phi}_\sigma(\text{listof}(S_1, \cdots, S_n) : L^p(A)) &= n \cdot p + \textstyle\sum_{i=1}^n \tilde{\Phi}_\sigma(S_i : A) \\
\tilde{\Phi}_\sigma(\text{leaf} : T^p(A)) &= 0 \\
\tilde{\Phi}_\sigma(\text{node}(S_0, S_1, S_2) : T^p(A)) &= p + \tilde{\Phi}_\sigma(S_0 : A) + \tilde{\Phi}_\sigma(S_1 : T^p(A)) + \tilde{\Phi}_\sigma(S_2 : T^p(A)) \\
\tilde{\Phi}_\sigma(\text{treeof}(S_1, \cdots, S_n) : T^p(A)) &= n \cdot p + \textstyle\sum_{i=1}^n \tilde{\Phi}_\sigma(S_i : A)
\end{aligned}$$

Fig. 11 shows the rules for worst-case input generation against structural rules. Our algorithm supports structural rules but forces these rules not to waste potential. The rule (WC-Weakening)

$\boxed{\Sigma;\Gamma;\gamma;\sigma \left|\frac{q}{q'}\right. e : A \Rightarrow \langle\phi, S, \sigma'\rangle}$ Under γ, σ, a worst-case path for e returns S, σ' with constraint ϕ

(WC-Weakening)

$$\frac{\Gamma;\gamma;\sigma \left|\frac{q}{q'}\right. e : A' \Rightarrow \langle\phi, S', \sigma'\rangle \quad \gamma(x) = S \quad \tilde{\Phi}_\sigma(S : A) = 0}{\Gamma, x : A;\gamma;\sigma \left|\frac{q}{q'}\right. e : A' \Rightarrow \langle\phi, S', \sigma'\rangle}$$

(WC-Relax)

$$\frac{\Gamma;\gamma;\sigma \left|\frac{p}{p'}\right. e : A \Rightarrow \langle\phi, S, \sigma'\rangle \quad q \geq p \quad q - p = q' - p'}{\Gamma;\gamma;\sigma \left|\frac{q}{q'}\right. e : A \Rightarrow \langle\phi, S, \sigma'\rangle}$$

(WC-Subtype)

$$\frac{\Gamma;\gamma;\sigma \left|\frac{q}{q'}\right. e : A \Rightarrow \langle\phi, S, \sigma'\rangle \quad A <: B \quad \tilde{\Phi}_{\sigma'}(S : A) = \tilde{\Phi}_{\sigma'}(S : B)}{\Gamma;\gamma;\sigma \left|\frac{q}{q'}\right. e : B \Rightarrow \langle\phi, S, \sigma'\rangle}$$

(WC-Supertype)

$$\frac{\Gamma, x : B;\gamma;\sigma \left|\frac{q}{q'}\right. e : C \Rightarrow \langle\phi, S', \sigma'\rangle \quad A <: B \quad \gamma(x) = S \quad \tilde{\Phi}_\sigma(S : A) = \tilde{\Phi}_\sigma(S : B)}{\Gamma, x : A;\gamma;\sigma \left|\frac{q}{q'}\right. e : C \Rightarrow \langle\phi, S', \sigma'\rangle}$$

Fig. 11. Rules of the resource-aware worst-case input generation algorithm (III)

requires the variable x that is thrown away to carry zero potential. The rule (WC-Relax) still permits adding some constant number to the potential functions, but the amounts added to the potential before evaluation of an expression and after the evaluation must be identical. Sub-typing is permitted if the skeleton has the same potential with respect to the types A, B where A is a sub-type of B.

After the worst-case input generation algorithm establishes a judgment $\Sigma;\Gamma;\gamma;\sigma \left|\frac{q}{q'}\right. e : A \Rightarrow \langle\phi, S, \sigma'\rangle$, we use the decision procedure for $\mathcal{L}[\mathsf{bool}^i, \mathsf{int}^i]$ to find a model M for the path constraint ϕ. If the model M is found, we can then use it to concretize the input skeleton γ, σ to a concrete input that will expose the worst-case resource consumption.

Example 5.2. *Recall the program in Fig. 1 with the function lpairs. Let an input skeleton be* $\gamma = \{l \mapsto \ell_1\}$, $\sigma = \{\ell_1 \mapsto \mathit{cons}(\mathsf{int}^1, \mathit{cons}(\mathsf{int}^2, \mathit{cons}(\mathsf{int}^3, \mathit{cons}(\mathsf{int}^4, \mathit{nil}))))\}$. *For the heap space metric, our algorithm derives*

$$l : L^3(\mathsf{int});\gamma;\sigma \left|\frac{2}{0}\right. \mathsf{app}(\mathsf{lpairs}, l) : L^0(\mathsf{int} \times \mathsf{int}) \Rightarrow \langle(\mathsf{int}^1 < \mathsf{int}^2) \wedge (\mathsf{int}^3 < \mathsf{int}^4), \ell_4, \sigma'\rangle$$

where $\sigma' = \sigma[\ell_2 \mapsto \mathit{nil}, \ell_3 \mapsto \mathit{cons}(\langle\mathsf{int}^3, \mathsf{int}^4\rangle, \ell_2), \ell_4 \mapsto \mathit{cons}(\langle\mathsf{int}^1, \mathsf{int}^2\rangle, \ell_3)]$. *The constraint* $(\mathsf{int}^1 < \mathsf{int}^2) \wedge (\mathsf{int}^3 < \mathsf{int}^4)$ *is satisfiable in the model* $M = \{\mathsf{int}^1 \mapsto 0, \mathsf{int}^2 \mapsto 1, \mathsf{int}^3 \mapsto 0, \mathsf{int}^4 \mapsto 1\}$. *Hence our algorithm finds a worst case input* $[0, 1, 0, 1]$ *for the function lpairs.*

Remark 5.3. *A practical relaxation of the formalization for worst-case input generation problem could be that we allow a bounded amount of resource waste from the inferred resource bound. We call the problem that allows d units of potential waste the* d-bounded worst-case input generation. *It is straightforward to extend our algorithm by adding a component to record current potential waste and forcing the waste not to exceed the specified bound d. For example, the rule (WC-Relax) can be modified as follows where* $w, w' \in \mathbb{Q}_0^+$ *stand for potential waste.*

(WC-Relax)

$$\frac{\Gamma;\gamma;\sigma \left|\frac{p}{p'}\right. e : A \Rightarrow \langle\phi, S, \sigma', w\rangle \quad q \geq p \quad q - p \geq q' - p' \quad w' = w + ((q - p) - (q' - p')) \quad w' \leq d}{\Gamma;\gamma;\sigma \left|\frac{q}{q'}\right. e : A \Rightarrow \langle\phi, S, \sigma', w'\rangle}$$

5.2 Proof

Complete proofs are included in the extended version of this paper [Wang and Hoffmann 2018].

Soundness. The soundness theorem states that if for a function f with a resource-annotated type, the worst-case input generation algorithm terminates with $\langle\phi, S, \sigma'\rangle$ under the skeleton

environment γ and the skeleton heap σ, then the evaluation of the function f under the concrete environment V that is the concretization of γ, σ that satisfies ϕ consumes the amount of resource *exactly* the same as the inferred upper bound.

