Implementing Hybrid Resource Analysis in Resource Aware ML 2

Arnav Sabharwal

May 2026

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

COMMITTEE

Jan Hoffmann

Submitted in partial fulfillment of the requirements
for the Senior Honors Thesis

Keywords: resource analysis, static analysis, data-driven analysis, Bayesian inference, Automatic Amortized Resource Analysis, Resource Aware ML, recursion depth, ghost variable, monad.

Abstract

Resource analysis refers to the task of deriving bounds on the worst-case resource use (eg. time, space, stack depth) of a program. There are two main classes of approaches to automatic resource analysis: *static* and *data-driven*. Static resource analysis considers all possible behaviors of the program to derive a *sound* worst-case cost bound i.e. the derived bound is a true upper bound on the program's cost over all inputs. However, due to the undecidability of sound automated worst-case cost analysis, there will always be programs whose cost cannot be bounded statically. Dually, data-driven analysis statistically infers a bound on the program's worst-case cost by running it on a finite set of inputs. Such methods are able to derive bounds for all programs, albeit at the expense of soundness. That is, the derived bound may not upper bound the program's cost over all inputs, since the set of inputs the program was run on may not contain the inputs that induce worst-case behavior.

Hybrid resource analysis is an alternative approach which combines the complementary strengths and weaknesses of static and data-driven worst-case resource analysis. At a high level, such methods allow the user to decide which fragments of the program are to be analyzed by static or data-driven analysis, after which the bounds derived by the two analyses are composed in a principled manner. *Resource decomposition* is a hybrid resource analysis technique which is able to derive tight, yet asymptotically sound cost bounds on functional programs which cannot be analyzed tightly (or at all) purely by static analysis. In this thesis, we describe the process of making resource decomposition an *automatic* hybrid resource analysis. We implement resource decomposition in Resource Aware ML 2 (RaML 2), an in-development automatic resource analysis tool for Standard ML programs. Finally, we empirically demonstrate how adding resource decomposition to RaML 2 allows for deriving asymptotically sound and tight cost bounds on common functional programs.

Acknowledgements

I would like to thank my advisor, Jan Hoffmann, and his students Ethan Chu and Long Pham (former) for their guidance and support.

Contents

1 Introduction

We introduce the resource analysis problem in Section 1.1, discussing static and data-driven techniques for resource analysis. In Sections 1.2 and 1.3, we specify the static and data-driven analysis we shall be considering throughout the thesis.

1.1 Automated resource analysis

Given a program P, the goal of resource analysis is to automatically derive a bound $f(x)$ on the worst-case resource use (eg. time, space, stack depth) of P in terms of its input x. We say that $f(x)$ is *sound* if for all possible arguments x, $f(x)$ is a valid upper bound on the worst-case cost of $P(x)$. This means that bounds derived by resource analysis are more informative than asymptotic bounds, which ignore constant factors and are only applicable for "sufficiently large" inputs.

There are two main classes of resource analysis techniques, with complementary strengths and weaknesses. Firstly, we have *static resource analyses*, which analyze the program's source code to reason about all possible behaviors. Their strength is that the bounds they derive are necessarily sound. However, due to the undecidability of resource analysis for Turing-complete languages, they are *incomplete*: there exist programs which cannot be analyzed tightly (or at all) by static resource analyses.

Conversely, we have *data-driven analyses*, which proceed by logging the program's worst-case cost when run on a finite set of inputs chosen according to some distribution. Then, a cost bound is statistically inferred using the collected data. Data-driven analyses are able to derive bounds on programs which cannot be analyzed statically. However, they are *unsound*: since the set of program inputs may not induce worst-case behavior, there is no guarantee that statistically inferred bounds are sound.

1.2 Automatic Amortized Resource Analysis

Automatic Amortized Resource Analysis (AARA) [4, 5, 3, 6] is a technique for statically deriving *sound, worst-case, polynomial* cost bounds on functional programs. Here, the notion of cost is specified by the user by annotating the program with the $\text{tick}(q), q \in \mathbb{Q}$ effect. When $q > 0$, the resource being specified is consumed, and when $q < 0$, it is freed up. For example, in Listing 1, we perform a $\text{tick}(1.0)$ whenever entering a function's body to specify the number of function calls as our notion of cost. The ability to free up resources means AARA is able to work with *non-monotone* cost metrics (eg. memory allocations, recursion depth, etc.). AARA has been implemented for OCaml as Resource Aware ML (RaML) [13], and is currently being improved and implemented for Standard ML (SML) as RaML 2 [2].

Listing 1: Mergesort with tick used to specify number of function calls as notion of cost

```
fun split l = tick 1.0; ...
fun merge l1 l2 = tick 1.0; ...
fun sort l =
  tick 1.0;
  case l of [] | [_] => l
  | _ :: _ :: _ =>
    let l1, l2 = split l in
    merge (sort l1) (sort l2)
```

As a static tool, AARA's strength lies in its soundness. Taking quicksort as an example, AARA is able to infer a quadratic worst-case bound of $1 + 2.5n + 0.5n^2$ on the number of function calls, where n is the length

of the input list. A purely data-driven analysis derives an asymptotically unsound sub-quadratic bound, since inputs that induce worst-case behavior are rare. In general, AARA yields tight cost bounds when the recursion scheme that induces worst-case behavior is structural.

However, as a manifestation of its incompleteness, AARA is currently unable to statically infer logarithmic cost bounds, which poses problems for divide-and-conquer style programs. Taking mergesort as an example, AARA infers an asymptotically loose $1 - 1.5n + 2.5n^2$ bound on number of function calls, due to its inability to tightly bound mergesort's recursion depth. In fact, for automated cost analysis tools that commit to soundness, correctly inferring logarithmic worst-case cost bounds remains an open problem.

1.3 Bayesian Inference

Bayesian inference is a data-driven analysis to derive (not necessarily sound) cost bounds on a program P. We shall explain it at a high level, a more detailed explanation can be found in [12]:

1. We define two random variables: y, the observed variable, represents a dataset of cost measurements and θ, the latent variable, represents the coefficients of a polynomial cost bound. We define the likelihood of observing any given dataset D given the bound θ as $\pi(y = D|\theta)$, and a prior distribution for the bounds $\pi(\theta)$.

2. Then, we collect a dataset $\mathcal{D}$ by running $P(x)$ on a randomly generated set of inputs.

3. We generate a sufficiently large set of samples $\theta_1, \theta_2, \ldots, \theta_M$ distributed according to the posterior distribution $\pi(\theta|y = \mathcal{D})$ through Markov Chain Monte Carlo (MCMC) methods. Concretely, this is done by defining our probabilistic model in STAN [1], a probabilistic programming language with support for MCMC.

4. We may then pick the most conservative sample from $\theta_1, \theta_2, \ldots, \theta_M$ as the coefficients for our final data-driven bound.

Hybrid resource analyses integrate the completementary strengths and weaknesses of static analyses like AARA and data-driven analyses like Bayesian inference. Informally, they allow the user to "delegate" whatever cannot be analyzed by purely static analysis to data driven analysis. Then, the static and data-driven bounds are composed in a principled manner to yield a final bound which (i) is closer to the ground truth than analyzing the entire program using a data-driven analysis and (ii) enables resource analysis for programs which cannot be analyzed tightly (or at all) by purely static methods.

1.4 Resource Decomposition

Resource decomposition [11] is a hybrid resource analysis combining AARA and Bayesian inference to derive tight bounds on purely functional programs. Given a program $P(x)$, it works as follows:

1. The user annotates $P(x)$ with a new cost annotation primitive $\text{mark}_l(z)$ ($z \in \mathbb{Z}$) used identically as $\text{tick}(q)$ to specify a quantitative semantic property ρ that cannot be tightly bound by AARA (eg. the recursion depth of a helper function within P). The semantic property tracked is called a *resource component*, and the annotated program is termed the *resource decomposed* program. Note that recursion depth is a *non-monotone* resource component, reflected in the "freeing up" of the resource through $\text{mark}_{\text{rd}}(-1)$ whenever exiting a function body.

