A Divide-and-Conquer Approach to Discovering Minimal Realizable Grammars

Will Thomas, Logan Schmalz, Sarah Johnson

Abstract

Motivating Ideas:

- Given a *sufficient* grammar, program synthesis *should* be faster on a smaller grammar than a larger one
- Given a grammar, determining unrealizability of a synthesis problem is fast

Goal:

- Modify the SemGuS implementation (Messy) to achieve better synthesis performance
  - Messy makes a call out to Z3 to determine realizability under the full grammar
  - Our project makes a call out to Z3 to determine realizability under each subgrammar

Method:

- Split a grammar into all possible subgrammars
- Convert the synthesis problem over each subgrammar to a proof search over CHCs
- Send each problem to Z3 from smallest to largest grammar until it returns realizable

Future Work:

- When Z3 returns realizable or takes a “long” time to prove unrealizability on the current subgrammar, attempt synthesis using a state-of-the-art tool on the current subgrammar

Background Info

Semantic-Guided Synthesis (SemGuS)

- A language-agnostic, logic-based framework for program synthesis problems over arbitrary semantics
- A SemGuS problem is defined using three components
  - Language syntax: Regular Tree Grammar
  - Language semantics: Constrained Horn Clauses
  - Problem specification: Constraints/Examples
- Accepts recursively defined big-step semantics
  - Allows synthesis over imperative programming languages
- Encodes problem as a proof search over CHCs
- Implementation in Scala called Messy

Very Simple Example

Problem:

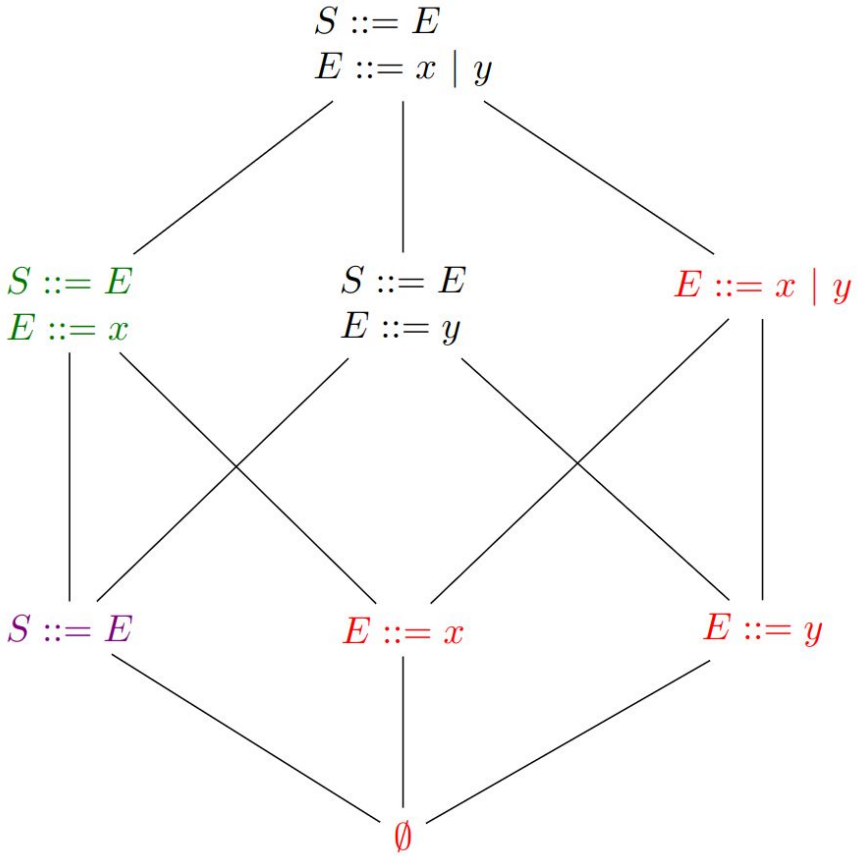
- Find a minimal realizable grammar to synthesis a function that return the first projection of an ordered pair

