Synthesis and Verification of Algorithms

Yican Sun

School of Computer Science, Peking University

Algorithm is Important but Hard

(1) Designing algorithms requires **human insight**
(2) Implementing algorithms is **error-prone**

e.g. Knapsack

Enumerative Search vs. **Dynamic Programming**

Easy, 5 Lines

Hard, 20+ Lines

Overlapping Subproblem
Optimal Substructure

Algorithm Synthesis

Algorithm Paradigm
e.g., divide-and-conquer, dynamic programming, etc.

Formal Specification

- **Brute-force Reference Program**: Synthesize an algorithm equivalent to the reference program
- **Declarative Specification**: e.g., MiniZinc, Weighted programming

↓

Desired Algorithm

How to Synthesize Algorithms?

LLM Approach

Can you optimize this program as a parallel program using D&C? The expected time complexity is O(n/p), where n is the length of the input list x, and p is the number of CPU cores.

```
mss = -INF
for i in range(len(x)):
  for j in range(i, len(x)):
    mss = min(mss, len(x[i: j+1]) + sum(x[i: j+1]))
return mss
```

Procedure
- Modify 'max segment sum'
- LLM produces incorrect answers

Conclusion
- LLM 'memorizes' problems
- **Slight modification stumps LLM**

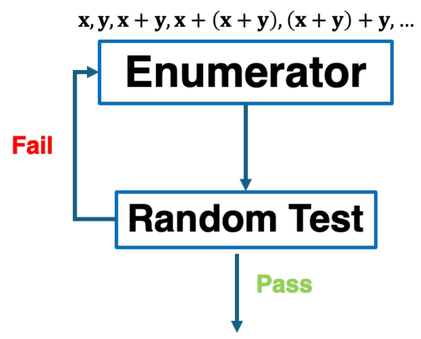
Deductive Approach

$$
\begin{aligned}
&\max \cdot \text{map sum} \cdot \text{segs} && \mathcal{O}(n^3)\ \text{Def segs}\\
=\ &\max \cdot \text{map sum} \cdot \text{concat} \cdot \text{map tails} \cdot \text{inits} && \mathcal{O}(n^3)\ \text{Map promotion}\\
=\ &\max \cdot \text{concat} \cdot \text{map (map sum)} \cdot \text{map tails} \cdot \text{inits} && \mathcal{O}(n^3)\ \text{Def max, Fold promotion}\\
=\ &\max \cdot \text{map max} \cdot \text{map (map sum)} \cdot \text{map tails} \cdot \text{inits} && \mathcal{O}(n^3)\ \text{Map distributivity}\\
=\ &\max \cdot \text{map (max} \cdot \text{map sum} \cdot \text{tails)} \cdot \text{inits} && \mathcal{O}(n^3)\ \text{Horner's rule *}\\
=\ &\max \cdot \text{map (foldl } (\otimes)\ 0) \cdot \text{inits} && \mathcal{O}(n^2)\ \text{Scan lemma *}\\
=\ &\max \cdot \text{scanl } (\otimes)\ 0 && \mathcal{O}(n)\ \text{Fold-scan fusion}\\
=\ &\text{fst} \cdot \text{foldl } (\odot)\ (0,0) && \mathcal{O}(n)
\end{aligned}
$$

$$
\begin{aligned}
x \otimes y &= (x+y) \uparrow 0\\
\langle u, v\rangle \odot x &= \textbf{let } w = (v+x) \uparrow 0 \textbf{ in } \langle u \uparrow w, w\rangle
\end{aligned}
$$

- Rewriting-based framework
- Design rules problem by problem
- **Semi-automated**

Inductive Approach: State of the art



Try each program from small to big

- Successful in D&C (like) algorithm
- **Widely open for other algorithms**
- **Synthesis result is not guaranteed to be correct**

Q1: Improve Inductive Algorithm Synthesizers

SynMem: Synthesize Dynamic Programming

MiniZinc Specification → SynMem → Dynamic Programming

MiniZinc: Encode combinatorial problems by variables, constraints and optimization function.

```
// Inputs
int: n, C;
array[1..n] of int: weight;
array[1..n] of int: value;
// Solutions
array[1..n] of var 0..1: p;
// Constraints
constraint ∑_{j=1}^{n} weight[j]·p[j] ≤ C;
// Objective
solve maximize ∑_{j=1}^{n} value[j]·p[j];
```

Challenges

- **Scalability Challenge:** Dynamic Programming programs are too large!
- **Efficiency Challenge:** Purely enumeration cannot guarantee efficiency.

