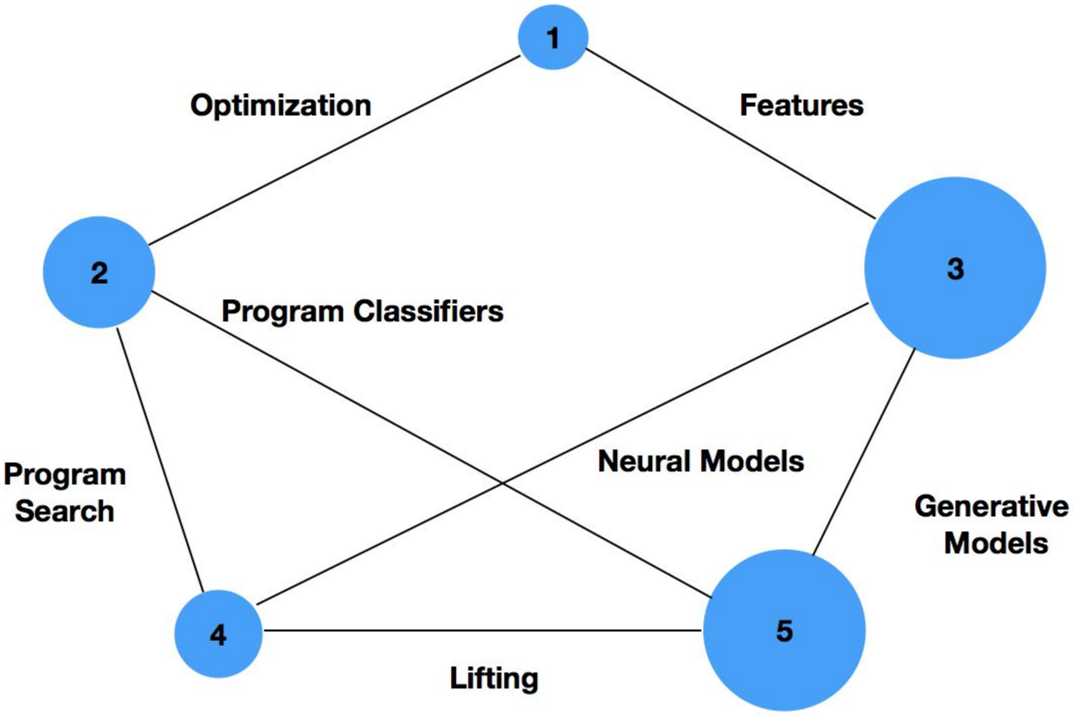
Rethinking Compilation: L4


1
2
3
4
5
Optimization
Features
Program Classifiers
Neural Models
Program
Search
Generative
Models
Lifting

Agenda

1. What is program synthesis?
2. Synthesis in compilers context
3. Fundamental techniques
4. Pruning strategies
5. The future
6. Bibliography

What is program synthesis?

What is program synthesis?

- Synthesis is **NOT**:
  - Compilation
  - Machine Learning

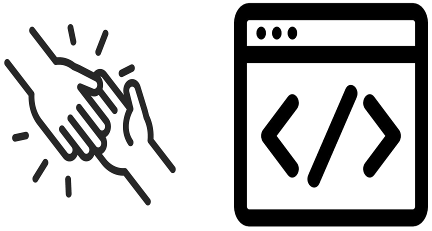
- According to Solar-Lezama **[1]**

> ***“Program Synthesis correspond to a class of techniques that are able to generate a program from a collection of artifacts that establish semantic and syntactic requirements for the generated code. ”***

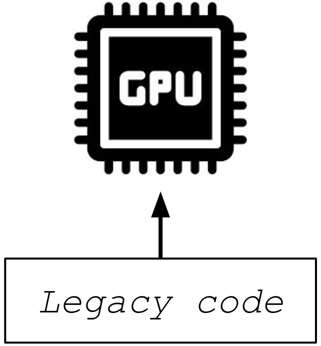
What is program synthesis?

- WHERE do we want to automatically generate software?