I/O Example:

- f(5,3) = 5

Subgrammars:

- The power set of the production rules forms a lattice
- The full grammar is the top element
- Red text denotes invalid grammars due to missing start symbol
- Purple text denotes invalid grammars due to missing nonterminal symbol(s)
- Green text denotes the minimal realizable grammar given the input-output example



Implementation Discussion

- Modifications to an existing implementation of Messy in Scala
- We added two modes NT and PROD:
- NT Mode:
  - Remove whole Non-Terminals from the Grammar (S, E)
  - NT Mode is faster, but much more coarse grained in its approach.
- PROD Mode:
  - Removal specific Production Rules from the Grammar (S -> E, E -> x, E -> y)
  - PROD mode is slower, but much more powerful in the sub-grammars it can find
- 2^N combinations must be checked, so the fewer variables (Non-Terminals or Production Rules) the better

$$S ::= E$$

$$E ::= x \mid y$$

Implementation Primary Algorithm

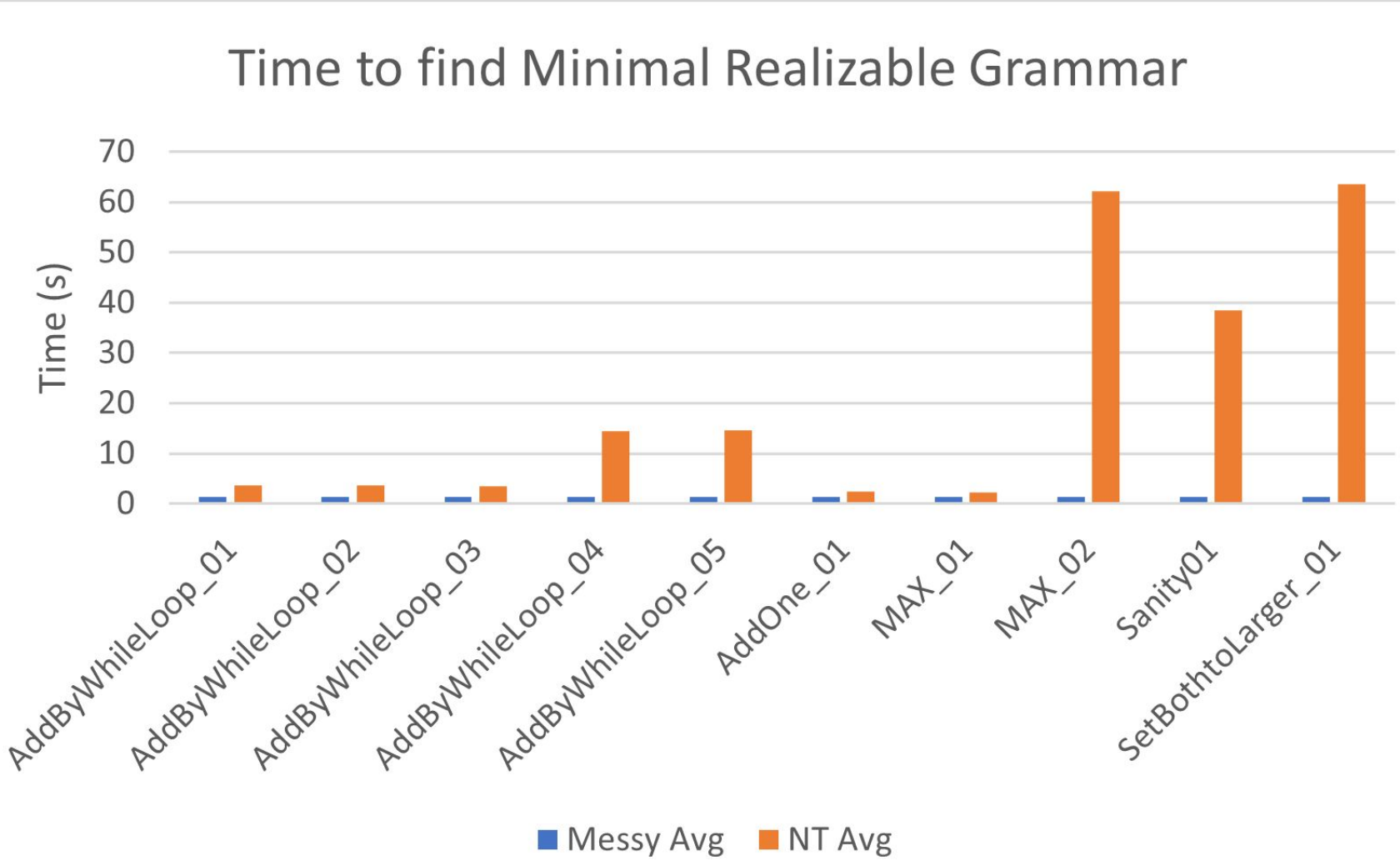
Algorithm 1: Minimal Sub-Grammar

Input: l : List[SpecEvents]
Output: l' : Option[List[SMTCommand]]

```
1  var declT, defT, hornC, constrs, synthFun = l.filter(classTypeFilter())
2  var requiredEvs = constrs ::: synthFun
3  var combinedEvs = defT ::: hornC
4  for i ← 1 to length(combinedEvs) do
5  |   for part ∈ combinedEvs.combinations(i) do
6  |   |   var smtEncoding = semgus2SMT(part ::: requiredEvs)
7  |   |   if checkSat(smtEncoding) then
8  |   |   |   return Some(smtEncoding)
9  |   |   end
10 |   end
11 end
12 Otherwise we return None;
```

Evaluation

- Unfortunately, both our NT and Prod approaches are slower than the original Messy Solver
- Additionally, our approaches are limited in exactly the same ways, cannot parse things that Messy cannot (despite being in Messy benchmark)
- However, our Prod approach was able to achieve a minimal sub-grammar that was smaller than the full grammar.
- We had to invalidate 8 tests from the 18 “Messy benchmarks” due to the fact that Messy itself could not parse them to even attempt solving.