Listing 2: Resource decomposed mergesort, resource component specified is the recursion depth of sort

```
fun sort l =
  tick 1.0; mark_rd(1);
  let val res =
    case l of [] | [_] => l
    | _ :: _ :: _ =>
      let l1, l2 = split l in
      merge (sort l1) (sort l2)
  in mark_rd(−1); res
```

2. Bayesian inference is used to derive a bound $g(x)$ on the peak value of ρ. In the case of Listing 2, we obtain a concrete logarithmic bound on the peak value of the resource component *rd* tracking the recursion depth of *sort*. Meanwhile, the resource decomposed program $P(x)$ is transformed to be parameterized over an additional variable r, which tracks the resource component ρ specified by the user. The resultant program $P(x, r)$ is termed the *resource guarded program*.

Listing 3: Resource guarded version of mergesort

```
fun sort l rd =
  case rd of
  Zero => raise exception
  | Succ rd' =>
    tick 1.0;
    case l of [] | [_] => l
    | _ :: _ :: _ =>
      let l1, l2 = split l in
      let l1_s = sort l1 rd' in
      let l2_s = sort l2 rd' in
      merge l1_s l2_s
```

For example, in Listing 3, sort l is given an additional ghost argument [7] *rd*, denoting the maximum recursion depth of *sort* for sorting any list of length $|l|$. Note that *rd* being 0 would imply a contradictory state of affairs, since no list can be sorted by *sort* with 0 recursion depth (hence the exception). The rest of the definition is forced to maintain the meaning of the ghost argument. Note also how the changes to *rd* are dictated exactly by the user's $\text{mark}_{\text{rd}}(-)$ annotations.

3. Finally, AARA is used to derive a sound worst-case bound $f(x, r)$ on the resource guarded program. Composing this with our data-driven bound, we report $f(x, g(x))$ as the final cost bound for $P(x)$. For example, AARA infers a bound of $1 + 3.5 * |l| *$ rd on Listing 3. If data-driven analysis reports that rd $\leq \log_2(|l|) + 2$, then the final bound on the number of function calls made by mergesort is $1 + 3.5 * |l| * (2 + \log_2 |l|)$.

The inability to tightly bound the cost of logarithmic recursion depth functions causes AARA to fail to yield useful bounds on many common functional programs. Thus, the artifact [10] associated with [11] gave users the ability to treat the worst-case recursion depth of any function in their program (OCaml) as a resource component. This allowed users to infer tight, and asymptotically sound cost bounds on commonly used programs like: mergesort (by treating its recursion depth as a resource component), AVL and Red-Black trees (by treating the recursion depth of tree insertion and lookups as resource components), etc.

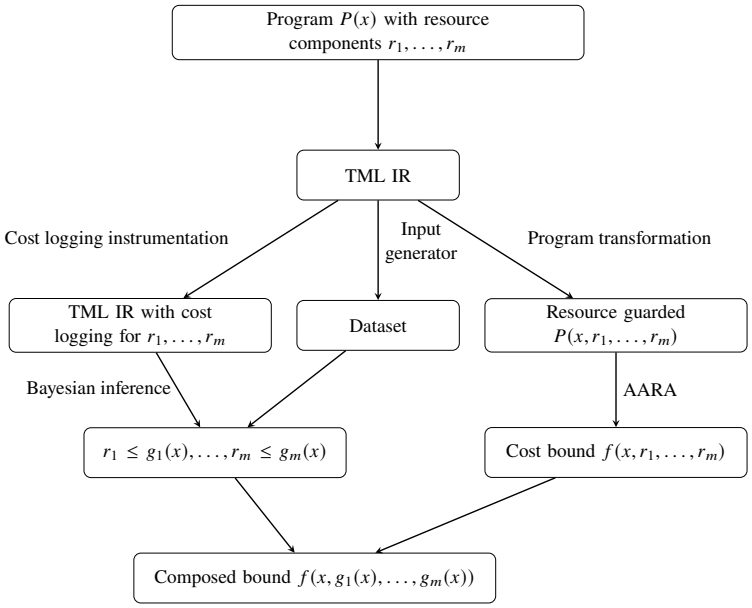

Figure 1: Architecture for the inclusion of resource decomposition in RaML 2

2 Structure of the thesis

In this thesis, we shall discuss the automation of resource decomposition and its implementation in RaML 2. Presently, we only allow users to treat the peak recursion depth of functions as resource components, which turns out to be sufficient for deriving tight bounds on various programs of interest. We provide a bird's eye view of the architecture in Figure 1, and organize resource decomposition as follows:

1. We provide source-level primitives for defining resource components $r_1, \ldots, r_m$ in the resource decomposed program $P(x)$. Here, $r_1, \ldots, r_m$ refer to the peak recursion depths of functions $f_1, \ldots, f_m$ defined in $P(x)$. Then, we transform $P(x)$ into a simplified intermediate representation called the TML IR henceforth. We define the user interface and TML in Section 3.

2. We then describe an automatic program transformation converting a resource decomposed program like Listing 2 to a resource guarded one like Listing 3, in Section 4. AARA analyzes the resulting program $P(x, r_1, \ldots, r_m)$ to yield a sound cost bound $f(x, r_1, \ldots, r_m)$.

3. Independently, our data-driven analysis takes the TML program $P(x : \alpha)$, and produces a set of randomly chosen values $v_1, \ldots, v_N : \alpha$ to run P on using an input generator. We also instrument $P(x)$ into an equivalent program $P'(x)$ which, for all $1 \leq i \leq m$, logs the cost of f_i every time it is called non-recursively. We then use Bayesian inference to derive bounds $r_1 \leq g_1(x), \ldots, r_m \leq g_m(x)$. This data-driven analysis is described in Section 5.

4. Finally, we compose the AARA bound $f(x, r_1, \ldots, r_m)$ with the data-driven bounds $r_1 \leq g_1(x), \ldots, r_m \leq g_m(x)$ to yield the bound $f(x, g_1(x), \ldots, g_m(x))$. We empirically demonstrate the usefulness of this approach on a set of benchmark programs in Section 6.

Note: Other than what is necessary for supporting resource decomposition, the design of the TML IR and the translation of source-level SML programs to TML is not a contribution of this thesis.

3 Setting the stage

In this section, we explain how users write resource decomposed SML programs in RaML 2. Then, we introduce our internal intermediate representation (IR) for resource decomposed programs, which we call TML.

3.1 RaML 2 cost annotation signature

Listing 4: Cost annotation signature

```
signature RAML = sig
  val tick : real -> unit
  val mark : string -> int -> unit
  val reset : string -> unit
end
```

All user code is typed against the signature in Listing 4. Here, `tick`(q) ($q \in \mathbb{Q}$) is the canonical AARA cost annotation primitive. The new `mark`$_s(z)$ ($z \in \mathbb{Z}$) construct is used to define the resource component called s that the user wishes to resource decompose against. Please refer to Section 1.4 for examples of how `tick` and `mark`$_s$ are meant to be used.

We also have `reset`$_s$, which resets the net cost associated with resource component s to 0. This is useful in the following scenario: let t be a balanced binary tree and l be a list of keys, and we interested in tracking the *maximum* number of *recursive* calls to `lookup` in `map l (fn x => lookup t x)`. Observe that `lookup` is called non-recursively multiple times. Thus, if the body of `lookup` was annotated by `mark`$_l(1)$ at the point of function entry, we'd end up tracking the *total* number of recursive calls to `lookup`. However, if we rewrote the code as `map l (fn x => reset l; lookup t x)`, we'd correctly track the maximum number of recursive calls.

Listing 5: Cost annotation structure

```
structure Raml : RAML = struct
  val tick = fn (r : real) => ()
  val mark = fn (s : string) => fn (z : int) => ()
  val reset = fn (s : string) => ()
end
```

When performing static analysis, the `tick`, `mark` and `reset` constructs have an uninteresting underlying implementation in Listing 5, since they are *primitives*. The reason why we disallow `real` arguments to `mark`$_s$ will become clearer later.