Data wrangling



Coding assistance



Program Lifting

Program → x86 → Hardware

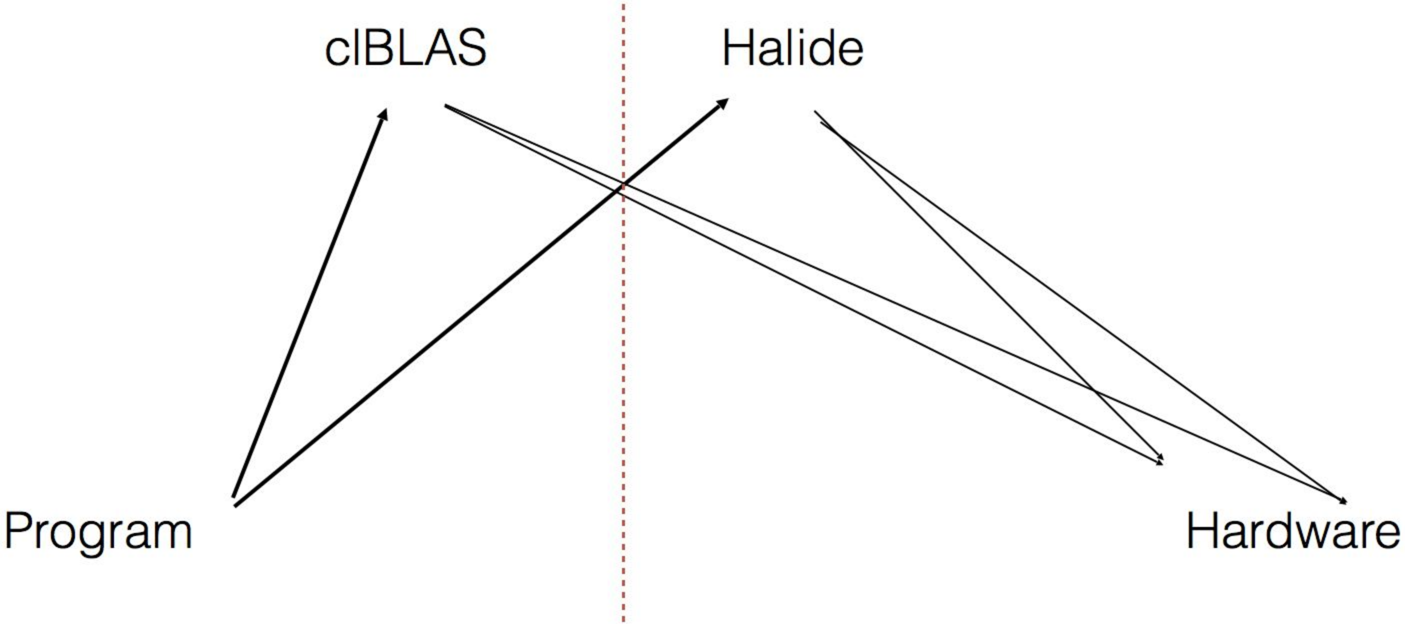
Program → OpenCL → Hardware

Program → clBLAS → Hardware

→ Halide → Hardware

→

LIFT code to API or DSL

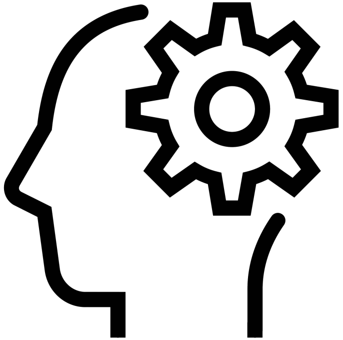


Vendor responsibility to map API/DSL to hardware - already the case

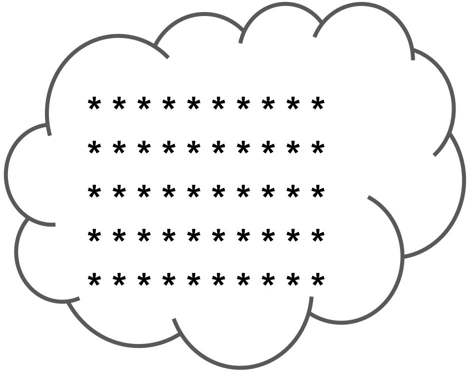
Our job - automatically lift it to API/DSL enabling hardware utilisation

What is program synthesis?

- WHY IS IT HARD to automatically generate software?
- Writing software:



Demands expertise



Search space is enormous

Hard to ensure correctness

What is program synthesis?

"Program Synthesis correspond to a class of techniques that are able to generate a program from a collection of artifacts that establish semantic and syntactic requirements for the generated code. "

Input/Output exs

```
in: [1,2,3,4], [2,5,7,8]
out: [5,11,16,20]
```

Logic specification

```
post(a, b) ≡ ∀ imin+1 ≤ i ≤ imax,
jmin ≤ j ≤ jmax.
a(i,j) = b(i-1,j) + b(i,j)
```

Grammar

```
<prog> ::= <var> = <expr>
<exp> ::= <exp> + <exp>
<exp> ::= <exp> * <exp>
<exp> ::= ( <exp> )
<var> ::= a | b | c
```

What is program synthesis?

1932

Constructive mathematics

Constructing algorithms by composition

60's - 70's

Rise of deductive synthesis

Deductive approaches generate programs out of formal specifications This rise leveraged automated theorem provers

2000's

Inductive synthesis systems

Program synthesis has a boom especially due to inductive reasoning, i.e., search for a program in a possible space

2013

FlashFill available on Microsoft Excel 13

Perhaps this is the best example of synthesis successfully applied to industry. FlashFill generates formulas given a few examples from the user.

Today

Spotlight and interdisciplinarity

Synthesis becomes a recurrent topics in top conferences on fields of programming languages (PLDI, POPL), formal systems (CAF, TACAS) and machine learning (NeurIPS, ICML). Algorithms are combined with other areas such as Neural networks

Synthesis in compilers context

Synthesis in compiler context

- High-level code translation

```
def max(a,b):
    return a if a > b else b
```

→

```
int max(int a, int b)
{
    return a > b ? a : b;
}
```

Synthesis in compiler context

- High-level code translation for:

Security:

Performance:

NVIDIA CUDA

Maintenance:

COBOL

Java

Synthesis in compiler context

- High-level code translation involves

Expertise

Time

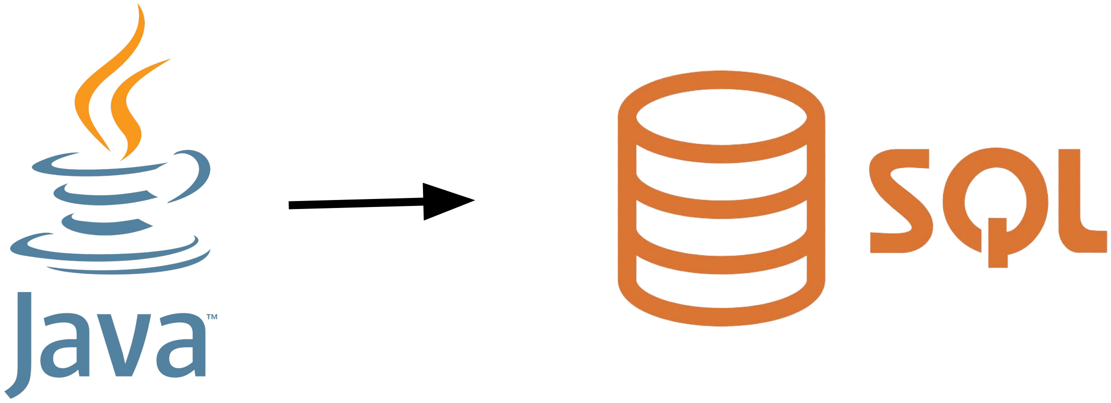
Costly

“Commonwealth Bank of Australia spent around $750 million and 5 years of work to convert its platform from COBOL to Java” **[2]**

Synthesis can automate tasks

Synthesis in compiler context

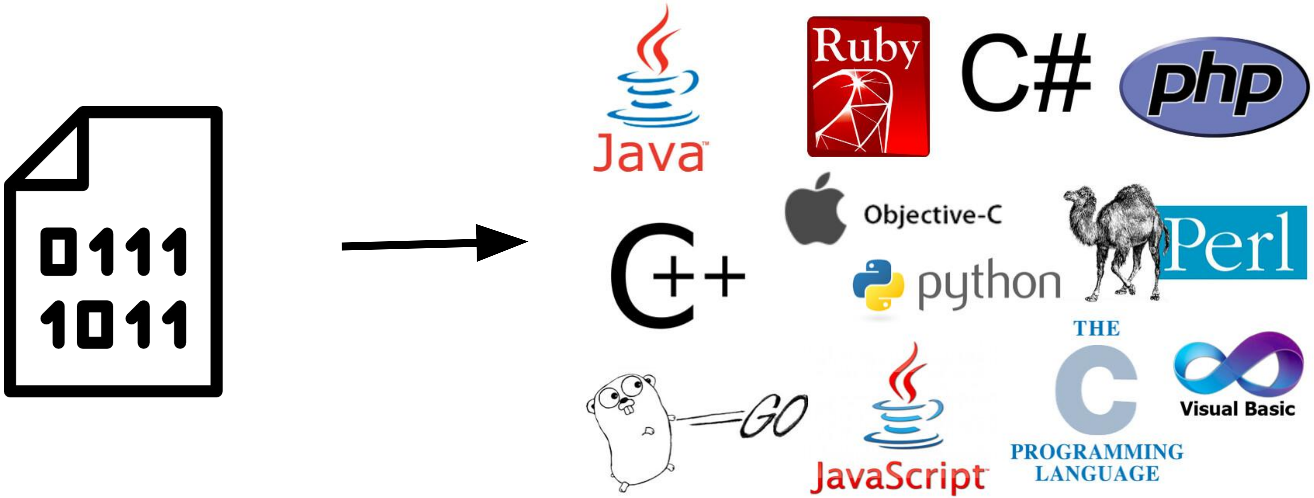
- Optimizing Database-Backed Applications with Query Synthesis (PLDI' 2013) **[3]**