Solution: Template-based approach

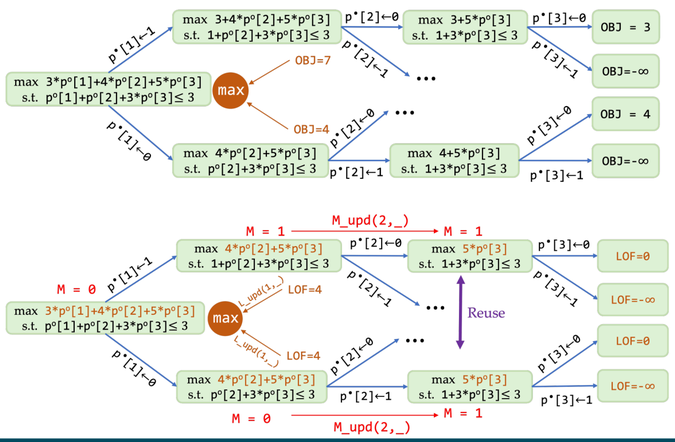
```
1  const enum_order E;
2  parameter β;
3  problem_instance I;
4  atom_var_info[] V_info;
5  map<(int, int^k), int> mem;
6  int L_leaf = ○_leaf, M_init = ○_minit;
7  int L_upd(int i,int L,agn_t V_i^•)
8    { return ○_Lupd;}
9  int^k M_upd(int i,int^k M,agn_t V_i^•)
10   { return ○_Mupd;}
11
12 int search(int i,int^k M,agn_t V^•) {
13   if (C(V^•)==False)
14     return obj_invalid;
15   if (i>length(V_info))
16     return L_leaf;
17   if(mem.find(i,M)!=mem.end())
18     return mem[(i,M)];
19
20   ans=obj_invalid;
21   for V_i^• ∈ V_info[i].dom
22     ans=obj_merge(ans,L_upd(i,L,V_i^•));
23     where L=search(i+1,M',V_2^•)
24     and M'=M_upd(i,M,V_i^•)
25     and V_2^•=V^• ++V_i^•
26   return mem[(i,M)]=ans;
27 }
28 int main() {
29   read(&β);
30   I = gen_instance(β);
31   V = [[E]];
32   return search(1,M_init,[]);
33 }
```

Why Template works?

- **Scalability Challenge:** Most part of codes is fixed.
- **Efficiency Challenge:** Reduce to optimization.

How to Design the Template?

Dynamic Programming = Exhaustive Search + ➢Local Objective Function + updating function ➢Memoization Partition Function + updating function



Q2: Verify the Synthesis Result in Practice

Simplified case: algorithm synthesis from Brute-force

Brute-force Reference Program ≡ **Synthesized Algorithm**

Both are *Functional*

Proving Equivalence Between Functional Programs

Basic Framework: Induction + Lemma

$\forall \textbf{xs: List. sum (rev xs)} = \textbf{sum (sort xs)}$

Base Case : $\textbf{xs} = \textbf{nil} \quad \vdash \textbf{sum (rev nil)} = \textbf{sum (sort nil)}$
Induction Case : $\textbf{xs} = \textbf{cons h t}$

$\textbf{sum (rev t)} = \textbf{sum (sort t)}$ ⊢ $\textbf{sum (rev (cons h t))} = \textbf{sum (sort (cons h t))}$

Induction Hypothesis

Rewriting the goal:

$\textbf{sum (snoc h (rev t))} = \textbf{sum (ins h (sort t))}$

Induction hypothesis cannot apply

Invent the lemma: $\forall \textbf{xs: List. sum (rev xs)} = \textbf{sum xs}$

```
Let rev (l:List) =
match l with
| nil → nil
| cons h t → snoc h (rev t)
end;

Let sort (l:List) =
match l with
| nil → nil
| cons h t → ins h (sort t)
end;

Let sum (l:List) =
match l with
| nil → 0
| cons h t → h + (sum t)
end;
```

Lemma Synthesis

- **Lack of Systematic study, wasting time in trying useless lemmas**
- Enumerate lemmas by Heuristics
  - Enumerate lemmas from small to large
  - Rank lemmas by text similarity
  - Predict lemmas via machine-learning

HipSpec tries the lemma $\forall \textbf{xs: List. rev (rev xs)} = \textbf{xs}$

Directed Lemma Synthesis: Transform goals into a induction-friendly form

Definition (Inductive-friendly forms)
For every proposition in this form, we can always apply inductive hypothesis.

Example
Either side is a single structural recursion

$\forall \textbf{xs: List. sum (rev xs)} = \textbf{sum xs}$

$\textbf{sum (rev xs)} = \textbf{sum (sort xs)}$

↓ **Synthesize structural recursion as a "bridge"**

$\textbf{sum (rev xs)} = \textbf{f xs}$
$\textbf{f xs} = \textbf{sum (sort xs)}$

Algorithm synthesis!