Theorem 5.4 (Soundness). *If* $\Sigma; x^f : A_1; \gamma; \sigma \left|\frac{q}{q'}\right. e^f : A_2 \Rightarrow \langle \phi, S, \sigma' \rangle$, $\sigma \vdash \gamma : (x^f : A_1)$, *$M$ is a model for ϕ, $M \vdash \sigma' \sqsubseteq H$, and $M; H \vdash \gamma \rightsquigarrow V$, then there exists a value v, satisfying* $V \left|\frac{q+\Phi_V(x^f:A_1)}{q'+\Phi(v:A_2)}\right. e^f \Downarrow v$, *and* $M; H \vdash S \rightsquigarrow v$.

To establish soundness, we prove the following generalized theorem.

Theorem 5.5. *If* $\Sigma; \Gamma; \gamma; \sigma \left|\frac{q}{q'}\right. e : A \Rightarrow \langle \phi, S, \sigma' \rangle$, $\sigma \vdash \gamma : \Gamma$, *$M$ is a model for ϕ, $M \vdash \sigma' \sqsubseteq H$, and $M; H \vdash \gamma \rightsquigarrow V$, then for all $p, r \in \mathbb{Q}_0^+$ such that $p = q + \Phi_V(\Gamma) + r$, there exist $p' \in \mathbb{Q}_0^+$ and a value v, satisfying* $V \left|\frac{p}{p'}\right. e \Downarrow v$, $p' = q' + \Phi(v : A) + r$, *and* $M; H \vdash S \rightsquigarrow v$.

Proof. By induction on the derivation of $\Sigma; \Gamma; \gamma; \sigma \left|\frac{q}{q'}\right. e : A \Rightarrow \langle \phi, S, \sigma' \rangle$. □

Remark 5.6. *Suppose f is a function with the signature $A_1 \xrightarrow{q/q'} A_2 \in \Sigma(f)$ with $q' = 0$ and $\curlyvee(A_2 \mid A_2, A_2)$, i.e., the result type carries only zero potential. Given an input skeleton γ, σ such that $\sigma \vdash \gamma : (x^f : A_1)$, let $\Psi = q + \tilde{\Phi}_\sigma(\gamma(x^f) : A_1)$, then for every concretization M, H such that $M \vdash \sigma \sqsubseteq H$, $M; H \vdash \gamma \rightsquigarrow V$, we have $\Psi = q + \Phi(V(x^f) : A_1) = q + \Phi_V(x^f : A_1)$. Hence by Thm. 3.3, for every V such that $\models V : (x^f : A_1)$, if $V \left|\frac{p}{p'}\right. e^f \Downarrow v$, then $p - p' \leq (q + \Phi_V(x^f : A_1)) - (q' + \Phi(v : A_2)) = \Psi$. If $\Sigma; x^f : A_1; \gamma; \sigma \left|\frac{q}{q'}\right. e^f : A_2 \Rightarrow \langle \phi, S, \sigma' \rangle$, M is a model for ϕ, $M \vdash \sigma' \sqsubseteq H$, and $M; H \vdash \gamma \rightsquigarrow V$, then by Thm. 5.5, there exists a value v such that $V \left|\frac{\Psi}{0}\right. e^f \Downarrow v$, and hence M, H exposes the resource usage that is greater or equal to the resource consumption of all other concretizations.*

Relative Completeness. We now want to study the *completeness* of our worst-case input generation algorithm. Although the theory $\mathcal{L}[\mathsf{bool}^i, \mathsf{int}^i]$ for booleans and integers might be undecidable, we prove our algorithm is complete *modulo constraint solving*. If a function f with a resource-annotated type has a worst-case input that is a concretization of the input skeleton γ, σ and exposes *exactly* the same resource usage as the inferred upper bound, then our algorithm is able to find a path constraint that corresponds to the concretization.

Theorem 5.7 (Completeness). *If* $\Sigma; x^f : A_1 \left|\frac{q}{q'}\right. e^f : A_2$, $\models V : \Gamma$, $V \left|\frac{q+\Phi_V(x^f:A_1)}{q'+\Phi(v:A_2)}\right. e \Downarrow v$, $\sigma \vdash \gamma : (x^f : A_1)$, *$M \vdash \sigma \sqsubseteq H$, and $M; H \vdash \gamma \rightsquigarrow V$, then there exist ϕ, S, σ', satisfying* $\Sigma; x^f : A_1; \gamma; \sigma \left|\frac{q}{q'}\right. e^f : A_2 \Rightarrow \langle \phi, S, \sigma' \rangle$, *and M is a model for ϕ.*

To establish completeness, we prove the following generalized theorem.

Theorem 5.8. *If* $\Sigma; \Gamma \left|\frac{q}{q'}\right. e : A$, $\models V : \Gamma$, $V \left|\frac{p}{p'}\right. e \Downarrow v$, $p = q + \Phi_V(\Gamma) + r$, $p' = q' + \Phi(v : A) + r$, $\sigma \vdash \gamma : \Gamma$, *$M \vdash \sigma \sqsubseteq H$, and $M; H \vdash \gamma \rightsquigarrow V$, then there exist ϕ, S, σ', H', satisfying* $\Sigma; \Gamma; \gamma; \sigma \left|\frac{q}{q'}\right. e : A \Rightarrow \langle \phi, S, \sigma' \rangle$, *$M$ is a model for ϕ, $H \subseteq H'$, $M \vdash \sigma' \sqsubseteq H'$, $M; H' \vdash \gamma \rightsquigarrow V$, and $M; H' \vdash S \rightsquigarrow v$.*

Proof. By induction on the derivation of $V \left|\frac{p}{p'}\right. e \Downarrow v$ and the derivation of $\Gamma \left|\frac{q}{q'}\right. e : A$, where the derivation of the evaluation judgment takes priority over the typing judgment. □

6 HEURISTICS FOR COMPOSITIONAL INPUT GENERATION

The type-guided worst-case input generation algorithm developed in §5 could become inefficient when the input skeleton is large and there remain a lot of candidate execution paths to investigate, even after the resource-annotated derivation has already helped prune the search space.

Let us first investigate the possible causes of inefficiency. As we already discussed in §5.1, most of the generation rules are deterministic, except the following three rules: (WC-Cond-True), (WC-Cond-False), and (WC-MatT-Tree-NonEmpty). For the first two rules, the nondeterminism occurs because our algorithm does not know the actual value of the predicate of a conditional expression. For the third rule, the nondeterminism comes from the enumeration of possible tree structures. When the size of the input skeleton increases, the total number of combinations that come from the nondeterministic rules is likely to exhibit an exponential blowup.

One way to improve the scalability of our input generation algorithm is to exploit *compositionality*—specifically, we hope to restrict the combinations of execution paths *inside* the function boundaries. Intuitively, when we search for a candidate path constraint for a function on an input skeleton, we want to first generate feasible path constraints for function calls inside the function body on subparts of the input skeleton, and then combine these constraints in a sound way.

Thm. 5.5 provides soundness guarantee for our input generation algorithm. The theorem implies that even if we only enable a subset of the generation rules, the algorithm always returns correct sufficient constraints for worst-case inputs, *if it terminates with some results*. This property gives us several opportunities to devise search heuristics that can enable, disable, and prioritize partial executions during the generation algorithm. In this section, we develop two search heuristics for compositional input generation.