- Synthesis from verification conditions
- Performance obtained

Synthesis in compiler context

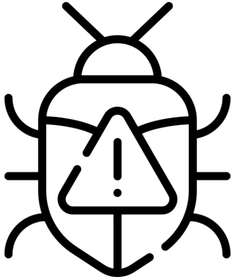
- Decompilation



Synthesis in compiler context

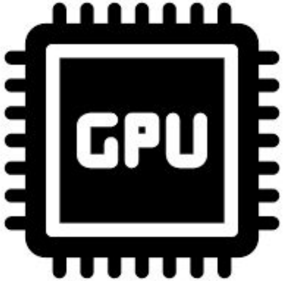
- Decompilation:

  Malware detection:

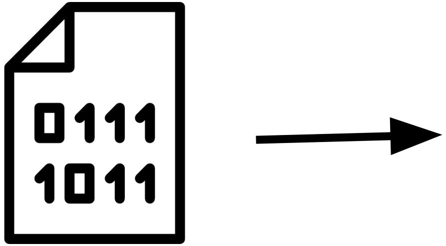
  Port code to new devices:

Synthesis in compiler context

- Helium: Lifting High-Performance Stencil Kernels from Stripped x86 Binaries to Halide DSL Code (PLDI' 2015) **[4]**




```
RDom r_0(input);
output(input(r_0.x,r_0.y)) =
cast<uint64_t>(output(input(r_0.x,r_0.y)) + 1);
```

- Synthesis leveraging binary traces
- 75% performance on Adobe PhotoShop

Synthesis in compiler context

- Api migration
- Maintenance
- Usability

Synthesis in compiler context

- SOAR: A Synthesis Approach for Data Science API Refactoring (ICSE' 2021) **[5]**

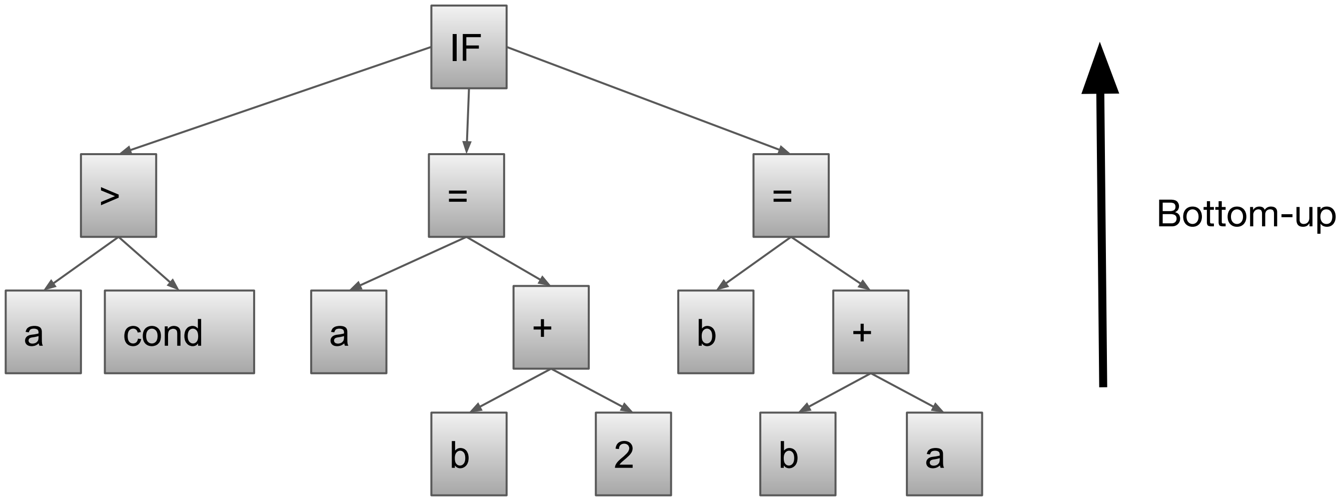
- Synthesis with NLP

Fundamental techniques

Fundamental techniques - Bottom-up

- Bottom-up search
  - Starts from small constructs
  - Combine them into larger programs

```
if(a > cond){
  a = b + 2
}else{
  b = b + a
}
```



Fundamental techniques - Bottom-up

Algorithm Bottom-up Enumerative Synthesis

```
function BOTTOM-UP(grammar, specification)
    c_list ← TERMINALS(grammar)
    while true do
        for r ∈ RULES(grammar) do
            c_list ← EXPAND(c_list, r)
        end for
        c_list ← ELIM_EQUIVALENTS(c_list)
        for c ∈ c_list do
            if IS_CORRECT(c, specifciation) then
                return c
            end if
        end for
    end while
end function
```

Fundamental techniques - Bottom-up

- Find the maximum of two numbers

<program> ::= <expr>
<expr> ::= <expr> **+** <expr>
| <expr> **>** <expr>
| **if** <expr> **then** <expr> **else** <expr>
| <var>
| **1**
<var> ::= **x** | **y**

