Resource-guided Program Synthesis

Tristan Knoth
tknoth@eng.ucsd.edu
UC San Diego
Advisor: Nadia Polikarpova

Introduction

In recent years, *program synthesis* has emerged as a promising technique for automating low-level aspects of programming [2, 19, 20]. Automatically optimizing program performance has long been a goal of synthesis, as real-world developers must take more than just the correctness of their code into account during the development process. Several existing techniques tackle this problem for *low-level programs* of restricted form [18, 15]. However, recent systems for the synthesis of *high-level* looping or recursive programs manipulating custom data structures do not take performance into account, instead simply returning the first program that satisfies the functional specification [12, 13, 1, 16, 10, 17]. Modern synthesis algorithms lack the means to analyze and understand the resource needs of these high-level programs. As a result, synthesized programs are prone to be inefficient, requiring the intervention of experienced developers to tune their performance.

In this work, we study the problem of program synthesis given *both* a functional specification of a program and a bound on its resource usage. Since both verification and synthesis are very expensive, we propose *resource-guided synthesis*: an approach that tightly integrates program synthesis and resource analysis, and uses the resource bound to guide the synthesis process, generating programs that are efficient by construction.

As an example, consider the programs in Fig. 1, both returning the intersection of two sorted lists. The implementation on the left, synthesized from a refinement-type specification by Synquid [16], is concise yet runs in quadratic time. Our technique guides the synthesizer to instead find the longer, yet linear-time, implementation shown on the right.

Our primary insight is that program synthesis via round-trip type checking, pioneered by the Synquid synthesizer [16], provides an extensible framework for type-directed synthesis. This allows us to leverage the existing body of work on resource type systems when incorporating resource analysis into the synthesis process [8, 4, 6, 14, 13, 16]. The main technical contribution of this project is a novel bidirectional type system, Re^2 (for *re*finements and *re*sources), allowing users to simultaneously specify both functional properties and resource bounds. We show how to then transform this bidirectional system into a synthesis algorithm, and use it to implement ReSyn, capable of synthesizing efficient functional programs.

The Re^2 Type System

Re^2 draws its inspiration primarily from Automatic Amortized Resource Analysis (AARA), an automated technique for deriving symbolic resource bounds on functional programs [9, 11, 3, 5]. Like AARA, Re^2 is affine in order to ensure that it infers strict upper bounds on resource usage.

AARA utilizes potential functions that map program states to non-negative numbers. To derive a bound, one must statically ensure that the potential at every program state is sufficient to cover the cost of the next transition and the potential of the subsequent state. Then, the top-level potential is an upper bound on the program's cost. To automate the technique, AARA uses template potential functions with a priori unknown coefficients, solved for by generating a system of linear constraints. [7, 5].

AARA is an excellent candidate for use in a program synthesizer, as the verification process is completely automated and it generates decidable and efficiently solvable constraints. However, AARA does lack two key features vital for working alongside a refinement type system to specify and verify general-purpose functions:

Polymorphism: Much of the expressive power and flexibility of refinement types comes from polymorphism, as it enables higher-order reasoning, and allows users to provide general-purpose components to the synthesizer. Polymorphic types also make the analysis compositional in a way that RAML's is not. For example, consider the following function that appends three lists:

```
append3 =λxs . λys . λzs . append xs (append ys zs)
```

To analyze the above, RAML will conduct a whole-program analysis and consider the body of `append` twice. Since Re^2 is polymorphic, `append` can have a single type signature that is instantiated twice with different

```
common l1 l2 = match l1 with
  Nil → Nil
  Cons x xs →
    if ¬(member x l2)
      then common xs l2
      else Cons x (common xs l2)
```

```
common l1 l2 = match l1 with
  Nil → Nil
  Cons x xs → match l2 with
    Nil → Nil
    Cons y ys → if x < y
      then common xs l2
      else if ts y < ts x
        then common l1 ys
        else Cons x (common xs ys)
```

Figure 1: Two synthesized programs that collect common elements between two sorted lists; a short solution from Synquid (left) and an efficient solution from ReSyn (right).

resource demands, making the analysis more modular and generating a smaller system of constraints than RAML would.