6.1 Uniform Execution

To get rid of nondeterministic rules for conditional expressions, one idea is to force the algorithm to choose the same branch for each conditional expression. Because on worst-case inputs the program always executes the same branch, we call this heuristic *uniform execution*. In this way, the algorithm only needs to enumerate a global configuration for conditional expressions. Formally, given a *global configuration* $\mathrm{config} : \mathrm{Exp} \to \{\leftarrow, \rightarrow\}$, the worst-case input generation algorithm proceeds as follows for conditional expressions.

(WC-Cond-True)

$$\frac{\mathrm{config}(\mathrm{if}(x, e_1, e_2)) = \leftarrow \qquad \gamma(x) = S \qquad \Gamma; \gamma; \sigma \left|\frac{q-K^{\mathrm{condT}}}{q'}\right. e_1 : A \Rightarrow \langle \phi, S', \sigma' \rangle}{\Gamma, x : \mathrm{bool}; \gamma; \sigma \left|\frac{q}{q'}\right. \mathrm{if}(x, e_1, e_2) : A \Rightarrow \langle S \wedge \phi, S', \sigma' \rangle}$$

(WC-Cond-False)

$$\frac{\mathrm{config}(\mathrm{if}(x, e_1, e_2)) = \rightarrow \qquad \gamma(x) = S \qquad \Gamma; \gamma; \sigma \left|\frac{q-K^{\mathrm{condF}}}{q'}\right. e_2 : A \Rightarrow \langle \phi, S', \sigma' \rangle}{\Gamma, x : \mathrm{bool}; \gamma; \sigma \left|\frac{q}{q'}\right. \mathrm{if}(x, e_1, e_2) : A \Rightarrow \langle \neg S \wedge \phi, S', \sigma' \rangle}$$

If for some function, the uniform-execution heuristic succeeds for every input skeleton then we can extract a compositional input generation procedure from the original function by embedding our type-guided input generation rules. In §2 we already showed the procedure wc_lpairs in Fig. 2 for the function lpairs in Fig. 1. As another example, Fig. 12b is the pseudocode of an input generation procedure extracted from an implementation of quicksort in Fig. 12a, where l_1 ++ l_2 returns the concatenation of two lists l_1, l_2.

6.2 Skeleton Similarity

The uniform-execution heuristic might fail when there does not exist a global configuration of conditional expressions such that on worst-case inputs the function always executes the same branch of a conditional expression. However, intuitively, a function is likely to execute the same execution path on worst-case inputs of the same *shape*. We then develop *skeleton similarity*, a heuristic that reuses the search results for input skeletons of similar shapes.

Formally, we define the *similarity* relation between skeletons in Fig. 13, written $\sigma, \sigma' \vdash_\rho S \sim S'$, where ρ is a mapping between indeterminates. We omit the fixed ρ from these rules. We also write $\vdash_\rho \phi \sim \phi'$ for the similarity of formulas, which is defined in an obvious way. Intuitively, if for a

```
let rec partition a = function
  | [] → ([], [])
  | x :: xs →
    let (cs, bs) = partition a xs in
    if (x:int) ≥ (a:int) then
      (cs, x :: bs)
    else
      (x :: cs, bs)

let rec qsort = function
  | [] → []
  | x :: xs →
    let (ys, zs) = partition x xs in
    let left = qsort ys in
    let right = qsort zs in
    left ++ (x :: right)
```

(a) Original code

```
let rec wc_partition a = function
  | [] → (⊤, ([], []))
  | x :: xs →
    let (ϕ, (cs, bs)) = wc_partition a xs in
    (¬ (x ≥ a) ∧ ϕ, (x :: cs, bs))

let rec wc_qsort = function
  | [] → (⊤, [])
  | x :: xs →
    let (ϕ_p, (ys, zs)) = wc_partition x xs in
    let (ϕ_l, left) = wc_qsort ys in
    let (ϕ_r, right) = wc_qsort zs in
    (ϕ_r ∧ ϕ_l ∧ ϕ_p, left ++ (x :: right))
```

(b) Pseudocode of compositional input generation

Fig. 12. The quicksort example

function call app(f, x), we already established $\Sigma; x^f : A_1; x^f \mapsto S_r; \sigma_r \mid\!\frac{q}{q'}\, e^f : A_2 \Rightarrow \langle \phi_r, S'_r, \sigma'_r \rangle$, and we want to find a worst-case execution path for another input skeleton with the same shape, i.e., S, σ_S such that $\sigma_S \vdash S : A_1$, $\sigma_r, \sigma_S \vdash_\rho S_r \sim S$ for some mapping ρ, then we can use ρ to substitute the boolean and integer indeterminates in S'_r, σ'_r as a candidate generation result, i.e., $\sigma'_r, \sigma'_S \vdash_\rho S'_r \sim S'$. Formally, we introduce the following rule, where $\sigma_1 \otimes \sigma_2$ is the conjunction of two separated skeleton heaps.

(WC-App-Skel-Sim)

$$\frac{\gamma(x) = S \qquad \sigma_S \vdash S : A_1 \qquad \sigma_r, \sigma_S \vdash_\rho S_r \sim S \qquad \sigma'_r, \sigma'_S \vdash_\rho S'_r \sim S' \qquad \vdash_\rho \phi_r \sim \phi}{x : A_1; \gamma; \sigma \otimes \sigma_S \mid\!\frac{q + K^{\text{app}}}{q'}\, \text{app}(f, x) : A_2 \Rightarrow \langle \phi, S', \sigma \otimes \sigma'_S \rangle}$$

Example 6.1. *Recall the program in Fig. 1 which defines the function lpairs. Let* $S_r = \ell_1$, $\sigma_r = \{\ell_1 \mapsto \textsc{cons}(\text{int}^1, \textsc{cons}(\text{int}^2, \textsc{nil}))$ *be a recorded input skeleton. Then a possible generation result is* $\phi_r = (\text{int}^1 < \text{int}^2)$, $S'_r = \ell_3$, $\sigma'_r = \sigma_r[\ell_2 \mapsto \textsc{nil}, \ell_3 \mapsto \textsc{cons}(\langle \text{int}^1, \text{int}^2 \rangle, \ell_2)]$. *Suppose later we encounter a function call with* $S = \ell_4$, $\sigma_S = \{\ell_4 \mapsto \textsc{cons}(\text{int}^3, \textsc{cons}(\text{int}^4, \textsc{nil}))\}$. *Let* $\rho = \{\text{int}^1 \mapsto \text{int}^3, \text{int}^2 \mapsto \text{int}^4\}$, *then we have* $\sigma_r, \sigma_S \vdash S_r \sim S$. *By substitution of integer indeterminates with respect to* ρ, *we derive* $\sigma'_S = \sigma_S[\ell_5 \mapsto \textsc{nil}, \ell_6 \mapsto \textsc{cons}(\langle \text{int}^3, \text{int}^4 \rangle, \ell_5)]$, $S' = \ell_6$, *and* $\phi = (\text{int}^3 < \text{int}^4)$. *In this way, our algorithm proceeds without investigating again the function body.*