iteration

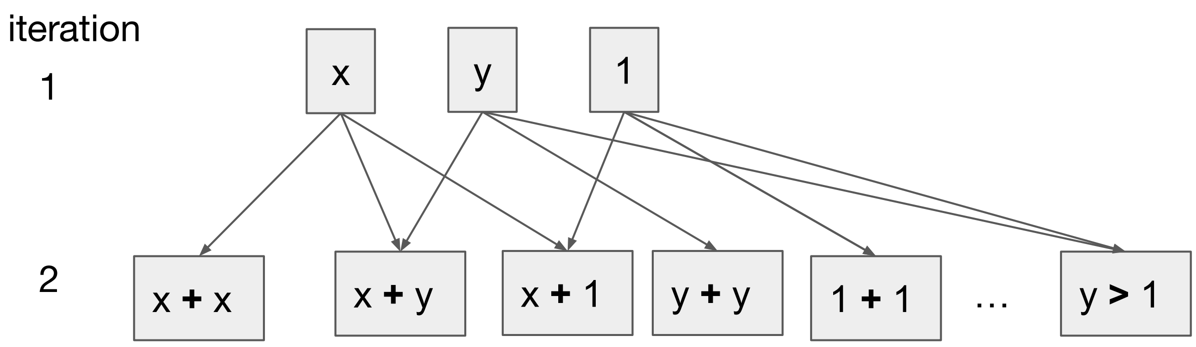
1

x y 1

Fundamental techniques - Bottom-up

- Find the maximum of two numbers

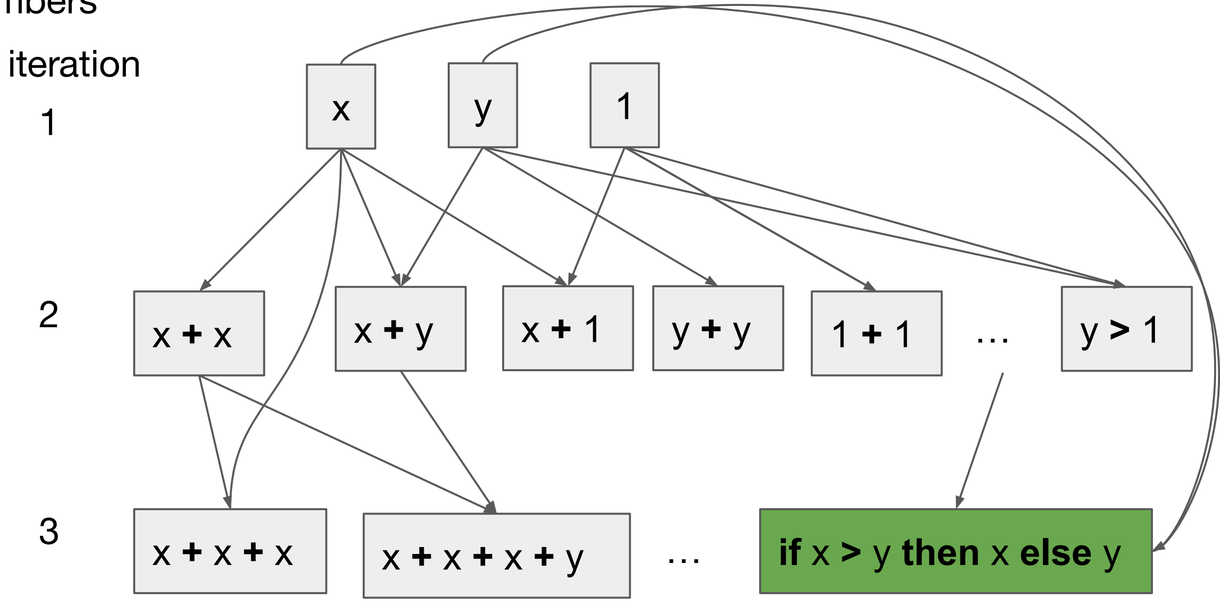
<program> ::= <expr>
<expr> ::= <expr> **+** <expr>
| <expr> **>** <expr>
| **if** <expr> **then** <expr> **else** <expr>
| <var>
| **1**
<var> ::= **x** | **y**



Fundamental techniques - Bottom-up

- Find the maximum of two numbers

<program> ::= <expr>
<expr> ::= <expr> **+** <expr>
| <expr> **>** <expr>
| **if** <expr> **then** <expr> **else** <expr>
| <var>
| **1**
<var> ::= **x** | **y**



Fundamental techniques - Bottom-up

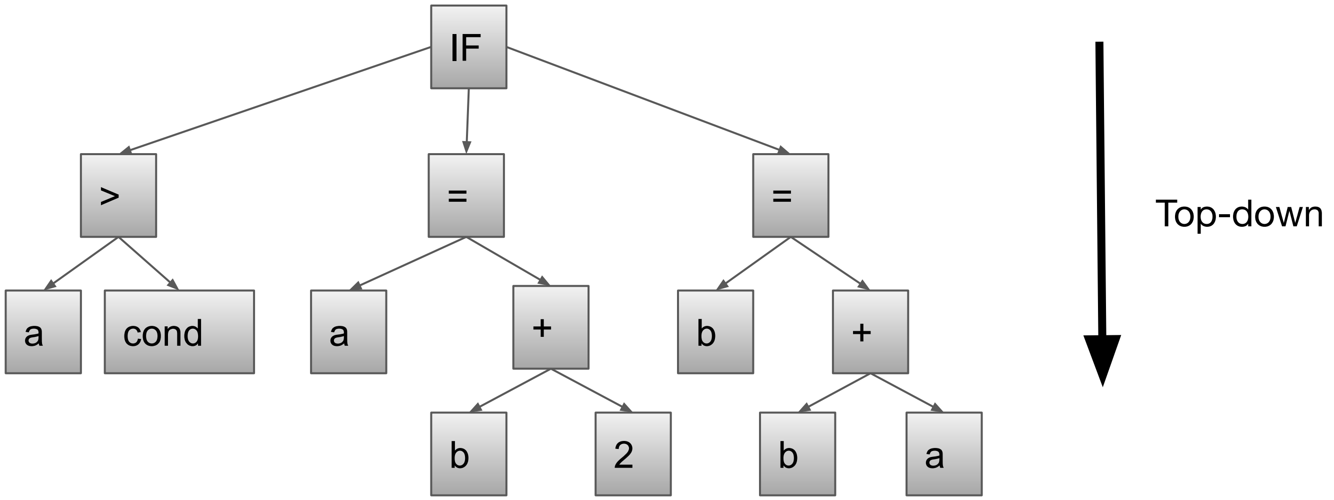
- Pros
  - Simple
  - Explores small programs first
  - Black-box language building

- Cons
  - Scalability
  - Poor performance for synthesizing constants

Fundamental techniques - Top-down

- Top-down search
  - Starts from high-level constructs
  - Fill the holes in temporary candidates

```
if(a > cond){
  a = b + 2
}else{
  b = b + a
}
```



Fundamental techniques - Top-down

Algorithm Top-down Enumerative Synthesis

```
function TOP-DOWN(grammar, specification)
    c_list ← ∅
    r_list ← RULES(grammar)
    while true do
        for r ∈ r_list do
            for t ∈ TERMINALS(grammar) do
                if r can expand to t then
                    c_list ← EXPAND(c_list, t)
                end if
            end for
            for r′ ∈ RULES(grammar) do
                if r can expand to r′ then
                    r_list ← EXPAND(r_list, r′)
                end if
            end for
        end for
        c_list ← ELIM_EQUIVALENTS(c_list)
        for c ∈ c_list do
            if IS_CORRECT(c, specifciation) then
                return c
            end if
        end for
    end while
end function
```

Fundamental techniques - Top-down

- Find the maximum of two numbers

<program> ::= <expr>
<expr> ::= <expr> **+** <expr>
| <expr> **>** <expr>
| **if** <expr> **then** <expr> **else** <expr>
| <var>
| **1**
<var> ::= **x** | **y**

