Chapter 14

Algorithmic Program Synthesis

Rastislav Bodík

1 Where we Were in 2008.

Our group was critical in rejuvenating the field of program synthesis by making it algorithmic, analogously to how model checking automated verification. Our contributions, which played a major role in a 2012 NSF Expedition, tell much of the story of modern synthesis:

- **Specification mining** introduced the benefits of synthesizing API specifications by learning from examples of API usage rather than from API implementations [7, 1, 2].

- **Prospector** observed that restricting the syntactic shape of synthesized code (to a jungloid) was key to efficient synthesis algorithms. Prospector inspired at least five systems.

- **Sketching** introduced two ideas with wide impact. First, partial programs (sketches) gave programmers control over the syntactic shape restriction [10, 11]. Under various names, partial programs are now a common method for “programming” synthesizers with human insight, including in Microsoft FlashFill.

- The second result from sketching was reduction of synthesis to symbolic constraint solving, by phrasing it as counterexample-guided inductive synthesis (CEGIS) [11], later extended for stencils [8] and concurrent programs [9]. CEGIS is now widely used in synthesis, for example, in superoptimization, and the Z3 solver.

2 Where we are now.

The 5-year UPCRC project examined the questions of how to make synthesis available to programmers and how to build synthesizers. Each led to a fundamental result that is likely to have a lasting influence.

- A **solver-aided** language includes programming constructs for synthesis, verification, bug finding, and angelic programming. The notion of solver-aided languages, formed in our group, may become the preferred method for giving programmers access to modern formal methods technology such as SAT/SMT solvers and model checkers. In a recent graduate cs294 course, synthesizers for fourteen distinct problems were built in the style of solver-aided languages, some by students who had little previous exposure to programming languages, synthesis, or formal methods.

- **Rosette** is a framework for construction of solver-aided domain-specific languages, including lightweight synthesizers. The key idea of Rosette is to create a regular interpreter, debug it with usual methods, and finally add solver-aided constructs such as “program holes” with minimal modifications to the interpreter. We have
given an invited tutorial on using Rosette, and it has been used to construct a synthesizer for cache coherence protocols (in collaboration with Intel’s Murali Talupur) and a synthesizer for an ultra-low-power many-core processors GreenArrays 144. Rosette was developed by Emina Torlak, a staff researcher who previously developed Kodkod, the solver for the Alloy checker.

3 Collaboration with Other Upcrc Efforts.

A lot of the work in the parallel web browser project exploited program synthesis. The project, described in a separate chapter, developed a synthesizer of parallel layout engines [5, 6] and several synthesizers based on for programming by demonstration, for example the Quicksilver relational query synthesizer.

Open questions at the beginning. When the UPCRC project started, we had developed the foundational synthesis technology but not the means to make it available to programmers. Three questions summarized our direction:

- Which parallel programming tasks can synthesis automate? We knew that synthesis cannot produce the entire program — analogously to how a programmer does not write the entire program in one swoop — but we did not know how to decompose the parallel programming process and which tasks are automatable. This question led to synthesis of parallel schedules (see the FTL tool in the parallel browser chapter), and synthesis of indexing expressions in parallel reductions (HotPar 2011).

- How will programmers communicate with synthesizers? In our papers up to 2008, the synthesizer was supplied necessary domain knowledge mostly in the form of a sketch — a program template whose holes were completed with the synthesizer. These sketches were written by paper authors rather than actual programmers. It was an open question whether programmers can ever learn how to write and use sketches. This question led to development of a DSL compiler for dynamic programming that hides sketches from the programmer inside the compiler.

- It was also unclear as to what language constructs should be designed to make programmers productive in writing sketches, thus giving them access to the power of the synthesizer. This question led to development of angelic programming, described below, among other results.

4 Angelic Programming.

The idea is to develop a language construct that allows the programmer to consult an oracle. The clairvoyant oracle, consulted at the program’s runtime, can, for example, place a queen on a chessboard so that the oracle can also successfully place the remaining queens. In parallel programming, the oracle can create an instance of a data structure that allows a program to process the data structure in parallel, even with GPU-friendly parallelism. The motivation for this work came from a user study, where we found that programmers found it difficult to debug sketches because they are not regular programs which you can run and trace but rather they are incomplete, unexecutable program templates with holes. The oracle supplies the values that the holes will compute (once the code for the holes are synthesized), thus allowing the sketch to run. This effort quickly became more general than sketch debugging, focusing on how to exploit constraint solvers in programming. The new construct lay foundation for what we now call solver-aided languages. The papers on this work appeared in [3, 4].

5 LL: A Language for Parallel Sparse Matrix Programming.

LL is a high-level language for sparse matrices. LL comes with a compiler that generates parallel code as efficient as a handwritten one, including for blocked sparse formats. This effort was born from our failure to synthesize sparse kernels at the level of C programs, where indexes into sparse matrices are explicit, complicating automated reasoning. After gaining experience with several proving methods, Gilad Arnold observed that it was necessary to lift the level of abstraction to a language with lists, which would allow representing sparse matrices as simple lists of lists. This dramatic change in the level of abstraction allowed proving the full correctness of sparse programs, as shown in ICFP 2010, reprinted in this book, which is the result of collaboration with Johannes Hölzl from the Isabelle theorem prover.
Algorithmic Program Synthesis

project. The other half of the problem—how to compile from this NESL-like language to a low, efficiency level—is described in Gilad’s PhD thesis. The long-term lesson from this effort was a new synthesizer architecture for realistic problems: synthesis and verification happen at a high abstraction level, from which a compiler translates the program to an efficiency language such as C. This compiler can be a classical, rewrite-based compiler, as is the case with the LL compiler, or the compiler could itself be a synthesizer. This lessons was used in the synthesizer for the spatial low-power hardware mentioned below, which lowers high-level code in three synthesis steps, using three carefully chosen abstraction levels.


This effort addressed the challenge of writing sketches by entirely hiding them from the programmers inside a compiler. In our case study, the compiler produced dynamic programming algorithms. More generally, the result is a new method for writing compilers for domain-specific languages. Rather developing tree-rewrite rules for the domain—which is hard, and in fact such rules have been invented only for a few DSL domains, such as regular expressions, grammars, relational queries, and some numeric algorithms—the compiler developer writes sketches (templates) of the target programs to be synthesized by the compiler. Evan Pu led this work, and the paper from SPLASH (OOPSLA) 2011 by is reprinted in this book.

7 Concurrency in Systems Biology.

With the arrival of the postdoc Saurabh Srivastava (an expert in synthesis with PhD from the University of Maryland), we started applying synthesis to biology. Cancer is a disease of cell regulation, and our synthesizer helps biologists distill mutation lab experiments into an executable model of how stem cells communicate to decide their fates, as well as how failures in protein signaling lead to undesirable cell growth. Saurabh’s acumen led him to realize that these models are concurrent (one process per cell) and could be synthesized by extending our algorithms for concurrent programs that appeared in PLDI 2008. The project, led by Ali Sinan Koksal, led to a paper that appeared in POPL 2013. The work is a collaboration with the Microsoft biologist Jasmin Fischer and Nir Piterman of University of Leicester.

8 Spatial Programming for Ultra-Low-Power Many-Cores.

GreenArrays 144 is a many-core processor with 10 times better energy efficiency than a leading embedded processor (TI MSP430). The efficiency comes at the cost of the programmability, from the reduced stack-based ISA to the minimal communication support and tiny cores with a few hundred bytes of memory. We are developing a language and a synthesis-aided compiler for this processor. The synthesizer first partitions the program across virtual cores, then places cores and routes the communication channels, and finally creates optimal machine code. Each of the steps is solved as a synthesis problem reduced to constraint solving. The project, led by Mangpo Phitchaya Phothilimthana, won the Qualcomm 2013 Innovation Fellowship.

Bibliography


Synthesis of First-Order Dynamic Programming Algorithms

Yewen Pu, Rastislav Bodík, and Saurabh Srivastava

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.

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 as 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 solutions. OtherSum and Extended Euclid algorithm, in Sections 4.1.7 and 4.1.6, respectively.

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 $apply([2, 3, 4, 1], [1, 0, 1, 0]) = 6$. The objective is to maximize the value of the assignment. For instance, $MIS([4, 1, 2, 3]) = 7$ 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 = [a_1, \ldots, a_n] )
    best = 0
    for all bitstrings {B = [b_1, \ldots, b_n]}:
        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\}
binar(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:

\[
\text{def mis ( A = [a_1, \ldots, a_n] )}
\]
\[
p1 = \text{array() } \quad // \text{solutions to subproblem 1}
p2 = \text{array() } \quad // \text{solutions to subproblem 2}
p1[0] = 0 \quad // \text{base case for subproblem 1}
p2[0] = 0 \quad // \text{base case for subproblem 2}
\]
\[
\text{for } i \text{ from } 1 \text{ to } n:
    p1[i] = p2[i-1] + A[i]
    p2[i] = \text{max}(p1[i-1], p2[i-1])
\]
\[
\text{return max(p1[n], p2[n])}
\]

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

\(P_1(i)\): solution to \(\text{MIS}(A[0:i])\) provided \(a_i\) was selected;

\(P_2(i)\): solution to \(\text{MIS}(A[0:i])\) 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.

\[
\begin{align*}
\text{MIS}(n) &= \max(P_1(n), P_2(n)) \\
P_1(n) &= \begin{cases} 
P_2(n - 1) + A(n) & \text{if } n > 0 \\
0 & \text{if } n = 0
\end{cases} \\
P_2(n) &= \begin{cases} 
\max(P_1(n-1), P_2(n-1)) & \text{if } n > 0 \\
0 & \text{if } n = 0
\end{cases}
\end{align*}
\]

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:

• \textit{Completeness}: How to define the space of DPAs so that it includes all DPAs of interest?

