Synthesis of First-Order Dynamic Programming Algorithms

Yewen Pu Rastislav Bodík Saurabh Srivastava
University of California, Berkeley

Abstract

To solve a problem with a dynamic programming algorithm, one must reformulate the problem such that its solution can be formed from solutions to overlapping subproblems. Because overlapping subproblems may not be apparent in the specification, it is desirable to obtain the algorithm directly from the specification. We describe a semi-automatic synthesizer of linear-time dynamic programming algorithms. The programmer supplies a declarative specification of the problem and the operators that might appear in the solution. The synthesizer obtains the algorithm by searching a space of candidate algorithms; internally, the search is implemented with constraint solving. The space of candidate algorithms is defined with a program template reusable across all linear-time dynamic programming algorithms, which we characterize as first-order recurrences. This paper focuses on how to write the template so that the constraint solving process scales to real-world linear-time dynamic programming algorithms. We show how to reduce the space with (i) symmetry reduction and (ii) domain knowledge of dynamic programming algorithms. We have synthesized algorithms for variants of maximal substring matching, an assembly-line optimization, and the extended Euclid algorithm. We have also synthesized a problem outside the class of first-order recurrences, by composing three instances of the algorithm template.

Categories and Subject Descriptors I.2.2 [*Automatic Programming*]: Program synthesis; D.3.3 [*Language Constructs and Features*]: Constraints

General Terms Algorithms, Languages, Verification

Keywords Synthesis, Constraint Solving

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

1. Introduction

Dynamic programming is an algorithmic technique that exploits the structure of an optimization problem [6, 13, 25]. While direct execution of a specification will typically enumerate all legal solutions, dynamic programming breaks down the problem into overlapping subproblems and evaluates the resulting identical subproblems only once. The challenge in obtaining a dynamic programming algorithm is in reformulating the problem into a recurrence that exposes identical subproblems. Even if the specification already is a recurrence, it is rare that its subproblems are overlapping. Human insight into the hidden structure of the shared subproblems is typically necessary.

We describe a synthesizer of dynamic programming algorithms (DPAs) that reduces the need for human insight. Our programmer (1) specifies the problem with a declarative specification, which can be implemented as an exhaustive, exponential-time search; and (2) provides the synthesizer with (a superset of) operators to be used in the recurrence underlying the DPA. Given the specification and the operators, the synthesizer automatically produces the DPA, if one exists that uses only the operators supplied. The programmer is asked neither for suitable identical subproblems nor for the problem is solved from the solutions to these subproblems.

We build on the idea of *synthesis with partial programs*, where program templates are completed by the synthesizer to yield a program functionally identical to the specification [1, 12, 20, 23]. We express a generic skeleton of a DPA as a template whose "holes" are parameterized with a suitable language of candidate expressions. The completion of a template can be viewed as a search, in the space of candidate DPAs syntactically defined by the template, for a DPA that implements the specification. The size of the space typically prohibits exhaustive search. Instead, efficient search techniques have been developed using machine learning [1], version space algebra [12], inductive synthesis via constraint solving [20], or invariant inference [23]. In this paper, we use solver of the Sketch synthesizer [20, 2], which reduces the problem to a SAT problem, but encoding techniques should carry over to other constraint solvers, such as SMT solvers.

Previously, a DPA has been successfully synthesized with a partial-program synthesizer, using a template tailored to

the problem [23]. Our goal is to develop a reusable template and thus develop a synthesizer for a broad class of DPAs. Our approach is to translate the programmer-provided hints into a template which serves are the partial program for the DPA synthesis. Our template generator thus constitutes a "domain theory" for dynamic programming, in that it embeds knowledge needed to synthesize any DPA in our class. While we study a class of dynamic programming, our lessons will hopefully inform creation of templates for other domains.

We focus on the domain of linear-time dynamic programming algorithms, specifically those that grow the solution by considering one input element at a time. We characterize DPAs for this class as first-order recurrences, but our template automatically expands to k-order recurrences. We can synthesize solutions with an arbitrary (bounded) number of subproblems.

We have synthesized algorithms for variants of maximal substring matching, an assembly-line optimization, and the extended Euclid algorithm. We have also demonstrated synthesis of a problem outside the class of first-order recurrences using a template composed of three instances of our single algorithm template.

We make the following contributions:

- We show that partial-program synthesis based on constraint solving is able to produce realistic linear-time dynamic programming algorithms. We synthesize these algorithms from a reusable template, asking the user only for operators that are to be used in the algorithm.
- We show how to produce templates that lead to efficient synthesis by relying on (i) the properties of the domain of dynamic programming and (ii) symmetry reduction.
- We show that several instances of our template can be composed to solve problems that are outside the class of first-order recurrences. Because we formulate the domain theory as a partial program, the user can easily modify and grow the template as she sees fit, without understanding the internals of the synthesizer.

2. Overview

We first introduce two classes of problems solvable by dynamic programming (Sections 2.1–2.4). Next, we describe the interactions of a programmer with our synthesizer (Section 2.5). Finally, we give an overview of the synthesizer algorithms and outline our encoding of the synthesis problem (Sections 2.6–2.8).

2.1 Dynamic Programming for Optimization Problems

Dynamic programming is an algorithmic technique that exploits the structure of an optimization problem [6, 13, 3, 25]. A naive approach to solving an optimization problem is to enumerate all legal solutions and selecting one with the optimum value—e.g., enumerating nonconsecutive subsequences of an array followed by selecting one with the largest sum. Often, exponentially many candidate solutions need to be explored.

Dynamic programming avoids enumeration by constructing an optimal solution from the solutions to subproblems. A problem can be solved with a *dynamic programming algorithm* (DPA) if

1. the problem exhibits an *optimal substructure*, i.e., an optimal solution can be composed from optimal solutions to subproblems; and
2. the problem can be broken down into subproblems that are *overlapping*, i.e., they share identical subproblems.

Because solutions to identical subproblems are reused, typically only a polynomial number of subproblems need to be evaluated.

2.2 Dynamic Programming for Functional Problems

Dynamic programming is also applicable to problems that compute the function of the input, instead of searching a space of solutions. One example is computing the Fibonacci sequence. We synthesize DPAs for two such problems, OtherSum and Extended Euclid algorithm, in Sections 4.1.7 and 4.1.6, respectively.

To be amenable to dynamic programming, these "functional" problems must also have a substructure property with overlapping subproblems. We cannot talk about *optimal* substructure, of course, as there is no notion of an optimal solution. We introduce functional problems separately from the optimization problems of Section 2.1 because optimization problems permit more efficient synthesis (cf. Section 3.3).

2.3 The Challenge of Designing a DPA

The design of a DPA amounts to defining a recurrence relation that gives rise to overlapping subproblems whose results can be stored in a table and computed by traversing the table, usually in an input-independent order.

Sometimes, the definition of the problem reveals the overlapping subproblems. For example, the n-th Fibonacci number, $F(n) = F(n-1) + F(n-2)$, is computed from two subproblems, $F(n-1)$ and $F(n-2)$, which share the subproblems $F(n-2)$ and $F(n-3)$. The DPA table stores the solutions to $F(n)$ and $F(n-1)$ and can be computed in $O(n)$ time.

Most problems must be reformulated to expose identical subproblems, a process that requires human insight. In particular, a suitable recurrence may involve enriched or even orthogonal subproblems that are not immediate from the problem specification but are essential because identical subproblems occur only in the modified problem space.

As an example of a subproblem that needs to be invented, consider the problem of finding a sub-string of an integer array that maximizes the sum of its elements, which could be negative. A naive algorithm searches over all $O(n^2)$ substrings, performing either linear work or if accumulating the sum then constant work for each substring, resulting in a

$O(n^3)$ or $O(n^2)$ algorithm. To obtain a linear-time DPA that scans the array left-to-right, we need two subproblems:

1. the maximal-sum substring to the left of index i; and
2. the maximal-sum substring to the left of index i that *ends at index* i.

The latter subproblem may seem non-intuitive but we need it to find an optimal substring that spans the index i.

The design of dynamic programming algorithms from their declarative specification requires insight and is therefore taught as an art.