To support polymorphism we annotate type variables with multiplicities in the tradition of linear logic. We can then perform potential-preserving type substitution, and thus compose resource bounds in a way that AARA could not.

Dependent annotations: Users should be able to express resource bounds with respect to variables and other program expressions used to specify functional properties. For example, analyzing the resource usage of a simple function like `replicate` :: `n`: `Nat` → `x`: `a` → {`List a` | `elems` $\nu =$ `[x]` $\wedge$ `len` $\nu =$ `n`}, which returns a list of `n` copies of some expression `x`, is outside the scope of AARA. However, Re^2 should be able to reason about dependent annotations, so we allow potential annotations ranging over linear combinations of program expressions.

As an example Re^2 specification, consider the following resource-annotated type signature for `common`, a function returning the intersection of two lists:

`common :: l1: IList {a | | 1}` → `l2: IList {a | | 1}` $\xrightarrow{1}$ {`List a` | `elems` $\nu =$ `elems l1 * elems l2`}

`common`'s type signature specifies a number of properties, both functional and resourceful:

- Both input lists are sorted – the `IList` type, while elided here, includes an invariant asserting that the head of the list is bounded above by every element in the tail.
- The function's return type asserts that the set of elements in the output list is the intersection of the elements of the two input lists (denoted with $*$).
- Any call to `common` incurs a cost of "1", indicated by the annotation on the function arrow. Here, the resource in question is simply recursive calls.
- Every element in the input lists carries a single unit of potential, indicated by the "1" annotation on the type variables. This essentially gives us enough potential to pay for one recursive call per element in each input list. A valid `common` implementation could make *at most* $|l1| + |l2|$ recursive calls according to this specification.

Given annotations of this form, checking resource bounds with Re^2 reduces to solving a system of quantified linear arithmetic expressions. Our syntax-directed typing rules generate constraints asserting that the context has enough potential to pay for every operation. In most cases, we can solve the system directly with a single solver query. However, when the resource bounds include program expressions, Re^2 generates exists-forall constraints, asserting that there exists a program expression greater than or equal to the relevant cost for all possible inputs. Solving constraints of this form is notoriously difficult, and in fact amounts to a separate program synthesis problem all together [19]. We solve the system with a standard counterexample-guided inductive synthesis (CEGIS) loop, which in practice is not too costly since the desired expressions are restricted to linear combinations of program terms.

From Verification to Synthesis with ReSyn

Like Synquid, we turn our bidirectional Re^2 rules into a round-trip system by propagating additional type information during the inference phase of the bidirectional system. This allows us in turn to implement ReSyn on top of SYNQUID's synthesis engine. [16].

In its current state, RESYN can analyze linear resource bounds, and on our benchmark suite, finds a linear-time implementation of every function for which SYNQUID could not. RESYN is slightly slower than SYNQUID, which is unsurprising given the additional constraint-solving burden and slightly reduced search pruning. However, RESYN massively outperforms naive approaches to the problem, and is the first high-level general-purpose program synthesizer that can reason about resource usage.

The primary technical challenge in program synthesis from Re^2 specifications is optimizing the search process after relaxing the assumptions upon which SYNQUID relied. For example, consider the issue of synthesizing a conditional. Rather than naively enumerating all branching expressions, SYNQUID instead searches for terms assuming some "condition unknown" C. Given a valid term, SYNQUID then searches for the weakest possible valuation for C under which the expression verifies.

Having found an expression and a guard condition, it is straightforward to convert the condition into a program expression and continue to synthesize the other branches.

The process, called condition abduction, is not so simple under Re^2, however, as condition abduction implicitly assumes that the order of checking premises does not matter. Since Re^2 is affine, each step of the type checking process can affect the context. In Re^2, to check a conditional expression, we first check the guard and then use the resulting context to check each branch, ensuring that the top-level context contains enough potential to execute the guard followed by any of the branches. Thus, the order in which we synthesize expressions in a branching term matters under Re^2, while in SYNQUID it did not. In RESYN, we extend condition abduction to our substructural type system by re-checking certain expressions. For example, suppose we synthesize an expression e from a context Γ. If e only checks under a condition C, we then generate the guard expression g in Γ again, leaving behind the context Γ'. We must then re-check $\Gamma' \vdash e$ in order to ensure that the context has enough potential to evaluate g and e. We can then proceed to generate the remaining branches in Γ'.

