Building Code Transpilers for Domain-Specific Languages Using Program Synthesis

Sahil Bhatia
University of California, Berkeley, CA, USA

Sumer Kohli
University of California, Berkeley, CA, USA

Sanjit A. Seshia
University of California, Berkeley, CA, USA

Alvin Cheung
University of California, Berkeley, CA, USA

Abstract

Domain-specific languages (DSLs) are prevalent across many application domains. Such languages let developers easily express computations using high-level abstractions that result in performant implementations. To leverage DSLs, however, application developers need to master the DSL's syntax and manually rewrite existing code. Compilers can aid in this effort, but part of building a compiler requires transpiling code from the source code to the target DSL. Such transpilation is typically done via pattern-matching rules on the source code. Sadly, developing such rules is often a painstaking and error-prone process.

In this paper, we describe our experience in using program synthesis to build code transpilers. To do so, we developed MetaLift, a new framework for building transpilers that transform general-purpose code into DSLs using program synthesis. To use MetaLift, transpiler developers first define the target DSL's semantics using MetaLift's specification language, and specify the search space for each input code fragment to be transpiled using MetaLift's API. MetaLift then leverages program synthesizers and theorem provers to automatically find transpilations expressed in the target DSL that is provably semantic equivalent to the input code. We have used MetaLift to build three DSL transpilers targeting different programming models and application domains. Our results show that the MetaLift-based compilers can translate many benchmarks used in prior work created by specialized implementations, but can be built using orders-of-magnitude fewer lines of code as compared to prior work.

2012 ACM Subject Classification Software and its engineering → Compilers

Keywords and phrases Program Synthesis, Code Transpilation, DSLs, Verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.38

Category Experience Paper

Supplementary Material *Software*: `https://github.com/metalift/metalift`

Funding This work was supported in part by a Google BAIR Commons project, by the DARPA LOGiCS project, by the NSF FMitF-1837132, IIS-1955488, IIS-2027575, CCF-1723352, ARO W911NF2110339, ONR N00014-21-1-2724, and DOE award DE-SC0016260.

Acknowledgements We would like to thank Shadaj Laddad, Maaz Ahmad and anonymous reviewers for their insightful feedback.

1 Introduction

Domain-specific languages (DSLs) are now popular means to develop applications across many domains. Besides improving programmability, modern DSLs often expose *domain-specific optimizations* via their interfaces for applications to leverage specialized hardware accelerators [16, 32, 6, 21], or domain-specific code transformation [15, 43, 12].

Yet, to capitalize on the benefits provided by DSLs, developers must learn the interfaces provided by the target DSL. For existing applications, developers need to painfully reverse-engineer legacy code, potentially convolved with optimizations, before rewriting it, only to realize that newly emerging frameworks can turn freshly rewritten code into legacy applications. Needless to say, each rewrite is another opportunity to introduce bugs into the application.

The classical mechanism to alleviate this problem is for DSL designers to build pattern-driven transpilers that translate programs, say written in general-purpose languages, into their DSLs [4]. While such transpilers are essentially part of all modern optimizing compilers, they often require developing a complex network of inter-connected translation rules [22, 31, 39], which is a highly tedious and error-prone task.

Instead of transpilation rules, researchers have leveraged advances in program synthesis [11, 17] for code transpilation, where the idea is to replace the rule-matching machinery with a code synthesizer that finds programs written in the target language that are semantically equivalent to the input code fragment [23, 14, 27, 20, 3]. However, using such techniques involves encoding the semantics of the input program as a synthesis problem. Furthermore, building such transpilers requires implementing specialized synthesis procedures, which relies on specialized knowledge of synthesis algorithms that most transpiler designers do not possess.

In this paper, we describe our experience in building code transpilers using program synthesis. To do so in a systematic manner, we designed a new framework called MetaLift. Our goal in building MetaLift is to free transpiler developers from designing a myriad of pattern matching rules for transpilation, while making synthesis technology easily accessible for code translation. Given input code written in the source language (MetaLift currently supports LLVM), developers can use MetaLift to implement a code transpiler that uses program synthesis to search for code in the target language that is *provably semantically equivalent* to the input. To use MetaLift, developers first define the semantics of their target DSL using MTL, i.e., MetaLift's Specification Language. Then, using MetaLift's search space API, developers specifies the search space of DSL programs for each input. The search space can be constructed programmatically by analyzing each input code to transpile. Given the DSL definition and search space description, developers then write a transpiler driver that orchestrates the transpilation process.

MetaLift is designed with developer usability in mind in constructing transpilers. As we will discuss in detail, MTL is designed as a specification language embedded within Python for ease of use. As MetaLift focuses on ensuring semantic equivalency between the source and transpiled code, MTL consists of a small number of constructs, and is designed to be high-level so that developers can easily use it to express the semantics of each construct in their target DSL, and yet simple enough for MetaLift to compile down as the input to different synthesizers and verifiers. Likewise, the MetaLift's API is also designed to abstract away the details of synthesis and verification from the developer.