Theorem 6.2. *The rule (WC-App-Skel-Sim) is sound.*

$\boxed{\sigma, \sigma' \vdash_\rho S \sim S'}$ Skeleton S under σ is similar to skeleton S' under σ'

$$\frac{}{\sigma, \sigma' \vdash \text{null} \sim \text{null}} \qquad \frac{b \in \{\text{true}, \text{false}\}}{\sigma, \sigma' \vdash b \sim b} \qquad \frac{\rho(\text{bool}^i) = \text{bool}^j}{\sigma, \sigma' \vdash \text{bool}^i \sim \text{bool}^j} \qquad \frac{n \in \mathbb{Z}}{\sigma, \sigma' \vdash n \sim n} \qquad \frac{\rho(\text{int}^i) = \text{int}^j}{\sigma, \sigma' \vdash \text{int}^i \sim \text{int}^j} \qquad \frac{\sigma, \sigma' \vdash \sigma(\ell) \sim \sigma'(\ell')}{\sigma, \sigma' \vdash \ell \sim \ell'}$$

$$\frac{}{\sigma, \sigma' \vdash \textsc{nil} \sim \textsc{nil}} \qquad \frac{\sigma, \sigma' \vdash S_h \sim S'_h \qquad \sigma, \sigma' \vdash S_t \sim S'_t}{\sigma, \sigma' \vdash \textsc{cons}(S_h, S_t) \sim \textsc{cons}(S'_h, S'_t)} \qquad \frac{\forall i \in \{1, \cdots, n\} : \sigma, \sigma' \vdash S_i \sim S'_i}{\sigma, \sigma' \vdash \textsc{listof}(S_1, \cdots, S_n) \sim \textsc{listof}(S'_1, \cdots, S'_n)}$$

$$\frac{}{\sigma, \sigma' \vdash \textsc{leaf} \sim \textsc{leaf}} \qquad \frac{\forall i \in \{0, 1, 2\} : \sigma, \sigma' \vdash S_i \sim S'_i}{\sigma, \sigma' \vdash \textsc{node}(S_0, S_1, S_2) \sim \textsc{node}(S'_0, S'_1, S'_2)} \qquad \frac{\forall i \in \{1, \cdots, n\} : \sigma, \sigma' \vdash S_i \sim S'_i}{\sigma, \sigma' \vdash \textsc{treeof}(S_1, \cdots, S_n) \sim \textsc{treeof}(S_1, \cdots, S_n)}$$

Fig. 13. Skeleton similarity relation

Proof. It suffices to show $x : A_1; \gamma; \sigma_S \left|\frac{q+K^{\text{app}}}{q'}\right. \text{app}(f, x) : A_2 \Rightarrow \langle \phi, S', \sigma'_S \rangle$. The proof proceeds by induction on the derivation of $x^f : A_1; x^f \mapsto S; \sigma_r \left|\frac{q}{q'}\right. e^f : A_2 \Rightarrow \langle \phi_r, S'_r, \sigma'_r \rangle$. □

Operationally, this heuristics can be implemented with a *skeleton cache* cache_f for a function f, such that $\text{cache}_f(S_r, \sigma_r) = (\phi_r, S'_r, \sigma'_r)$. When the input generation algorithm encounters a function call, it first looks up the cache to see if there is a similar input skeleton that has been processed. If there is a cache record then the algorithm tries the recorded path constraint. Otherwise, it proceeds as with the original rules and, after generating a satisfiable path constraint for the function call, records the result into the cache.

7 EVALUATION

In this section, we describe the implementation of our worst-case generation algorithm building on RaML, a summary of an evaluation with 22 benchmark programs, and multiple detailed case studies. The source code of the benchmark programs is included in the extended version of this paper [Wang and Hoffmann 2018].

7.1 Implementation

We integrate our type-guided worst-case input generation algorithm in the existing RaML system [Hoffmann et al. 2017]. The algorithm is implemented in OCaml and consists of about 1600 lines of code. To generate a worst-case input for a top-level function in a source program, the user needs to specify a resource metric, a maximal degree of the resource bounds, and an input skeleton. We then invoke RaML's type inference to derive an upper bound on the resource usage and a resource-annotated type-derivation tree. The input generation rules are implemented as a recursive function on the derivation tree in continuation-passing style. Our implementation resolves nondeterminism in the rules systematically via two continuations, one for generation success and one for generation failure. When a path constraint is generated, we use the off-the-shelf SMT solver Z3 [de Moura and Bjørner 2008] to check its satisfiability and generate models to resolve boolean and integer indeterminates in the input skeleton. If the SMT solver succeeds, we use the generated model to obtain a concrete heap via the relation $M \vdash \sigma \sqsubseteq H$ and concretize the input skeleton via the relation $M; H \vdash S \rightsquigarrow v$. Otherwise, we continue to search for other path constraints.

We have also implemented the two heuristics for compositional worst-case input generation, which can be enabled by the user. The *uniform-execution* heuristic is implemented by enumerating global configurations for all conditional expressions in the given program before the input generation. The *skeleton-similarity* heuristic is implemented by employing a hash table as the generation cache. Instead of the similarity relation, we define signatures for input skeletons such that skeletons of the same signature are similar to each other. Then we use the signature as the hash key in the generation cache. When processing function calls, we extract the signature of the current input skeleton and look it up in the cache. If a recorded generation result does not exist, we use the original rules to generate a worst-case path constraint as well as the corresponding output skeleton, and record them in the cache. Otherwise, we instantiate the recorded constraint and output skeleton for the current input skeleton.

We also apply several simple optimizations. First, we cache the results of potential functions to eliminate redundant computation. Second, we try to simplify the skeletons during the input generation via partial evaluation, in order to deduce the value of predicates in the conditional expressions. Third, we insert satisfiability checking of path constraints during the input generation to get rid of unsatisfiable execution paths as early as possible.

Table 1. Case studies. In the bounds n is the size of the first argument, m_i are the sizes of the elements of the first argument, and x is the size of the second element.