3.2 TML IR

RaML 2 converts source-level resource decomposed programs into the TML IR, which we define through its statics in Figures 2 and 3. Note that TML maintains a modal separation between values and expressions. Whenever it is unnecessary to differentiate between the two, we will refer to expressions and values as simply expressions. For ease of analysis, in Figure 3, we present named, n-ary sums and products as their corresponding binary versions. From Figure 2, we elide values of base types like `int`, `bool` and `real` since their treatment is similar to the unit type. We also elide the syntactic forms for values of base types SML

$$\frac{}{\Gamma \vdash \langle\rangle : 1} \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda(x.e) : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash v : [\mathtt{rec}(t.\tau)/t]\tau}{\Gamma \vdash \mathtt{fold}(v) : \mathtt{rec}(t.\tau)} \qquad \frac{\Gamma \vdash v : \tau_i}{\Gamma \vdash i \cdot v : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \qquad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash \langle v_1, v_2 \rangle : \tau_1 \times \tau_2}$$

Figure 2: Statics for TML Values

$$\frac{\Gamma \vdash e \div \tau \qquad \Gamma, x : \tau \vdash e' \div \rho}{\Gamma \vdash \mathtt{let}\, x \leftarrow e \;;\; e' \div \rho} \qquad \frac{\Gamma \vdash v : \tau}{\Gamma \vdash \mathtt{ret}(v) \div \tau}$$

$$\frac{\begin{array}{c}\forall 1 \le i \le n \quad \Gamma, f_1 : \tau_1^1 \to \tau_2^1, \ldots, f_n : \tau_1^n \to \tau_2^n, x_i : \tau_1^i \vdash e_i \div \tau_2^i \\ \Gamma, f_1 : \tau_1^1 \to \tau_2^1, \ldots, f_n : \tau_1^n \to \tau_2^n \vdash e \div \rho\end{array}}{\Gamma \vdash \mathtt{letfun}[\mathtt{fun}(f_1, x_1.e_1), \ldots, \mathtt{fun}(f_n, x_n.e_n)] \;;\; f_1, \ldots, f_n.e \div \rho}\ \textsc{mutfun}$$

$$\frac{\Gamma \vdash f : \tau_1 \to \tau_2 \qquad \Gamma \vdash v : \tau_1 \qquad \Gamma, x : \tau_2 \vdash e \div \rho}{\Gamma \vdash \mathtt{app}(f \;;\; v) \;;\; x.e \div \rho} \qquad \frac{q \in \mathbb{Q} \qquad \Gamma \vdash e \div \rho}{\Gamma \vdash \mathtt{tick}(q) \;;\; e \div \rho}\ \textsc{tick}$$

$$\frac{z \in \mathbb{Z} \qquad \Gamma \vdash e \div \rho}{\Gamma \vdash \mathtt{mark}_s(z) \;;\; e \div \rho}\ \textsc{mark} \qquad \frac{\Gamma \vdash e \div \rho}{\Gamma \vdash \mathtt{reset}_s \;;\; e \div \rho}\ \textsc{reset}$$

$$\frac{\Gamma \vdash v : \tau_1 \times \tau_2 \qquad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e \div \rho}{\Gamma \vdash \mathtt{split}\, v\, \{x_1, x_2.e\} \div \rho} \qquad \frac{\Gamma \vdash v : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash e_1 \div \rho \qquad \Gamma, x : \tau_2 \vdash e_2 \div \rho}{\Gamma \vdash \mathtt{case}\ v\, \{x_1.e_1 \mid x_2.e_2\} \div \rho}$$

$$\frac{\Gamma \vdash v : \mathtt{rec}(t.\tau) \qquad \Gamma, x : [\mathtt{rec}(t.\tau)/t]\tau \vdash e \div \rho}{\Gamma \vdash \mathtt{unfold}(v) \;;\; x.e \div \rho}$$

Figure 3: Statics for TML Expressions

arithmetic primitives like `+`, `-`, `%`, `<`, ... The rule MUTFUN is used to define a collection of mutually recursive functions. Any purely functional resource-decomposed SML program can be translated into the TML IR.

We assume a call-by-value cost semantics $e \longmapsto^* v|(h, r); \sigma$ for closed, well-typed TML expressions e. The $(h, r) \in \mathbb{Q}^2$ means that in the evaluation of e to v, the net amount of ticks consumed is h and the peak amount of ticks consumed is r. Similarly, for each resource component s, $\sigma(s) = (h_s, r_s) \in \mathbb{Z}^2$ represents the peak and net resources consumed by $\texttt{mark}_s$. We distinguish between peak and net costs due to the presence of non-monotone cost metrics (specified through negative ticks/marks). For example, when the cost metric is heap space, the net cost may be zero, but the peak cost may be significantly larger.

4 Program Transformation

In this section, we explain the automatic conversion of resource decomposed TML to resource guarded TML in RaML 2, which builds on the resource guarding translation specified in [11].

4.1 Resource effects

We define all uses of $\texttt{mark}_s(z)$ and $\texttt{reset}_s$ in TML as *resource effects*, and define the class of *resource effectful* expressions in TML using the e eff judgment in Figure 4. Note that e eff is defined over *open* expressions e, so long as there exists a typing context Γ such that $\Gamma \vdash e \div \tau$. Of note is the APPEFF rule, which demonstrates our assumption that function bodies are always effectful. Whilst this makes e eff a conservative check for effectfulness, it greatly simplifies the implementation of the transformation.

$$\frac{e\ \mathsf{eff}}{\texttt{let}\, x \leftarrow e\ \texttt{;}\ e'\ \mathsf{eff}} \qquad \frac{e'\ \mathsf{eff}}{\texttt{let}\, x \leftarrow e\ \texttt{;}\ e'\ \mathsf{eff}}$$

$$\frac{e\ \mathsf{eff}}{\texttt{letfun}[\texttt{fun}(f_1, x_1.e_1), \ldots, \texttt{fun}(f_n, x_n.e_n)]\ \texttt{;}\ f_1, \ldots, f_n.e\ \mathsf{eff}} \qquad \frac{}{\texttt{app}(f\ \texttt{;}\ v)\ \texttt{;}\ x.e\ \mathsf{eff}}\ \textsc{appeff}$$

$$\frac{}{\texttt{mark}_s(z)\ \texttt{;}\ e\ \mathsf{eff}} \qquad \frac{e\ \mathsf{eff}}{\texttt{tick}(z)\ \texttt{;}\ e\ \mathsf{eff}} \qquad \frac{}{\texttt{reset}_s\ \texttt{;}\ e\ \mathsf{eff}} \qquad \frac{e\ \mathsf{eff}}{\texttt{split}\, v\, \{\, x_1, x_2.e\,\}\ \mathsf{eff}}$$

$$\frac{e_1\ \mathsf{eff}}{\texttt{case}\ v\, \{\, x_1.e_1 \mid x_2.e_2\,\}\ \mathsf{eff}} \qquad \frac{e_2\ \mathsf{eff}}{\texttt{case}\ v\, \{\, x_1.e_1 \mid x_2.e_2\,\}\ \mathsf{eff}} \qquad \frac{e\ \mathsf{eff}}{\texttt{unfold}(v)\ \texttt{;}\ x.e\ \mathsf{eff}}$$

Figure 4: Characterization of resource effectful expressions

We also define what it means for TML values and expressions to be *pure*, in the sense that – eff cannot be derived for them. All values are thus pure: note that variable values are pure since TML is a call-by-value IR where variables range over (closed) values. We elide the definition of a corresponding e pure judgment to characterize expressions for which e eff is not derivable.