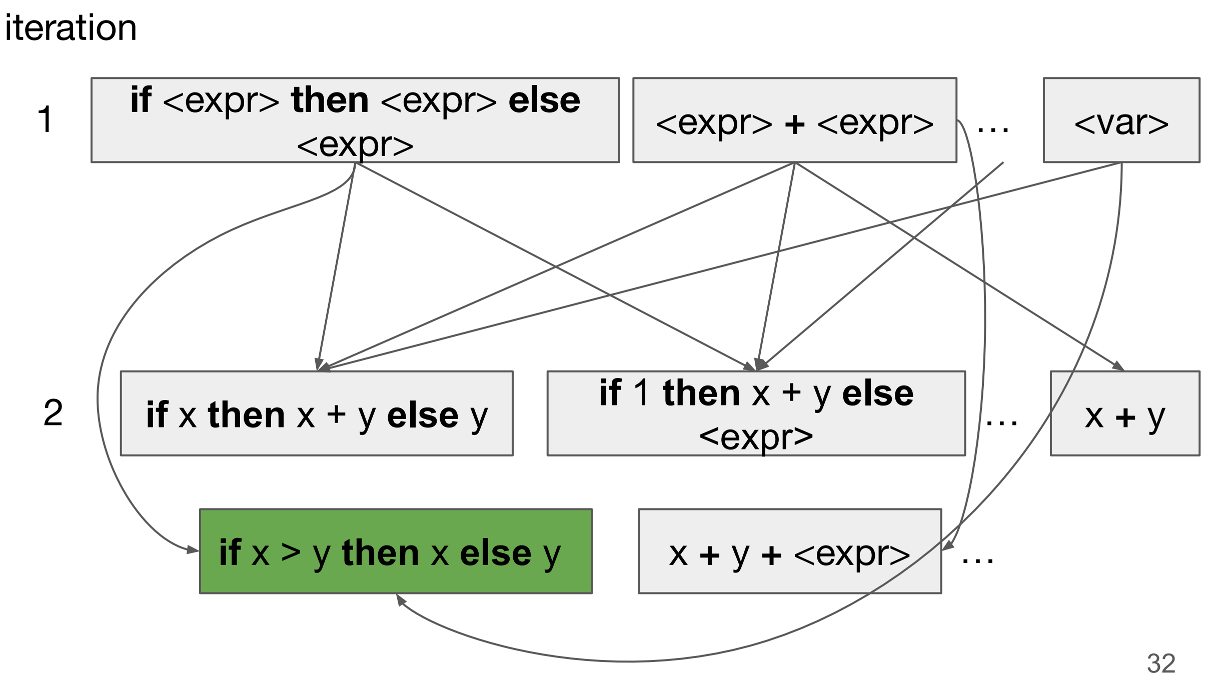
iteration

1	**if** <expr> **then** <expr> **else** <expr>	<expr> **+** <expr>	…	<var>

Fundamental techniques - Top-down

- Find the maximum of two numbers

<program> ::= <expr>
<expr> ::= <expr> **+** <expr>
| <expr> **>** <expr>
| **if** <expr> **then** <expr> **else** <expr>
| <var>
| **1**
<var> ::= **x** | **y**

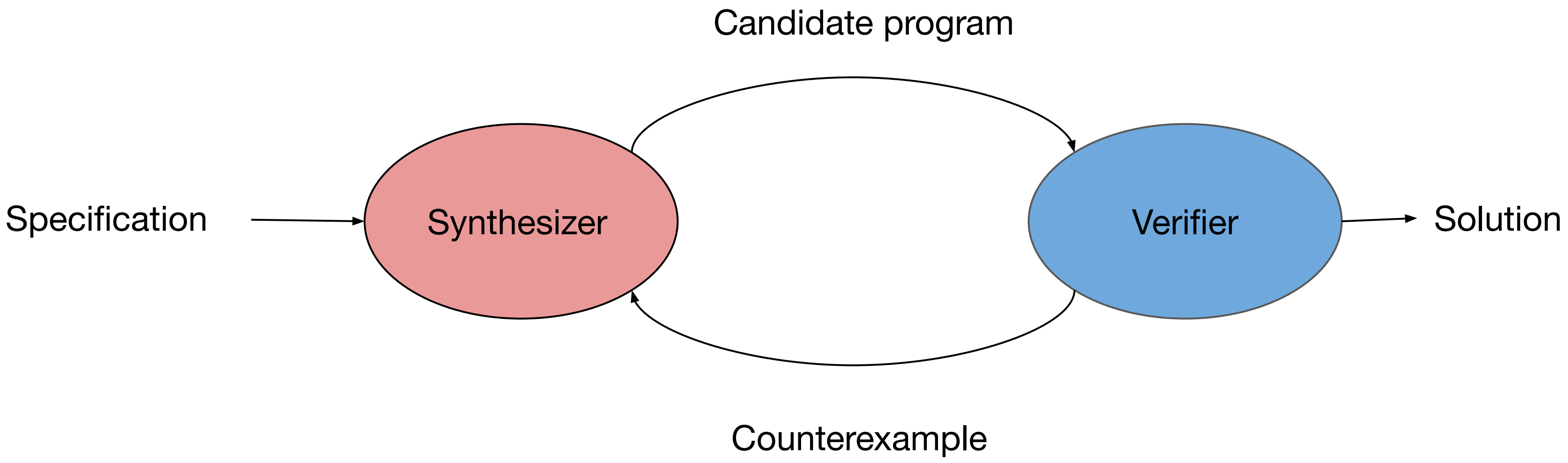


Fundamental techniques - Top-down

- Pros
  - Efficient with a good initial guess
  - Effective when dealing with recursion
- Cons
  - It is hard to execute temporary programs
  - Scalability

Fundamental techniques - CEGIS

- Counterexample Guided Inductive Synthesis **[6]**
- Synthesizer + Verifier
- Feedback to synthesizer



Fundamental techniques - Verification conditions

- Convert a program into a logical statement:
  - $\forall x.Q(x) \Rightarrow$ program is correct
- Verification conditions (vc's) can be used to synthesize programs
- Given a verification condition, the goal is to synthesize a predicate that makes such a verification valid
- Usually, VC's contain a *loop invariant*
- A loop invariant is a formula that must hold in every iteration of a loop

Fundamental techniques - Verification conditions

- Verified Lifting of Stencil Computations (PLDI' 2015) **[7]**
- The goal is to synthesize a verification condition that summarizes the computation above.

```
procedure sten(imin,imax,jmin,jmax,a,b)
  real (kind=8), dimension(imin:imax,jmin:jmax) :: a
  real (kind=8), dimension(imin:imax,jmin:jmax) :: b
  do j=jmin,jmax
    t = b(imin, j)
    do i=imin+1,imax
      q = b(i,j)
      a(i,j) = q + t
      t = q
    enddo
  enddo
end procedure
```

(a)

Fundamental techniques - Verification conditions

- Verification condition is formed by:
    - Pre-condition
    - Loop invariant
    - Pos-condition

- Such that:
    - $\forall s.\mathit{pre}(s) \rightarrow \mathit{invariant}(s)$
    - $\forall s.\mathit{invariant}(s) \wedge \mathit{cond}(s) \rightarrow \mathit{invariant}(\mathit{body}(s))$
    - $\forall s.\mathit{invariant}(s) \wedge \neg \mathit{cond}(s) \rightarrow \mathit{post}(s)$