Function	Description	Metric	Inferred Bound	Time
lpairs : $L(\text{int}) \to L(\text{int}^2)$	Example in Fig. 1	Heap space	$3n + 2$	0.01s
lpairs_alt : $L(\text{int}) \to L(\text{int}^2)$	Example in Fig. 3	Heap space	$3n + 2$	0.01s
find : $\text{int} \times L(\text{int}) \to \text{bool}$	Find an element in a list	Eval. steps	$12x + 3$	0.01s
compare : $L(\text{int})^2 \to \text{int}$	Lexicographic comparison	Eval. steps	$20x + 5$	0.01s
opairs : $L(\text{int}) \to L(\text{int}^2)$	Generate ordered pairs	Eval. steps	$26\binom{n}{2} + 17n + 3$	0.02s
queue : $L(\text{bool} \times \text{int}) \to \text{unit}$	Functional queue	Eval. steps	$34.5n + 12$	0.01s
eratos : $L(\text{int}) \to L(\text{int})$	Sieve of Eratosthenes	Eval. steps	$21\binom{n}{2} + 25n + 3$	0.02s
isort : $L(\text{int}) \to L(\text{int})$	Insertion sort	Eval. steps	$20\binom{n}{2} + 15n + 10$	0.02s
qsort : $L(\text{int}) \to L(\text{int})$	Quicksort	Eval. steps	$29\binom{n}{2} + 28n + 10$	0.04s
qsort_pairs : $L(\text{int}^2) \to L(\text{int}^2)$	Tail-recursive quicksort of pairs	Eval. steps	$37\binom{n}{2} + 32n + 13$	0.04s
qsort_lists : $L(L(\text{int})) \to L(L(\text{int}))$	Lexicographic quicksort	Eval. steps	$\sum_{1 \le i < j \le n} 20m_j + 39\binom{n}{2} + 34n + 10$	0.33s
sort_all : $L(L(\text{int})) \to L(L(\text{int}))$	Quicksort all buckets	Eval. steps	$\sum_{1 \le i \le n} (33\binom{m_i}{2} + 34m_i) + 20n + 3$	0.18s
zigzag : $T(\text{unit}) \to \text{unit}$	Zigzag on a tree	Eval. steps	$11n + 3$	0.01s
subtrees : $T(\text{unit}) \to L(T(\text{unit}))$	Collect all subtrees	Eval. steps	$9\binom{n}{2} + 26n + 3$	0.03s
find_tree : $\text{int} \times T(\text{int}) \to \text{bool}$	Find an element in a search tree	Eval. steps	$18n + 3$	0.01s
build_tree : $L(\text{int}) \to T(\text{int})$	Build a search tree by insertion	Eval. steps	$16\binom{n}{2} + 15n + 3$	0.02s
hashtbl : $L(\text{int}^8) \to L(\text{int} \times L(\text{int}^8))$	Create a hash table for 8-char strings	Ticks	$\binom{n}{2}$	0.14s
split_sort : $L(\text{int}^2) \to L(\text{int}^2)$	Group pairs by key and sort each bucket	Ticks	$2\binom{n}{2} + n$	0.14s
kth : $\text{int} \times L(\text{int}) \to \text{int}$	Quickselect	Ticks	$\binom{x}{2}$	0.08s
sum_avl : $T(\text{int}^2) \to \text{int}$	Sum all nodes of an AVL tree	Ticks	n	0.01s
dfs_avl : $T(\text{int}^2) \to L(\text{int})$	Depth-first-search and sort the nodes	Ticks	$\binom{n}{2} + n$	0.06s
bfs_avl : $T(\text{int}^2) \to L(\text{int})$	Breadth-first-search and sort the nodes	Ticks	$\binom{n}{2} + 9n + 4$	0.27s

7.2 Evaluation Setup

Research Questions. We evaluate our algorithm to answer the following questions.

- **RQ1:** Is our algorithm able to generate worst-case inputs for OCaml programs in practice?
- **RQ2:** Is our algorithm scalable to large input skeletons?
- **RQ3:** How does our algorithm compare to existing methods in terms of effectiveness and efficiency?

Evaluated programs. Tab. 1 gives an overview of 22 programs on which we evaluate our algorithm. It lists each case study's function name,[6] description, resource metric, inferred upper bound, and time of type inference in RaML. The functions lpairs and lpairs_alt are the running examples we use in §2. The functions isort, qsort, and hashtbl are similar to the benchmarks used by Noller et al.'s Badger [Noller et al. 2018]. We collect some interesting programs from RaML's examples [Hoffmann et al. 2017]. We also implement new benchmarks such as the functions sum_avl, dfs_avl, bfs_avl that operate on AVL trees. In most of these functions, we specify a standard heap space metric or an evaluation step metric. We also include some case studies where we use a customized metric (that we refer to as "ticks"), for example, for the function hashtbl we specify a metric to count the number of hash collisions.

Experiment Execution. For all functions we ran three variations: (i) ALG: our type-guided worst-case input generation algorithm, (ii) ALG+H1: the algorithm with the *uniform-execution* heuristic enabled, and (ii) ALG+H2: the algorithm with the *skeleton-similarity* heuristic enabled. For each function, we evaluated all these algorithms on four input skeletons of different sizes. We ran our experiments fo 5 times with a 15-minute timeout and computed the 20% trimmed mean of the running time. Tab. 2 presents the statistics of running time of all the experiments.

[6] Although our implementation takes a top-level function as its input, the program can contain auxiliary functions that could be invoked by the analyzed function.

Table 2. Running time statistics (in seconds). "T/O" stands for timeout.

Function	ALG	ALG+H1	ALG+H2	ALG	ALG+H1	ALG+H2	ALG	ALG+H1	ALG+H2	ALG	ALG+H1	ALG+H2
lpairs	$n = 10$			$n = 50$			$n = 100$			$n = 200$		
	0.01	0.01	0.06	0.01	0.01	0.26	0.02	0.02	0.57	0.03	0.03	1.15
lpairs_alt	$n = 10$			$n = 30$			$n = 100$			$n = 200$		
	0.11	0.79	0.08	321.83	T/O	0.25	T/O	T/O	0.84	T/O	T/O	1.73
find	$n = 10$			$n = 50$			$n = 100$			$n = 200$		
	0.01	0.01	0.11	0.01	0.01	0.55	0.02	0.02	1.11	0.03	0.03	2.34
compare	$n = 10, x = 10$			$n = 50, x = 50$			$n = 100, x = 100$			$n = 200, x = 200$		
	0.01	0.01	0.12	0.02	0.02	0.64	0.03	0.03	1.31	0.07	0.07	2.91
opairs	$n = 10$			$n = 50$			$n = 100$			$n = 200$		
	0.03	0.03	0.14	1.52	1.52	2.41	20.70	20.71	25.24	353.85	354.55	389.12
queue	$n = 10$			$n = 50$			$n = 100$			$n = 200$		
	0.04	0.09	0.15	3.54	35.33	7.77	36.90	709.71	109.35	444.64	T/O	T/O
eratos	$n = 10$			$n = 14$			$n = 18$			$n = 20$		
	2.19	2.19	12.62	2.70	2.70	19.75	4.20	4.19	35.77	T/O	T/O	T/O
isort	$n = 10$			$n = 50$			$n = 100$			$n = 200$		
	0.02	0.02	0.14	0.29	0.26	1.24	1.33	1.20	7.07	7.74	6.97	94.81
qsort	$n = 10$			$n = 64$			$n = 100$			$n = 200$		
	1.38	0.07	0.19	T/O	2.99	4.84	T/O	8.67	15.34	T/O	53.23	157.21
qsort_pairs	$n = 10$			$n = 50$			$n = 100$			$n = 200$		
	0.03	0.03	0.25	0.51	0.50	2.07	2.56	2.50	9.35	14.96	14.79	71.68
qsort_lists	$n = 10, m_i = n - i + 1$			$n = 50, m_i = n - i + 1$			$n = 75, m_i = n - i + 1$			$n = 100, m_i = n - i + 1$		
	0.19	0.19	0.33	16.83	16.80	33.87	113.47	113.47	662.13	439.35	438.79	T/O
sort_all	$n = 10, m_i = 10$			$n = 50, m_i = 10$			$n = 100, m_i = 10$			$n = 200, m_i = 10$		
	T/O	0.32	0.67	T/O	1.46	0.73	T/O	2.95	0.89	T/O	6.52	1.66
zigzag	$n = 10$			$n = 15$			$n = 100$			$n = 200$		
	3.47	6.96	0.16	110.35	222.40	0.25	T/O	T/O	1.74	T/O	T/O	4.87
subtrees	$n = 10$			$\mathbf{n = 13}$			$n = 100$			$n = 200$		
	0.23	0.23	0.12	1.75	1.76	0.16	T/O	T/O	8.79	T/O	T/O	112.35
find_tree	$n = 10$			$n = 50$			$n = 100$			$n = 200$		
	0.01	0.01	0.11	0.02	0.02	0.63	0.03	0.03	1.26	0.06	0.06	2.78
build_tree	$n = 10$			$\mathbf{n = 50}$			$n = 100$			$n = 200$		
	0.02	0.02	0.23	0.32	0.32	1.48	1.55	1.53	6.69	9.22	9.16	88.56
hashtbl	$n = 5$			$n = 10$			$n = 30$			$n = 64$		
	0.50	0.49	0.68	2.16	2.16	16.30	3.07	3.08	60.14	7.64	7.62	181.74
split_sort	$n = 10$			$n = 50$			$n = 100$			$n = 200$		
	703.22	0.12	1.99	T/O	3.02	T/O	T/O	14.60	T/O	T/O	85.70	T/O
kth	$n = 10$			$n = 50$			$n = 100$			$n = 200$		
	0.03	0.03	0.11	0.35	0.35	1.24	1.60	1.57	6.02	8.78	8.67	54.36
sum_avl	$n = 5$			$n = 10$			$n = 30$			$n = 50$		
	0.18	0.18	0.17	70.06	70.13	0.93	T/O	T/O	33.39	T/O	T/O	240.88
dfs_avl	$n = 5$			$n = 8$			$n = 30$			$n = 40$		
	2.72	72.09	0.37	805.29	T/O	1.46	T/O	T/O	260.55	T/O	T/O	T/O
bfs_avl	$n = 5$			$n = 8$			$n = 12$			$n = 14$		
	4.77	136.14	1.60	T/O	T/O	16.54	T/O	T/O	492.49	T/O	T/O	T/O