2.4 Running Example: Maximum Independent Sum

Given an array of positive integers $A = [a_1, \ldots, a_n]$, The *Maximum Independent Sum* (MIS) problem selects a subset of non-consecutive array elements that maximizes the sum of its elements [15]. Formally, MIS finds an *assignment* of boolean values $B = [b_1, \ldots, b_n]$, where $b_i = 1$ iff a_i is selected. Array B is legal if it contains no substring of contiguous 1s. If $A = [2, 3, 4, 1]$ and the assignment to B is $[1, 0, 1, 0]$, the value of applying the assignment is $\mathit{apply}([2, 3, 4, 1], [1, 0, 1, 0]) = 6$. The objective is to maximize the value of the assignment. For instance, $\mathit{MIS}([4, 1, 2, 3]) = 7$ via the assignment $[1, 0, 0, 1]$.

Note that this problem is not given as a recurrence; instead, the definition gives the legality and optimality conditions. Therefore, it does not suggest any obvious overlapping subproblems.

2.5 Synthesizer Input and Output

The user provides a problem specification and operator hints, and the synthesizer outputs a DPA. Next, we illustrate the inputs and outputs of the synthesizer.

Specification The user can give the specification in one of two ways. A functional specification computes the solution to a given problem instance; a declarative specification checks whether a value is a correct output for a given instance. The specification for the MIS problem is interesting in that it is functional but is written in a declarative style of universally quantifying over all solutions:

```
def spec( A = [a1, ..., an] )
    best = 0
    for all bitstrings {B = [b1, ..., bn]}:
        if B is not consecutive:
            best = max(best, apply(A,B))
    return best
```

Programmer hints The synthesizer asks the programmer for hints on which operators will form the recurrence. In the case of MIS, the programmer supplies two unary operators (the identity and the zero functions); one binary operator (addition); and the optimality function (maximum):

```
unary(x) = {x, 0}
binary(x,y) = {x+y}
opt(x,y) = max(x,y)
```

The good news is that a sufficient set of operators can often be obtained from the specification. In MIS, the addition and the maximum arise from maximizing the sum of selected array elements; the identity and the zero function correspond to the decision of selecting (identity) or not selecting (zero) the current input element. We have not attempted to automate the extraction of the operator hints from the specification. The programmer specifies the operators and then iteratively increases the number of subproblems the synthesizer is to consider. For small values that do not contain any solution the synthesizer informs the user as such. The programmer iterates until a solution is obtained.

Synthesized Solution The synthesizer outputs a linear-time DPA that implements the specification. The synthesized algorithm follows the common table-filling pattern, shown here for the MIS problem:

```
def mis ( A = [a1,...,an] )
    p1 = array() // solutions to subproblem 1
    p2 = array() // solutions to subproblem 2
    p1[0] = 0 // base case for subproblem 1
    p2[0] = 0 // base case for subproblem 2
    for i from 1 to n:
        p1[i] = p2[i−1] + A[i]
        p2[i] = max(p1[i−1], p2[i−1])
    return max(p1[n], p2[n])
```

The synthesizer created two subproblems, which we can interpret as follows:

$P_1(i)$**:** solution to MIS(A[0:i]) provided a_i was selected;

$P_2(i)$**:** solution to MIS(A[0:i]) provided a_i was not selected.

Notice how the synthesizer invented subproblems whose iterative computation implicitly encodes the condition that no two consecutive elements are picked. The current element is added only to P_2 which holds solutions that exclude the previous element.

The table of the DPA is formed by arrays p1 and p2, which store the values of the solutions to the subproblems $P_1(i)$ and $P_2(i)$, respectively. When $A = [3, 1, 2, 4, 4, 6]$, then p1[5]$= 9$, corresponding to the selection $[1, 0, 1, 0, 1]$, and p2[5]$= 7$, corresponding to the selection $[1, 0, 0, 1, 0]$.

The synthesized DPA initializes the base cases of the two subproblems to 0 and iterates through the arrays left to right, filling the table. The synthesizer determines that the solution to the overall problem is the larger of the solutions to the subproblems from the last iteration.

If desired, we can manually translate the synthesized program to a recurrence relation. We obtain a *first-order recurrence*, i.e., solutions to subproblems of size i depend solely on the solutions of size $i - 1$. All DPAs that we synthesize are of this form; we discuss the expressiveness of first-order

recurrences in Section 3. Notice that the recurrence is composed from the hints provided by the programmer.

$$\begin{aligned} \mathit{MIS}(n) &= \max(P_1(n), P_2(n)) \\ P_1(n) &= \begin{cases} P_2(n-1) + A(n) & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases} \\ P_2(n) &= \begin{cases} \max(P_1(n-1), P_2(n-1)) & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases} \end{aligned}$$

The synthesizer does not generate a proof that DPA correctly implements the specification. The synthesizer's verifier checks the correctness of the synthesized DPA via bounded model checking [20], which ensures that the specification and the DPA agree on all inputs of a small size. In our experiments, we verified the solution on all arrays of sizes 1 to 4. We have found this small-world-assumption test sufficient.

2.6 Synthesis as Constraint Solving

We view synthesis as a search in the space of candidate programs. We define a space of DPAs and the synthesizer is asked to find a DPA that agrees with the specification on a small number of representative inputs I. The space of DPAs is defined with a template program, which in our system is called a sketch [20]. The template is translated into a constraints system whose solution serves as arguments to the template. The instantiated template yields a DPA that agrees with the specification on all inputs from I. If inputs in I yield a DPA that fails the bounded model checking test, the counterexample input from the test is added to I and constraint solving is repeated. This is the core solving technique underlying the Sketch synthesizer [20].

To make constraints-based synthesis suitable for synthesis of DPAs, we need to address two interrelated problems:

- *Completeness:* How to define the space of DPAs so that it includes all DPAs of interest?
- *Efficiency:* How to define the space of DPAs that induces a constraint problem that is solvable in a few minutes.

The next two subsections overview our solution to these two problems.

2.7 First-Order Dynamic Programming Algorithms

It may be tempting to define the space of DPAs with a rather general recurrence template, such as this one that breaks the problem P into two unspecified subproblems:

$$P(n) = \underline{f}(P(\underline{g}(n)), P(\underline{h}(n)), n)$$

This template asks the synthesizer to find functions g, h that suitably break the problem into subproblems, and a function f that composes the solution to $P(n)$ from the solutions to the two subproblems $P(g(n))$ and $P(h(n))$. It is straightforward to define a complete template in this general manner, but only at the cost of two problems:

- Allowing the synthesizer to select functions g, h from a large space of functions would likely produce a space of DPAs that is too large for all state-of-the-art constraint-based synthesizers [2, 24, 10].
- The recurrence template does not insist that the synthesized subproblems are overlapping. As a result, not all recurrences synthesized from this template could be translated to a DPA. Another challenge, specific to the Sketch synthesizer, is that due to unrolling of calls to P, this template would result in a constraint system whose size is exponential in n, and thus likely too large even for the small inputs on which Sketch performs the synthesis.

To overcome the first problem, we restrict the space of DPAs to first-order recurrences (FORs), which have the form

$$P(n) = \underline{f}(P(n-1), n)$$

that is, we hardcode that a problem of size n is decomposed into a subproblem of size $n-1$. Our space of DPAs is more general than this recurrence template may suggest: our template allows (i) k-order recurrences and (ii) multiple subproblems (cf. Section 3.2). We believe that this template covers most linear-time DPAs.

To overcome the second problem, rather than synthesizing a recurrence, we synthesize directly a table-filling form of a DPA. By storing solutions to subproblems in a linear-size table, a table-filling template insists that the synthesized recurrence will exhibit overlapping subproblems. We encode a table-filling FOR algorithms with the following template:

```
def attempt ( a = [a_1,...,a_n] ):
    // create arrays to hold the values of subproblems
    p1 = array()
    p2 = array()
    // initialize
    p1[0] = init() // some initial value
    p2[0] = init() // some initial value
    // update the subproblem values
    for i from 1 to n:
        p1[i] = update_1(p1[i−1], p2[i−1], a[i])
        p2[i] = update_2(p1[i−1], p2[i−1], a[i])
    // the terminate function composes the final solution
    return terminate(p1[n], p2[n])
```

The template we illustrate here uses two subproblems; the number of subproblems is adjusted accordingly by our synthesizer that auto-generates the template from user-input. The structure of this template is identical to that of the solution for the MIS problem in Section 2.4; the differences are in the underlined functions, which the synthesizer selects from a space of functions by constraint solving. The initialization function, $\underline{int}$, returns an integer constant (in our experiments, the constant was restricted to $-\infty$, 0, and ∞). The propagation function, $\underline{update}$, and the termination function, $\underline{terminate}$, are selected from a space of functions that are compositions of the user-provided operators. For MIS, the search spaces are defined by the sets below. The

{| e1 | e2 | ... |} operator asks the synthesizer to select between expressions e1 and e2.