4.2 Introducing a type distinction for resource effects

From the rules MARK and RESET in Figure 3, it is evident that TML does not make a type distinction between effectful and pure expressions. We introduce this type distinction in the TMLF IR in Figure 5, which performs resource effects monadically. We present the only the rules added to and modified from Figures 2 and 3. The remaining rules stay unchanged, barring that all functions must have monadic return type. TMLF is a call-by-value IR and all variables range over values of value type.

$$\textbf{Value types } \tau \coloneqq 1 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \texttt{rec}(t.\tau_1) \mid \tau \to C$$

$$\textbf{Monadic type } C \coloneqq F(\tau)$$

$$\frac{\Gamma \vdash f : \tau_1 \to F(\tau_2) \qquad \Gamma \vdash v : \tau_1 \qquad \Gamma, x : \tau_2 \vdash e \div F(\rho)}{\Gamma \vdash \texttt{app}(f\ ;\ v)\ ;\ x.e \div F(\rho)}\ \textsc{appf} \qquad \frac{z \in \mathbb{Z} \qquad \Gamma \vdash e \div F(\rho)}{\Gamma \vdash \texttt{mark}_s(z)\ ;\ e \div F(\rho)}\ \textsc{markf}$$

$$\frac{\Gamma \vdash e \div F(\rho)}{\Gamma \vdash \texttt{reset}_s\ ;\ e \div F(\rho)}\ \textsc{resetf} \qquad \frac{\Gamma \vdash v : \rho}{\Gamma \vdash \texttt{mret}(v) \div F(\rho)}\ \textsc{lift}$$

$$\frac{\Gamma \vdash e \div F(\tau) \qquad \Gamma, x : \tau \vdash e' \div F(\rho)}{\Gamma \vdash \texttt{bind}\, x \leftarrow e\ ;\ e' \div F(\rho)}\ \textsc{bind}$$

Figure 5: Definition of TMLF

Next, we define a translation of closed, well-typed TML expressions to closed, well-typed TMLF expressions. Our translation in Figure 6 preserves I/O and cost semantics, although we do not prove this. The following typing invariants guide the translation:

1. $\Gamma \vdash v : \tau \implies \|\Gamma\| \vdash \|v\| : \|\tau\|$
2. $\Gamma \vdash e \div \tau \wedge e\,\mathsf{pure} \implies \|\Gamma\| \vdash \|e\| \div \|\tau\|$
3. $\Gamma \vdash e \div \tau \wedge e\,\mathsf{eff} \implies \|\Gamma\| \vdash \|e\| \div |\tau|$

4.3 Expressing resource effects through resource guards

We now define a translation from TMLF to TML, which eliminates resource effects by representing effectful expressions as functions which accept the *resource state* σ and return their result along with the (updated) resource state σ'. This is akin to the handling of effects via the state monad. Our translation in Figure 7 is derived from what was presented in [11].

We identify the n distinct resource components used in the source through numeric identifiers in $[n]$, and define the resource state type $\sigma = \text{nat}^{2^n}$. A value of resource effect type has the form $((h_1, r_1), \ldots, (h_n, r_n))$, where h_i is the ghost variable representing the peak cost that will be incurred on resource component i. Correspondingly, r_i is the remaining amount of resource, instantiated at h_i at the toplevel. In the $\texttt{mark}_i(z)\ ;\ e$ rule, $(s - z)^i$ is the expression that decrements r_i, and diverging if $r_i = 0$ before the decrement. This reifies the invariant that h_i is the peak cost on component i, since if r_i is instantiated to h_i, a decrement on $r_i = 0$ represents a contradictory state of affairs. If the decrement succeeds, we continue threading through the new resource state s' to $[\![e]\!]$. In the $\texttt{reset}_i\ ;\ e$ rule, s^i is the expression that sets $r_i = h_i$ in the current state s, which reifies the cost semantics for $\texttt{reset}_i$.

4.4 Soundness of program transformation

We now present the central theorem (without proof) that allows for the bound AARA derives on the resource guarded program $P(x, r_1, \ldots, r_m)$ to compose with the bounds that Bayesian inference derives on $r_1, \ldots, r_m$.

Translation of types: $\|\tau\|, |\tau|$

$$\|1\| = 1 \qquad \|\tau_1 \times \tau_2\| = \|\tau_1\| \times \|\tau_2\| \qquad \|\tau_1 + \tau_2\| = \|\tau_1\| + \|\tau_2\| \qquad \|\tau_1 \to \tau_2\| = \|\tau_1\| \to |\tau_2|$$

$$\|\mathtt{rec}(t.\tau)\| = \mathtt{rec}(t.\|\tau\|) \qquad \|t\| = t \qquad |\tau| = F(\|\tau\|)$$

Translation of values: $\|v\|$

$$\|\langle\rangle\| = \langle\rangle \qquad \|x\| = x \qquad \|\langle v_1, v_2\rangle\| = \langle\|v_1\|, \|v_2\|\rangle \qquad \|i \cdot v\| = i \cdot \|v\| \qquad \|\mathtt{fold}(v)\| = \mathtt{fold}(\|v\|)$$

$$\|\lambda(x.e)\| = \lambda(x.F(e))$$

Translation of expressions: $\|e\|$

$$F(e) = \|e\| \text{ if } e \,\mathsf{eff} \text{ else } \mathtt{let}\, x \leftarrow \|e\| \;;\; \mathtt{mret}(x)$$

$$\|\mathtt{let}\, x \leftarrow e \;;\; e'\| = \text{if } e \,\mathsf{eff} \text{ then } \mathtt{bind}\, x \leftarrow \|e\| \;;\; F(e') \text{ else } \mathtt{let}\, x \leftarrow \|e\| \;;\; \|e'\| \qquad \|\mathtt{ret}(v)\| = \mathtt{ret}(\|v\|)$$

$$\|\mathtt{letfun}[\mathtt{fun}(f_1, x_1.e_1), \ldots, \mathtt{fun}(f_n, x_n.e_n)] \;;\; f_1, \ldots, f_n.e\| =$$

$$\mathtt{letfun}[\mathtt{fun}(f_1, x_1.F(e_1)), \ldots, \mathtt{fun}(f_n, x_n.F(e_n))] \;;\; f_1, \ldots, f_n.\|e\|$$

$$\|\mathtt{app}(f \;;\; v) \;;\; x.e\| = \mathtt{app}(\|f\| \;;\; \|v\|) \;;\; x.F(e) \qquad \|\mathtt{tick}(q) \;;\; e\| = \mathtt{tick}(q) \;;\; \|e\|$$

$$\|\mathtt{mark}_s(z) \;;\; e\| = \mathtt{mark}_s(z) \;;\; F(e) \qquad \|\mathtt{reset}_s \;;\; e\| = \mathtt{reset}_s \;;\; F(e)$$

$$\|\mathtt{split}\, v\, \{\, x_1, x_2.e \,\}\| = \mathtt{split}\, \|v\|\, \{\, x_1, x_2.\|e\| \,\}$$

$$\|\mathtt{case}\; v\, \{\, x_1.e_1 \mid x_2.e_2 \,\}\| = \begin{cases} \mathtt{case}\, \|v\|\, \{\, x_1.F(e)_1 \mid x_2.F(e)_2 \,\} & \text{if } e_1 \,\mathsf{eff} \vee e_2 \,\mathsf{eff} \\ \mathtt{case}\, \|v\|\, \{\, x_1.\|e\|_1 \mid x_2.\|e\|_2 \,\} & \text{otherwise} \end{cases}$$

$$\|\mathtt{unfold}(v) \;;\; x.e\| = \mathtt{unfold}(\|v\|) \;;\; x.\|e\|$$

Figure 6: Translation of TML into TMLF

Translation of types: $[\![\tau]\!]$, $[\![C]\!]$

$$[\![F(\tau)]\!] = \sigma \to \sigma \times [\![\tau]\!] \qquad [\![\tau_1 \times \tau_2]\!] = [\![\tau_1]\!] \times [\![\tau_2]\!] \qquad [\![\tau_1 + \tau_2]\!] = [\![\tau_1]\!] + [\![\tau_2]\!] \qquad [\![\tau \to C]\!] = [\![\tau]\!] \to [\![C]\!]$$

$$[\![\mathtt{rec}(t.\tau)]\!] = \mathtt{rec}(t.[\![\tau]\!]) \qquad [\![t]\!] = t$$