Evaluation Platform. Our experiments were performed on a machine with an Intel Core i7 3.6 GHz processor and 16GB of RAM under macOS High Sierra 10.13.5.

7.3 Case Studies

For every function in Tab. 1, our type-guided worst-case input generation algorithm is able to find worst-case inputs for some input skeletons of 5–200 nodes. This suggests that the inferred bounds by RaML are tight for all these functions. We present a detailed description of the experiments for several functions below.

Example 1: Quicksort of Integers. We use a mutually recursive implementation of the quicksort algorithm in [Xi 2002]. This implementation is interesting because the worst-case inputs are not

reversely ordered lists as usual. Although ALG runs out of time for input lists of length 64, 100, and 200, both ALG+H1 and ALG+H2 are able to generate a worst-case input for each of these lengths in 3 minutes. Intuitively, the reason why ALG fails is that the number of candidate execution paths is $O(2^n)$ where n is the length of the input list. For example, for the input list of length 10, ALG generates the worst-case input $[0, -2, -4, -6, -8, -9, -7, -5 - 3, -1]$.

Example 2: Sequential Insertions in a Hash Table. We implement an OCaml program that models the hash table function from Badger [Noller et al. 2018]. We insert an expression tick(1.0) when a hash collision happens. By specifying the number of ticks as the resource metric, RaML derives an upper bound $\binom{n}{2}$ on the number of collisions, where n is the number of insertions. In this function, each key in the hash table has a length of 8 characters, and we model it as a tuple type (abbreviated as int^8). The hash values are in the range $[0, 64)$. We implement the DJBX33A hash function used in a vulnerable PHP implementation [Website 2011]. The program performs 64 insertions into an empty hash table and we want to generate an insertion sequence to trigger the worst-case number of hash collisions. ALG and ALG+H1 are able to generate a list of 64 strings of length 8 in 20 seconds that cause the greatest number of hash collisions, i.e., all keys are different from each other but have the same hash value, hence this insertion sequence triggers $\binom{64}{2}$ hash collisions, while ALG+H2 takes a longer time. We think the reason why ALG+H2 runs much slower is that the typing information is able to prune a sufficiently large part of the search space so the overheads of caching dominate the running time.

Example 3: Lexicographic Quicksort of Lists of Lists. This function is from RaML's standard benchmark set. It implements a standard quicksort that lexicographically sorts lists of lists. To lexicographically compare two lists, one needs linear time in the length of the shorter list. For the worst-case input generation, we specify input skeletons such that the lengths of inner lists are strictly decreasing. ALG and ALG+H1 succeed in generating worst-case inputs for input lists of length 100, while ALG+H3 runs out of time. The worst-case inputs they generate set all integers in the inner lists to zero. However, if the inner lists of the input skeleton are not reversely ordered by length, these algorithms report a generation failure. It suggests that the inferred bound by RaML is not tight for these input skeletons. We think it is because currently, RaML does not support the min operator in the resource polynomials, and in this example, it always assigns potential to the second argument of a list comparison, hence when the first list has a shorter length, there exists potential waste.

Example 4: Zigzagging on a Binary Tree. We implement a tree traversal that visits the left and right child alternatively. For a fixed size, the worst-case tree should arrange all its nodes on a "zigzag" path so that the traversal needs to visit all its nodes. ALG and ALG+H1 become inefficient when the size of the tree is 15, while ALG+H2 can easily generate a worst-case input for a tree of size 200, because a subtree of a zigzagging tree is indeed zigzagging.

Example 5: Summing up nodes of an AVL Tree. We implement another tree traversal that simply sums up the values of all nodes but expresses some constraints on the tree structure. Basically, we record a height in each node and then we require the height of a node should be one plus the maximum of the heights of its children and the difference of heights of its left child and its right child should not exceed one. This corresponds to AVL trees that are well-known balanced search trees. The worst-case input generation algorithm is then able to generate valid AVL trees for a given size. Like the last example, ALG and ALG+H1 time out on small input skeletons, but ALG+H2 is able to scale to large input skeletons. The reason is that every subtree of an AVL tree is an AVL tree.

Discussion.

- **RQ1**: Our evaluation shows that our type-guided worst-case input generation algorithm is able to handle a broad suite of OCaml programs, on condition that RaML infers tight bounds on the programs. Moreover, as we discussed earlier in the paper, our algorithm is easy to modify to handle d-bounded worst-case inputs, so if the RaML-inferred bound is not tight but only differs from the original bound by a constant, our algorithm should also work.
- **RQ2**: Our evaluation shows that, in general, the time complexity of our input generation algorithm is exponential in the size of the input skeleton. Nevertheless, the two heuristics, *uniform-execution* and *skeleton-similarity*, can be helpful in practice. For example, if the worst-case input data structure satisfies some inductive properties, e.g., it is a zigzagging tree or an AVL tree, then the *skeleton-similarity* heuristic can scale to large input skeletons.
- **RQ3**: Although we do not perform a systematic comparison to existing techniques, we argue that we make significant progress on some benchmark functions. For the quicksort and hash table examples, Noller et al. evaluated Badger [Noller et al. 2018] on Java implementations for 5 hours, but did not generate an input that exposes worst-case resource consumption among all possible inputs, e.g., on the hash table example, Badger produced an insertion sequence with half of the worst-case number of hash collisions. Moreover, they only ran their tool to generate inputs of size smaller or equal to 64 for sorting algorithms and hash tables. In contrast, we ran our tool on several benchmarks including sorting algorithms with input size up to 200.

