Automatic Dependent Resource Analysis

DAVID CAO, University of California, San Diego, USA

1 INTRODUCTION

There exists a litany of existing systems which can analyze the resource usage of an input program. These systems lie on a spectrum of expressiveness and automation. Inference-based systems can automatically generate an upper bound on the resource use of a program, at the cost of generating less precise analyses of resource usage. Resource-Aware ML (RaML) [Hoffmann et al. 2012], for instance, can infer polynomial bounds on programs by distributing *potential* among data structures, but cannot infer *dependent bounds*: resource bounds which include program variables or conditional expressions. Meanwhile, verification-based systems are more expressive but lack automation; ReSyn [Knoth et al. 2019, 2020], for instance, requires the programmer to provide a resource bound themselves, but can verify bounds with conditional expressions and program variables as a result.

This article presents *automated dependent resource analysis*, a set of techniques for inferring dependent bounds on program resource consumption. Our approach aims to combine the best of both worlds, introducing new techniques to infer a richer set of resource bounds. Our system takes as input a cost model for the program and automatically infers a fine-grained upper bound on resource usage, a bound which can reference program variables and include conditional expressions.

To accomplish this, we build on ReSyn's liquid resource types [Knoth et al. 2020], which integrate resource annotations with liquid types [Vazou et al. 2013], allowing ReSyn to use refinements to reason about resource consumption by adding potential annotations. We augment ReSyn with liquid resource type inference, synthesizing top-level resource annotations and optimizing these annotations to the find a minimal fine-grained upper bound on a program's resource use.

2 A MOTIVATING EXAMPLE: RANGE

We present the range function as a motivating example for our work. An implementation of range, annotated with the costs of individual operations via *ticks*, is included in fig. 1. This function produces a list of numbers between given minimum and maximum values, inclusive, and its resource use is defined as the length of the output list (i.e. the difference between the input maximum and minimum); in particular, the resource use of this function is *dependent* on the value of its arguments. Because of this, this example would be beyond the scope of existing resource inference systems.

```
range = λlo. λhi. if lo ≥hi then Nil else Cons lo (tick 1 (range (lo + 1) hi))
```

Fig. 1. The range function

3 EXISTING WORK: LIQUID RESOURCE TYPES

To infer the resource use of examples like range, we build upon ReSyn and its liquid resource type system. ReSyn combines liquid types with a resource analysis system that revolves around annotating types with a numeric quantity called *potential*. For instance, a value of type Int^1 carries one unit of potential, and a value of type $\mathsf{List}\ \mathsf{Int}^1$ carries a unit of potential for each element in the list. ReSyn is also capable of distributing potential non-uniformly throughout a data structure via *abstract potentials*, allowing programmers to express polynomial or exponential resource use. And because ReSyn's type system integrates refinement types and resource bounds, these potential annotations can also contain conditional expressions and program variables. These units of potential can be spent on operations that might occur during evaluation. These annotations on

Author's address: David Cao, University of California, San Diego, USA, dmcao@ucsd.edu.

resource distribution and use are eventually reduced to a set of conditional linear arithmetic (CLIA) constraints, dispatched to ReSyn's existing constraint-solving infrastructure.

3.1 Range in ReSyn

As a result of having these features, ReSyn is able to verify the correct upper resource bound on range's resource use. The liquid resource type for the `range` function is provided below:

$$\mathsf{range} :: lo : \mathsf{Int} \to hi : \{\, \mathsf{Int}^{\nu - lo} \mid \nu \geq lo \,\} \to \mathsf{List\ Int}$$

In this signature, we require that hi must be greater than or equal to the value of lo, and that hi must contain hi − lo units of *potential*, which can be used to pay off resource use during execution of the function. (ν in the type of hi represents the value of hi itself.) Note that this resource annotation is dependent: the resource use of range depends on the values of lo and hi.

The challenge in this case is to extend ReSyn to be able to *infer* these liquid resource types. In the spirit of RaML, this boils down to finding the minimum potential needed to account for all the resource use in a program; this is equivalent to finding minimal top-level resource annotations. Thus, our approach must be able to synthesize new resource annotations, some of which could contain conditional expressions and program variables, and optimize these synthesized resource annotations. Together, these constitute an inference system which could automatically infer the most precise resource bounds ReSyn can currently verify on examples like range.

This is less trivial than it seems. For one, there are many possible resource expressions, and we need a strategy for efficiently generating resource expressions based on the structure of the given program. Additionally, there is no clear formal way of ordering resource expressions. For instance, out of the expressions `if a < b then 1 else 0` and `if a < b then (if c < d then 1 else 0) else 0`, how can we tell which is "smaller?" We present two extensions to ReSyn in order to overcome these hurdles, allowing it to infer the resource use of a much wider array of programs.

4 ADDING A DEPENDENT OPTIMIZATION LOOP

In order to verify that a resource bound is valid for a given cost model of a program, ReSyn must ensure that the provided potential can be distributed to all of the operations that need it. ReSyn does this by first introducing *resource unknowns*: stand-ins for resource expressions which will be synthesized during the constraint solving process. ReSyn then generates constraints over these unknowns, corresponding to using and splitting resources across the program.