Translation of values: $[\![v]\!]$

$$[\![\langle\rangle]\!] = \langle\rangle \qquad [\![x]\!] = x \qquad [\![\langle v_1, v_2\rangle]\!] = \langle [\![v_1]\!], [\![v_2]\!]\rangle \qquad [\![i \cdot v]\!] = i \cdot [\![v]\!] \qquad [\![\mathtt{fold}(v)]\!] = \mathtt{fold}([\![v]\!])$$

$$[\![\lambda(x.e)]\!] = \lambda(x.\,\mathtt{ret}(\lambda(s.\mathtt{let}\, c \leftarrow [\![e]\!]\ ;\ \mathtt{app}(c\ ;\ s)\ ;\ r.\,\mathtt{ret}(r))))$$

Translation of expressions: $[\![e]\!]$

$$[\![\mathtt{let}\, x \leftarrow e\ ;\ e']\!] = \mathtt{let}\, x \leftarrow [\![e]\!]\ ;\ [\![e']\!] \qquad [\![\mathtt{ret}(v)]\!] = \mathtt{ret}([\![v]\!])$$

$$[\![\mathtt{letfun}[\mathtt{fun}(f_1, x_1.e_1), \ldots, \mathtt{fun}(f_n, x_n.e_n)]\ ;\ f_1, \ldots, f_n.e]\!] =$$

$$\mathtt{letfun}[\mathtt{fun}(f_1, x_1.[\![e]\!]_1), \ldots, \mathtt{fun}(f_n, x_n.[\![e]\!]_n)]\ ;\ f_1, \ldots, f_n.[\![e]\!]$$

$$[\![\mathtt{app}(f\ ;\ v)\ ;\ x.e]\!] = \mathtt{ret}(\lambda(s.\mathtt{app}([\![f]\!]\ ;\ [\![v]\!])\ ;\ c.\mathtt{app}(c\ ;\ s)\ ;\ \langle s', x\rangle.\mathtt{let}\, g \leftarrow [\![e]\!]\ ;\ \mathtt{app}(g\ ;\ s')\ ;\ r.\,\mathtt{ret}(r)))$$

$$[\![\mathtt{tick}(q)\ ;\ e]\!] = \mathtt{tick}(q)\ ;\ [\![e]\!]$$

$$[\![\mathtt{mark}_i(z)\ ;\ e]\!] = \mathtt{ret}(\lambda(s.\mathtt{let}\, s' \leftarrow (s - z)^i\ ;\ \mathtt{let}\, c \leftarrow [\![e]\!]\ ;\ \mathtt{app}(c\ ;\ s')\ ;\ r.\,\mathtt{ret}(r)))$$

$$[\![\mathtt{reset}_i\ ;\ e]\!] = \mathtt{ret}(\lambda(s.\mathtt{let}\, s' \leftarrow s^i\ ;\ \mathtt{let}\, c \leftarrow [\![e]\!]\ ;\ \mathtt{app}(c\ ;\ s')\ ;\ r.\,\mathtt{ret}(r)))$$

$$[\![\mathtt{split}\, v\, \{\, x_1, x_2.e\,\}]\!] = \mathtt{split}\, [\![v]\!]\, \{\, x_1, x_2.[\![e]\!]\,\}$$

$$[\![\mathtt{case}\ v\, \{\, x_1.e_1 \mid x_2.e_2\,\}]\!] = \mathtt{case}\, [\![v]\!]\, \{\, x_1.[\![e]\!]_1 \mid x_2.[\![e]\!]_2\,\} \qquad [\![\mathtt{unfold}(v)\ ;\ x.e]\!] = \mathtt{unfold}([\![v]\!])\ ;\ x.[\![e]\!]$$

$$[\![\mathtt{mret}(v)]\!] = \mathtt{ret}(\lambda(s.\,\mathtt{ret}(\langle s, [\![v]\!]\rangle)))$$

$$[\![\mathtt{bind}\, x \leftarrow e\ ;\ e']\!] = \mathtt{ret}(\lambda(s.\mathtt{let}\, c \leftarrow [\![e]\!]\ ;\ \mathtt{app}(c\ ;\ s)\ ;\ \langle s', x\rangle.\mathtt{let}\, c' \leftarrow [\![e']\!]\ ;\ \mathtt{app}(c'\ ;\ s')\ ;\ r.\,\mathtt{ret}(r)))$$

Figure 7: Resource guarding of TMLF programs into TML

Theorem 1 *Let $P : \alpha \to 1$ be a TML program. Then, for all values $v : \alpha$, if $P(v)$ evaluates to $\langle\rangle$ with cost (h, r) and resource effects $\sigma = [(h_1, r_1), \ldots, (h_m, r_m)]$, then for all $h'_i \geq h_i$, $[\![\|P(v)\|]\!]((h'_1, h'_1), \ldots, (h'_m, h'_m))$ evaluates to $\langle\rangle$ with cost (h, r) whilst performing no resource effects.*

Importantly, if $f(x, r_1, \ldots, r_m)$ is a AARA-derived bound on the resource guarded program $P_{rd} : \alpha \to \sigma \to \sigma \times 1$, then as long as the bounds $r_1 \leq g_1(x), \ldots, r_m \leq g_m(x)$ derived by Bayesian inference are sound, $f(x, g_1(x), \ldots, g_m(x))$ is a sound bound on the original program $P(x)$. Note that Theorem 1 may be stated for arbitrary result types, we only state it at the unit result for simplicity.

5 Data-driven analysis

We shall now explain the implementation of the Bayesian inference based data-driven analysis when performing resource decomposition.

5.1 Random input generator

Given the source level resource decomposed program $P(x : \alpha)$, where α is guaranteed to be a monomorphic, polynomial type due to upstream code transformations, the goal of the input generator is generate a set of valid SML values $v : \alpha$.

5.1.1 Types

We parse the types used in the source program into a format which captures SML's invariants that (i) inlined sum types are disallowed and (ii) `datatype` definitions are always sum types. Fix a set $\mathcal{T}$ of defined type names and let c denote data constructors.

$$
\begin{array}{lcll}
\tau & ::= & 1 & \\
 & | & \text{int} & \\
 & | & t & \text{defined type } t \in \mathcal{T} \\
 & | & \tau_1 \times \cdots \times \tau_n & \\
\text{type definition} & ::= & t \in \mathcal{T} \mapsto c_1 \text{ of } \tau_1 \ldots c_n \text{ of } \tau_n &
\end{array}
$$

5.1.2 Sizes of values

In addition to a type τ, the generator requires a target "size" s which it uses to guide its choices when generating a value $v : \tau$. Given a type $t \in \mathcal{T}$ defined as c_1 of $\tau_1 \ldots c_n$ of τ_n, we define its values to have the form $\text{fold}_t(c_i \cdot v_i)$, where value $v_i : \tau_i$.

Now, given any type τ, and any defined type t, we define the size of value $v : \tau$ with respect to t, written $\text{size}_t(v)$, as the number of times fold_t appears in v. Finally, assuming $\mathcal{T} = \{t_1, t_2, \ldots, t_k\}$ for some fixed ordering, we define $\text{size}(v : \tau)$ as $(\text{size}_{t_1}(v), \text{size}_{t_2}(v), \ldots, \text{size}_{t_k}(v))$.

For example, we may define the type of rose trees containing integer data as {rtree $\coloneqq$ Node of int$\times$rlist, rlist $\coloneqq$ Nil of 1 | Cons of rtree $\times$ rlist}. Consider the value v : rtree = $\text{Node}(1, \text{Cons}(\text{Node}(2, \text{Nil}), \text{Nil}))$. Then, $\text{size}_{\text{rtree}}(v) = 2$, $\text{size}_{\text{rlist}}(v) = 3$, and so $\text{size}(v) = (2, 3)$.

5.1.3 Generator

Given a type τ and a target size σ, the generator we describe in Algorithm 1 produces a value v such that $\text{size}(v)$ is "close enough" to σ. It is challenging to meet σ exactly since it is possible for sizes to be infeasible, i.e. there exist sizes σ' such that there are no values v satisfying $\text{size}(v : \tau) = \sigma'$. For example, $(0, 1)$ is an infeasible size for the rtree type.