8 RELATED WORK

Input Generation. Most closely related to our work are techniques for generating worst-case inputs based on symbolic execution. WISE [Burnim et al. 2009] exhaustively explores all program paths for small inputs to find worst-case paths. These paths are then used as a heuristic to limit the search space for inputs of larger sizes. Similarly, SPF-WCA [Luckow et al. 2017] uses path policies to prune parts of the search space during symbolic execution. It also takes into account calling contexts and "execution histories" to guide the search. Badger [Noller et al. 2018] combines symbolic execution with fuzz testing for generating resource intensive inputs to entirely avoid exhaustive exploration. There are also pure fuzzers like SlowFuzz [Petsios et al. 2017] that aim at generating inputs that cause programs to have high resource consumption. The main difference in our work is that we use RaML's type derivations to prune the search space. Advantages of this approach are that it is more efficient, guarantees that the generated inputs are indeed witnesses for the worst-case behavior, and, as a side effect, proves that the bounds derived by RaML are tight. A disadvantage is that the technique is only applicable to programs for which RaML derives a bound.

There are tools for random testing such as QuickCheck [Claessen and Hughes 2000], Smallcheck [Runciman et al. 2008], and QuickChick [Lampropoulos et al. 2018] that use type information and additional properties to generate random tests. However, we are not aware that these tools have been used to generate worst-case inputs or tests for exposing high resource usage.

Resource Analysis. Automatic resource bound analysis has been extensively studied.

AARA has been introduced [Hofmann and Jost 2003] for automatically deriving linear worst-case bounds for first-order functional programs. The technique has been generalized to derive polynomial bounds [Hoffmann et al. 2011; Hoffmann and Hofmann 2010; Hofmann and Moser 2015], lower bounds [Ngo et al. 2017], higher-order functions [Hoffmann et al. 2017; Jost et al. 2010], lazy functional programs [Simões et al. 2012; Vasconcelos et al. 2015], user defined data types [Hoffmann et al. 2017; Jost et al. 2009], and numeric imperative program [Carbonneaux et al. 2017, 2015]. It

also has been integrated into separation logic [Atkey 2010] and proof assistants [Charguéraud and Pottier 2015; Nipkow 2015].

Beyond AARA, there exist many other approaches to automatic worst-case resource bound analysis. They are based on sized types [Vasconcelos 2008], linear dependent types [Lago and Gaboardi 2011; Lago and Petit 2013], refinement types [Çiçek et al. 2017, 2015; Wang et al. 2017], annotated type systems [Crary and Weirich 2000; Danielsson 2008], defunctionalization [Avanzini et al. 2015], recurrence relations [Albert et al. 2015; Danner et al. 2015; Flores-Montoya and Hähnle 2014; Kincaid et al. 2017], abstract interpretation [Blanc et al. 2010; Gulwani et al. 2009; Sinn et al. 2014; Zuleger et al. 2011], and techniques from term rewriting [Avanzini and Moser 2013; Brockschmidt et al. 2014; Frohn et al. 2016; Noschinski et al. 2013].

In contrast to all the aforementioned works, we study the problem of automatically deriving worst-case inputs. These inputs are also witnesses for the tightness of the derived bounds. We are not aware of existing works that leverage automatically-derived bounds to compute worst-case inputs.

Symbolic Execution. A lot of techniques have been developed to improve effectiveness and efficiency of symbolic execution in practice. Dynamic symbolic execution [Godefroid et al. 2005; Sen et al. 2005] uses a specific concrete execution to drive the symbolic execution in the sense that the concrete execution provides resolution of branches in the program. Selective symbolic execution [Chipounov et al. 2012] interleaves concrete and symbolic executions in order to explore only some components of a program. Symbolic backward execution [Chandra et al. 2009; Dinges and Agha 2014] performs in the reverse direction of normal execution to identify an input instance to satisfy a given post-condition. Different path selection strategies are proposed for different analysis goals [Cadar et al. 2008; Ma et al. 2011; Zhang et al. 2015]. Our worst-case input generation algorithm essentially performs symbolic execution with a depth-first path selection strategy, but utilizes typing derivations to prune the search space as well as guide the search.

9 CONCLUSION

We have presented a type-guided worst-case input generation algorithm for functional programs that is based on automatic amortized resource analysis. We have proved of soundness and relative completeness of our algorithm and developed sound heuristics to find worst-case inputs more efficiently. Finally, an implementation of our algorithm has been integrated with RaML and evaluated with benchmark programs.

In the future, we plan to add support for negative resources to generate inputs that trigger worst-case high-water marks. We will also work on mechanisms that use the absence of worst-case inputs to improve the precision of resource-bound analyses. Another research direction is to support side effects and more complex resource bounds such as those involving heights of trees. We are also looking into symbolic execution techniques that can further improve the scalability of the worst-case input generation algorithm.

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 and SHF Award 1812876. 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

E. Albert, P. Arenas, S. Genaim, and G. Puebla. 2011. Closed-Form Upper Bounds in Static Cost Analysis. *J. Automated Reasoning* 46 (February 2011). Issue 2.

E. Albert, J. C. Fernández, and G. Román-Díez. 2015. Non-cumulative Resource Analysis. In *Tools and Algs. for the Construct. and Anal. of Syst. (TACAS'15)*.

R. Atkey. 2010. Amortised Resource Analysis with Separation Logic. In *European Symp. on Programming (ESOP'10)*.

M. Avanzini, U. D. Lago, and G. Moser. 2015. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *Int. Conf. on Functional Programming (ICFP'15)*.

M. Avanzini and G. Moser. 2013. A Combination Framework for Complexity. In *Int. Conf. on Rewriting Techniques and Applications (RTA'13)*.

R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. 2010. ABC: Algebraic Bound Computation for Loops. In *Logic for Prog., AI., and Reasoning (LPAR'10)*.

M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. 2014. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Algs. for the Construct. and Anal. of Syst. (TACAS'14)*.

J. Burnim, S. Juvekar, and K. Sen. 2009. WISE: Automated Test Generation for Worst-case Complexity. In *Int. Conf. on Softw. Eng. (ICSE'09)*.

C. Cadar, D. Dunbar, and D. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Op. Syst. Design and Impl. (OSDI'08)*.

Q. Carbonneaux, J. Hoffmann, T. Reps, and Z. Shao. 2017. Automated Resource Analysis with Coq Proof Objects. In *Computer Aided Verif. (CAV'17)*.

Q. Carbonneaux, J. Hoffmann, and Z. Shao. 2015. Compositional Certified Resource Bounds. In *Prog. Lang. Design and Impl. (PLDI'15)*.

S. Chandra, S. J. Fink, and M. Sridharan. 2009. Snugglebug: A Powerful Approach To Weakest Preconditions. In *Prog. Lang. Design and Impl. (PLDI'09)*.

A. Charguéraud and F. Pottier. 2015. Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In *Interactive Theorem Proving (ITP'15)*.

V. Chipounov, V. Kuznetsov, and G. Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *Trans. on Comp. Syst.* 30 (February 2012). Issue 1.

E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann. 2017. Relational Cost Analysis. In *Princ. of Prog. Lang. (POPL'17)*.

E. Çiçek, D. Garg, and U. A. Acar. 2015. Refinement Types for Incremental Computational Complexity. In *European Symp. on Programming (ESOP'15)*.