• \textit{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:

\[
P(n) = f(P(g(n)), P(h(n)), n)
\]

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

\[
P(n) = f(P(n-1), n)
\]
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:

```python
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, `init`, returns an integer constant (in our experiments, the constant was restricted to $-\infty, 0, \text{and } \infty$). The propagation function, `update`, and the termination function, `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 $\{|e_1|e_2|\ldots|e_k\}$ operator asks the synthesizer to select between expressions $e_1$ and $e_2$.

- `init() := \{ |-\infty | 0 | \infty | \}\`
- `update(x,y,aval) := \{| x | aval | y+x | x+max(y,aval) | */...*/\}`
- `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 $\text{opt}$ 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

```
update(x,y,aval) := \text{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 decisions, 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 \( \text{opt} \) 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:

\[
\begin{align*}
\text{update}(x,y,\text{aval}) & := \max(\text{choose\_subset}(f_1(x,\text{aval}),f_2(y,\text{aval}))) \\
\hat{f}_i(x,\text{aval}) & := \{|x|, \text{aval}, \max(x,\text{aval}) | \ldots \} 
\end{align*}
\]

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

**FOR Algorithm:**

\[
\begin{align*}
\text{FOR}(a) & = \text{terminate}(p_1(n), \ldots, p_k(n)) \\
p_1(0) & = \text{init}_1() \\
\vdots \\
p_k(0) & = \text{init}_k() \\
p_1(m) & = \text{update}_1(p_1(m-1), \ldots, p_k(m-1), a(m)) \\
\vdots \\
p_k(m) & = \text{update}_k(p_1(m-1), \ldots, p_k(m-1), a(m))
\end{align*}
\]

Yewen Pu, Rastislav Bodík, and Saurabh Srivastava 453
The correctness condition asserts that the synthesized algorithm is correct on all problem instances from a set $I$ of small-size instances:

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

We define some terminology:

- A subproblem, denoted $p_i$, is a series of subproblem instances that are solved with the same update function.
- A subproblem 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 $update_i(p_1(m-1), \ldots, p_k(m-1), a_m)$. 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:

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

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 $k$th-order recurrences via the following reduction:

$$p(m) = update(p(m-1), \ldots, p(m-k))$$

$$\rightarrow$$

$$p(m) = p_1(m)$$
$$p_1(m) = update(p_1(m-1), \ldots, p_k(m-1))$$
$$p_2(m) = p_1(m)$$
$$\vdots$$
$$p_k(m) = p_{k-1}(m)$$

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, $a[1]$.  

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 \( \text{update}_i \) 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.

The termination function is a one-step update function that operates on the subproblems from the last, \( n \)th 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 subproblems:

```python
def algorithm ( a = [a_1, ..., a_n] ):
    # create arrays to hold the values of subproblems
    p_1 = array()
    ...
    p_k = array()
    # initialize
    p_1[0] = init()
    ...
    p_k[0] = init()
    # update the subproblem 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:

```python
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 \( \text{opt}(S_i(m)) \), i.e., the \( \text{opt} \) function is at the root of the computation tree. During synthesis, we can thus rule out update functions where \( \text{opt} \) is syntactically in a non-root position.

2. Each solution \( s_{i,j}(m) \in S_i(m) \) is computed from exactly one subproblem \( p'(m-1) \). 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.

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 \( \text{ext}_{j,l} \), \( l \leq k \), \( j \leq r \). The solutions \( \text{ext}_{j,l} \) 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.

\[
\begin{align*}
\text{ext}_{1,1} &= \text{combiner}(p_1, a) \\
& \vdots \\
\text{ext}_{r,1} &= \text{combiner}(p_1, a) \\
& \vdots \\
\text{ext}_{1,k} &= \text{combiner}(p_k, a) \\
& \vdots \\
\text{ext}_{r,k} &= \text{combiner}(p_k, a)
\end{align*}
\]

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

\[
p_i(m) = \text{opt}(\text{select}_s(\text{ext}_{1,1}, \ldots, \text{ext}_{r,k}))
\]

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 \( \text{pick} \) and \( \text{no\_pick} \). The synthesizer first creates the solutions: We can extend the subproblem \( \text{no\_pick} \) into two solutions, by picking versus not picking the current array element:

\[
\begin{align*}
\text{ext}_{\text{pick}}(\text{no\_pick}(m-1)) &= \text{no\_pick}(m-1) + a_m \\
\text{ext}_{\text{no\_pick}}(\text{no\_pick}(m-1)) &= \text{no\_pick}(m-1) + 0
\end{align*}
\]

In contrast, we can extend the subproblem \( \text{pick} \) only into one solution, by not picking the current element, because we cannot pick contiguous array elements:

\[
\text{ext}_{\text{no\_pick}}(\text{pick}(m-1)) = \text{pick}(m-1) + 0
\]
Next, to find an update function that solves the no_pick subproblem, the synthesizer needs to select from the three solutions, by resolving the select template:

\[
\text{no\_pick}(m) = \max(\text{select}_a(\text{ext\_pick}(\text{no\_pick}(m - 1)) \\
\text{ext\_no\_pick}(\text{pick}(m - 1)), \\
\text{ext\_no\_pick}(\text{no\_pick}(m - 1))))
\]

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.

\[
\text{no\_pick}(m) = \max(\text{ext\_no\_pick}(\text{pick}(m - 1)), \\
\text{ext\_no\_pick}(\text{no\_pick}(m - 1)))
\]

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 \( \text{identity}(\text{zero}(\text{max}(x, y) + z)) \) as well as the semantically equivalent expression \( \text{max}(\text{zero}(x), \text{zero}(y)) + \text{zero}(z) \). 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 leaves of the expression tree:

\[
\text{combiner}(x_1, \ldots, x_k) := \text{reduce}(u(x_1), \ldots, u(x_k))
\]

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: \text{negate}, \text{identity}, \text{zero}; the set of binary operators are \(+, -, *, /, \text{max}, \text{min}, \%\). The reduce 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, \( \text{negate} \circ \text{identity} \circ \text{zero} \circ \text{negate} \equiv \text{zero} \).

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 \( \text{max}, \text{min}, \% \), while all other unary operators distribute over all binary operators. For instance, \( \text{negate}(x/y) \equiv \text{negate}(x)/\text{identity}(y) \).

In cases when a unary operator \( u’ \) does not distribute across a binary operator \( \odot’ \), such as \( \text{negate}(\text{max}(x, y)) \), 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:

\[
\begin{align*}
\text{base case} & \\
\text{by property 1} & \\
\text{by property 2} & 
\end{align*}
\]

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:
Each element 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:

\[
\begin{align*}
c(leaf) &= \text{true} \\
c(tree(l,r)) &= c(l) \land c(r) \land l < r
\end{align*}
\]

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 subtree 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 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 that defines the expression tree and evaluates the predicate:

```python
def reduce(A):
    if (len(A) == 1):
        return the only element in A
    else:
        A1, A2 = split A to two arbitrary subsets
        assert A1, A2 are nonempty, disjoint unions of A
        assert weight(A1) < weight(A2)
        return reduce(A1) \odot reduce(A2)
```

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 | (x \odot z) \odot y | (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 \( \text{spec}(\text{input}, \text{output}) \) 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:

```
nai\ve(A):
    \text{if } (A = [0, 0, \ldots, 0, 0]) \text{ return out}_1
    \vdots
    \text{if } (A = [M, M, \ldots, M, M]) \text{ 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 ($\text{stay}_i$) or switching to a different line ($\text{switch}_i$), 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:

\[
\begin{align*}
\text{unary}(x) &= x \mid 0 \\
\text{binary}(x,y) &= x + y \mid \max(x,y) \\
\text{opt} &= \max
\end{align*}
\]

Synthesized Recurrence Relation:

\[
\begin{align*}
\text{mis}(n) &= \max(\text{pick}(n), \text{no_pick}(n)) \\
\text{pick}(n) &= \max(\text{no_pick}(n-1) + \text{array}(n)) \\
\text{no_pick}(n) &= \max(\text{pick}(n-1), \text{no_pick}(n-1)) \\
\text{pick}(0) &= 0 \\
\text{no_pick}(0) &= 0
\end{align*}
\]

Here pick(n) is the subproblem 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:

\[
\begin{align*}
\text{unary}(x) &= x \mid 0 \\
\text{binary}(x,y) &= x + y \mid \max(x,y) \\
\text{opt} &= \max
\end{align*}
\]

Synthesized Recurrence Relation:
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:

\[
unary(x) = x | 0 | -x \\
binary(x, y) = x + y | \max(x, y) \\
opt = \max
\]

Synthesized Recurrence Relation:

\[
mas(n) = \max(\text{suffix\_pos}(n), \text{suffix\_neg}(n), \text{best}(n)) \\
\text{suffix\_pos}(n) = \max(0, \text{suffix\_2}(n-1)) + \text{array}(n) \\
\text{suffix\_neg}(n) = \max(0, \text{suffix\_1}(n-1)) - \text{array}(n) \\
\text{best}(n) = \max(\text{suffix\_pos}(n-1), \text{suffix\_neg}(n-1), \text{best}(n-1)) \\
\text{suffix\_pos}(0) = 0 \\
\text{suffix\_neg}(0) = 0 \\
\text{best}(0) = 0
\]

Here suffix\_1(n) denotes the best legal suffix up to the \(n^{th}\) element that ends in 1 while suffix\_2(n) ends in -1. best(n) denotes the best legal assignment up to the \(n^{th}\) element.

4.1.4 MMM

User Hints:

\[
unary(x) = x | 0 | -x \\
binary(x, y) = x + y | \max(x, y) \\
opt = \max
\]

Synthesized Recurrence Relation:

\[
mmm(n) = \max(\text{mark\_0}(n), \text{mark\_1}(n), \text{mark\_{-1}}(n)) \\
\text{mark\_ig}(n) = \max(\text{mark\_{-1}}(n-1), \text{mark\_1}(n-1)) \\
\text{mark\_pi}(n) = \max(\text{mark\_0}(n-1)+\text{array}(n), \text{mark\_{-1}}(n)+\text{array}(n-1)) \\
\text{mark\_ne}(n) = \max(\text{mark\_0}(n-1)-\text{array}(n), \text{mark\_1}(n)-\text{array}(n-1)) \\
\text{mark\_ig}(0) = 0 \\
\text{mark\_pi}(0) = 0 \\
\text{mark\_ne}(0) = 0
\]

Here mark\_ig(n), mark\_pi(n), mark\_ne(n) are the subproblems 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 | 0
binary(x, y) = x + y | 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 \( a \times x + b \times y = 1 \). The greatest common divisors 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: \( q1 = \text{reverse}(p1), q2 = \text{reverse}(p2) \), 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 \times x + b \times y == 1 OR a \times y + b \times 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 | 0
binary(x, y) = x + y | x * y | x%y | x - y | x/y

Synthesized Recurrence Relation:
Here there is no obvious concise interpretation for the meanings of the subproblems. 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 \( a = [a_1, \ldots, a_n] \), compute the array \( b = [s - a_1, \ldots, s - a_n] \) where \( s = \sum_{i=1}^{n} a_i \). That is, \( b[i] \) equals the sum of every element in \( a \) other than the \( i \)th 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:

```python
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:

```python
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 subproblem arrays (as in Extended Euclid algorithm, we extract the entire subproblem 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:
    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 subproblems, 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 MAS problem, perhaps 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
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.

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 subproblems, 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 algorithm of interest without making constraint solving prohibitively expensive.

7 Acknowledgements

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.

Bibliography


Abstract

Sparse matrix formats are typically implemented with low-level imperative programs. The optimized nature of these implementations hides the structural organization of the sparse format and complicates its verification. We define a variable-free functional language (LL) in which even advanced formats can be expressed naturally, as a pipeline-style composition of smaller construction steps. We translate LL programs to Isabelle/HOL and describe a proof system based on parametric predicates for tracking relationship between mathematical vectors and their concrete representations. This proof theory automatically verifies full functional correctness of many formats. We show that it is reusable and extensible to hierarchical sparse formats.

1 Introduction

Sparse matrix formats compress large matrices with a small number of nonzero elements into a more compact representation. The goal is to both reduce memory footprint and increase efficiency of operations such as sparse matrix-vector multiplication (SpMV). More than fifty formats have been developed; the reason for this diversity is that a format may improve memory locality in a given memory hierarchy, expose parallelism that fits the hardware, and tailor the layout to the operations that will be performed on the matrix. The development of a sparse matrix format is nontrivial; formats exploit algebraic properties such as commutativity, associativity and zero; have to judiciously choose between linear and random access to array data to improve cache locality, memory bandwidth and use of vector instructions. Sparse codes are used heavily in scientific applications, simulations and data mining, as well as other domains. It is expected that more formats will be designed to support future (parallel) platforms. Our goal is to simplify their development.

Sparse matrix codes are typically implemented using imperative languages like C and Fortran. This gives programmers control over low-level details of the computation, allowing them to create optimized implementations. However, imperative implementations obfuscate the structure of the format because logically independent steps of sparse matrix construction are fused, resulting in code with loop nests that contain complex array indirections, in-place data mutation and other low-level optimizations. Not only is the code hard to read, it is also challenging to verify. In fact, we failed to verify the functional correctness of even simple formats using several state-of-the-art tools. The key reason was that describing the properties of the format construction expressed using such low-level implementations required complex invariants that were hard to formulate. Consequently, we sought to raise the level of abstraction in programming sparse matrix formats.

*Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems.
We describe a new approach to implementing and verifying sparse matrix codes. The main idea is to specify sparse codes as functional programs, where a computation is a sequence of high-level transformations on lists. We then use Isabelle/HOL to verify full functional correctness of programs.

We identify a “little language” (LL) for specifying a variety of sparse matrix formats. LL is a strongly typed, variable-free functional programming language in the spirit of FP [1]. It is also influenced by such languages as APL, J, NESL and Python, but favors simplicity and ease of programming over generality and terseness. LL provides several built-in functions and combinators for operations over vectors and matrices common in sparse formats. LL is restricted by design, lacking custom higher-order functions, recursive definitions, and a generic reduction operator. These limitations of LL, as well as its purely functional semantics, facilitate automatic verification of sparse codes.

The contributions of this paper can be summarized as follows.

• We design a variable-free functional language for sparse matrix codes. We show how interesting and complex sparse formats can be naturally and concisely expressed in LL (Section 3).

• We describe a powerful proof method for automatic verification of sparse matrix codes using Isabelle/HOL [11] (Section 4).

• We evaluate the reusability of proof rules in our theory and its extensibility to proving additional formats. We show that our language and verifier can accommodate complex formats including Jagged Diagonals (JAD) and Coordinate (COO), as well as hierarchical formats including Sparse CSR (SCSR), register- and cache-blocking schemes (Section 5). As far as we know, this is the first successful attempt in proving full functional correctness of operations on such formats.

We are currently writing a compiler which automatically generates efficient low-level code from LL programs.

2 Overview

This section outlines our solutions for implementing and verifying sparse matrix programs. We demonstrate our language using the JAD sparse format, and the proof system using the CSR sparse format. These formats are introduced properly in Section 3; in this section, we will make do with an informal overview of the formats and the examples shown in Fig. 1.

2.1 Sparse Matrix Codes in the LL Language

Sparse matrix formats are usually constructed with a sequence of transformations. For example, a JAD sparse matrix is constructed in three steps, by (i) compressing each row in the dense matrix; (ii) sorting compressed rows by their length; and (iii) transposing the rows. Efficient imperative implementations usually fuse these distinct steps, which complicates code comprehension and maintenance. We define a small functional language that keeps these steps separate. The fusion, necessary for performance, will be performed by a data-parallel compiler (which is under development and outside the scope of this paper).

Let us compare the characteristics of imperative and functional implementations of JAD format construction. Consider first the C code that compresses a dense matrix \( M \) into the JAD format, represented by arrays \( P \), \( D \), \( J \), and \( V \).