Algorithm 1 Random input generator

Require: D: type definitions for each $t \in \mathcal{T}$
Require: τ: the type being generated
Require: σ: desired sizes for each defined type in τ

function GENERATE(τ, σ)
  if τ is a custom type t **then**
    Get definition c_1 of $\tau_1 \mid \cdots \mid c_n$ of τ_n of t from D
    if $\sigma[t] > 1$ **then**
      Choose a recursive constructor c_i uniformly at random
    else
      Choose a base constructor c_i uniformly at random
    end if
    return c_i(GENERATE($\tau_i, \sigma -_t 1$))
  else if τ is a product type $\tau_1 \times \cdots \times \tau_m$ **then**
    for each defined type t **do**
      Uniformly distribute $\sigma[t]$ among τ_i that can reach t
      Set $\sigma_i[t] = 0$ for components that cannot reach t
    end for
    for $1 \leq i \leq m$ **do**
      Create i-th component $P_i \leftarrow$ GENERATE(τ_i, σ_i)
      Add any size "slack" (difference) to σ_{i+1}
    end for
    return $(P_1, \ldots, P_m)$
  else ▷ τ is a base type
    return a value chosen uniformly at random from τ
  end if
end function

Finally, to create our dataset of size N for type τ, we simply decide on sizes $[\sigma_1, \ldots, \sigma_N]$ and then invoke generate(τ, σ_i) for each i. The generator's output is pretty-printed into a valid SML value.

5.2 Cost logging instrumentation

When performing data-driven analysis, we wish to track the costs incurred due to the `tick` and $\texttt{mark}_s$ annotations at runtime. Thus, in data collection phase for Bayesian inference, we run the source-level resource decomposed program against a different `structure RamlData : RAML`. It maintains a state $s : \mathbb{Q}^2$ for tracking the (peak, net) cost due to `tick`, and a state $\sigma : \mathbb{Z}^{2^k}$, where $\sigma.i$ is the (peak, net) cost for the i^{th} resource component toggled by the $\texttt{mark}_i$ annotations. The `tick`, $\texttt{mark}_i$, and $\texttt{reset}_i$ annotations are implemented exactly as described in [11] to update the states s and σ appropriately. Finally, we include a $\texttt{restart}_i$ construct for resource component i, which has the effect of setting $\sigma.i = (0, 0)$.

Let $\mathcal{F} = \{f_1, \ldots, f_k\}$ be the set of recursive functions in the TML program P whose definitions contain $\texttt{mark}_{f_i}(1| - 1)$ annotations for tracking their recursion depth, like `sort` does in Listing 2. The goal now is to transform P into an equivalent program P' in which every function $f_i \in \mathcal{F}$ logs its argument sizes and peak recursion depth for every call made to it at runtime. Then, P' may be straightforwardly converted to SML, and run on the generated set of SML values against `structure RamlData` to produce our cost

dataset.

5.2.1 User-defined size functions

We define a *size function* for type τ as any function with type $\tau \to \mathbb{N}^k$, for any k. That is, a size function for τ yield a (multi)-dimensional notion of size for values $v : \tau$. For example, size$(- : \tau)$ defined in Section 5.1.2 is a size function for τ.

We expect the user to define size functions for each argument type for each $f_i \in \mathcal{F}$. That is, if f_i is a curried function with the type $\tau_1^i \to \ldots \tau_{n_i}^i \to \rho_i$, then we expect size functions for each of $\tau_1^i, \tau_2^i, \ldots, \tau_{n_i}^i$. If the user does not provide a particular τ_j^i, then the data-driven bound derived on the recursion depth of f_i cannot be a function its j^{th} argument. This means that for any position $1 \leq j \leq n_i$, if the recursion depth of f_i could depend on its j^{th} argument, then the user must provide a size function for τ_j.

5.2.2 Transformation

Given an $f_i \in \mathcal{F}$, we define its *instrumented* version f_i^{inst} as its η-expansion with the following modifications:

1. The user-defined size functions are called on each argument $y_j : \tau_j^i$ for $1 \leq j \leq n_i$ to yield $s = \langle \text{size}_{\tau_1^i}(y_1), \ldots, \text{size}_{\tau_{n_i}^i}(y_{n_i}) \rangle$. If there is no provided size function for τ_j^i, $\text{size}_{\tau_j^i}(y_j)$ is seen as contributing no size.

2. The resource component associated with the recursion depth of f_i is reset to have peak and net cost 0, using $\texttt{restart}_i$. This prevents the measurements from previous calls to f_i from affecting the current measurement.

3. A call is made to f_i with arguments $y_1, y_2, \ldots, y_{n_i}$. Once the call returns, the peak recursion depth along with the argument sizes s is logged.

The cost logging transformation is largely performed pointwise on a given TML expression. The only interesting case in the transformation is the treatment of the MUTFUN variant in Figure 3, since this declares recursive functions which may be in $\mathcal{F}$. Given an expression $\texttt{letfun}[\texttt{fun}(f_1, x_1.e_1), \ldots, \texttt{fun}(f_n, x_n.e_n)]$; $f_1, \ldots, f_n.e$:

1. Perform the transformation recursively on the function bodies.

2. For each defined function f that is in $\mathcal{F}$, insert a declaration for f^{inst} preceeding e. Then, in e, replace all calls to those $f \in \mathcal{F}$ with calls to f^{inst}.

By performing this transformation on the TML program P, we obtain P', which is equivalent to P' and correctly makes cost measurements for each $f \in \mathcal{F}$, so long as each $f \in \mathcal{F}$ meets the following conditions:

1. f must not perform computations between taking curried arguments, i.e., the η-expansion of f should be equivalent to f.

2. All callsites of f outside its definition must have the property that, once called, they can only be called again after the original call returns. This is necessary since f^{inst} calls $\texttt{restart}_i$ before calling f, and it is undesirable to have a cost measurement zeroed-out before it is completed.

3. The recursion depth of f must be a function of (and only of) its arguments, for the cost measurements to be useful. That is, f's recursion depth should not depend on value(s) bound in its closure.

5.3 Inferring recursion depth bounds

Upon running the transformed program $P'(x)$ on the generated set of values against `structure RamlData`, we obtain a cost dataset $\mathcal{D} = \bigcup_{f \in \mathcal{F}} \mathcal{D}_f$. $\mathcal{D}_f[i]$ contains cost data for the i^{th} call to f, where the recursive calls f makes to itself do not count. Let σ_f^i be the peak recursion depth of f for the i^{th} time it is called. Let $v_1^i : \tau_1, v_2^i : \tau_2, \ldots, v_{n_f}^i : \tau_{n_f}$ be the (top-level) arguments to f the i^{th} time it is called. Then:

$$\mathcal{D}_f[i] = \langle [\text{size}_{\tau_1}(v_1^i), \text{size}_{\tau_2}(v_2^i), \ldots, \text{size}_{\tau_{n_f}}(v_{n_f}^i)], \sigma_f^i \rangle$$

Now, given $\mathcal{D}_f$, the next step is to infer a bound on its recursion depth in terms of the sizes of its arguments. For simplicity, we commit to only inferring univariate bounds, as follows:

1. For each f, we use mutual information to choose the distinguished size component of the particular type with highest dependence on the recursion depth data in $\mathcal{D}_f$.

2. Finally, we use Bayesian inference model of [11], implemented in STAN, to infer a univariate linear or logarithmic bound on the recursion depth of f.

It's worth noting that the bound derived on f is in terms of its *own* arguments, as opposed to the *global* argument x of the program $P(x)$ described in the original definition of resource decomposition [11]. This localizes the application of Bayesian inference, the one and only source of unsoundness in RaML 2, to those functions whose recursion depths cannot be tightly bounded by AARA.

The final step is to *statically* derive a bound on the size component s^* that recursion of depth of f is a function of in terms of the global program argument x. We tried instrumenting AARA itself for this purpose, treating size as a notion of cost. However, we could not arrive at a general solution to this problem, and thus perform the size bounding analysis manually.