$\underline{int}$() := {| $-\infty$ | 0 | ∞ |}
$\underline{update}$(x,y,aval) := {| x | aval | y+x | x+max(y,aval) | /*...*/ |}
$\underline{terminate}$(x,y) := {| x | y+x | max(x,y) | /*...*/ |}

Our space of candidate DPAs is thus formed by this FOR template with the underlined functions selected from function spaces described in the next subsection.

2.8 Efficient Constraint Solving

Even after we restricted DPAs to first-order recurrences, the search space remains excessively large. For instance, in MIS, which is a relatively simple problem, the space contains 84,934,656 candidate DPAs. To further reduce the space, we apply space and symmetry reductions on the space of update and termination functions. In *space reduction*, we exploit properties of the optimization problems. Specifically, we restrict the syntactic forms of our functions, ruling out DPAs we know to be incorrect. In *symmetry reduction*, we prune away functions that are semantically equivalent (symmetric) to other programs despite being syntactically distinct.

Space Reduction Space reduction is made possible by two observations on the nature of the optimization problems (Section 2.1). First, each step of the algorithm makes an optimal choice from several legal alternative solutions, each computed from optimal solutions of smaller size. The update function thus needs to have the optimality function *opt* in the root of the expression—selecting the optimal solution is the last step of its computation. In other words, the syntactic form of the update function thus must be

$\underline{update}$(x,y,aval) := opt(f_1(x,y,aval),...,f_k(x,y,aval));

where functions f_i compute the values of legal solutions. For example, in MIS the goal is to return the largest sum, corresponding to the best assignment of non-contiguous elements. We can insist that update functions have the max operator at the root of the expression.

Second, we observe that in FOR algorithms, each of the legal solutions can depend on at most one subproblem. This argument becomes clearer if we view an optimization problem as returning n optimal decision, one for each element of the input. In the case of MIS, for the legal solution, if the value for $P_1(i)$ was constructed from the values to both $P_1(i-1)$ and $P_2(i-1)$, these two solutions would somehow need to be combined and the overall problem would return more than n decisions. For instance, in MIS the optimal solution corresponds to an array of 1 and 0. This observation allows us to syntactically restrict functions f_i such that each consumes only one optimal subproblem. Note that each optimal subproblem can still be used to compute multiple alternative solutions. These solutions however cannot be combined other than in the *opt* function that selects one of these alternatives. The form of the update function is now restricted to this form, which reduces the size of the candidate DPAs that we need to search:

$\underline{update}$(x,y,aval) := max(choose_subset(f_1(x,aval),f_2(y,aval)))
$\underline{f_i}$(x,aval) := {| x | aval | max(x,aval) | /*...*/|}

Symmetry Reduction We further reduce the number of programs by noticing that the operators such as $+$ and max are commutative. For example, the expression $x+max(y,z)$ is identical to the expression $max(z,y)+x$. We prune away this symmetry by defining canonical expression trees and ensuring that only canonical trees are constructed. For MIS, the symmetry reduction, along with space reduction, reduced the number of update functions from 589,824 to 65,536. Symmetry reduction becomes vital if the problem is not an optimization problem, in which case we cannot apply space reduction.

3. The DPA Synthesizer

This section describes the implementation of the DPA synthesizer. The synthesizer consists of two parts, a front-end template generator and a back-end synthesis constraint solver. The front-end translates the user-provided problem specification and hints into a template that efficiently encodes the space of DPAs. The backend resolves the template to a desired algorithm. We use the Sketch [2] solver, which was outlined in Section 2.6.

This section details template generation, which was overviewed in Sections 2.7 and 2.8. Recall that the main concern is balancing expressiveness and efficiency: the template must define a space of programs that includes all DPAs of interest yet must be small enough to give rise to easy-to-solve constraint systems. We first present a space of DPAs that is sufficiently expressive and then gradually prune it by exploiting domain-specific space reduction and symmetry reduction.

3.1 First-Order Recurrences

Here we define the space of *general* FORs, which contains all DPAs of interest to us. Subsequent subsections will narrow down the definition of the recurrence, tailoring it to dynamic programming.

Assume that the specification is a predicate $spec(a, o)$ which holds when o is a solution to the problem instance a. Assume that array $a = [a_1, ..., a_n]$, where each a_m is a scalar or a tuple of values. We encode our algorithm as a first-order recurrence.

FOR Algorithm:

$$
\begin{aligned}
FOR(a) &= terminate(p_1(n), ..., p_k(n)) \\
p_1(0) &= init_1() \\
&\vdots \\
p_k(0) &= init_k() \\
p_1(m) &= update_1(p_1(m-1), ..., p_k(m-1), a(m)) \\
&\vdots \\
p_k(m) &= update_k(p_1(m-1), ..., p_k(m-1), a(m))
\end{aligned}
$$

The correctness condition asserts that the synthesized algorithm is correct on all problem instances from a set I of small-size instances:

$$\forall a \in I.\, spec(a, FOR(a))$$

We define some terminology:

- A *subproblem*, denoted p_i, is a series of subproblem instances that are solved with the same update function.
- A *sub-problem instance* is a particular instance in a subproblem, denoted $p_i(m)$.
- An *update* is a function of the form:

  $$update_i : (p_1, \ldots, p_k, a) \rightarrow p_i$$

  There is one update function per subproblem. When we need to compute the value of a subproblem instance $p_i(m)$, we invoke the update function $update_i(p_1(m-1), ..., p_k(m-1), a_m)$. This update follows a first-order recurrence because subproblem instances at step m depend only on subproblem instances at step $m-1$.

  $$\vec{p}(m) = update(\vec{p}(m-1), a(m))$$

- A *termination* is a function of the form:

  $$terminate : (p_1, \ldots, p_k) \rightarrow output$$

  The termination function computes the solution to the original problem from the solutions to subproblems. Since subproblems range over a synthesized domain, the termination function maps subproblems back to the domain of the original problem.

The synthesizer determines the minimal number of subproblems by attempting to solve the problem with one subproblem, $k = 1$, and gradually increasing k as the solver determines that a solution cannot be found for a given value of k. The user can, of course, fix k should she have an intuition as to how many subproblems are needed.

We want to remark that our FOR space includes kth-order recurrences via the following reduction:

$$
\begin{aligned}
p(m) &= update(p(m-1), ..., p(m-k)) \\
&\rightarrow \\
p(m) &= p_1(m) \\
p_1(m) &= update(p_1(m-1), \ldots, p_k(m-1)) \\
p_2(m) &= p_1(m) \\
&\vdots \\
p_k(m) &= p_{k-1}(m)
\end{aligned}
$$

We now describe the function spaces of the functions $init$, $update$, and $terminate$ with context free grammars.

init

$$init(a) := -\infty \mid 0 \mid \infty$$

The $init$ function establishes the initial conditions of the recurrence. The user may suggest values other than those used in our experiments, including other constants or a function parameterized by the problem instance a, such as the first element of the array, $a[1]$.

update

$$
\begin{aligned}
&update_i(p_1, ..., p_k, a) := p_1 \mid \ldots \mid p_k \mid a \\
&\qquad \mid unary_?(update_i(p_1, ..., p_k, a)) \\
&\qquad \mid binary_?(update_i(p_1, ..., p_k, a), update_i(p_1, ..., p_k, a)) \\
&\text{where} \\
&unary_?(x) := unary_1(x) \mid \ldots \mid unary_l(x) \\
&binary_?(x, y) := binary_1(x, y) \mid \ldots \mid binary_m(x, y)
\end{aligned}
$$

The update function computes the solution to a subproblem at step m given a_m and the solutions at step $m-1$. The function space includes all expressions that can be constructed from the user-provided unary and binary operators. The recursive formulation of $update$ does not scale in general. Since $update$ sends k values to 1 output, if we let m be the total number of user specified binary operators, the total number of functions for each $update_i$ is at least m^k as it takes k binary operators to reduce $k+1$ values to 1 output. In section 3.3 and 3.4 we discuss how to search this space efficiently.

terminate

$$terminate(p_1, ..., p_k) := update(p_1, ..., p_k)$$

The termination function is a one-step update function that operates on the subproblems from the last, nth step. It is drawn from the language of the $update$ grammar.