SYNQUID also assumes that any expression that type checks will appear in the final program, perhaps behind a guard. In RESYN, this is no longer true, as it might synthesize an expression with the desired functional properties, but where any program containing said expression exceeds the resource budget. Thus, RESYN must often explore possible programs that SYNQUID never did, often again hurting its performance relative to SYNQUID's.

References

[1] FESER, J. K., CHAUDHURI, S., AND DILLIG, I. Synthesizing data structure transformations from input-output examples. In *Programming Language Design and Implementation (PLDI)* (2015).

[2] GULWANI, S., HARRIS, W. R., AND SINGH, R. Spreadsheet data manipulation using examples. *Commun. ACM 55*, 8 (Aug. 2012), 97–105.

[3] HOFFMANN, J. *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis.* PhD thesis, Ludwig-Maximilians-Universität München, 2011.

[4] HOFFMANN, J., AEHLIG, K., AND HOFMANN, M. Multivariate Amortized Resource Analysis. In *38th Symp. on Principles of Prog. Langs. (POPL'11)* (2011), pp. 357–370.

[5] HOFFMANN, J., AEHLIG, K., AND HOFMANN, M. Multivariate Amortized Resource Analysis. In *38th Symposium on Principles of Programming Languages (POPL'11)* (2011).

[6] HOFFMANN, J., AEHLIG, K., AND HOFMANN, M. Resource Aware ML. In *24rd International Conference on Computer Aided Verification (CAV'12)* (2012), vol. 7358 of *Lecture Notes in Computer Science*, Springer, pp. 781–786.

[7] HOFFMANN, J., AND HOFMANN, M. Amortized Resource Analysis with Polynomial Potential. In *19th European Symposium on Programming (ESOP'10)* (2010).

[8] Hoffmann, J., and Hofmann, M. Amortized Resource Analysis with Polynomial Potential - A Static Inference of Polynomial Bounds for Functional Programs. In *In Proceedings of the 19th European Symposium on Programming (ESOP'10)* (2010), vol. 6012 of *Lecture Notes in Computer Science*, Springer, pp. 287–306.

[9] Hofmann, M., and Jost, S. Type-Based Amortised Heap-Space Analysis. In *15th Euro. Symp. on Prog. (ESOP'06)* (2006), pp. 22–37.

[10] Inala, J. P., Polikarpova, N., Qiu, X., Lerner, B. S., and Solar-Lezama, A. Synthesis of recursive ADT transformations from reusable templates. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I* (2017), pp. 247–263.

[11] Jost, S., Hammond, K., Loidl, H.-W., and Hofmann, M. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *37th ACM Symp. on Principles of Prog. Langs. (POPL'10)* (2010), pp. 223–236.

[12] Kneuss, E., Kuraj, I., Kuncak, V., and Suter, P. Synthesis modulo recursive functions. In *OOPSLA* (2013).

[13] Osera, P., and Zdancewic, S. Type-and-example-directed program synthesis. In *PLDI* (2015).

[14] Peng Wang, Di Wang, A. C. Timl: A functional language for practical complexity analysis with invariants. In *OOPSLA* (2017).

[15] Phothilimthana, P. M., Thakur, A., Bodík, R., and Dhurjati, D. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016* (2016), pp. 297–310.

[16] Polikarpova, N., Kuraj, I., and Solar-Lezama, A. Program synthesis from polymorphic refinement types. In *Programming Language Design and Implementation (PLDI)* (2016), pp. 522–538.

[17] Qiu, X., and Solar-Lezama, A. Natural synthesis of provably-correct data-structure manipulations. *PACMPL 1*, OOPSLA (2017), 65:1–65:28.

[18] Schkufza, E., Sharma, R., and Aiken, A. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013* (2013), pp. 305–316.

[19] Solar-Lezama, A. Program sketching. *STTT 15*, 5-6 (2013), 475–495.

[20] Torlak, E., and Bodík, R. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014* (2014), p. 54.