However, this process requires that we already have a top-level resource annotation for our program. Additionally, the resource constraint solver only checks for that there is *some* valid assignment for these resource unknowns under a set of resource constraints; no machinery exists to *optimize* these resource unknowns. Utilizing existing CLIA solvers would also be inadequate, as they are too slow to find valuations for these resource unknowns in a reasonable amount of time, since we're quantifying over all possible inputs, and don't support optimization.

Our first insight is that, given a set of dependent resource expressions, we can create a simple optimization loop which requires that on each iteration, no resource unknown has a larger value than it did previously for all inputs, and at least one resource unknown has a smaller value for some input. We dub this technique *dependent optimization*. Specifically, before starting optimization, we introduce top-level unknowns as resource annotations that we will solve for during optimization. For instance, with range, we introduce four top-level resource unknowns, I1 through I4:

$$\mathsf{range} :: lo : \mathsf{Int}^{I1} \to hi : \{\, \mathsf{Int}^{I2} \mid \nu \geq lo \,\} \to (\mathsf{List\ Int}^{\mathsf{I3}})^{I4}$$

On each iteration of the optimization loop, we first run ReSyn's resource constraint solver to find a valuation for our resource unknowns before repeatedly attempting to find a better resource bound;

"better" means that one of the resource unknowns gets smaller for at least one set of program variable values, and none of the resource unknowns get larger for all program variable values. When this fails, we know that we have the tightest upper bound.

For range, there is one recursive call for each difference between hi and lo. Thus, the main resource constraint of interest is that there's enough potential to pay for each of these calls: $I1 + I2 \geq hi - lo$. In the case that the resource solver initially finds a non-optimal assignment for the resource unknowns which satisfies this constraint (*e.g.* $I2 = 2v - lo$, $I1 = I3 = I4 = 0$), on the next iteration of the optimization loop, we would add the constraint that for all inputs, none of the unknowns got worse, and there exists an input where one or more unknowns improve. For brevity, we only include the added optimization constraints on I1 and I2:

$$((\forall v \in \mathbb{Z}.\ I1 \leq 0) \wedge (\forall v, lo \in \mathbb{Z} \times \mathbb{Z}.\ I2 \leq 2v - lo)) \wedge (\exists v \in \mathbb{Z}.\ I1 < 0 \vee \exists v, lo \in \mathbb{Z} \times \mathbb{Z}.\ I2 < 2v - lo)$$

This would eventually lead the resource solver to find the optimal assignment for $I1$: $I1 = v - lo$.

We have implemented a prototype of the dependent optimization loop within ReSyn which can infer the resource use of many existing ReSyn verification tests, including the asymptotic bounds of several sorting algorithms and several tests translated from RaML. Our prototype can also infer the resource use of the above range example, which wasn't possible with previous inference systems.

5 FUTURE WORK: GENERATING CONDITIONAL RESOURCE EXPRESSIONS

While ReSyn can synthesize resource polynomials, it cannot synthesize conditional expressions, and for good reason. Conditional expressions can be arbitrarily nested, and each can contain one of many possible boolean expressions in their guards, so naive enumeration would be extremely inefficient. This is one reason why existing CLIA solvers are impractically slow for this use case.

In order to address this, our second insight is that the conditional structure of our generated resource bounds should mirror the conditional structure of the corresponding program. Thus, any conditional expressions in our generated resource expressions should have the same level of nesting and should have similar guards as the conditionals in our program. We propose a future extension to our dependent optimization system, dubbed *conditional structure inference*. When generating resource expressions, we first extract the general conditional structures of the functions in the program. We then use these conditional structures as templates for our generated resource expressions, substituting in in-scope variables as needed and generating fresh resource variables to replace the leaves of these conditional expressions. With these two extensions, our approach can infer a vast majority of the bounds that ReSyn can currently only verify.

REFERENCES

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *Computer Aided Verification*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, P. Madhusudan, and Sanjit A. Seshia (Eds.). Vol. 7358. Springer Berlin Heidelberg, Berlin, Heidelberg, 781–786. https://doi.org/10.1007/978-3-642-31424-7_64 Series Title: Lecture Notes in Computer Science.

Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2019*. ACM Press, Phoenix, AZ, USA, 253–268. https://doi.org/10.1145/3314221.3314602

Tristan Knoth, Di Wang, Adam Reynolds, Jan Hoffmann, and Nadia Polikarpova. 2020. Liquid resource types. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 1–29. https://doi.org/10.1145/3408988

Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *Programming Languages and Systems*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Matthias Felleisen, and Philippa Gardner (Eds.). Vol. 7792. Springer Berlin Heidelberg, Berlin, Heidelberg, 209–228. https://doi.org/10.1007/978-3-642-37036-6_13 Series Title: Lecture Notes in Computer Science.