```c
lenperm (M, P); /* obtain row permutation */
for (d = k = 0; d < n; d++) {
    kk = k;
    for (i = 0; i < n; i++) {
        for (j = nz = 0; j < m; j++)
            if (M[P[i]][j])
                if (++nz > d) break;
        if (j < m) {
            J[k] = j;
```

The low-level code reads and writes a single word at a time, relies heavily on array indirections (i.e., array accesses whose index expressions are themselves array accesses), and explicitly spells out loop boundaries. The code does not distinguish the three construction steps provides little insight into the JAD format:
Specifying and Verifying Sparse Matrix Codes

\[ V[k] = M[P[i]][j]; \]
\[ k++; \]
\} 
\} 
\if (k == kk) break; 
\D[d] = k; \} 

Contrast the C code with this LL program, which is a composition of three functions corresponding to the steps in JAD construction. The function composition operator is \(-\rightarrow\).

\[
\text{def jad: csr -> lenperm -> (fst, snd -> trans)}
\]

\[
\begin{pmatrix}
  a & 0 & 0 & 0 \\
  b & c & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  0 & d & 0 & e
\end{pmatrix}
\]

\(P\): \([1\ 3\ 0\ 2]\)
\(D\): \([3\ 5]\)
\(J\): \([0\ 1\ 0\ 1\ 3]\)
\(V\): \([b\ d\ a\ c\ e]\)

(a) Dense matrix. (b) JAD sparse format. (c) CSR sparse format.

Figure 1: Two sparse matrix formats. Shown are “imperative” representations; their LL counterparts are in Figures 2 and 4.

LL is a functional language rooted in the variable-free style of FP/FL [1], which means that functions do not refer to their arguments by name; instead, they transform a single, unnamed input parameter. For example, if the input to a function is a pair, then a function extracts the first element using the built-in function \(\text{fst}\). LL is strongly typed and datatypes include numbers, Boolean values, pairs and lists. Vectors are represented by lists, and matrices by lists of (row) vectors. Compressed matrix representations use a variety of nested data structures built of lists and pairs.

The three steps of JAD construction in LL are visualized in Fig. 2, which shows the dense matrix, the resulting JAD matrix, as well as the intermediate values of JAD construction. Notice that the JAD representation in LL (the result in Fig. 2) is more abstract than the JAD format in C (Fig. 1(b)). Where LL formats rely on lists of lists, the C formats linearize the outer list and create explicit indexing structures to access the inner lists. LL thus frees the programmer from reasoning about these optimized data structure layouts, eliminating dependence on explicit array indirection.

The first step compresses rows by invoking the constructor for the sparse format CSR. In the second step, the function \(\text{lenperm}\) sorts the compressed rows by decreasing length:

\[
\text{def lenperm:} \quad [[\text{len}, (\#, \text{id})]] \rightarrow \text{sort} \rightarrow \text{rev} \rightarrow [\text{snd}] \rightarrow \text{unzip}
\]

Here, the syntax \([f]\) denotes a map that applies the function \(f\) over the elements of the input list: \(\text{len}\), \(\#\) and \(\text{id}\) return the length of the current element, the position index of that element in the list, and the element itself (identity), respectively. The third-step function \((\text{fst, snd} \rightarrow \text{trans})\) takes a pair and produces a pair in which first element is unchanged and the second element is transposed.

In summary, LL lifts an intricate imperative computation into a cleaner functional form, exposes high-level stages and the flow of data from one stage to another, and encourages the programmer to think about invariants over intermediate results. These benefits are not merely due to the use of functional programming. We believe that they are equally attributed to our careful selection of a very simple subset of functional language features, designed with the sparse matrix domain in mind. In particular, LL does not support lambda abstractions, which encourages expressing computations as pipelines of functions. LL also excludes definitions of recursive functions and a general fold operator, both of which are compensated for by a versatile set of built-ins (\(\text{e.g.}\), \(\text{zip}\) and \(\text{sum}\)) and combinators for handling lists (\(\text{e.g.}\), \(\text{map}\) and \(\text{filter}\)). These restrictions contribute to our ability to automatically verify LL programs because they sidestep the need to infer induction invariants, a hard task for automated tools. The LL language is introduced in detail in Section 3.

We have recently developed compiler for LL that relies on optimization techniques pioneered in NESL [3] and later generalized in Data Parallel Haskell [4]. Thanks to LL’s simplicity, we were able to simplify the compilation and identify more opportunities for optimization. Initial results indicate that code generated for real-world formats such as register-blocked CSR (see Section 5.2) runs as fast as a hand-optimized code and scales well to multiple cores.
2.2 Verifying Sparse Matrix Codes

There are at least two arguments for full functional verification of sparse matrix codes. First, classical static typing is insufficient for static bug detection because these programs contain array indirection, whose memory safety would be typically guaranteed only with run-time safety checks. Dependent type systems may be able to prove memory safety but, in our experience, the necessary dependent-type predicates would need to capture invariants nearly as complex as those that we encountered during full functional verification. For example, to prove full functional correctness, one may need to show that a list is some permutation of a subset of values in another list; to prove memory safety, one may need to show that the values in a list are smaller than the length of another list. It thus seemed to us that with a little extra effort, we can use theorem proving to extend safety to full functional correctness.

The second reason for full functional verification is synthesis of sparse matrix programs, including the discovery of new formats. In inductive synthesis, which is conceptually a search over a space of plausible (i.e., potentially semantically incorrect) implementations, a full functional verifier is a prerequisite for synthesis because it is an arbiter of correctness of the selected implementation. Synthesis, however, is outside the scope of this paper.

Before settling on the design presented in this paper, we set as our goal the full functional verification of imperative sparse code, in the style presented in Section 2.1. However, even the simple CSR format turned out to be rather overwhelming. We attempted to verify its correctness in multiple ways: (i) manually with Hoare-style logic, both with first-order predicates and inductive predicates; (ii) with ESC/Java [6]; (iii) with TVLA [13]; and (iv) using a SAT-based bounded model checker. The results were unsatisfactory either because it took weeks to develop the necessary invariants (i, ii), the abstraction was too complex for us to manage (iii), or because the checker scaled poorly (iv). Eventually, we concluded that we needed to verify sparse codes at a higher level of abstraction (and separately compile the verified code into efficient low-level code). Turning our attention to functional programs allowed us to replace explicit loops over arrays with maps and a few fixed reductions over lists, which in turn simplified the formulation and encapsulation of inductive invariants.

Let us the simple CSR format to give the rationale for the design of our proof system. Suppose that \( A \) and \( x \) are concrete language objects that, respectively, contain dense representations of a mathematical matrix \( B \) and a vector \( y \). We want to prove that the product of the CSR-compressed \( A \) with \( x \) produces an object that is a valid (dense) representation of the vector \( B \cdot y \). Note that the product is CSR-specific. Formally, our verification goal is

\[
\text{csr}(A) \cdot x \vdash B \cdot y
\]

Figure 2: The three steps of JAD format construction. Shown are the dense matrix, the JAD matrix, and the two intermediate values.
into simpler, equivalent ones; and (b) introduction, which substitutes a proof goal with a certain term for alternative goal(s) that do not contain the term, and whose validity implies the validity of the original goal. In our example, term simplification unfolds the definitions of `csrmv` and `csr` and applies standard rules for simplifying function application and composition, map and filter operations on lists, and extraction of elements from pairs. This results in the goal

\[ \text{enum} \rightarrow [\text{snd} \neq 0?] \rightarrow \text{[snd} \ast x[\text{fst}]\] \rightarrow \text{sum}(A) \]

The LL function on the left enumerates each row of \(A\) into a list of (column index, value) pairs, then filters out pairs whose second element is zero ((\text{snd} \neq 0?)). For the remaining pairs, it multiplies the second (nonzero) component with the value of \(x\) at the index given by the first component ((\text{snd} \ast x[\text{fst}])). Finally, it sums the resulting products (\text{sum}). So far, simplification has done a good job.

To carry out the next step of the proof, we observe that the missing zeros do not affect the result of the computation, so we would like to simplify the left-hand-side by rewriting away the filter ((\text{snd} \neq 0?)); this would effectively “desparsify” the left-hand side, moving it closer to the mathematical right-hand-side. Unfortunately, standard simplification available in prover libraries cannot perform the rewrite; we would need to add a rule tailored to this format. The hypothetical rule, shown below, would match \(p\) with \(\text{snd} \neq 0\) and \(f\) with \(\text{snd} \ast x[\text{fst}]\).

\[
\forall y. \neg p(y) \implies f(y) = 0 \\
[p?] \rightarrow [f] \rightarrow \text{sum} = [f] \rightarrow \text{sum}
\]

The rule would achieve the desired simplification but we refrain from adding such a rule because it would take a considerable effort to prove it. Additionally, the rule would be of little use in cases where the LL operations appear in just a slightly syntactically different way.

We will instead rely on introduction which, by substituting the current goal with a set of goals, isolates independent pieces of reasoning. Introduction rules tend to be more general than simplification rules because they are concerned with a single construct from the current goal. Also, the validity of introduction rules is easier to establish.

Our first introduction rule substitutes in the goal (1) the whole result vector with a single element of that vector. In effect, this removes the outermost map from the LL function on the left-hand side. Semi-formally, the rule for map can be stated as follows:

\[
\text{length of } A \text{ is } m \quad \forall i < m. f(A[i]) = B_i \\
[f](A) \triangleright B
\]

In goal (1), \(f\) matches the entire chain of \text{enum} \rightarrow \ldots \rightarrow \text{sum} and the new subgoals are

(i) length of \(A\) is \(m\)

(ii) \(\forall i < m. \text{enum} \rightarrow [\text{snd} \neq 0?] \rightarrow [\text{snd} \ast x[\text{fst}]] \rightarrow \text{sum}(A[i]) = \sum_{j<n} B_{i,j} \cdot y_j\)

We now need a second introduction step to remove the summation on both sides of the equality: instead of requiring equivalence between sums of sequences of numbers, we will require equivalence between the values in the sequences themselves. In order for such a rule to be general enough, we need to permit arbitrary permutations of the values in a sequence to prove programs that exploit associativity and commutativity of addition. A hypothetical rule may look as follows, where \([x_i | p(x_i)]_{i=a_0, \ldots, a+\delta}\) denotes a construction of an ordered list of elements out of \(x_{a_0}, \ldots, x_{a+\delta}\) that satisfy \(p\).

\[
\exists n' \leq n, \text{ permutation } P. \\
f(A[i]) \triangleright [B_{i,j} \mid B_{i,j} \neq 0]_{j=p_0, \ldots, p_{n-1}} \\
\sum(f(A[i])) = \sum_{j<n} B_{i,j}
\]
This rule is problematic for two reasons. First, it is more complex than what we may want to prove. For example, the premise constructs a filtered and permuted mathematical vector on the right-hand side (via list comprehension), rather than keeping the mathematical object untouched. This might hinder our ability to link our proof goal to the original input matrix in the assumptions of the theorem. Second, the rule is not as general as we would like because a concrete representation may contain zeros.

Our approach is to enrich the representation relation \( (a \triangleright b) \). This relation uses plain equality to relate single elements from the two vector objects, which limits its applicability to more subtle mappings. To express a relation where, say, each element in a concrete representation equals the corresponding vector element multiplied by some value, we parameterize the representation relation with an inner relation that describes how individual elements represent their mathematical counterparts. Individual elements need not be scalars; they could be, recursively, lists. Therefore, inner relations could be parameterized by further inner relations.

Our domain proof theory for sparse matrices is novel in two ways. First, we define common representation relations that occur in our domain. Our infrastructure is powerful because we (a) insist on relaxing invariants as much as possible (e.g., zeros may still be present in a compressed representation); (b) encapsulate many quantifications and implications in the representation relations (e.g., universal quantification on all indexes of a vector, existence of a permutation); (c) include necessary integrity constraints in the representation relations (e.g., lengths must match). The representation relations we define include indexed list \((i\text{-}list})\), where the element at position \( i \) represents the \( i \)th vector element; value list \((v\text{-}list})\), in which all nonzero values are represented; and associative list \((a\text{-}list})\), which contains index-value pairs. These representation relations raise the level of abstraction and focus theory development on these prevalent data representation. The use of representation relations also prevents oversimplification of proof terms by concealing their internal conjuncts from Isabelle’s simplifier.

The second novelty is parameterizing the inner predicate, which describes how the vector elements represent their mathematical counterparts. In the case of a vector of numbers, we use equality. For matrices, the inner relation relates a single row to its concrete indexed-list representation \((i\text{-}list})\); technically, the inner relation predicate is a parameter to the (outer) representation predicate for the whole matrix. In addition to reducing the number of rules, parameterization helps with syntactic matching and substitution of inner comparators during introduction. For example, with a parameterized relation, an introduction rule for map similar to that in Eq. (2.2) can be written more generally and concisely: the conclusion of the rule contains an indexed-list representation relation where the concrete object is the term \( f \{ x \} \) (i.e., map with an arbitrary function \( f \) over \( x \)) and the inner representation relation is some arbitrary predicate \( P \)—our parameter. The premise of the rule is again an indexed-list representation relation where the concrete object is \( x \) and the inner representation relation is \( \lambda i \ a \ b \ . \ P(i, a, f(b)) \). Fortunately, Isabelle can match and substitute terms that contain parameters such as \( P \) (as well as \( f \) and \( x \)); these rules can thus be applied automatically.

The representation relations are described in Section 4. Section 5 evaluates whether they improve reuse of rules and thus simplify theory development; we argue that the principles used in our approach are crucial for proofs on nested data representations. It may be interesting to apply such parameterized representation relations also in other domains.

3 High-Level Sparse Matrix Programming

Sparse matrix codes can often be decomposed into sequences of high-level transformations. This section describes LL and its use for expressing such computations naturally and concisely.

3.1 Introduction to LL

The LL language constructs are presented in Fig. 3. The semantics of each construct is shown, either by translation to Isabelle/HOL \( \lambda \)-calculus and standard library for lists [11], or by de-sugaring to simpler LL constructs. The language includes (a) general functions such as identity, equality, constants, conditional branching, and a name binding form used for assigning names to components of an input value; (b) construction of pairs/tuples and extraction of values from pairs; (c) pipeline- and application-style composition, as well as a curried application operator; (d) standard arithmetic operators and comparators; (e) Boolean logic operators; and (f) list handling functions (e.g., distribution of values onto lists, zipping, enumeration, concatenation) and combinators (map, filter, and a unified comprehension syntax).
<table>
<thead>
<tr>
<th>ID</th>
<th>Equation/Notation</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>id</code></td>
<td>$\lambda x. x$</td>
<td>$\lambda x, y. x = y, \lambda(x, y). x \neq y$</td>
</tr>
<tr>
<td><code>eq (=), neq (!=)</code></td>
<td>$\lambda(x, y). x = y, \lambda(x, y). x \neq y$</td>
<td>$\lambda y, n, \lambda y. \text{true}, \lambda y. \text{false}$</td>
</tr>
<tr>
<td><code>n, true, false</code></td>
<td>$\lambda x. \text{if } f \text{ then } g \text{ else } h \text{ x}$</td>
<td>$\lambda x_1, \ldots, x_k. g[l_i/\lambda y. x_i] (x_1, \ldots, x_k) \circ f$</td>
</tr>
<tr>
<td>`f ? g</td>
<td>h`</td>
<td>$\lambda x_1, \ldots, x_k = f \cdot g \uparrow$</td>
</tr>
<tr>
<td><code>l_1, \ldots, l_k = f : g \uparrow</code></td>
<td>$(f_1, f_2, \ldots, f_k)$</td>
<td>$\lambda x. (f_1 x, f_2 x, \ldots, f_k x)$</td>
</tr>
<tr>
<td><code>fst, snd</code></td>
<td><code>f -&gt; g</code></td>
<td>$\lambda(x, y). x, \lambda(x, y). y$</td>
</tr>
<tr>
<td><code>g(f_1, \ldots, f_k)</code></td>
<td><code>g : f</code></td>
<td>$(f_1, \ldots, f_k) \to g$</td>
</tr>
<tr>
<td><code>add (+), sub (-), mul (*), div (/), mod (%)</code></td>
<td><code>leq (&lt;=), lt (&lt;), geq (&gt;=), gt (&gt;)</code></td>
<td>$\lambda(x, y). x + y, \lambda(x, y). x - y, \ldots$</td>
</tr>
<tr>
<td><code>sum (+), prod (*)</code></td>
<td><code>foldl (op +), foldl (op *) 1</code></td>
<td>$\lambda(x, y). x \leq y, \lambda(x, y). x &lt; y, \ldots$</td>
</tr>
<tr>
<td>`and (&amp;&amp;), or (</td>
<td></td>
<td>)`</td>
</tr>
<tr>
<td><code>neg (!)</code></td>
<td>$\lambda x, \neg x$</td>
<td>$\lambda x, \neg x$</td>
</tr>
<tr>
<td>`conj (/&amp;&amp;), disj (/</td>
<td></td>
<td>)`</td>
</tr>
<tr>
<td><code>len</code></td>
<td><code>rev</code></td>
<td>$\lambda(v, i). v!i$</td>
</tr>
<tr>
<td><code>rev</code></td>
<td><code>length</code></td>
<td>$\lambda(v, s). \text{map } (\lambda i. v!i) s$</td>
</tr>
<tr>
<td><code>sub (f[g])</code></td>
<td><code>subseq (f[g])</code></td>
<td>$\lambda(x, v). \text{map } (\lambda(y, (x, y))) v, \lambda(v, x). \text{map } (\lambda y. (y, x)) v$</td>
</tr>
<tr>
<td><code>distl. distr</code></td>
<td><code>zip, unzip</code></td>
<td>$\lambda(x, v). \text{map } (\lambda y. (x, y)) v, \lambda(v, x). \text{map } (\lambda y. (y, x)) v$</td>
</tr>
<tr>
<td><code>elem, unsplit</code></td>
<td><code>concat</code></td>
<td>$\lambda(s, i, n, v). \text{foldr } (\lambda(i, x)\ v. v[i := x]) v (\text{replicate } n d)$</td>
</tr>
<tr>
<td><code>enum</code></td>
<td><code>concat</code></td>
<td>$\lambda v. \text{zip } \left[0 \ldots &lt; \text{length } v \right] v$</td>
</tr>
<tr>
<td><code>concat</code></td>
<td><code>concat</code></td>
<td>$\lambda i \left[0 \ldots &lt; \text{if } v = \right] \text{then } 0 \text{ else length } (v!1)]]$</td>
</tr>
<tr>
<td><code>infl</code></td>
<td><code>filter</code></td>
<td>$\lambda d.\ n.\ v.\ \text{foldr } (\lambda(i, x)\ v.\ v[i := x]) v (\text{replicate } n d)$</td>
</tr>
<tr>
<td><code>gather</code></td>
<td><code>filter</code></td>
<td>$\lambda k.\ k.\ \text{map } (\lambda (k', v).\ k = k') (xs)) (\text{rendups } (\text{map } f (xs)))$</td>
</tr>
<tr>
<td><code>sort</code></td>
<td><code>filter</code></td>
<td>$\lambda v.\ \text{map } (\lambda v. v!1) (\text{takeWhile } (\lambda v. i &lt; \text{length } v) v). i \left[0 \ldots &lt; \text{if } v = [] \text{then } 0 \text{ else length } (v!1)]]$</td>
</tr>
<tr>
<td><code>trans</code></td>
<td><code>filter</code></td>
<td>$\lambda i_1, \ldots, l_n = f : g \uparrow$</td>
</tr>
<tr>
<td><code>map f</code></td>
<td><code>filter f</code></td>
<td>$\lambda i_1, \ldots, l_n: h$</td>
</tr>
<tr>
<td><code>filter f</code></td>
<td><code>filter f</code></td>
<td>$\lambda i_1, \ldots, l_n = f : g \uparrow$</td>
</tr>
</tbody>
</table>

Figure 3: LL constructs and their translation to Isabelle/HOL. Here, $f$, $g$ and $h$ denote functions, $n$ a number, and $l$ a label. Alternative infix, prefix or mixfix notation is shown in parentheses. $^\dagger$ $f$ defaults to $id$. $^\ddagger$ Value naming is optional, $f$ and $h$ default to $id$ and $g$ to true.
3.2 Specification of Sparse Codes Using LL

Compressed sparse rows (CSR). This format compresses each row by storing nonzero values together with their column indexes. The resulting sequence of compressed rows is not further compressed, so empty (all zero) rows are retained. This enables random access to the beginning of each row, but requires linear traversal to extract a particular element out of a row. CSR is widely used because it is relatively simple and entails good memory locality for row-wise computations such as SpMV.

Implementing CSR in C, shown below,1 is not trivial. Traversal of the dense matrix (construction) or the compressed rows (SpMV) is done with nested loops. Single values are copied (construction) or extracted (SpMV) through array indirection. Compressed row boundaries need to be stored and observed. That said, the resulting SpMV code is rather efficient as the inner product of each row is incrementally accumulated, using very few instructions and avoiding unnecessary memory accesses. Applying CSR construction to the 4-by-4 matrix in Fig. 1(c) yields the data structure in Fig. 1(c).