6 Evaluation

In this section, we empirically demonstrate how resource decomposition yields asymptotically tight and sound cost bounds for purely functional SML programs containing one or more functions with logarithmic recursion depth. We choose the following examples:

1. `MergeSort`: performs mergesort on a list of natural numbers. Resource components: recursion depth of `mergesort` : list(nat) $\to$ list(nat).

2. `HeapInsert`: creates a leftist heap from a list of natural numbers, by repeated insertion. Since leftist heap operations are implemented in terms of an efficient `merge` : heap $\times$ heap $\to$ heap, our resource component is the recursion depth of `merge`.

3. `BalancedBST`: given keys : list(nat) and queries : list(nat), (i) sorts keys using `mergesort` (ii) creates a balanced BST t using the sorted keys, and (iii) calls BST lookup for each key in queries on t. Resource components are recursion depths of (i) `mergesort` and (ii) `lookup` : tree $\times$ nat $\to$ bool.

4. `RedBlackTree`: given keys : list(nat) and queries : list(nat), (i) creates a red-black tree t using repeated insertions of keys and (ii) calls red-black tree lookup for each key in queries on t. Resource components are recursion depths of (i) lookup : rbtree $\times$ nat $\to$ bool and (ii) insert : rbtree $\times$ nat $\to$ rbtree.

5. `RedBlackJoin`: given two red-black trees t_1, t_2 containing natural numbers and a natural number n, joins the components into a red black tree containing all the keys. Assuming the implementation described in [8], resource components are the recursion depths of (i) `joinleft` : rbtree$\times$rbtree$\times$nat $\to$ rbtree and (ii) the symmetric function `joinright`.

Our notion of cost for each example is the number of *recursive* function calls. That is, we perform `Raml.tick`(1.0) when entering the body of a recursive function. The one and only exception to this rule is `BalancedBST`, in which we do not perform ticks for the recursive functions `split` and `merge`.

We present our results in Tables 1 and 2. For the ground truth costs for operations on leftist heaps and red-black trees, we refer to existing results: `merge` on leftist heaps (trees) has worst-case recursion depth $O(\log|t_1|+\log|t_2|)$ [9], insertions and lookups on red-black trees have $O(\log|t|)$ depth, and joining two red-black trees has depth $O(\log|t_1|+\log|t_2|)$ [8]. Here, $|t|$ refers to the number of nodes in the tree t. All ground truth and AARA bounds are reported in terms of x, the *global* argument to the program $P(x)$. Conversely, Bayesian inference bounds are in terms of the arguments of the specific function whose recursion depth is being bounded. For example, in `HeapInsert`, x_1 and x_2 refers to the list of keys and queries respectively, and the bound on the recursion depth of `merge` is $O(\log(|h_1|+|h_2|))$, where h_1 and h_2 are the leftist heaps being merged. Then, the maximum sizes of the arguments to such functions are bounded *manually* in terms of the global argument size $|x|$. For instance, the sum of the sizes of the heaps being merged $|h_1|+|h_2|$ is at most the number of keys $|x|$.

The Bayesian inference results in Table 2 are derived by running each program on 200 inputs of increasing sizes produced by our generator. The guarded AARA column represents AARA's bounds on the resource guarded versions of the programs produced by our automatic program transformation. A small caveat is that since RaML 2's implementation of AARA is currently incomplete, all AARA results are produced using the (old) implementation of RaML for OCaml programs [13]. That is, the plain AARA results in Table 1 are produced by pretty-printing resource decomposed source level SML programs to OCaml, and the guarded AARA results in Table 2 are produced by pretty-printing resource guarded TML programs to OCaml.

We provide an example of workflow for applying resource decomposition to `HeapInsert` in Appendix A.

Table 1: Resource Decomposition Results

Benchmark	**Ground Truth**	**Plain AARA**	**Resource Decomposition**
`MergeSort`	$O(\|x\| \log \|x\|)$	$1.5\|x\|^2 - 1.5\|x\|$	$O(\|x\| \log \|x\|)$
`HeapInsert`	$O(\|x\| \log \|x\|)$	$0.5\|x\|^2 + 0.5\|x\|$	$O(\|x\| \log \|x\|)$
`BalancedBST`	$O((\|x_1\| + \|x_2\|) \log \|x_1\|)$	$0.4\|x_1\|+\|x_1\|\|x_2\|+0.7\|x_1\|^2+\|x_2\|$	$O((\|x_1\| + \|x_2\|) \log \|x_1\|)$
`RedBlackTree`	$O((\|x_1\| + \|x_2\|) \log \|x_1\|)$	$1.5\|x_1\|+\|x_1\|\|x_2\|+0.5\|x_1\|^2+\|x_2\|$	$O((\|x_1\| + \|x_2\|) \log \|x_1\|)$
`RedBlackJoin`	$O(\log \|x_1\| + \log \|x_2\|)$	$\|x_1\| + \|x_2\|$	$O(\log \|x_1\| + \log \|x_2\|)$

Table 2: Resource Decomposition: Intermediate Stage Results

Benchmark	**Resource Decomposition**		
	Bayesian Inference	**Size Bound**	**Guarded AARA**
`MergeSort`	$d_1 \in O(\log \|l\|)$	$\|l\| \leq \|x\|$	$2\|x\|d_1$
`HeapInsert`	$d_1 \in O(\log(\|h_1\| + \|h_2\|))$	$\|h_1\| + \|h_2\| \leq \|x\|$	$\|x\| + \|x\|d_1$
`BalancedBST`	$d_1 \in O(\log \|l\|)$ $d_2 \in O(\log \|t\|)$	$\|l\| \leq \|x_1\|$ $\|t\| \leq \|x_1\|$	$1 + 0.5d_1\|x_1\| + 0.5\|x_1\| + \|x_2\|d_2$
`RedBlackTree`	$d_1 \in O(\log \|t\|)$ $d_2 \in O(\log \|t\|)$	$\|t\| \leq \|x_1\|$ $\|t\| \leq \|x_1\|$	$\|x_1\|d_1 + \|x_1\| + \|x_2\|d_2 + \|x_2\|$
`RedBlackJoin`	$d_i \in O(\log(\|t_1\| + \|t_2\|))$	$\|t_1\| + \|t_2\| \leq \|x_1\| + \|x_2\|$	$d_1 + d_2$

7 Comparison with prior work

Our work is an extension of the original formulation of resource decomposition presented in [11] in the following aspects:

1. We automate the translation of resource decomposed to resource guarded programs, which was originally done by hand.

2. We automate the generation of inputs for programs with arbitrary input type (so long as said type is polynomial). Previously, generation was supported for only list and graph types.

3. We minimize the dependence on data-driven analysis in the following sense: instead of bounding the recursion depth of functions in terms of the global argument x in program $P(x)$, recursion depth bounds are expressed in terms of the sizes of the function's own arguments. Our plan then was to employ a separate, *sound* analysis to upper bound the sizes of the arguments in terms the global argument size $|x|$. Although the size bounding phase is not complete yet, in principle, our approach localizes the application of unsound empirical analyses only where necessary.

8 Future Work

The translation from resource decomposed TML to resource guarded TML leads to a non-trivial increase in the program size. This manifests in AARA taking a long time to analyze the resource guarded versions of more complicated programs like Bellman Ford, Least Common Ancestor, Subset Sum, etc. It could thus be fruitful to add an semantics-preserving optimization phase to resource guarded TML programs.

Secondly, of the goals we weren't able to meet was statically deriving bounds on the worst-case sizes of arguments of functions whose recursion depths we analyze via Bayesian inference. A problematic case we considered was handling programs of the form:

```
fun mergesort_all (l : nat list list) =
  List.map l mergesort
```

If we treat the recursion depth of `mergesort` as a resource component d, we'd derive a logarithmic bound on d in terms of the length of the list argument to `mergesort`. If l is a nested list of the form $[l_1, l_2, \ldots, l_m]$, AARA would derive the bound $d * \sum_{i=1}^{m} |l_i|$ on the resource guarded version of `mergesort_all`. However, the only possible worst case bound we can give on the size of the largest argument to `mergesort` in terms of the global argument l is $\sum_{i=1}^{m} |l_i|$, which is too conservative of a bound to be useful. The core issue is that we use a single resource component d for *all* calls to `mergesort`. We can see this problem more clearly in the resource guarded version of `mergesort_all`, observe that the same ghost variable d is used for all calls to `mergesort`:

```
fun mergesort_all l d =
  List.map l (mergesort d)
```

A potential solution we explored was developing ways to use different resource components for each individual call to `mergesort`, but we have not yet arrived at a general solution that works for all programs. Going forward, we intend to fully explore the extent to which this idea works.