    - The exact reasons that the benchmarks are not parsable by the solver they are designed for is certainly odd


Time to find Minimal Realizable Grammar
Time (s)
70
60
50
40
30
20
10
0
AddByWhileLoop_01
AddByWhileLoop_02
AddByWhileLoop_03
AddByWhileLoop_04
AddByWhileLoop_05
AddOne_01
MAX_01
MAX_02
Sanity01
SetBothtoLarger_01
Messy Avg
NT Avg

Production Rule Mode Results

- Quite slow so we did not profile all tests
- Able to generate a smaller realizable grammar
- For test AddByWhileLoop_1:
  - In 434 seconds we are able to generate a grammar that uses 11 less variables, 2 less relations, and 7 less semantic rules, but is still realizable
  - This reduction implies that we are able to get rid of certain production rules, but not the non-terminals related to those production rules

Issues Encountered

- Attempt 1: Translate Messy release into OCaml
  - Trouble with subclasses and subtyping in OCaml
- Attempt 2: Translate Messy release into Python
  - Realized Messy uses a custom parsing library only written in Java
- The Messy parser cannot parse many of the Messy Benchmarks

Conclusion

- We were able to find smaller grammars to solve problems
- We need optimizations to reduce the number of subgrammars that are checked
  - A grammar without a start symbol or a grammar using a nonterminal with no productions is invalid and this should not be too hard to check
  - We were unable to do this due to the current representation in Messy being hard to work with

Future Work

- Optimizations
- Galois connection between grammars and their languages could help determine which grammars would yield the most information
- Incorporate an actual synthesizer after we have found our minimal realizable grammar
- Improve the Messy parser

Bibliography

Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. 2021. Semantics-guided synthesis. Proc. ACM Program. Lang. 5, POPL, Article 30 (January 2021), 32 pages. https://doi.org/10.1145/3434311

Bonus Slides: The Galois Connection