```
/* CSR construction */
for (i=k=0; i<m; i++) {
  for (j=0; j<n; j++)
    if (M[i][j] != 0) {
      J[k] = j;
      V[k] = M[i][j];
      k++;
    }
R[i] = k;
}
```

Fig. 4 shows the high-level stages in CSR compression mentioned above. Given (a), each row is enumerated with column indexes, resulting in (b). Pairs containing a zero value are then filtered, yielding (c). A dataflow view of such a computation is shown in Fig. 5. Notice how similar it is to the following LL function.

```
def csr: [enum -> [snd != 0 ? ]]
```

Using name binding in comprehensions may improve clarity.

```
[enum -> [j, v: v != 0 ? ]]
```

1For brevity, we omit memory allocation and initialization and assume that matrix dimensions are known at compile-time.
Alternatively, one can use an explicit enumeration operator inside comprehensions. The following variant appears more “integrated”, but in fact entails the exact same semantics.

\[
[[v: v \neq 0 ? (#, v)]]
\]

A more verbose variant uses Python-style comprehension. This variant is de-sugared to the original definition.

```python
def csr(A):
    \[
    [[(j, v) \text{ for } j, v \text{ in } \text{enum}(r) \text{ if } v \neq 0] \text{ for } r \text{ in } A]
    \]
```

Fig. 6 shows the stages in CSR SpMV. Each compressed row is multiplied separately, as shown in (b). First, column indexes are separated from nonzero values as in (c). They are used to retrieve corresponding values from \(x\), pairing them with their respective row values as in (d). Finally, values in pairs are multiplied and the products are summed, yielding the inner-product in (e). This maps to the following LL function.

```python
def csrsmv(A, x):
    A ->
    \[
    [J, V = \text{unzip: (V, x[J]) \rightarrow zip} \rightarrow [\text{mul}] \rightarrow \text{sum}]
    \]
```

Here, too, it is possible to write a more integrated variant that bundles multiplication with the extraction of single values. Although semantically equivalent, the resulting code is less amenable to vectorization due to the use of word-level operations.

**A \rightarrow [[j, v: v * x[j]] \rightarrow \text{sum}]**

**Jagged diagonals (JAD).** This format deploys a clever compression scheme that allows handling of sequences of nonzeros from multiple rows, taking advantage of vector instructions. The \(i\)th nonzero values from all rows are laid out consecutively in the compressed format, constituting a “jagged diagonal”. Since nonzeros are distributed differently in each row, column indexes need to be stored as well. These steps can be thought of as per-row compression (as shown above for CSR), followed by transposition to invert the direction of compressed rows and \(i\)th-element columns.

However, packing \(i\)th elements in a predetermined order—e.g., from the first to the last row—induces a problem: one needs to account for compressed rows that are shorter than other rows that succeed them.\(^2\) This is addressed by adding a sorting step between row compression and transposition, in which rows are ordered by decreasing number of nonzeros. The sort permutation is stored with the resulting diagonals, so the correct order of rows can be restored. These conceptual steps in JAD compression are visualized in Fig. 2 and the LL implementation is shown in Section 2.1.

Fig. 7 shows the high-level steps in JAD SpMV. (b) is obtained by computing, for each diagonal, the cross-product of its induced vector of values with the elements of \(x\) corresponding to their column indexes. These are transposed to obtain the lists of products in each (nonzero) row as in (c). Products in corresponding rows are summed, obtaining (d). In (e), each inner product is paired with its row index, which are then “inflated” to obtain the dense result vector in (f). The following LL function implements these steps.

\(^2\)Transposition inverts columns up to the first missing element, below which all other elements are omitted. In this respect it is “lossy” and the equality \(A = A^{TT}\) only holds for matrices whose rows are sorted by length.
The Berkeley Par Lab: Progress in the Parallel Computing Landscape

Figure 7: Conceptual phases in JAD SpMV.

```
def jadmv((P, D), x):
    (P,
     D -> [unzip -> snd * x[fst]] -> trans -> [sum]) ->
    zip -> infl(0, m, id)
```

**Other formats.** Two additional standard formats are Coordinate (COO) and Compressed Sparse Columns (CSC) [10]. COO is a highly-portable compression in which nonzeros are stored together with their row and column indexes in a single, arbitrarily ordered sequence. Construction can be implemented in LL as follows.

```
def coo: [i = #: [v: v != 0 ? (i, #, v)]] -> concat
```

COO SpMV is less straightforward: one needs to account for the fact that nonzeros of a particular row might be scattered along the compressed list. It is necessary to *gather* those values prior to computing the inner-product. This is expressed as follows.

```
def coomv (A, x):
    A-> gather ->
    [(fst, snd -> [j, v: v * x[j]] -> sum)] ->
    infl(0, m, id)
```

A CSC representation is obtained by compressing the nonzero values in the column direction, instead of row direction as in CSR. In C, it is done by swapping the order of the loops iterating over the dense matrix, and storing the row index with the nonzero values. In LL, it amounts to prepending a transposition to CSR construction.

```
def csc: trans -> csr
```

Like COO, CSC SpMV calls for a gather operation prior to summing row cross-products.