3.2 The FOR Template

We now implement the FOR algorithm as a template program that executes a bottom-up, table-filling dynamic programming algorithm with explicit memoization of overlapping sub-problems:

```
def algorithm ( a = [a_1,...,a_n] ):
    // create arrays to hold the values of sub−problems
    p_1 = array()
    ...
    p_k = array()
    // initialize
    p_1[0] = init()
    ...
    p_k[0] = init()
    // update the sub−problem values
    for i from 1 to n:
        p_1[i] = update_1(p_1[i−1], p_k[i−1], a[i])
        ...
        p_k[i] = update_k(p_1[i−1], p_k[i−1], a[i])
    // terminate
    return terminate(p_1[n],...,p_k[n])
```

The FOR template uses k arrays to hold solutions to k subproblems. The underlined functions are defined in their own templates. As an example, consider the templates for the function $init$, which is a direct translation of the $init$ context-free grammar from the previous section:

```
def basecase ():
    case = 0 | 1 | 2
    case 0: return −∞
    case 1: return 0
    case 2: return ∞
```

3.3 Update Functions for Optimization Problems

In the FOR template, the update function can be any expression constructed from the user-provided operators. We will now restrict the syntactic form of the update function to reflect the structure of an optimization problem. Our encoding is applicable to the problems defined in Section 2.1 but not to the functional problems in Section 2.2. The functional problems can, however, take advantage of the more general (but weaker) optimizations described in the following subsection.

Recall the two special structural properties of optimizations problems (cf. Section 2.8):

1. Assume that $S_i(m)$ is the set of solutions (not all optimal) to the subproblem instance $p_i(m)$. The value of $p_i(m)$ is the optimal solution from $S_i(m)$. The computation of $p_i(m)$ thus has the form $opt(S_i(m))$, i.e., the opt function is at the root of the computation tree. During synthesis, we can thus rule out update functions where opt is syntactically in a non-root position.
2. Each solution $s_{i,j}(m) \in S_i(m)$ is computed from exactly one subproblem $p'_i(m-1)$. If $s_{i,j}(m)$ were to combine solutions to multiple subproblems, it would have to combine the histories of optimal decisions from both of these subproblems into a single history of m decisions, forcing the examination of the two histories for the purpose of deciding which decisions to preserve. In order to avoid examining histories, an optimal solution in an FOR DPA is constructed by extending the history of *one* optimal solution with a single optimal decision, based on the current input element.

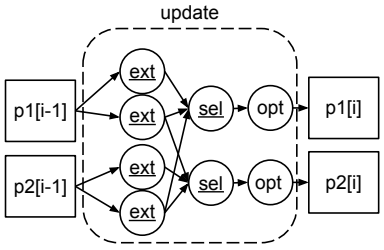

Figure 1. The update function for optimization problems.

To capitalize on these restrictions, we synthesize an update function by asking the synthesizer to make the following decisions:

- Synthesize functions that compute rk solutions, with r solutions for each of the k optimal subproblems p_i. These functions *extend* solutions to subproblems of size $m-1$ into solutions for a problem of size m. The solutions are stored in variables $ext_{j,l}$, $l \leq k$, $j \leq r$. The solutions $ext_{j,l}$ are computed with a *combiner* function that consumes a solution and the current input element. The template of the combiner function is defined in the next subsection.

$$\begin{aligned} ext_{1,1} &= \underline{combiner}(p_1, a) \\ &\vdots \\ ext_{r,1} &= \underline{combiner}(p_1, a) \\ &\vdots \\ ext_{1,k} &= \underline{combiner}(p_k, a) \\ &\vdots \\ ext_{r,k} &= \underline{combiner}(p_k, a) \end{aligned}$$

- Next we ask the synthesizer to decide, for each subproblem p_i, which of the solutions $ext_{j,l}$ solve p_i. This decision populates the sets S_i defined above with the solutions $ext_{j,l}$. The template $\underline{select}_s$ synthesizes into a function that selects up to s of its arguments.

$$p_i(m) = opt(\underline{select}_s(ext_{1,1}, \ldots, ext_{r,k}))$$

Figure 1 shows the structure of the update function.

As an example of how the update function is constructed, consider the MIS problem defined in Section 2.4. Let us call its two subproblems $pick$ and no_pick. The synthesizer

first creates the solutions: We can extend the sub-problem no_pick into two solutions, by picking versus not picking the current array element:

$$ext\_pick(no\_pick(m-1)) = no\_pick(m-1) + a_m$$
$$ext\_no\_pick(no\_pick(m-1)) = no\_pick(m-1) + 0$$

In contrast, we can extend the subproblem $pick$ only into one solution, by not picking the current element, becaus we cannot pick contiguous array elements:

$$ext\_no\_pick(pick(m-1)) = pick(m-1) + 0$$

Next, to find an update function that solves the no_pick subproblem, the synthesizer needs to select from the three solutions, by resolving the $\underline{select}_s$ template:

$$no\_pick(m) = max(\underline{select}_2(ext\_pick(no\_pick(m-1)) \\ ext\_no\_pick(pick(m-1)), \\ ext\_no\_pick(no\_pick(m-1))))$$

The synthesized $select$ function picks two solutions across which the no_pick problem optimizes. This gives us the final update function for the no_pick subproblem.

$$no\_pick(m) = max(ext\_no\_pick(pick(m-1)), \\ ext\_no\_pick(no\_pick(m-1)))$$

Discussion: In this section, we have restricted the update function syntactically. By doing so, we sought to reduce the function space to be explored during constraint solving. Whenever possible, syntactic restrictions seem preferable over symmetry-breaking predicates [17] used in Section 3.4. This is because syntactic restrictions offer the advantage of simultaneously reducing the size of the constraint system. In contrast, symmetry-breaking predicates prune the space by adding clauses to the constraint system; these clauses prevent symmetric candidates from arising as solutions but do so at the cost having the solver maintain conflict clauses over variables in the predicate.

3.4 Encoding of the Combiner Function Template

In the previous subsection, we restricted the form of the update function to reflect the properties of optimization problems. Here, we develop a template for the combiner function that is invoked from the update function. The combiner is used by both the optimization problems (Section 2.1) and the functional problems (Section 2.2). The combiner function reduces k inputs, $x_1, \ldots, x_k$, to an output value using the unary and binary operators provided by the programmer. Recall that in an optimization problem, we have $k = 2$, and the combiner computes a solution from a solution to a single subproblem and the current input element. In the functional problems, $k \geq 2$ because the combiner is allowed to combine the current input element multiple subproblems. For this reason, functional problems in particular benefit from the optimizations of this subsection.

The goal of the two optimizations is to reduce the space of combiner functions by eliminating those functions that are syntactically distinct but semantically equivalent. First, we normalize the combiners by distributing unary operators to the leaves of the expression. Second, we eliminate combiners that are identical up to commutativity of binary operators.

We adopt the restriction that each input x_i is used in the combiner exactly once. This restriction will allow us to work with trees, rather than dags. Should the DPA recurrence require a duplicate use of x_i, the synthesizer will implicitly work around this restriction by introducing an additional subproblem, whose value will be equal to x_i.

For the sake of conciseness, we introduce an infix form of the grammar $binary$ as $\odot$, and abbreviate the grammar $unary$ as u.

Distribute Unary Operators to Leaves In the $update$ grammar of Section 3.1, unary operators can appear at any position of the expression tree; they can also appear multiple times. For instance, the grammar generates the expression $identity(zero(max(x, y) + z))$ as well as the semantically equivalent expression $max(zero(x), zero(y)) + zero(z)$. We consider the latter expression canonical and eliminate the former. To distribute unary operators across binary operators, we introduce a combiner grammar where the unary operators appear only in the leafs of the expression tree:

$$combiner(x_1, \ldots, x_k) := reduce(u(x_1), \ldots, u(x_k))$$

Combiners thus first apply the unary operators on the inputs, then reduce them exclusively with binary operators. The $reduce$ grammar is introduced in the following section.