Fundamental techniques - Verification conditions

- A pre-condition can be obtained by syntax-guided algorithms
  - Pre-condition (outermost loop)

$$I_j(\mathtt{a}, \mathtt{b}, \mathtt{jmin})$$

- Loop invariant and post-condition must be synthesized
- Verified lifting uses CEGIS with a SMT solver as verifier to find the last two

- Invariant:

$$\begin{aligned} invariant(\mathtt{a}, \mathtt{b}, \mathtt{j}) \equiv \mathtt{j} \le \mathtt{jmax} + 1 \ \wedge \\ \forall\, \mathtt{imin+1} \le \mathtt{i} \le \mathtt{imax},\ \mathtt{jmin} \le j' < \mathtt{j}. \\ \mathtt{a(i,}j'\mathtt{)} = \mathtt{b(i-1,}j'\mathtt{)\ +\ b(i,}j'\mathtt{)} \end{aligned}$$

(c)

- Post-condition (outermost loop):

$$\begin{aligned} post(\mathtt{a}, \mathtt{b}) \equiv \forall\, \mathtt{imin+1} \le \mathtt{i} \le \mathtt{imax},\ \mathtt{jmin} \le \mathtt{j} \le \mathtt{jmax}. \\ \mathtt{a(i,j)} = \mathtt{b(i-1,j)\ +\ b(i,j)} \end{aligned}$$

(b)

Fundamental techniques - Verification conditions

- Once loop invariant and post-condition are synthesized, SMT proves that the VC holds

- The VC is then translated to the target language

```
int main() {
  ImageParam b(type_of<double>(),2);
  Func func; Var i, j;
  func(i,j) = b(i-1,j) + b(i,j);
  func.compile_to_file("ex1", b);
  return 0; }
```

(d)

Fundamental techniques - Verification conditions

- Synthesized code is faster:

Benchmark	Kernel	Halide Speedup	icc Before Speedup	icc After Speedup	Halide GPU Speedup	Halide GPU Speedup (no transfer)
	akl81	7.44	1.73	4.53	4.09	10.88
	akl83	4.57	0.95	0.95	3.29	8.27
	akl84	4.51	0.71	0.94	2.94	7.88
	akl85	4.05	0.95	0.77	2.68	6.85
	akl86	4.04	0.97	0.79	2.27	6.00
	ackl95	4.19	1.00	1.00	1.94	7.47
	amkl100	3.84	0.93	0.89	1.57	6.72
	amkl101	3.64	1.01	1.00	1.52	5.44
	amkl103	3.42	0.93	0.93	1.99	6.77
	amkl105	3.37	0.98	0.98	1.50	5.17
	amkl107	3.95	0.94	0.96	2.31	8.71
	amkl97	4.19	1.00	1.01	1.96	6.98

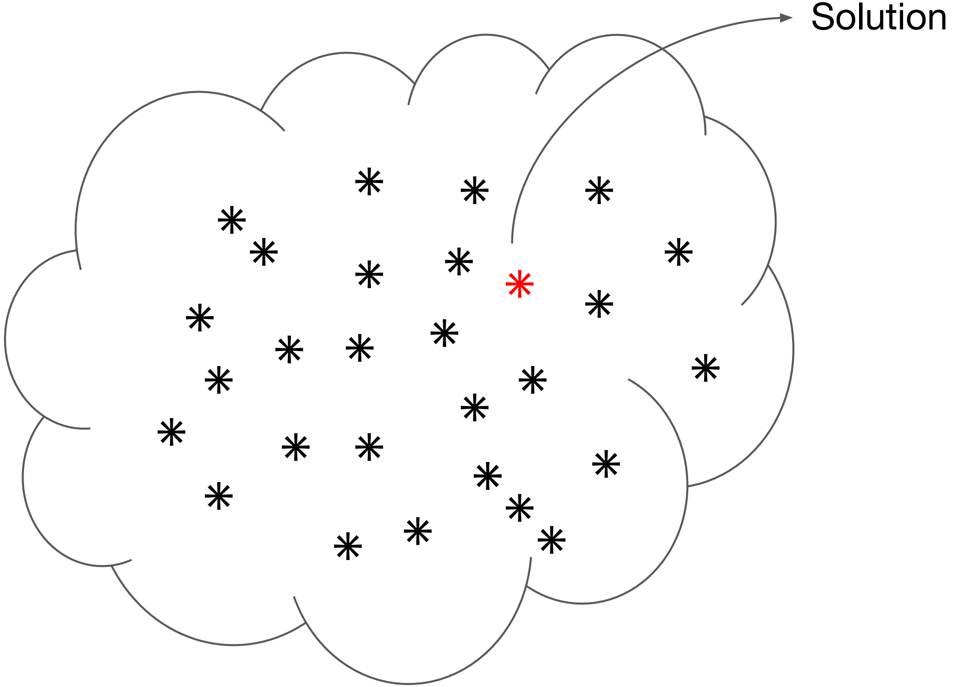
Fundamental techniques - Verification conditions

- Limitations of verified lifting:
  - Conditional statements
  - Loop invariables that increase non-monotonically
  - Demands a lot of user specification

Pruning Strategies

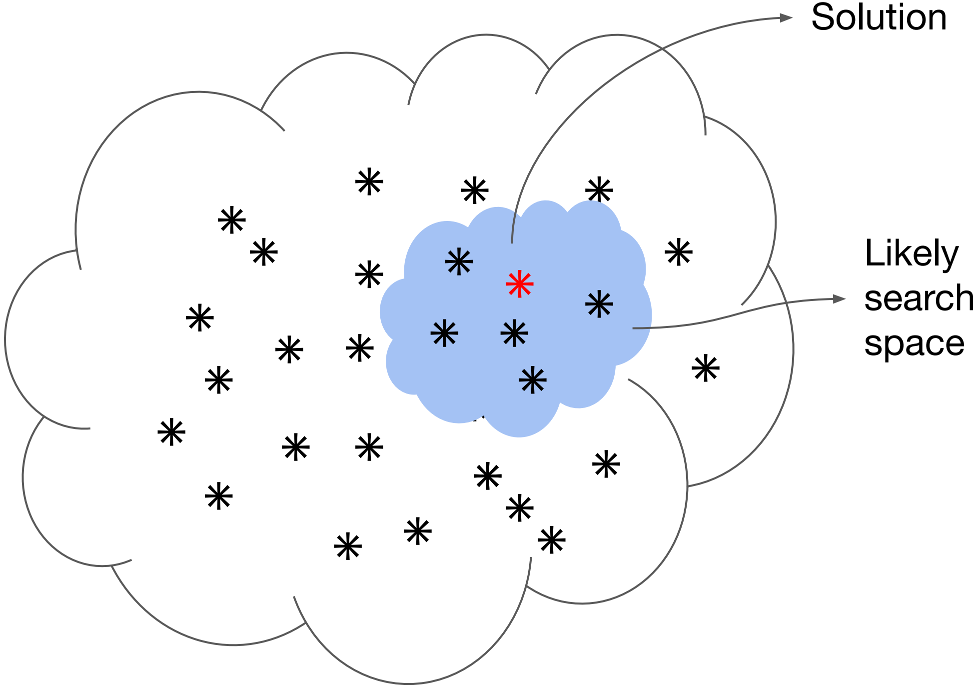
Pruning strategies

- Problem: large search space
- Brute force is unfeasible



Pruning strategies

- Solution: pruning search space
- Several techniques
- Focus on 3:
  - Type-directed
  - Sketch
  - Neural guidance