```
def scsmv:
    zip -> [cj, xj: cj -> [i, v: (i, v * xj)]] ->
    concat -> gather ->
    [(fst, snd -> sum)] -> infl(0, m, id)
```

Here, too, the fact that data layout is not in line with the computation entailed by matrix-vector multiplication calls for additional steps to massage the result into a proper vector form.

In addition to the above formats, LL can naturally and succinctly describe hierarchical compression. This includes Sparse CSR (SCSR) and different block variants of all of the above. These will be described and studied in Section 5.
4 Verifying Sparse Codes Using Isabelle/HOL

We make use of Isabelle’s rich infrastructure in implementing a proof method for sparse matrix codes. This includes the simplifier and a powerful tactical language, which is used to combine existing proof methods in forming new ones. All parts of our proofs are checked from first principles by a small LCF-style kernel.

4.1 Translating LL to Isabelle/HOL

Fig. 3 constitutes a shallow embedding [16] of LL in Isabelle/HOL, a standard technique when the goal is to verify correctness of programs in some language. In this approach, the functions and types of an object language (LL) are written directly in the language of the theorem prover (typed $\lambda$-calculus). Subsequent logical formulas relate to these translated programs as ordinary HOL objects, which allows to leverage existing support for proving properties of them. The CSR implementation in Section 3.2 translates to the following definitions, which will be used in our proofs.

$$
csr = (\text{filter}(\lambda(j,v).v \neq 0)) \circ \text{enum} \quad \text{and} \quad 
csr \text{mv} (A,x) = \text{map} (\text{listsum} \circ \text{map} (\lambda(x,y).x \cdot y) \circ 
\text{unsplit} \circ 
(\lambda(J,V). (V, \text{map} (\lambda(x!i).J)) \circ 
\text{map} \text{unzip})
$$

(4.1)

We now pose the verification theorem: when $A$ index-represents the $m \times n$-matrix $A'$ and $x$ the $n$-vector $x'$, the result of CSR SpMV applied to a CSR version of $A$ and to $x$ represents the $m$-vector that is equal to $A' \cdot x'$.

$$
ilist_M m n A' A \land \text{ilist}_v n x' x \longrightarrow \text{ilist}_v m (\lambda_i. \Sigma_j < n. A' i j \cdot x' j) (\text{csrmv} (\text{csr} A, x))
$$

(4.2)

The remainder of this section presents the formalism and explains the reasoning used in proving this goal.

4.2 Formalizing Vector and Matrix Representations

We begin by formalizing vectors and matrices in HOL. Mathematical vectors and matrices are formalized as functions from indexes to values, namely $\text{nat} \rightarrow \alpha$ and $\text{nat} \rightarrow \text{nat} \rightarrow \alpha$, respectively; note that the $\rightarrow$ type constructor is right-associative, hence a matrix is a vector of vectors. Dimensions are not encoded in the type itself, and values returned for indexes exceeding the dimensions can be arbitrary, which means that many functions can represent the same mathematical entity. Concrete representations of dense and sparse vectors/matrices are derived from the LL implementation and consist of lists and pairs. Commonly used representations include indexed lists, value lists and associative lists, all of which are explained below.

We introduce representation relations (defined as predicates in HOL) to link mathematical vectors and matrices with different concrete representations, for three reasons. First, in proving correctness of functions we map operations on concrete objects to their mathematical counterparts. This is easy to do for indexed list representations but gets unwieldy with others. We hide this complexity inside the definitions of the relations. Second, predicates can be used to enforce integrity constraints of the representation. For example, an associative list representation requires that index values are unique; or the lengths of a list of indexed list representations need to be fixed. Third, for some representations (e.g., value list) there exists no injective mapping from concrete objects to abstract ones, forcing us to use relations rather than representation functions. Using relations across the board yields a more consistent and logically lightweight framework.

An indexed list representation of an $n$-vector $x'$ by a list $x$ is captured by the $\text{ilist}$ predicate. Note that we refrain from fixing vector elements to a specific type (e.g., integers) and instead use type parameters $\alpha$ and $\beta$ to denote the types of inner elements of the mathematical and concrete vectors, respectively.

$$
ilist :: \text{nat} \rightarrow (\text{nat} \rightarrow \alpha \rightarrow \beta \rightarrow \text{bool}) \rightarrow 
(\text{nat} \rightarrow \alpha) \rightarrow [\beta] \rightarrow \text{bool}
$$

$$
ilist n P x' x \iff 
(length x = n) \land (\forall i < n. P i (x' i) (x ! i))$$
The parameter $P$ is a relation that specifies the representation of each element in the vector. For ordinary vectors, it is equality of elements. However, $P$ turns useful for matrix representation, as we can use arbitrary relations to determine the representation of inner vectors. We introduce abbreviations for the common cases of indexed list representations.

\[
\text{ilist}_v \ n \ x' \ x \iff \text{ilist} \ n \ (\lambda j. \ \text{op} =) \ x' \ x \\
\text{ilist}_M \ m \ n \ A' \ A \iff \text{ilist} \ m \ (\lambda i. \ \text{ilist}_v \ n) \ A' \ A
\]

An associative list representation is central to sparse matrix codes as it is often used in vector compression. It is captured by the \textit{alist} predicate.

\[
\text{alist} :: \text{nat} \to (\text{nat} \to \alpha \to \beta \to \text{bool}) \to \\
(\alpha \text{ set}) \to (\text{nat} \to \alpha) \to [\text{nat}, \beta] \to \text{bool} \\
\text{alist} \ n \ P \ D \ x' \ x \iff \\
\text{distinct} \ (\text{map} \ \text{fst} \ x) \land \\
(\forall (i, v) \in \text{set} \ x. \ P \ i \ (x' \ i) \ v \land i < n) \land \\
(\forall i < n. \ x' \ i \not\in \text{D} \rightarrow \exists v. \ (i, v) \in \text{set} \ x)
\]

Here, \textit{distinct} is a predicate stating the uniqueness of indexes (i.e., keys) in $x$. Each element in an associative list must relate to the respective vector element, also requiring that index values are within the vector length. Finally, each element in the vector that is not a default value (specified by the set of values $D$) must appear in the representing list. Note that a set of default values accounts for cases where more than one such value exists, as in the case of nested vectors where each function mapping the valid dimensions to zero is a default value. Also note that \textit{alist} does not enforce a particular order on elements in the compressed representation, nor does it insist that all default values are omitted.

Sometimes concrete objects contain only the values of the elements in a given vector, without mention of their indexes. This value list representation often occurs prior to computing a cross- or dot-product. It is captured by the \textit{vlist} predicate, which states that the list of values can be zipped with some list of indexes $p$ to form a proper associative list representation. (The length restriction ensures that no elements are dropped from the tail of $x$.)

\[
\text{vlist} :: \text{nat} \to (\text{nat} \to \alpha \to \beta \to \text{bool}) \to \\
(\alpha \text{ set}) \to (\text{nat} \to \alpha) \to [\beta] \to \text{bool} \\
\text{vlist} \ n \ P \ D \ x' \ x \iff \\
\exists p. \ \text{length} \ p = \text{length} \ x \land \\
\text{alist} \ n \ P \ D \ x' \ (\text{zip} \ p \ x)
\]

Additional representations can be incorporated into our theory. For example, when a matrix is compressed into an associative list, a dual-index representation relation can be defined similarly to \textit{alist}.

### 4.3 Proving Correctness of Sparse Matrix Computations

We prove Eq. (4.2) using term rewriting and introduction rules. Introduction rules are used whenever further rewriting cannot be applied. An introduction rule is applied by resolution: applying the rule $G \ x \land H \ y \rightarrow F \ xy$ to the goal $F \ a \ b$ yields two new subgoals, $G \ a$ and $H \ b$.

The theorem in Eq. (4.2) makes the following two assumptions,

\[
\text{ilist}_M \ m \ n \ A' \ A \\
\text{ilist}_v \ n \ x' \ x
\]

which are added to the set of available introduction rules as true $\rightarrow \ldots$. The conclusion of Eq. (4.2) is our initial proof goal,

\[
\text{ilist}_v \ m \ (\lambda i. \ \Sigma j < n. \ A' \ i \ j \ast x' \ j) \ (\text{csrmv}(\text{csr} \ A) \ x)
\]
Simplifying the goal. We begin by applying Isabelle’s simplifier using Eq. (4.1) and standard rules for pairs, lists, arithmetic and Boolean operators. This removes most of the function abstractions, compositions and pair formations due to the translation from LL. Our new goal is analogous to Eq. (2.1) in Section 2.2.

\[
\text{ilist}_v \ m \ (\lambda i. \Sigma j < n. \ A' \ i \ j * x' \ j) \\
\quad (\text{map} \ (\lambda v. \ \text{listsum} \ (\text{map} \ (\lambda v. \ \text{snd} \ v * x \ ! \ \text{fst} \ v) \ (\text{filter} \ (\lambda v. \ \text{snd} \ v \neq 0) \ (\text{enum} \ r)))) \ A)
\]

(4.6)

Solving the entire goal using rewriting alone calls for simplification rules that are too algorithm-specific. For example, the rule

\[
(\forall x \in \text{set} \ xs. \neg P \ x \rightarrow f \ x = 0) \rightarrow \text{listsum} \ (\text{map} \ f \ (\text{filter} \ P \ xs)) = \text{listsum} \ (\text{map} \ f \ xs)
\]

(4.7)

allows further simplification of Eq. (4.6), but fails for all formats that introduce more complex operations between \text{map} and \text{filter}.

Introduction rules on representation relations. Consider the equation in the conclusion of Eq. (4.7). We know that it holds when the two lists, \(xs\) and \(\text{filter} \ P \ xs\), value-represent the same vector. By introducing rules, describing when it is allowed to apply \text{map}, \text{filter} and \text{enum} operations to value list representations, we prove that the result of \text{listsum} in Eq. (4.6) equals the mathematical dot-product.

Fig. 8 shows the introduction rules used in proving Eq. (4.2). Application of introduction rules is syntax directed, choosing rules whose conclusion matches the current goal. Given Eq. (4.6), the prover applies \text{ILIST-MAP}, which moves the map from the representing object into the inner representation relation, followed by \text{ILIST-LISTSUM}, which substitutes \text{listsum} with an equivalent notion of value-represented rows. This results in

\[
\text{ilist} \ v \ m \ (\lambda i. \Sigma j < n. \ A' \ i \ j * x' \ j) \\
\quad (\text{map} \ (\lambda v. \ \text{listsum} \ (\text{map} \ (\lambda v. \ \text{snd} \ v * x \ ! \ \text{fst} \ v) \ (\text{filter} \ (\lambda v. \ \text{snd} \ v \neq 0) \ (\text{enum} \ r)))) \ A
\]

(4.8)

Further simplification is not possible at this point, nor can we modify the \text{vlist} relation inside \text{ilist}. Luckily, \text{ILIST-VLIST} matches our goal, lifting the inner \text{vlist} to the outermost level and permitting to further operate on the concrete parameters of \text{vlist}. Note that \text{ILIST-VLIST} has two assumptions, resulting in new subgoals

\[
\text{ilist} \ m \ ?Q \ ?B' \ A
\]

and

\[
\forall i < m. \ \text{vlist} \ n \ (\lambda j. \ \text{op} =) \ {\{0\}} \ (\lambda j. \ A' \ i \ j * x' \ j) \\
\quad (\text{map} \ (\lambda v. \ \text{snd} \ v * x \ ! \ \text{fst} \ v) \ (\text{filter} \ (\lambda v. \ \text{snd} \ v \neq 0) \ (\text{enum} \ (A \ ! \ i))))
\]

(4.9)

In Eq. (4.8), \(?Q\) and \(?B'\) are existentially quantified variables. They do not get instantiated when we apply \text{ILIST-VLIST}, and the subgoal Eq. (4.8) merely certifies that \(A\) has length \(n\). Therefore, the prover is allowed to instantiate them arbitrarily and Eq. (4.8) is discharged by the assumption Eq. (4.3).

The rules \text{VLIST-MAP}, \text{ALIST-FILTER} and \text{ALIST-ENUM} can now be applied to Eq. (4.9). Note that applying them amounts to the effect of simplification using Eq. (4.7). However, they can be applied regardless of the way in which the three operations—\text{map}, \text{filter} and \text{enum}—are intertwined. Therefore, they are applicable in numerous cases where the context imposed by Eq. (4.7) is too restrictive.