We now give conditions under which this $combiner$ grammar defines the same space of (semantic) functions as the $update$ grammar. In situations when the $combiner$ grammar loses some functions from the $update$ grammar, we show how to recover the lost expressiveness by adding binary operators. As a running example, we use the set of unary operators that we used most frequently in our experiments: $negate, identity, zero$; the set of binary operators are $+, -, *, /, max, min, \%$.

The $reducer$ grammar is equivalent to the $update$ grammar if the following conditions hold:

1. The set of unary operators is closed under composition, i.e., the grammar $u \circ \ldots \circ u$ and the grammar u generate the same space of (semantic) functions.
2. Each unary operator distributes over each binary operator, i.e., the grammar $u(x \odot y)$ generates the same space of semantic functions as the grammar $u(x) \odot u(y)$.

We currently check these properties manually. On our running example, the first property is easy to show. Any composition containing the unary operator $zero$ yields the $zero$ operator; any composition without $zero$ is either $negate$ or

$identity$ depending on the number of $negate$'s used. For example, $negate \circ identity \circ zero \circ negate \equiv zero$.

The second property, $u(x \odot y) = u(x) \odot u(y)$, requires some thought. We need to show that for all instances of $\odot$ and u on the left-hand side there exists an equivalent instance of $\odot$ and u on the right-hand side. We find that the operator $negate$ does not distribute over the operators $max, min, \%$, while all other unary operators distribute over all binary operators. For instance, $negate(x/y) \equiv negate(x)/identity(y)$.

In cases when a unary operator u' does not distribute across a binary operator $\odot'$, such as $negate(max(x, y))$, we extend the grammar of $\odot$ with a new binary operator $u'(x \odot' y)$, which restores the expressiveness by hard-coding the combination of the two operators. In practice, we found that it was not necessary to keep the $combiner$ grammar equivalent to the $update$ grammar by adding these new operators.

To see that the $combiner$ grammar is equivalent to the $update$ grammar if the two conditions hold, observe that any expression from the latter grammar can be rewritten into an expression of the former grammar with these transformations:

$u(x_i) \rightarrow u(x_i)$ // base case
$u \circ \ldots \circ u(exp) \rightarrow u(exp)$ // by property 1
$u(exp_1 \circ exp_2) \rightarrow u(exp_1) \circ u(exp_2)$ // by property 2

While we are currently performing the legality check manually, in the future, this step can be automated using the synthesizer, which can automatically find the equivalent instance on the right-hand side, if one exists.

Symmetry reduction for commutative binary operators We now write the template for the function $reduce$, which encodes all possible ways of reducing k inputs $x_1, \ldots, x_k$, into one output with exclusively binary operators. If any operators in $reduce$ are commutative, then the space of reducers includes "symmetric" expressions that are identical up to commutativity. In this section, we explain the symmetry reduction for commutative binary operators, which prunes the search space without losing any expressiveness. For the sake of presentation, we assume all binary operators are commutative; we explain below how to handle $reduce$ that may include non-commutative operators. Note that the binary operators may or may not be associative. We do not address symmetry reduction based on associativity because we found symmetries due to associativity less harmful in our experiments.

The space of functions defined by $reduce$ is defined with this grammar:

$reduce(A) := reduce(A_1) \odot reduce(A_2)$
$reduce(x) := x$

Here, A denotes the set of inputs, $A = x_1, \ldots, x_n$, and A_1, A_2 an arbitrary partition of the set A. The $reduce$ function takes in a set of inputs, arbitrarily splits the set into two partitions, recursively reduces each partition, combining the results with an arbitrary binary operator.

Because all operators in $\odot$ are commutative, two expression trees using the operators $\odot$ are symmetric if one tree can be transformed into another tree by swapping children at the $\odot$ nodes. The symmetry is an equivalence relation. For example, these two trees belongs to the same equivalent class: $x \odot (y \odot z)$ and $(y \odot z) \odot x$. To reduce the symmetry, we define a canonical tree for each equivalence class and admit only the canonical expression tree while rejecting non-canonical trees.

We now define a predicate that identifies canonical trees. Consider the power set 2^A, where A is the set of inputs. Each element B of 2^A has a natural correspondence to a bit mask b, where the bit at $b[i]$ marks whether B contains element x_i. The lexicographic order on bit-masks defines a total order over 2^A, where $B_1 < B_2$ if $b_1 < b_2$. We extend the total order on 2^A to a partial order on T, the set of all expression trees with inputs $B \in 2^A$. Let t_1 be an expression tree with inputs B_1, and t_2 with inputs B_2, we say $t_1 < t_2$ iff $B_1 < B_2$. Notice that this is a partial order because given a subset of inputs B, there are many expression trees over B, which are un-ordered. The canonical predicate c states that an expression tree t is canonical if its children are canonical and that its left child is less than its right child:

$$\begin{aligned} c(leaf) &= true \\ c(tree(l, r)) &= c(l) \wedge c(r) \wedge l < r \end{aligned}$$

We now give an outline of a proof that each equivalence class has a unique canonical element:

1. *Existence of a canonical element.* Any tree can be canonicalized by identifying all nodes whose left child is greater than its right child, and swapping the children. Note that swapping the children at a particular node does not affect the lexicographic orders of children at any other sub-trees. Thus, the canonical tree of an equivalence class can be obtained by selecting any tree from the class and canonicalizing it. Since the canonicalization is performed by swapping, the canonicalized tree will be in the same class, proving every class has a canonical element. Figure 2 illustrates the canonicalization process.
2. *Uniqueness of the canonical element.* Suppose an equivalence class contains two distinct canonical expression trees t_1 and t_2. Then there exists a sequence of swaps that transform t_1 into t_2. Consider a particular sub-tree which has its children swapped. Since neither subsequent not previous swaps can change the order of the children of this particular sub-tree, either t_1 or t_2 must have its children ordered in descending order at this sub-tree, and is thus not canonical.

Up to this point we have assumed that all binary operators are commutative. To canonicalize trees with non-commutative operators, we define for every non-commutative

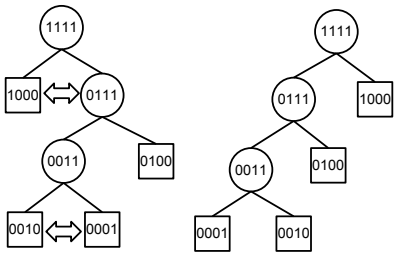

Figure 2. Canonicalization of the left tree into the right tree. The process swaps the mis-ordered children at two nodes. Each node is labeled with the subset of inputs to the subtree rooted at that node. The labels 0001, 0010, 0100, 1000 denote x_1, x_2, x_3, x_4, respectively.

binary operator $\ominus$ a companion operator, $\ominus'$, defined with $\ominus'(x, y) = \ominus(y, x)$. With the companion operator, we can swap children of a non-commutative nodes by substituted the operator with its companion.

Having defined the canonical predicate c, we can explain how it is used during synthesis. The synthesizer consults the predicate to rule out trees that are not canonical. The predicate is evaluated as the tree is generated, to allow for early pruning of the search compared to the alternative of evaluating the predicate on a complete tree. We show here the template the defines the expression tree and evaluates the predicate:

```
def reduce( A ):
    if (|A| == 1):
        return the only element in A
    else:
        A_1, A_2 = split A to two arbitrary subsets
        assert A_1, A_2 are nonempty, disjoint unions of A
        assert weight(A_1) < weight(A_2)
        return reduce(A_1) ⊙ reduce(A_2)
```

The synthesizer always selects a canonical tree. After we have added the companion operators to the grammar $\odot$, the may use either the original operator or its companion, depending on how inputs are used in the expression tree.

In our experiments, we have found that this template did not lead to sufficient scalability. This was in part because at the time of our experiments, the Sketch synthesizer translated recursive templates like $reduce$ eagerly into an exponentially large formula.

Therefore, we precomputed the $reduce$ template statically, effectively evaluating the canonical predicate c on all subtrees, and inlined the resulting grammar into the combiner template. For illustration, here is the resulting encoding for a combiner of three elements:

$$combiner(x, y, z) := reduce(u(x), u(y), u(z))$$
$$reduce(x, y, z) := (x \odot y) \odot z \mid (x \odot z) \odot y \mid (y \odot z) \odot x$$

3.5 Constraint Solving

In addition to suitably encoding a template for the DPA, we need to assert a correctness condition that guides the synthesizer towards a correct program. The specification is given as a predicate $spec(input, output)$ that the output of the DPA must satisfy. This predicate is usually implemented with a naive, exponential-time algorithm. We assert the correctness condition over a bounded domain I of inputs. We define I with arrays of length N, where each array element contains integer or tuple of integers ranging between $[0, M]$.