K. Claessen and J. Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Int. Conf. on Functional Programming (ICFP'00)*.

K. Crary and S. Weirich. 2000. Resource Bound Certification. In *Princ. of Prog. Lang. (POPL'00)*.

S. A. Crosby and D. S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Sec. Symp. (USENIX'03)*.

N. A. Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Princ. of Prog. Lang. (POPL'08)*.

N. Danner, D. R. Licata, and R. Ramyaa. 2015. Denotational Cost Semantics for Functional Languages with Inductive Types. In *Int. Conf. on Functional Programming (ICFP'15)*.

L. de Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algs. for the Construct. and Anal. of Syst. (TACAS'08)*.

P. Dinges and G. Agha. 2014. Targeted Test Input Generation Using Symbolic–Concrete Backward Execution. In *Automated Softw. Eng. (ASE'14)*.

A. Flores-Montoya and R. Hähnle. 2014. Resource Analysis of Complex Programs with Cost Equations. In *Asian Symp. on Prog. Lang. and Systems (APLAS'14)*.

J. E. Forrester and B. P. Miller. 2000. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *USENIX Windows Syst. Symp. (WSS'00)*.

F. Frohn, M. Naaf, J. Hensel, M. Brockschmidt, and J. Giesl. 2016. Lower Runtime Bounds for Integer Programs. In *Int. Joint Conf. on Automated Reasoning (IJCAR'16)*.

P. Godefroid, N. Klarlund, and K. Sen. 2005. DART: Directed Automated Random Testing. In *Prog. Lang. Design and Impl. (PLDI'05)*.

P. Godefroid, M. Levin, and D. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Network and Dist. Syst. Security (NDSS'08)*.

S. Gulwani. 2009. SPEED: Symbolic Complexity Bound Analysis. In *Computer Aided Verif. (CAV'09)*.

S. Gulwani, K. K. Mehra, and T. M. Chilimbi. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Princ. of Prog. Lang. (POPL'09)*.

R. Harper. 2016. *Practical Foundations for Programming Languages.* Cambridge University Press.

J. Hoffmann, K. Aehlig, and M. Hofmann. 2011. Multivariate Amortized Resource Analysis. In *Princ. of Prog. Lang. (POPL'11)*.
J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *Princ. of Prog. Lang. (POPL'17)*.
J. Hoffmann and M. Hofmann. 2010. Amortized Resource Analysis with Polynomial Potential. In *European Symp. on Programming (ESOP'10)*.
M. Hofmann and S. Jost. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Princ. of Prog. Lang. (POPL'03)*.
M. Hofmann and G. Moser. 2015. Multivariate Amortised Resource Analysis for Term Rewrite Systems. In *Int. Conf. on Typed Lambda Calculi and Applications (TLCA'15)*.
S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. 2010. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *Princ. of Prog. Lang. (POPL'10)*.
S. Jost, H.-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. 2009. Carbon Credits for Resource-Bounded Computations using Amortised Analysis. In *Symp. on Form. Meth. (FM'09)*.
Z. Kincaid, J. Breck, A. F. Boroujeni, and T. Reps. 2017. Compositional Recurrence Analysis Revisited. In *Prog. Lang. Design and Impl. (PLDI'17)*.
U. D. Lago and M. Gaboardi. 2011. Linear Dependent Types and Relative Completeness. In *Logic in Computer Science (LICS'11)*.
U. D. Lago and B. Petit. 2013. The Geometry of Types. In *Princ. of Prog. Lang. (POPL'13)*.
L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. 2018. Generating Good Generators for Inductive Relations. In *Princ. of Prog. Lang. (POPL'18)*.
K. Luckow, R. Kersten, and C. Păsăreanu. 2017. Symbolic Complexity Analysis Using Context-Preserving Histories. In *Int. Conf. on Softw. Testing, Verif. and Validation (ICST'17)*.
K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. 2011. Directed symbolic execution. In *Static Analysis Symp. (SAS'11)*.
M. D. McIlroy. 1999. A Killer Adversary for Quicksort. *J. Softw.-Practice & Experience* 29 (April 1999). Issue 4.
V. C. Ngo, Mario Dehesa-Azuara, M. Fredrikson, and J. Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *Symp. on Sec. and Privacy (SP'17)*.
T. Nipkow. 2015. Amortized Complexity Verified. In *Interactive Theorem Proving (ITP'15)*.
Y. Noller, R. Kersten, and C. S. Păsăreanu. 2018. Badger: Complexity Analysis with Fuzzing and Symbolic Execution. In *Int. Symp. on Softw. Testing and Analysis (ISSTA'18)*.
L. Noschinski, F. Emmes, and J. Giesl. 2013. Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs. *J. Automated Reasoning* 51 (June 2013). Issue 1.
T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Conf. on Comp. and Comm. Sec. (CCS'17)*.
C. Runciman, M. Naylor, and F. Lindblad. 2008. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Vaues. In *Symp. on Haskell (Haskell'08)*.
K. Sen, D. Marinov, and G. Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Found. of Softw. Eng. (FSE'05)*.
H. R. Simões, P. B. Vasconcelos, M. Florido, S. Jost, and K. Hammond. 2012. Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In *Int. Conf. on Functional Programming (ICFP'12)*.
M. Sinn, F. Zuleger, and H. Veith. 2014. A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verif. (CAV'14)*.
R. E. Tarjan. 1985. Amortized Computational Complexity. *SIAM J. Algebraic Discrete Methods* 6 (August 1985). Issue 2.
P. B. Vasconcelos. 2008. *Space Cost Analysis Using Sized Types*. Ph.D. Dissertation. School of Computer Science, University of St Andrews.
P. B. Vasconcelos, S. Jost, M. Florido, and K. Hammond. 2015. Type-Based Allocation Analysis for Co-recursion in Lazy Functional Languages. In *European Symp. on Programming (ESOP'15)*.
D. Walker. 2002. Substructural Type Systems. In *Advanced Topics in Types and Programming Languages*. MIT Press.
D. Wang and J. Hoffmann. 2018. Type-Guided Worst-Case Input Generation. Available on https://www.cs.cmu.edu/~diw3/papers/WangH18.pdf.
P. Wang, D. Wang, and A. Chlipala. 2017. TiML: A Functional Language for Practical Complexity Analysis with Invariants. In *Object-Oriented Prog., Syst., Lang., and Applications (OOPSLA'17)*.
Website. 2011. CVE - CVE-2011-4885. Available on: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4885.
Website. 2012a. PHP 5.3.8 - Hashtables Denial of Service. Available on https://www.exploit-db.com/exploits/18296/.
Website. 2012b. PHP: PHP 5 ChangeLog. Available on http://www.php.net/ChangeLog-5.php#5.3.9.
Website. 2015. Space/Time Analysis for Cybersecurity (STAC). Available on https://www.darpa.mil/program/space-time-analysis-for-cybersecurity.
H. Xi. 2002. Dependent Types for Program Termination Verification. *J. Higher-Order and Symbolic Comp.* 15 (2002). Issue 1.

Y. Zhang, Z. Chen, J. Wang, W. Dong, and Z. Liu. 2015. Regular Property Guided Dynamic Symbolic Execution. In *Int. Conf. on Softw. Eng. (ICSE'15)*.

F. Zuleger, M. Sinn, S. Gulwani, and H. Veith. 2011. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *Static Analysis Symp. (SAS'11)*.