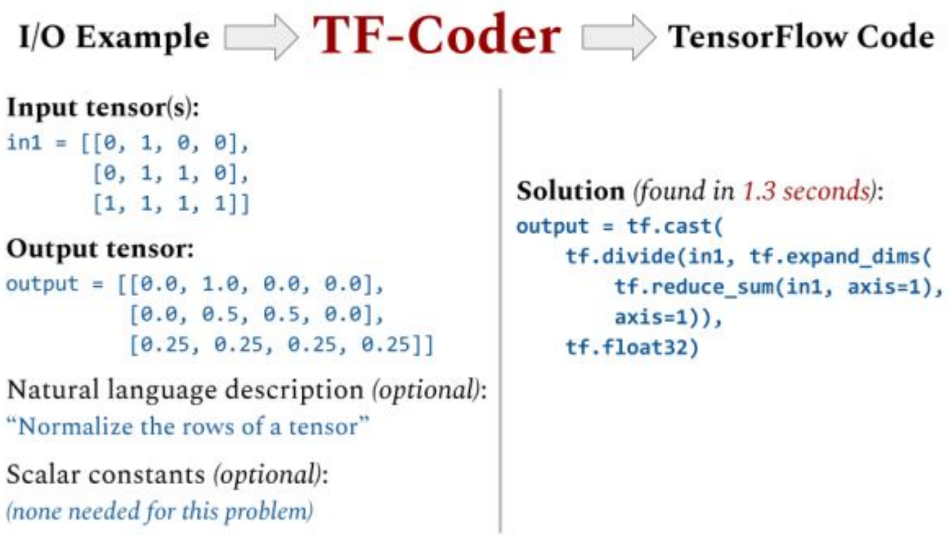


Pruning strategies

- Type-directed synthesis
- Use type checking to discard candidates
- Synthesis looks for compatible programs

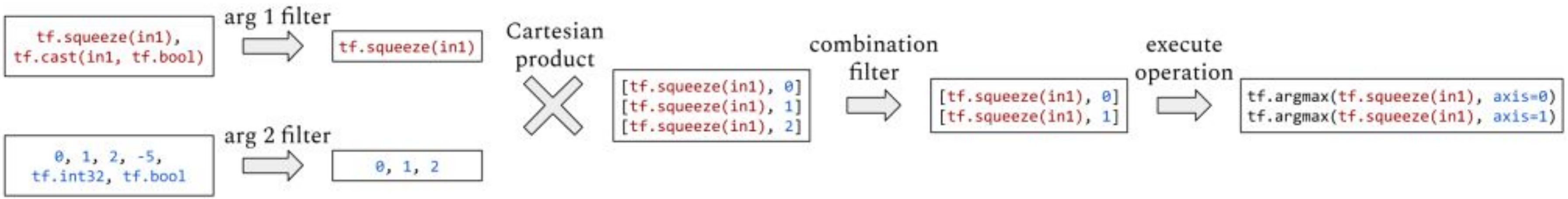
Pruning strategies

- TF-Coder: Program Synthesis for Tensor Manipulations (ACM Transactions on Programming Languages 2021) **[8]**



Pruning strategies

- Determine the arguments of TensorFlow API calls
- Type mismatch and compatibility among arguments themselves



Pruning strategies

- Sketch
- Provide a *draft* of solution
    - Given by user
    - Learnt some preprocessing step
- Synthesizer needs to fill the draft

Pruning strategies

- Program synthesis by sketching. **[9]**

- Sketch basics:
  - Unknown constants
  - Test harness
  - Generator functions

Pruning strategies

- Program synthesis by sketching. **[9]**
- Consider the problem of transposing a 5x5 matrix
- Sketch:

```
int[25] transpose5x5(int[25] mat){
        int[25] out;
        for(int i=0; i<5; ++i) for(int j=0; j<5; ++j){
                out[ ??*i + ??*j + ??] = mat[??*i + ??*j + ??];
        }
        return out;
}
```

Pruning strategies

- Expression ??*i + ??*j + ?? is repeated twice
- Generator encompasses the set of possible linear algebra expressions involving i and j

```
generator int legen(int i, int j){
        return ??*i + ??*j + ??;
}
int[25] transpose5x5(int[25] mat){
        int[25] out;
        for(int i=0; i<5; ++i) for(int j=0; j<5; ++j){
                out[ legen(i,j) ] = mat[ legen(i,j) ];
        }
        return mat;
}
```

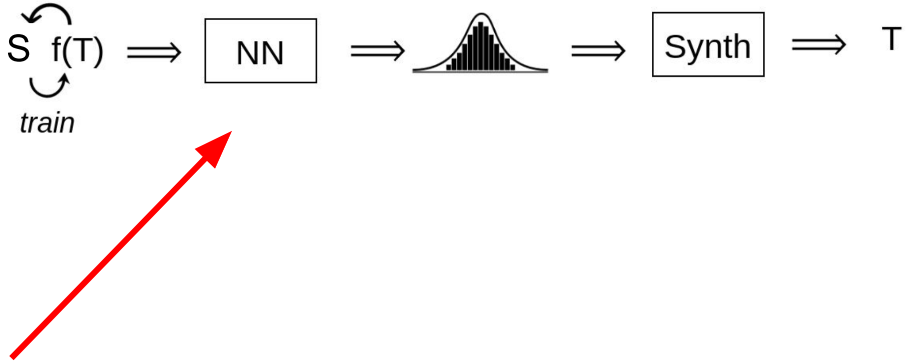
Pruning strategies

- Each call to the generator will resolve to a different expression, resulting in a correct implementation

```
int[25] transpose5x5(int[25] mat){
        int[25] out;
        for(int i=0; i<5; ++i) for(int j=0; j<5; ++j){
                out[ 5*i + j ] = mat[ i + 5*j ];
        }
        return mat;
}
```