We have found that a naive algorithm is too expensive to encode in Sketch because it is translated to exponentially large formulas. Therefore, we precompute the specification as a table, by evaluating the naive algorithm offline, with a script. The specification is then supplied to Sketch in a table-lookup form:

```
naive( A ):
    if (A = [0, 0, ..., 0, 0]) return out_1
    ⋮
    if (A = [M, M, ..., M, M]) return out_m
```

We have several options in how we define the correctness condition. First, one can assert that the template is equivalent to the naive algorithm, by asserting, at once, that the program must be correct on all inputs from I. This is inefficient because it create a large constraint system.

Instead, we take advantage of the counterexample-guided inductive synthesis (CEGIS) loop in the Sketch synthesizer. This refinement procedure reduces the function space iteratively, by gradually asserting that the synthesized program must be correct on one more input. This input is obtained by checking the correctness of the synthesized candidate algorithm. When we get a correct algorithm, we stop. If the algorithm is incorrect, we have an input-output pair on which the candidate algorithm and the naive algorithm diverge. We assert this particular input-output pair as an additional constraint to the template. This reduces the load in the synthesizer significantly by having it consider only the inputs that the previous algorithm failed on.

Our final option is to assert correctness on all arrays of a small size, then gradually consider bigger input arrays.

Finally, we want to note that whenever a naive algorithm is difficult to write, the input-output pairs needed for synthesis can be generated manually from a declarative specification. Usually, only a small number of examples is needed to synthesize a correct algorithm, and these can be obtained manually.

4. Experiments

In this section we evaluate our approach by synthesizing various algorithms. We wish to evaluate whether the symmetry and space reduction techniques we developed in Sections 3.4 and 3.3, along with the assertion techniques in Section 3.5 make the synthesizer efficient while retaining completeness

of the template. The synthesizer is evaluated on the following benchmark problems.

The first four benchmark problems are from a class of "multi-marking" problems. In these problems, given an array of integers, the objective is to find another array of integers, an assignment, such that the dot product of the two arrays is maximized. The problems differ in their requirements for the assignment arrays. The *Maximal Independent Sum* (MIS) problem takes an array of positive integers and finds a selection array consisting of 0 and 1 with no adjacent 1s. The *Maximal Segment Sum* (MSS) problem takes an array of positive or negative integers and finds a selection of 0 and 1 such that all the 1s are consecutive. The *Maximal Alternating Sum* (MAS) problem takes an array of positive or negative integers and finds a selection of 0, 1, and -1 such that all the 1s and -1s are consecutive, and that the 1s and -1s must interleave. The *Maximal Multi-Marking* (MMM) problem takes an array of positive or negative integers and finds a selection of 0, 1, and -1 such that no two 0, 1, or -1 are consecutive.

Our next benchmark is the *Assembly Line* (ASSM) problem. Given two assembly lines A and B, and the costs for staying on a line ($stay_i$) or switching to a different line ($switch_i$), the problem is to find the minimal cost of traversing the assembly.

Our last two benchmarks are the *OtherSum* (OSUM) and the *Extended Euclid* (EUC) problems, which will be described below.

We will first show the synthesized solutions, followed by empirical evaluation on the effects of our encodings.

4.1 Solutions to Synthesis Problems

4.1.1 MIS

User Hints:

$unary(x) = x \mid 0$
$binary(x, y) = x + y \mid max(x, y)$
$opt = max$

Synthesized Recurrence Relation:

mis(n) = max(pick(n),no_pick(n))
pick(n) = max(no_pick(n−1) + array(n))
no_pick(n) = max(pick(n−1),no_pick(n−1))
pick(0) = 0
no_pick(0) = 0

Here pick(n) is the sub-problem of the best legal assignment up to the n^{th} array element where we are forced to pick the n^{th} element. no_pick(n) denotes the best legal assignment up to the n^{th} element and that we are forced to avoid the n^{th} element.

4.1.2 MSS

User Hints:

$unary(x) = x \mid 0$
$binary(x, y) = x + y \mid max(x, y)$
$opt = max$

Synthesized Recurrence Relation:

mss(n) = max(suffix(n),best(n))
suffix(n) = max(max(0,suffix(n−1))+array(n))
best(n) = max(suffix(n−1),best(n−1))
suffix(0) = 0
best(0) = 0

Here suffix(n) denotes the best suffix assignment ending at the n^{th} array element, and best(n) denotes the best legal assignment up to the n^{th} element.

4.1.3 MAS

User Hints:

$unary(x) = x \mid 0 \mid -x$
$binary(x, y) = x + y \mid max(x, y)$
$opt = max$

Synthesized Recurrence Relation:

mas(n) = max(suffix_pos(n),suffix_neg(n),best(n))
suffix_pos(n) = max(0,suffix2(n−1)) + array(n)
suffix_neg(n) = max(0,suffix1(n−1)) − array(n)
best(n) = max(suffix_pos(n−1),suffix_neg(n−1),best(n−1))
suffix_pos(0) = 0
suffix_neg(0) = 0
best(0) = 0

Here suffix1(n) denotes the best legal suffix up to the n^{th} element that ends in 1 while suffix2(n) ends in -1. best(n) denotes the best legal assignment up to the n^{th} element.

4.1.4 MMM

User Hints:

$unary(x) = x \mid 0 \mid -x$
$binary(x, y) = x + y \mid max(x, y)$
$opt = max$

Synthesized Recurrence Relation:

mmm(n) = max(mark_0(n),mark_1(n),mark_−1(n))
mark_ig(n) = max(mark_−1(n−1),mark_1(n−1))
mark_pi(n) = max(mark_0(n−1)+array(n),
mark_−1(n)+array(n−1))
mark_ne(n) = max(mark_0(n−1)−array(n),
mark_1(n)−array(n−1))
mark_ig(0) = 0
mark_pi(0) = 0
mark_ne(0) = 0

Here mark_ig(n),mark_pi(n),mark_ne(n) are the sub-problems of the best legal assignment up to the n^{th} array element where we are forced to ignore,pick,and negate the n^{th} element, respectively.

4.1.5 ASSEM

User Hints:

$unary(x) = x \mid 0$
$binary(x, y) = x + y \mid min(x, y)$
$opt = min$

Synthesized Recurrence Relation:

assem(n) = min(line1(n),line2(n))
line1(n) = min(line1(n−1)+stay1(n),line2(n−1)+switch1(n))
line2(n) = min(line2(n−1)+stay2(n),line1(n−1)+switch2(n))
line1(n) = 0
line2(n) = 0

Here line1(n),line2(n) denote the optimal cost of n assemblies that end in line1 and line2, respectively.

4.1.6 Extended Euclid Algorithm

Here we attempt to synthesize the extended Euclid Algorithm (EUC): Given two integers x and y with greatest common divisor (gcd) of 1, find coefficients a and b such that $a * x + b * y = 1$. The greatest common devisors of 2 numbers can be found by the Euclid algorithm as follows:

```
euclid(x,y):
  p1 = array()
  p2 = array()
  p1[0] = x
  p2[0] = y
  i = 0
  while(p2[i] != 0):
    i += 1
    p1[i] = p2[i−1]
    p2[i] = p1[i−1] % p2[i−1]
  return p1[i]
```

Suppose the user vaguely remembers that EUC is performed by traversing the computing histories of Euclid's Algorithm, p1 and p2, backwards. The user first reverse the histories: $q1 = reverse(p1), q2 = reverse(p2)$, and asks the synthesizer to formulate a DPA that computes the coefficients a and b by expressing the following constraints:

```
on input (x,y):
    (a,b) = DPA(q1,q2)
    assert(a∗x+b∗y == 1 OR a∗y+b∗y == −1)
```

Note that the user was not completely sure if the coefficient's parities, hence she expresses a relaxed constraint, and is happy if the coefficients can compute either positive or negative 1.