References

[1] Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., and Riddell, A. Stan: A probabilistic programming language. *Journal of Statistical Software 76*, 1 (2017), 1–32.

[2] Chu, E. The theory and implementation of resource aware ml 2. Master's thesis, Carnegie Mellon University, Pittsburgh, PA, December 2023. Available at `https://www.andrew.cmu.edu/user/ethanchu/raml-thesis.pdf`.

[3] Hoffmann, J., Aehlig, K., and Hofmann, M. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst. 34* (2012), 14:1–14:62.

[4] Hoffmann, J., and Hofmann, M. Amortized resource analysis with polynomial potential. In *European Symposium on Programming* (2010).

[5] Hoffmann, J., and Jost, S. Two decades of automatic amortized resource analysis. *Mathematical Structures in Computer Science 32* (2022), 729 – 759.

[6] Hofmann, M., and Jost, S. Static prediction of heap space usage for first-order functional programs. In *ACM-SIGACT Symposium on Principles of Programming Languages* (2003).

[7] Hofmann, M., and Pavlova, M. Elimination of ghost variables in program logics. In *Proceedings of the 3rd Conference on Trustworthy Global Computing* (Berlin, Heidelberg, 2007), TGC'07, Springer-Verlag, p. 1–20.

[8] Li, R., Grodin, H., and Harper, R. A verified cost analysis of joinable red-black trees.

[9] Okasaki, C. *Purely Functional Data Structures*. Cambridge University Press, USA, 1999.

[10] Pham, L., Niu, Y., Glover, N., Saad, F., and Hoffmann, J. Artifact for resource decomposition, Aug. 2025. Available at `https://doi.org/10.5281/zenodo.16916701`.

[11] Pham, L., Niu, Y., Glover, N., Saad, F., and Hoffmann, J. Integrating resource analyses via resource decomposition. *Proc. ACM Program. Lang. 9*, OOPSLA2 (Oct. 2025).

[12] Pham, L., Saad, F. A., and Hoffmann, J. Robust resource bounds with static analysis and bayesian inference. *Proc. ACM Program. Lang. 8*, PLDI (June 2024).

[13] Wang, D., Kahn, D. M., and Hoffmann, J. Resource aware ml, 2020. Available at `https://doi.org/10.1145/3410233`.

Appendices

A Workflow for `HeapInsert`

At the very beginning, the user must write a source level resource decomposed program, as shown in Figure 8:

```
datatype nat = Succ of nat | Zero
datatype binary_heap = Leaf | Node of nat * nat * binary_heap *
   binary_heap

fun num_nodes h = ...

(* size function for cost logging merge *)
fun size_merge (h1, h2) = fn i => if i = 0 then num_nodes h1 +
   num_nodes h2 else ~1

fun rd_merge ((h1, h2) : binary_heap * binary_heap) =
  let
    val _ = Raml.mark "merge" 1
    val res = (case (h1, h2) of
    (Leaf, h) => h | (h, Leaf) => h
    | (Node (v1, rank1, l1, r1), Node (v2, rank2, l2, r2)) =>
      if v1 > v2 then
        let
          val () = Raml.tick 1.0
          val merged = rd_merge (r2, Node (v1, rank1, l1, r1))
          val rank_left = rank l2 val rank_right = rank merged
        in
          if rank_right <= rank_left
          then Node (v2, Succ rank_right, l2, merged) else Node (v2,
             Succ rank_left, merged, l2)
        end
      else (* symmetric *))
    in Raml.mark "merge" (~1); res end

fun insert (v, heap) = rd_merge ((singleton v), heap)

fun create_heap_from_list xs =
  case xs of
  [] => Leaf
  | hd :: tl => let val () = Raml.tick 1.0 in insert (hd,
     (create_heap_from_list tl)) end

fun main (xs : nat list) = create_heap_from_list xs
```

Figure 8: Resource decomposed program for `HeapInsert`

The function treats the recursion depth of `merge` as a resource component, hence we perform $\texttt{mark}_{merge}(1)$ when entering and and $\texttt{mark}_{merge}(-1)$ when leaving the function body. We perform a `tick`(1.0) upon recurring in `merge` amd `create_heap_from_list`.

Firstly, we define the size function for the binary_heap × binary_heap type through `size_merge`. This is necessary for the data-driven analysis as described in Section 5.2.1. Here, we add the number of nodes in the two heaps to create a single dimensional notion of size. This is because we want the data-driven bound on the recursion depth of `merge` to be function of the sizes of both heaps. Alternatively, we could define a two-dimensional notion of size such as $(|h_1|, |h_2|)$, and let Bayesian inference choose the component that the recursion depth is most dependent on.

We have had to choose a peculiar type for the size functions. Ideally, for a given type τ, the user would define $\text{size}_\tau : \tau \to \text{list(int)}$, to define multidimensional notions of size. However, the conversion to TML uses MLton to monomorphize the program, which renames the names and constructors of algebraic datatypes in unpredictable ways. Thus, we encode the desired list(int) as a primitive int → int type.

Thereafter, RaML 2 observes the argument type of `main` and produces 200 instances of list(nat) of increasing sizes. Running the data-driven analysis workflow of Section 5, we obtain a bound of $0.6 + 1.8\log(0.6 + 4.2(|h_1| + |h_2|))$ on the recursion depth of `merge`. We must bound $|h_1| + |h_2|$ by *hand* in terms of the global argument $|xs|$, and we can see that $|h_1| + |h_2| \leq |xs|$. So, we derive an asymptotically sound and tight $O(\log |xs|)$ bound on the recursion depth of `merge`.

In parallel, our automatic program transformation converts `HeapInsert` into its resource guarded version, which we prettify for the purposes of demonstration in Figure 9. RaML reports an $|xs| + |xs| * \text{net}$ bound on this program, and we use our data-driven $\text{net} \in O(\log |xs|)$ bound to produce our final $O(|xs| \log |xs|)$ bound.

```
datatype nat = Succ of nat | Zero
datatype binary_heap = Leaf | Node of nat * nat * binary_heap *
   binary_heap
fun diverge () = diverge ()

fun rd_merge ((h1, h2) : binary_heap * binary_heap) (net, _) =
  case net of
  | Zero => diverge ()
  | Succ net' =>
    case (h1, h2) of
    (Leaf, h) => (h, (net, peak)) | (h, Leaf) => (h, (net, peak))
    | (Node (v1, rank1, l1, r1), Node (v2, rank2, l2, r2)) =>
      if v1 > v2 then
        let
          val () = Raml.tick 1.0
          val merged, (net'', peak) = rd_merge (r2, Node (v1, rank1,
             l1, r1)) (net', peak)
          val rank_left = rank l2 val rank_right = rank merged
        in
          (if rank_right <= rank_left
          then Node (v2, Succ rank_right, l2, merged) else Node (v2,
             Succ rank_left, merged, l2),
          (Succ net'', peak))
        end
      else (* symmetric *)

fun insert (v, heap) (net, peak) = rd_merge ((singleton v), heap)
   (net, peak)

fun create_heap_from_list xs (net, peak) =
  case xs of
  [] => (Leaf, (net, peak))
  | hd :: tl => let val () = Raml.tick 1.0 in
  let val tl_res, (net', peak) = create_heap_from_list tl (net, peak)
     in
  insert (hd, tl_res) (net', peak) end

fun main (xs : nat list) (net, peak) = create_heap_from_list xs (net,
   peak)
```

Figure 9: Resource guarded program for `HeapInsert`