\[3\]The predicate \(P\) and the vector \(z'\) are arbitrary, they just help to state that \(x\) is a list of length \(m\).
Figure 8: Introduction rules used in the proof of CSR SpMV.
\[
\begin{array}{l}
k \neq 0 \\
\text{ILIST}_n (\lambda i. \text{block} k 1 (\lambda i'. j. A' (i \cdot k + i'))) A \\
\text{ILIST-CONCAT_VECTORS}
\end{array}
\]

\[
\begin{array}{l}
l \neq 0 \\
\text{ILIST}_m (m \cdot l) x' x \\
\text{ILIST-BLOCK VECTOR}
\end{array}
\]

\[
\begin{array}{l}
k \neq 0 \\
l \neq 0 \\
\text{ILIST}_m n (\lambda j. \text{block} k l (\lambda i'. j'. x (i \cdot k + i'))(j \cdot l + j')) A \\
\text{ILIST-BLOCK_MATRIX}
\end{array}
\]

Figure 9: Introduction rules used for proving blocked format operations.

The ALIST-FILTER rule forces us to prove that filter only removes default values, in the form of the following new subgoals,

\[
\forall i < m. \forall j < n. \forall v \cdot \\
\neg \text{snd} (j, v) \neq 0 \land v' = v \cdot x ! j \rightarrow v' \in \{0\}
\]

Fortunately, subgoal Eq. (4.10) is completely discharged by the simplifier. The remaining goal is solved using the ILIST-MULT, ILIST-NTH, and ILIST\_v\rightarrow\text{ILIST}_M, as well as the assumptions Eq. (4.3) and Eq. (4.4).

4.4 Automating the Proof

The above proof outline already dictates a simple proof method. Isabelle’s tactical language [15] provides us with ample methods and combinators that can be used to implement custom proof tactics. Our proof method is implemented as follows.

1. The simplifier attempts to rewrite the goal until no further rewrites are applicable, returning the new goal. If no rewrite rule could be applied, it returns an empty goal.

2. The resolution tactic attempts to apply each of the introduction rules and returns a new goal state for each of the matches. It is possible that more than one rule matches a given goal, e.g. \text{ILIST-MAP} and \text{ILIST-NTH} both match \text{ILIST}_n (\lambda i. v' v'. v' = v \cdot x ! i) (\lambda j. A' i j \cdot x' j) (A ! i), resulting in a sequence of alternative goal states to be proved.

Invoking the proof method leads to a depth-first search on the combination of the two sub-methods. It maintains a sequence of goal states, initially containing only the main goal. After each successful application of either sub-method, the result is prepended to the head of the sequence. A failure at any level causes the search to backtrack and continue with the next available goal state. When the top element of the goal state sequence is empty, the main goal has been discharged and the proof is complete.

5 Evaluation

In this section we evaluate programmability of sparse codes in LL and the extensibility of our verification method to new formats.
5.1 Verifying Additional Sparse Formats

We examine to what extent our prover design allows us to verify additional formats without adding excessively many rules. Recall that our initial implementation of the prover for CSR SpMV (Section 4) insisted on minimizing reliance on format-specific rules, avoiding duplication of logic, and keeping representation relations general, for example by keeping the type of the value stored in the matrix parametric. In this section, we extend our prover to verify several formats that are strictly more complex than CSR.

Our experience indicates that our prover can overcome variations in both format construction and matrix-vector multiplication. The variations were both syntactic (i.e., due to syntactic sugar) and structural (i.e., inducing a different dataflow structure). This benefit is thanks to Isabelle’s simplifier, which successfully canonicalizes these differences, requiring only minor tweaks to the prover’s rule base. Therefore, we consider below only a single implementation for each format and argue that the single variant represents a larger class of similar implementations.

Jagged Diagonals (JAD). A prominent feature of JAD’s proof goal is the double use of transpose, once during compression (jad) and once during multiplication (jadm). This form can be simplified to the Isabelle `takeWhile` list operator on the premise that compressed rows are sorted by length prior to being transposed. The form is matched by a rewrite rule for `transpose` (transpose `xs`). Adding introduction rules for `infl`, `takeWhile`, `rev` and `sort_key` was sufficient for our verifier to complete the proof.

The ability to prove full functional correctness of JAD SpMV documents the strength of our prover; no other verification framework that we know of can (i) handle the complex data transformations in JAD compression, and (ii) prove correctness of arithmetic operations on the resulting sparse representation (see Section 6).

Coordinate (COO). As mentioned in Section 3.2, the COO format is challenging because it associates matrix values with both row and column coordinates, and also because it requires concatenation and gather operations. It turns out that the COO pair coordinates do not call for a new representation relation. In fact, thanks to how the functions `coo` and `coomv` are composed, we need to handle the pair coordinates only between concatenation (in `coo`) and gather (in `coomv`). The simplifier moves these two functions together; therefore, we introduce a rule to relate the representation of the input and output of `gather` (concat `xs`), allowing the prover to automatically complete the proof.

\[
\begin{align*}
\text{ALIST-GATHER-CONCAT} & \\
\text{vlist } n \ (\lambda i. \ \text{vlist } m \ (\lambda j \ a. \ a = \text{snd } b \ \land i = \text{fst } b) \ \{0\} ) \\
& \quad \{x. \forall j < m. x j = 0\} \ M \ xs \\
\text{alist } n \ (\lambda i. \ \text{vlist } m \ (\lambda i. \ \text{op } =) \ \{0\} ) \\
& \quad \{x. \forall j < m. x j = 0\} \ M \ (\text{gather } (\text{concat } xs))
\end{align*}
\]

Compressed Sparse Columns (CSC). As CSC exhibits a peculiar use of concatenation and gather operations, it is handled similarly to COO. In contrast to COO, the input list to `concat` represents a transposed matrix, hence we use a rule similar to `ALIST-GATHER-CONCAT`, but with a transposed matrix `M`.

How many introduction rules did we need to prove our sparse formats? In total, 24 rules were needed, including both introduction and simplification rules. Introduction rules were typically used to (i) reason about some language construct such as `map`, `sum` and `filter`, in the context of a certain representation (e.g., rules `ILIST-MAP`, `ILIST-LISTSUM`, `ALIST-FILTER` in Fig. 8); and (ii) formalize algebraic operations on vector and matrix representations, such as extracting an inner representation relation (ILIST-VLIST) and substituting a vector representation with a matrix representation (ILIST\textsuperscript{x} \rightarrow ILIST\textsubscript{M}). Most operators were handled by a single introduction rule; a few (e.g., `map`) required one rule per representation relation.

To quantify rule reuse in our prover, we summarize the reuse of the 24 rules that were needed for proving five sparse formats (see Fig. 10). On average, fewer than 19% of rules used by a particular format are specific to this format, while over 66% of these rules are used by at least three additional formats, a significant level of reuse. Even of the rules needed for more complex formats (CSC and JAD), only up to a third are format-specific. On the other hand, format-specific rules tend to be harder to prove, as indicated by the average number of lines of Isar code required to prove the rules. A detailed examination reveals that two rules for handling a `gather-concat` sequence (used in CSC and COO) account for over a hundred lines each. We believe that these rules can be refactored for better reuse of simpler lemmas and greater automation. Note that most of the effort in proving JAD was invested in stating and proving
specifying and verifying sparse matrix codes

<table>
<thead>
<tr>
<th>Reuse degree</th>
<th># Rules</th>
<th>Avg. LOC</th>
<th>CSR</th>
<th>SCSR</th>
<th>COO</th>
<th>CSC</th>
<th>JAD</th>
<th>Total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>28.3</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>11</td>
<td>18.3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6.3</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>10.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>6.0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6.4</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>20</td>
<td>33.3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>2.3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>20</td>
<td>33.3</td>
</tr>
</tbody>
</table>

Figure 10: Reusability analysis of our sparse-matrix code prover.

simplification of transpose-transpose composition. Fig. 10 does not account for these rules, as they are quite general and were implemented as an extension to Isabelle’s theory of lists.

5.2 Case Study: Hierarchical Compression Formats

This subsection evaluates expressiveness of the LL language. This question is motivated by the absence from LL of some powerful constructs, such as first-class functions and folds. We show that LL can express advanced formats even without these constructs.

Sparse CSR (SCSR). The SCSR format extends CSR with another layer of compression. SCSR compresses the list of (compressed) rows, by filtering out empty rows (i.e., those rows that have only zero-valued elements). The remaining rows are associated with their row index.

Implementing SCSR in LL amounts to obtaining the CSR format, which compresses individual rows, followed by compression of the resulting list of compressed rows. Again, LL manages to express format construction as a pipeline of stages.

\[
\text{def scsr: csr -> } [\text{len != 0 ? (, id)}]
\]

The corresponding SpMV implementation needs to account for the row indexes. It must also inflate the resulting sparse vector into dense format:

\[
\text{def scsrmv(A, x):}
\]
\[
\text{A -> [i, r: r -> unzip -> snd * x[fst] ->}
\]
\[
\text{sum -> (i, id)] ->}
\]
\[
\text{inf1(0, m, id)}
\]

Alternatively, we can reuse SpMV for CSR:

\[
\text{A -> unzip -> (fst, csrmv(snd, x)) -> zip -> inf1(0, m, id)}
\]

SCSR demonstrates the ability of our prover to peel off the additional compression layer and prove correctness of the overall result, while requiring only two rules in addition to those needed by CSR (see Fig. 10).

Next, we investigate two optimizations for SpMV—register blocking and cache blocking—designed to improve temporal locality of the vector \(x\) at two different levels of the memory hierarchy. The locality is improved by reorganizing the computation to operate on smaller segments of the input matrix, which in turn allows the reuse of a segment of \(x\).

Register blocking. This optimization is useful when the nonzero values in the matrix appear in clusters [14]. The idea is to place the cluster of nonzeros in a small dense matrix. To obtain register-blocked format, instead of compressing single-cell values (i.e., numbers), a matrix is partitioned into uniformly sized rectangular blocks. These dense blocks form the base elements for compression: a block is filtered away if all its elements are zeros; if the block is
nonzero, it is represented as a dense matrix. The size of these blocks is chosen so that the corresponding portion of the vector \( x \) can reside in registers during processing of a block.

Register blocking can be applied to all sparse formats described in Section 3.2. The \( 2 \times 2 \) blocked representation of Fig. 1(a) can be seen in Fig. 11(a). Applying CSR compression to this blocked matrix results in the register-blocked CSR format in Fig. 11(b).

To construct the register-blocked CSR (RBCSR) in LL, we first “blockify” the dense matrix with the block function, which transforms a dense matrix \( A \) of size \( mr \times nc \) to an \( m \times n \) matrix of \( r \times c \) dense blocks. Next, we pair these blocks with their column indices using \([\text{enum}]\), and filter out the all-zero blocks.