User Hints:

$unary(x) = x \mid 0$
$binary(x, y) = x + y \mid x * y \mid x\%y \mid x - y \mid x/y$

Synthesized Recurrence Relation:

e_euc(n) = (p1(n),p2(n))
p1(n) = p2(n−1)
p2(n) = p1(n−1)+p2(n−1)∗(q1(n)/q2(n))
p1(n) = 1
p2(n) = 1

Here there is no obvious concise interpretation for the meanings of the sub-problems. In short, solution takes advantage of the fact that top(n-1) = bot(n) and bot(n-1) = top(n) mod bot(n).

4.1.7 Other Sum

In this section we study the composability of the FOR templates, and argue that extending the template for a specific problem can be done without expertise in program synthesis.

OtherSum problem: given an array of integers $a = [a_1, ..., a_n]$, compute the array $b = [s - a_1, \ldots, s - a_n]$ where $s = \Sigma_{i=1}^{n} a_i$. That is, $b[i]$ equals the sum of every element in a other than the ith one. The catch is that we cannot use subtraction and the algorithm must be $O(n)$ time. We use the easier-to-express subtraction algorithm can as specification:

```
def spec (A = [a_1,...,a_n])
    total = 0
    for i from 1 to n: total += A[i]
    ret = array()
    for i from 1 to n: ret[i] = total − A[i]
    return ret
```

Next, we asked the synthesizer to produce a DPA restricted to use only addition. The synthesizer answers that no such algorithm exists, in less than a second.

Since the domain theory does not include the desired algorithm, we make some conjectures on the properties of the desired algorithm:

1. In the specification, the answer is computed with multiple loops. Perhaps multiple passes over the array are needed also in the desired algorithm.
2. Since the synthesizer failed to identify a recurrence that works around the lacks of subtraction, the key to an efficient algorithm seems to be a different traversal order.

We then wrote a more general template that encoded these conjectures. The first conjecture is encoded by composing multiple DPA templates:

```
def sketch(A):
    temp1 = DPA1(A)
    temp2 = DPA2(A,temp1)
    result = DPA3(temp1,temp2)
    return result
```

Both DPA1 and DPA2 are modified to return the entire sub-problem arrays (as in Extended Euclid algorithm, we extract the entire sub-problem p1 and p2) so that it can be consumed by the next DPA as inputs, whereas DPA3 is acting in place of the function $terminate$ except it now attempts to summarize entire arrays rather than just values of the last iteration.

To encode the second conjecture, we relaxed the loop iteration order from the default left-to-right to any arbitrary array traversal order, to be selected by the synthesizer. We do this by asking the synthesizer to produce an arbitrary reordering $reorder$ that translates the iteration space $[1, \ldots, n-1]$ to the space $[r(1), \ldots, r(n-1)]$. The map $reorder$ is implemented in DPA1 and DPA2 using the array reorder that is initialized by the synthesizer. The synthesizer finds an initialization of reorder that leads to a correct algorithm.

```
def DPA1(A):
    // initialize the array with n synthesized constants
    int[n] reorder = [1, ..., 1] | ... | [n − 1, ..., n − 1] // all reorders
    ret = array()
    ret[reorder[0]] = 0
    for i from 1 to n−1:
        ret[reorder[i]] =
            combiner2(ret[reorder[i−1]], A[reorder[i−1]])
    return ret

def DPA2(A,B):
    int[n] reorder = [1, ..., 1] | ... | [n − 1, ..., n − 1] // all reorders
    ret = array()
    ret[reorder[0]] = 0
    for i from 1 to n−1:
        ret[reorder[i]] = combiner3(ret[reorder[i−1]],
                                    A[reorder[i−1]],B[reorder[i−1]])
    return ret

def DPA3(A,B,C):
    ret = array()
    for i from 0 to n−1:
        ret[i] = combiner3(ret[i−1],A[i−1],B[i−1])
    return ret
```

The observant reader will notice that we have asked the synthesizer to produce a map $reorder$ that is fixed to a particular value of n, and will (only) work for a fixed value n.

We have a reason to ask the synthesizer for $reorder$ bound to a fixed n. The synthesized $reorder$ serves as a demonstration of a particular traversal order that allows us to solve OtherSum in $O(n)$-time. As such, it reveals the algorithm on a given n, in the spirit of angelic programming [5]. The demonstration provides hints for the user on how the problem might be solved in the general case for all n.

There exist many maps $reorder$ that lead to a correct algorithm, and the synthesizer is capable of returning any of such maps. One possible value of reorder is $[3, 4, 0, 1, 2]$. This may seem like a random traversal. However, a closer inspection yields a remarkable observation: The iteration reordering on DPA1 and DPA2 are always the reverse of one another! For example, if the reorder is $[3, 4, 0, 1, 2]$ in DPA1, then the reorder is $[2, 1, 0, 4, 3]$ for DPA2. This suggested to us that the traversal in DPA1 could be left-to-right and the traversal in FOR2 could be right-to-left. To synthesize a suitable map $reorder$ in DPA2, we replaced reorder[i] with n−i+??, which then synthesized the following final program:

```
otherSum(A):
    temp1 = array()
    temp1[0] = 0
    for i from 1 to n−1:
        temp1[i] = temp1[i−1]+A[i−1]
    temp2 = array()
    temp2[0] = 0
    for i from 1 to n−1:
        temp2[n−i−1] = temp[n−i]+A[n−i]
    ret = array()
    for i from 0 to n−1:
        ret[i] = temp1[i] + temp2[i]
    return ret
```

4.2 Empirical Studies

In this section we show the effects of different encodings of the recurrence and different assertion schemes on the scalabilities of the solver.

We ran all our experiments on a four CPU, 2GHz machine, with 4GB of memory, since we iterate through the numbers of sub-problems, here we shown only the data of the last iteration.

Figure 3 compares the representation size and the function space size of our encodings. The representation size is measured in nodes, where each node is an operator in the formula constructed from the template by the Sketch synthesizer. The function space size is measured in bits, where each combination of bits correspond to a distinct completion of the partial program, i.e., a candidate program. We remark that the naive grammar encoding described in Section 3.2 creates a constraint system so large that the synthesizer runs out of memory while trying to construct it. In contrast, the combiner (Section 3.4) and extender (Section 3.3) encoding significantly reduce the formula size and the space of functions explored. For the harder problems, the extender encodings reduce the representation and function space size dramatically. For instance, MAS has a function space of 172 bits with the combiner encoding, but a mere 40 bits for the extender encoding, which is a 10^{40}-fold improvement.

Figure 4 shows that reducing the problem's representation size and function space reduce the synthesis time for a particular algorithm. However, it is not true across algorithms. For instance, MMM has fewer nodes and function space size than MAS, yet it took significantly longer to synthesize. This maybe caused by an easier constraint of the

Benchmk	GR		COMB		EXT	
	rep	space	rep	space	rep	space
MIS	mem	N.A.	871	25	453	9
MSS	mem	N.A.	876	25	458	9
MAS	mem	N.A.	4785	172	1101	40
MMM	mem	N.A.	4251	71	634	19
ASM	mem	N.A.	mem	N.A.	1905	33
OSM	mem	N.A.	1023	23	N.A.	N.A.
EUC	mem	N.A.	38837	766	N.A.	N.A.

Figure 3. Constraint Size and Function Space. GR denotes the naive grammar encoding of the update functions, while COMB and EXT employ combiners and extenders, respectively. Note that the extender encoding is only available for optimization problems.

Prob.	COMB		EXT	
	time	space	time	space
MIS	16	355	9	95
MSS	35	784	10	41
MAS	N.A.	mem	467	193
MMM	N.A.	mem	5909	790
ASM	N.A.	mem	101	90
OSM	2	54	N.A.	N.A.
EUC	N.A.	mem	N.A.	N.A.

Figure 4. Synthesis Time (s) and Memory Usage (MB).

Benchmk	ALL		CEG		INC	
	time	space	time	space	time	space
MIS	9	355	9	95	10	31
MSS	10	784	10	41	14	48
MAS	689	mem	467	193	58	160
MMM	N.A.	mem	5909	790	626	316
ASM	N.A.	mem	101	90	30	60
OSM	15	377	2	54	N.A.	N.A.
EUC	N.A.	mem	N.A.	mem	316	1410

Figure 5. Solving Time (s) and Memory Usage (MB). ALL denotes we assert all the correctness conditions at once, CEG uses the default CEGIS loop, while INC asserts correctness conditions from small to large.