Pruning strategies

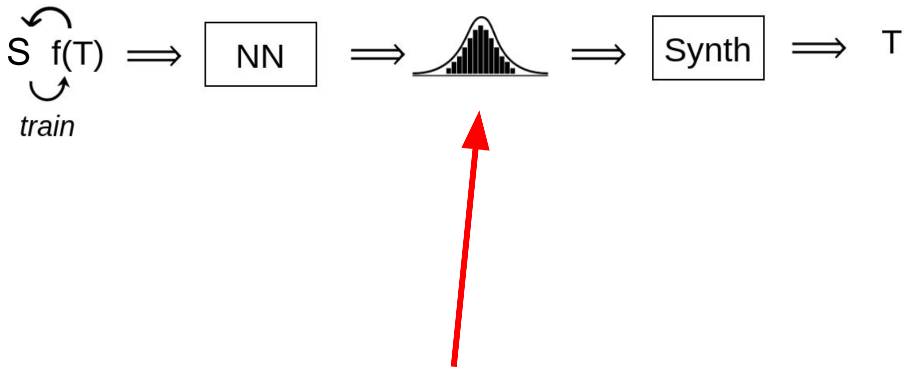
- Neural-guided synthesis



- Use of neural networks to help synthesis

Pruning strategies

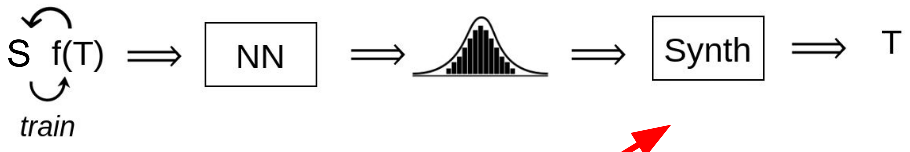
- Neural-guided synthesis



- Neural network outputs a probability distribution
- Indicates likelihood of tokens in target language

Pruning strategies

- Neural-guided synthesis



- Distribution leads the search

Pruning strategies

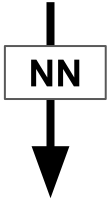
- Deepcoder: Learn to Write Programs (ICLR 2017) **[10]**

- Maps IO examples to program properties

`a ← [int]`	**An input-output example:**
`b ← FILTER (<0) a`	*Input*:
`c ← MAP (*4) b`	`[-17, -3, 4, 11, 0, -5, -9, 13, 6, 6, -8, 11]`
`d ← SORT c`	*Output*:
`e ← REVERSE d`	`[-12, -20, -32, -36, -68]`

Pruning strategies

a ← [int]	**An input-output example:**
b ← FILTER (<0) a	*Input*:
c ← MAP (*4) b	[-17, -3, 4, 11, 0, -5, -9, 13, 6, 6, -8, 11]
d ← SORT c	*Output*:
e ← REVERSE d	[-12, -20, -32, -36, -68]



(+1)	(-1)	(*2)	(/2)	(*-1)	(**2)	(*3)	(/3)	(*4)	(/4)	(>0)	(>0)	(%2==1)	(%2==0)	HEAD	LAST	MAP	FILTER	SORT	REVERSE	TAKE	DROP	ACCESS	ZIPWITH	SCANL1	+	-	*	MIN	MAX	COUNT	MINIMUM	MAXIMUM	SUM
.0	.0	.1	.0	.0	.0	.0	.0	1.0	.0	.0	1.0	.0	.2	.0	.0	1.0	1.0	1.0	.7	.0	.1	.0	.4	.0	.0	.1	.0	.2	.1	.0	.0	.0	.0

Pruning Strategies

- Deepcoder uses Depth-First Search
- Extends programs based on the probabilities
- Speedup on search

Table 1: Search speedups on programs of length $T = 3$ due to using neural network predictions.

Timeout needed	DFS			Enumeration			λ^2			Sketch		Beam
to solve	20%	40%	60%	20%	40%	60%	20%	40%	60%	20%	40%	20%
Baseline	$41ms$	$126ms$	$314ms$	$80ms$	$335ms$	$861ms$	$18.9s$	$49.6s$	$84.2s$	$>10^3 s$	$>10^3 s$	$>10^3 s$
DeepCoder	$2.7ms$	$33ms$	$110ms$	$1.3ms$	$6.1ms$	$27ms$	$0.23s$	$0.52s$	$13.5s$	$2.13s$	$455s$	$292s$
Speedup	**15.2×**	**3.9×**	**2.9×**	**62.2×**	**54.6×**	**31.5×**	**80.4×**	**94.6×**	**6.2×**	**>467×**	**>2.2×**	**>3.4×**

Pruning Strategies

- Limitations of neural-guided synthesis
  - Performance of synthesizer depends on the model
  - Therefore, it depends on the quality of training data
  - Languages targeted are still domain-specific/small
  - Need to find the most relevant features

The future

The future

- Sketching = how to find the best initial sketch?
- Machine Learning + Synthesis
- Loop invariants
- How to strength verification?
- How to generate good specifications (IO)?

Summary

- Program synthesis emerge as a technique to generate programs automatically with more control to the user
- As computers evolve, the inductive approach has risen as the main synthesis method
- Formal verification and machine learning incorporated into synthesis
- Challenges for the future include finding the best starting point and handle more complex languages and domains

Overview

- L1: Motivation and brief survey of auto-tuning/machine learning

for compilers

- L2: Program rewriting schemes - e-graphs and equality saturation
- L3: Program embeddings and Graph Neural Networks
- This lecture: Program synthesis and neural synthesis
- Next L5: Neural Machine Translation,Transformers and Large language models

Bibliography

Survey:

- GULWANI, Sumit et al: **Program synthesis**. Foundations and Trends® in Programming Languages, v. 4, n. 1-2, p. 1-119, 2017.

Papers

1. Armando Solar-Lezama: **Introduction to Program Synthesis**. Lecture Notes. 2018
2. Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, Guillaume Lample: **Unsupervised Translation of Programming Languages**. NeurIPS 2020
3. Alvin Cheung, Armando Solar-Lezama, Samuel Madden: **Optimizing Database-Backed Applications with Query Synthesis**. PLDI 2013
4. Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, Saman P. Amarasinghe: **Helium: lifting high-performance stencil kernels from stripped x86 binaries to halide DSL code**. PLDI 2015
5. Ansong Ni, Daniel Ramos, Aidan Z. H. Yang, Inês Lynce, Vasco M. Manquinho, Ruben Martins, Claire Le Goues: **SOAR: A Synthesis Approach for Data Science API Refactoring**. ICSE 2021
6. Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, Vijay A. Saraswat: **Combinatorial sketching for finite programs**. ASPLOS 2006

Bibliography

7. Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, Armando Solar-Lezama: **Verified lifting of stencil computations.** PLDI 2016
8. Kensen Shi, David Bieber, Rishabh Singh: **TF-Coder: Program Synthesis for Tensor Manipulations**. ACM Trans. Program. Lang. Syst. 44(2): 10:1-10:36 (2022)
9. Armando Solar-Lezama: **Program Synthesis By Sketching.** Ph.D. Thesis. University of California, Berkeley 2008
10. Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, Daniel Tarlow:**DeepCoder: Learning to Write Programs.** ICLR 2017