```python
def rbcsr(A):
    block(r, c, A) ->
    [\text{enum} -> \text{snd} -> \{\text{neq ' 0} \rightarrow \text{disj} \rightarrow \text{disj ?} \}]
```

In SpMV of an \( r \times c \)-RBCSR matrix \( A \) and a dense vector \( x \), we first bind the names \( B \) and \( l \) to each dense block and its index, respectively, and perform dense matrix-vector multiplication (densemv) on each block and the corresponding \( c \)-sub-vector of \( x \). The latter is obtained by breaking \( x \) into a list of \( c \)-vectors using \( \text{block}(c, x) \) and selecting the appropriate sub-vector. The result vectors in a row block are summed, and the final result is obtained by concatenating the result sub-vectors from all row blocks.

```python
def rbcsrmv(A, x):
    A ->
    \{\text{sum} \rightarrow \text{concat} \}
```

Our prover allowed us to easily extend proofs to blocked formats because our matrices are of parametric type; the prover can work with matrices of numbers as well as with matrices whose elements are matrices. Parameterization of matrices was expressed with Isabelle/HOL type classes, which are used to restrict types in introduction rules.

We use a theory of finite matrices [12]. Here, too, the size of a matrix is not encoded in the matrix type (denoted \( \alpha \) matrix) but it is required that matrix dimensions are bounded. To represent matrices as abstract values, we introduce the matrix conversion function:

\[
\text{matrix} :: \text{nat} \rightarrow \text{nat} \rightarrow (\text{nat} \rightarrow \text{nat} \rightarrow \alpha) \rightarrow \alpha \text{ matrix}
\]

The first two parameters specify the row and column dimensions, respectively. The third parameter is the abstract value encoded into the matrix. Implementing functions on compressed matrices necessitates a few more conversion functions:

\[
\text{block} \_\text{vector} :: \text{nat} \rightarrow \text{nat} \rightarrow [\alpha] \rightarrow [\alpha \text{ matrix}]
\]

\[
\text{block} \_\text{matrix} :: \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \rightarrow [\alpha] \rightarrow [[\alpha]] \rightarrow [[\alpha \text{ matrix}]]
\]

\[
\text{concat} \_\text{vectors} :: \text{nat} \rightarrow [\alpha \text{ matrix}] \rightarrow [\alpha]
\]

The operation \( \text{block} \_\text{matrix} m n k l A \) transforms the object \( A \), representing an \( mk \times nl \)-matrix, into an object representing an \( m \times n \)-matrix of \( k \times l \)-blocks; \( \text{block} \_\text{vector} m k x \) transforms the list \( x \) of length \( nk \) into a list of \( n \) \( k \)-vectors; \( \text{concat} \_\text{vectors} k x \) is the inverse operation, unpacking the \( k \)-vectors in \( x \).

This code shows register-blocked CSR code in Isabelle.

\[
\text{rbcsr} m n k l A = \text{map} (\text{filter} (\lambda(i, v). v \neq 0) \odot \text{enum}) (\text{block} \_\text{matrix} m n k l A)
\]
Specifying and Verifying Sparse Matrix Codes

\[ \text{rbcsrmv } m \ n \ k \ l \ (A, \ x) = \]
\[ \text{concat\_vectors } k \]
\[ (\text{map} \ (\text{lists\_sum} \circ \]
\[ (\text{map} \ (\lambda v. \ \text{snd} \ v \ * \ \text{block\_vector} \ n \ l \ x \ ! \ \text{fst} \ v)) \) \ A) \]

We require that block dimensions are greater than zero and properly divide the respective matrix and vector dimensions. The correctness theorem follows.

\[ k \neq 0 \land l \neq 0 \land \text{ilist}_M (m \ * \ k) \ (n \ * \ l) \ A' \ A \land \text{ilist}_V (n \ * \ l) \ x' \ x \]
\[ \rightarrow \text{ilist}_V (m \ * \ k) \ (\lambda i. \ \Sigma j < n \ * \ l. \ A' \ i \ j \ * \ x' \ j) \]
\[ \text{rbcsrmv } m \ n \ k \ l \ (\text{rbcsr } m \ n \ k \ l \ A, \ x) \] \hfill (5.1)

After adding the introduction rules in Fig. 9 and a few rewrite rules for matrix, the prover automatically proves Eq. (5.1).

**Cache blocking.** The idea in cache blocking is to reduce cache misses for the source vector \( x \) when it is too large to entirely fit in the cache during SpMV. We consider static cache blocking [7]. The sparse matrix is partitioned into rectangular sub-matrices of size \( r \times c \). While in register blocking these sub-matrices were kept dense, in the cache-blocked format they are compressed.

Our cache blocking scheme differs from the one in [7] in that we only allow cache blocks to start at column indices which are multiples of \( c \); this restriction leads to suboptimal compression. We believe that this restriction can be relaxed by augmenting LL with a blocking function that creates optimally placed blocks.

Notice that the construction of a cache-blocked matrix is very similar to the construction for register blocking. The only difference is the additional compression applied to each block. The LL code for the CSR compressed cache-blocked matrix, whose blocks are stored in CSR format, is shown below.

```python
def cbcsr(A):
    block(r, c, A) ->
    [enum -> [snd -> [[neq ' 0] -> disj] -> disj ? ] ->
     [1, B: (1, B -> csr)])]
```

The corresponding cache-blocked SpMV in LL:

```python
def cbcsrmv(A, x):
    A ->
    [[1, B: (B, block(c, x)[1]) -> csrmv] -> sum] ->
    concat
```

In the cache-blocked SpMV, we again notice the similarity to the register-blocked SpMV. The two codes are identical except for the function used for multiplying a block by a vector. It is somewhat desirable to factor out these inner multiplications (dense\_mv and csrmv) but this is not possible in LL. The reason is that LL does not support lambda abstraction, which would allow reuse of code common to register- and cache-blocked versions. We have refrained from enriching LL with first-order functions for now because this allows for simpler verification and gives us broad verification coverage. We do not consider the absence of lambda abstraction a significant disadvantage because even optimized LL programs are small. In the future, we may decide to extend LL with a template mechanism that will be used to instantiate such hierarchical composition, allowing code reuse.

The verification of cache-blocked sparse formats was not yet implemented. We expect that the amount of work will not be substantial, based on our experience with other hierarchical formats.

6 Related Work

**Specifying sparse matrix code** Bernoulli [9, 8] is a system that synthesizes efficient low-level implementations of matrix operations given a description of the sparse format using relational algebra. This is impressive and permits rapid development of fast low level implementations. However, the functionality of the system was limited and it had limited impact. Instead, we are expressing formats using a functional programming language which can be mechanically verified. We believe that function-level programming provides the right level of abstraction for expressing the
desired transformations. Moreover, LL can be embedded in existing call-by-value functional programming languages. Compiling LL into low-level code is a work in progress.

The synthesizer by Bik et al. [2] produces efficient implementations by replacing $A[i, j]$ in dense-matrix code with a representation function that maps to the corresponding sparse element; powerful compiler optimizations then yield efficient code.

**Verifying sparse matrix code** We are not aware of previous work on verifying full functional correctness of sparse matrix codes. We are not even aware of work that verified their memory safety without explicitly provided loop invariants. Our own attempts at verification included ESC/Java, TVLA and SAT-based bounded model checking, neither of which was satisfactory. Furthermore, neither of these tools was capable of proving higher-order properties like the ones we currently prove. This led us to raising the level of abstraction and deferring to purely functional programs where loops are replaced with comprehensions and specialized reduction operators.

**Higher order verification** Duan et al. [5] verified a set of block ciphers using the interactive theorem prover HOL-4. They proved that the decoding of an encoded text results in the original data. Their proofs are mostly done using inversion rules, namely rules of the form $f(f^{-1}x) = x$, and algebraic rules on bit-word identities. For the block ciphers used by AES and IDEA special rules where needed. The domain of block cipher verification does not seem to require more complicated rules than bit-word identities.

7 Conclusion

In this paper we showed how to raise the level of abstraction for sparse matrix programs from imperative code with loops to functional programs with comprehensions and limited reductions. We also developed an automated proof method for verifying a diverse range of sparse matrix formats and their SpMV operations. This was accomplished by introducing relations that map a sparse representation to the abstract (mathematical) one. Through a clever definition of these representation relations we were able to build a reusable set of simplification and introduction rules, which could be applied to a variety of computations.

We are currently working on the problem of compiling the functional code into an efficient C code. Deploying techniques from predecessor functional and data parallel languages, we already exhibit promising performance results with real-world sparse formats.

Bibliography