MAS problem, perhapse multiple candidate algorithms are correct. Also notice that the EUC algorithm failed to synthesize with the default CEGIS assertions since the constraint system is too difficult.

Next, we experiment with the different encodings of correctness. Figure 5 shows the effect of different encodings of the correctness assertions, as discussed in Section 3.5 on the runtime and memory usage of the solver. Notice the dramatic improvement when the incremental assertion is employed, turning the EUC from unsynthesizable to synthesizable.

5. Related Work

We relate our work to alternative approaches addressing the two novel ideas we present here, namely that of semi-automatic synthesis of DPA, and that of using partial programs as domain theories.

5.1 Derivation/synthesis of DPA

Most prior work concerned with the derivation of efficient dynamic programming algorithms exploits an *optimization theorem*, while some prior work has attempted to formulate the synthesis of DPA as a constraint-solving synthesis problem. We will first discuss the refinement transformation-based approaches below, followed by program synthesis-based DPA generation.

With optimization theorem-based approaches, it is unclear how a user may take her algorithmic insight and find a suitable transformations that exploits the insight. She would have to link her program insight to the meta-reasoning at the level of transformations. In our approach, the programmer does not need to reason about transformations. She thus reasons directly about programs (if at all the domain template needs tweaking), and not about transformations that produce programs. With prior synthesis-based approaches, a constraint-system has to be setup for each DPA instance.

Refinement theorem-based Yao shows that if a problem, formulated as a recurrence relation, satisfies the quadrangle inequality [26], then the problem can be solved in quadratic time, which improves on the asymptotic complexity of a naive cubic-time algorithm that implements the specification directly. The quadrangle inequality allows us to safely restrict the range of subproblems that need to be considered to produce an optimal solution. Phrased in terms of the "memoization table" computed by a dynamic programming algorithm, the inequality prunes the number of cells that the algorithm needs to visit. The inequality does not seems to applicable to linear-time problems that we consider in this paper. Additionally, the problem must be formulated in a form that satisfies the preconditions of the theorem.

Derivation of linear-time dynamic programming algorithms has been considered by Bird and de Moor [4]. The optimization theorem, called the thinning theorem, transforms problems into an efficient solution if two suitable preorders are found. To reduce the burden of the work needed before the theorem can be applied, Sasano *et al* restrict their domain to maximum-weightsum problems, which can be automatically translated to efficient algorithms as long as the problem can be expressed as a mutumorphism [16]. As far as we can tell, this process requires the programmer to identify the suitable sub-problems, which we seek to avoid with our synthesis methodology.

Existing approaches for derivation of dynamic programming algorithms rely on instantiating the program from an algorithmic theory [14]. The approach relies on the user to

invent or symbolically calculate dominance relations for the algorithmic theory.

Program synthesis-based In most partial-program synthesizers [20, 23], each partial program/template is specific to the synthesis problem at hand. In other words, only one functionally-unique program can be synthesized from a partial program. (One exception may be synthesis, i.e., inference, of program invariants in the context of program analysis, where a single template can synthesize into many possible invariants [7, 8, 22, 21].) Srivastava *et al* synthesized the first dynamic programming algorithm from a partial program. For each problem they have a separate template that may capture multiple orthogonal solutions which verifiably meet the specification, but which is restricted to that problem. Here, we generalize templates enough to be reusable across problems.

5.2 Partial programs as domain theories

Our work is motivated by the desire to equip programmers with practical program synthesis tools. We believe that it is difficult to achieve this goal if we ask programmers to develop optimizing program transformations and/or instantiate algorithmic theories. Everyday programmers may miss the necessary formal background; another open question is teachability of program derivation tools. For that reason, our "domain theory" for dynamic programming is purely syntactic, expressed as a program template called a *partial program*. With partial programs, programmers can write skeletons of desired code rather than a transformation that produces that code.

We develop our partial program-based domain theory in the SKETCH language, which has been used to synthesize cryptographic algorithms, stencil kernels and concurrent data structures [20, 18, 19]. While previous uses of SKETCH have developed one partial program per desired synthesized program, we have developed a partial program that serves as a domain theory from which an entire class of dynamic programming problems can be synthesized. This is the first use of programmer-editable partial programs as a domain theory.

Informally, we say that a partial program forms a domain theory if the partial program can synthesize programs for a range of specifications. The first synthesizer with such a partial-program domain theory was SmartEdit, which learnt editor macros from programmer demonstrations [12]. The synthesis algorithm was the version space algebra; the partial program encoded a language of learnable macros. While a single partial program could learn a large range of macros, the domain theory was produced by the tool creator and was not modifiable by the user, while we attempt to make the domain theory user-modifiable.

Itzhaky *et al* [11] developed partial-program-based domain theories that were programmable by the synthesizer expert. Their partial programs, called target languages, could produce linear-time graph classifiers and perform finite differencing on set operations. It will be interesting to explore whether dynamic programming can be expressed in their language. Their partial programs were expressed in a restricted language dictated by their synthesis algorithm and are likely accessible only to the synthesis expert.

6. Conclusion

We have shown that linear-time dynamic programming algorithms can be synthesized with rather small guidance from the programmer. Our synthesizer is based on constraints solving, where a program template defines the space of programs explored by the synthesizer. This paper focuses on how to encode the dynamic programming template so that it includes all algorithsm of interest without making constraint solving prohibitively expensive.

Acknowledgments

This material is based on work supported by U.S. Department of Energy grant under Grant Number DE–SC0005136, by NSF under Grant Number CCF–0916351, by a grant from University of California Discovery program, as well as by contracts with Intel and Microsoft.

References

[1] D. Andre and S. J. Russell. State abstraction for programmable reinforcement learning agents. In *AAAI/IAAI*, pages 119–125, 2002.

[2] e. a. Armando Solar-Lezama. The Sketch synthesizer. http://sketch1.csail.mit.edu/demo/.

[3] R. Bellman. The theory of dynamic programming. *Bull. Amer. Math. Soc.*, 60:503–515, 1954.

[4] R. Bird and O. de Moor. *The Algebra of Programming*. Prentice-Hall, 1996.

[5] R. Bodík, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with angelic nondeterminism. In Hermenegildo and Palsberg [9], pages 339–352.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, New York, 2001.

[7] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, 2008.

[8] S. Gulwani, S. Srivastava, and R. Venkatesan. Constraint-based invariant inference over predicate abstraction. In *VMCAI '09: Proceedings of the 2009 Conference on Verification Model Checking and Abstract Interpretation*, 2009.

[9] M. V. Hermenegildo and J. Palsberg, editors. *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. ACM, 2010.

[10] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, pages 36–46, 2010.

[11] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In

W. R. Cook, S. Clarke, and M. C. Rinard, editors, *OOPSLA*, pages 36–46. ACM, 2010.

[12] T. A. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.

[13] U. Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1989.

[14] S. Nedunuri and W. R. Cook. Synthesis of fast programs for maximum segment sum problems. In J. G. Siek and B. F. 0002, editors, *GPCE*, pages 117–126. ACM, 2009.

[15] S. Nedunuri, D. R. Smith, and W. R. Cook. A class of greedy algorithms and its relation to greedoids. In A. Cavalcanti, D. Déharbe, M.-C. Gaudel, and J. Woodcock, editors, *ICTAC*, volume 6255 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2010.

[16] I. Sasano, Z. Hu, M. Takeichi, and M. Ogawa. Make it practical: A generic linear-time algorithm for solving maximum-weightsum problems. In *In Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP'00*, pages 137–149. ACM Press, 2000.

[17] I. Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Applied Mathematics*, 155(12):1539–1548, 2007.

[18] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.

[19] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent data structures. In *PLDI*, 2008.

[20] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 404–415. ACM, 2006.

[21] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, 2009.

[22] S. Srivastava, S. Gulwani, and J. S. Foster. VS3: SMT solvers for program verification. In *CAV*, 2009.

[23] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, 2010.

[24] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In Hermenegildo and Palsberg [9], pages 313–326.

[25] Wikipedia. Dynamic programming, Aug. 2011.

[26] F. F. Yao. Efficient dynamic programming using quadrangle inequalities. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, STOC '80, pages 429–435, New York, NY, USA, 1980. ACM.