To evaluate MetaLift, we have used the framework to reproduce three different transpilers described in prior work spanning multiple application domains and programming models. Our evaluation shows that they require order-of-magnitude fewer lines of code to build as compared to prior work, with the resulting transpiler generating the same (or very similar) code as the original implementations.

In summary, this paper makes the following contributions:

- We design MTL for developers to specify the semantics of different constructs in their DSLs. The design of MTL is general enough to support many real-world DSLs, yet simple enough to be translated as the input to various program synthesizers and verification engines.

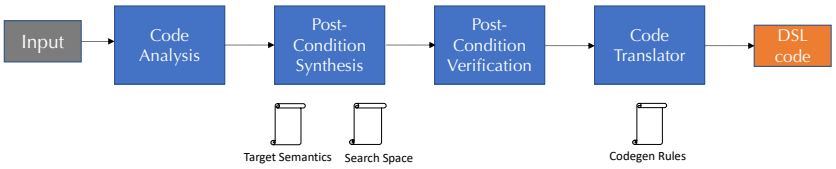

Figure 1 An overview of the MetaLift architecture.

- We describe MetaLift, a unified framework for developing transpilers for DSLs. Rather than designing pattern matching rules, MetaLift enables designers to use program synthesis to easily translate input code to their DSLs without requiring any program synthesis expertise. Furthermore, the translations generated by MetaLift are formally verified for semantic equivalence to the source code, so developers are assured of an accurate translation.
- We show how MetaLift generalizes previous work on building transpilers using program synthesis by creating two DSL transpilers that translate from general-purpose code to these DSLs. These three DSLs are aimed at different application domains. Our evaluations demonstrate how MetaLift dramatically reduces the effort required to build these compilers.

We organize the rest of the paper as follows. We provide an overview of MetaLift (Section 2). We describe MetaLift in more detail (Section 3) using a representative example. We evaluate MetaLift in Section 4 and review the prior approaches for building transpilers in Section 5. Finally, we conclude (Section 6) with directions for future work.

2 Overview

In this section, we provide an overview of the MetaLift framework. The high-level architecture of MetaLift is shown in Figure 1. For the majority of the paper, we use the example in Figure 2 as our running example and describe how to build a transpiler using MetaLift that translates sequential Java code to the Spark DSL. Spark [43] provides users with an interface to efficiently process large-scale distributed computations in a parallel and fault-tolerant manner.

Concretely, the example in Figure 2a takes as input a list of words and counts the frequency of each word in input list. Figure 2b shows the equivalent implementation using map reduce operators in the Spark DSL. The `map` operator returns `(word,1)` for each word in the input list, and the `reducebykey` operator then uses the reducer function `(v1 + v2)` to aggregate each unique key in the map operator's output.

The Mold compiler [31] has implemented a syntax-driven compiler that automatically translates sequential Java code to Spark. To perform this translation, Mold uses rewrite rules that pattern match on the input source code. However, these rules can be hard to implement as

1) they must be expressive in order to capture all of the different coding patterns,
2) they must ensure semantic equivalence to the source code, and
3) they must be maintained as the DSLs change.

For instance, as described in their paper, Mold requires 22 different rules to generate the corresponding Spark program for the same word count program shown in Figure 2a.

```cpp
map<string, int> countWords(vector<string> words) {
   map<string, int> counts;
       for (int j = 0; j < words.size(); j++) {
              string word = words[j];
              int prev = 0;
              if (counts.find(word) != counts.end())
                     prev = counts[word];
              counts[word] = prev + 1;
       }
       return counts;
}
```

(a) Input Source Code: Sequential C++ Code (Java benchmarks are converted to C++ since the MetaLift front-end currently supports C++).

```scala
Map<String, Integer> countWords(List<String> words) {
       Map<String, Integer> counts = new HashMap<String, Integer>();
       counts = words.mapToPair(v -> new Tuple2<String,Integer>(v, 1))
                .reduceByKey((v1,v2) -> v2 + v1).collectAsMap();
       return counts;
       }
```

(b) Output: Apache Spark Code.

Figure 2 Translation from sequential C++ code to Apache Spark DSL.

```python
# target DSL definition, see Figure 11
# grammar description, see Figure 13
# code generation rules, see Figure 18

def transpiler(source): # driver program
    liveVars, modVars, VC = analyze(source)
    verifiedSummaries = synthesize(VC, targetLang(), grammar(liveVars, modVars))
    transpiledCode = codeGen(verifiedSummaries)
    return transpiledCode
```

Figure 3 Example of the MetaLift driver code to transpile the running example shown in Figure 2.

Instead of designing syntax-driven rules, developers can use MetaLift to build this transpiler. MetaLift leverages program synthesis to search for transformations that are semantically equivalent to the source code. Figure 3 shows the driver code developers will write to implement a compiler using MetaLift, where they provide the following inputs to MetaLift:

1. **Target DSL Definition**. First, using MTL and the programming interface provided by MetaLift, developers define the semantics of the operators in their target DSL . Each operator represents a program construct in the target language. For instance, for our Spark compiler, we define the semantics of the map and reduce operators as shown in Figure 11. In Section 3.3, we discuss how MTL can be used to define the semantics for each construct in the target language.
2. **Search Space Definition**. Besides the target language, developers also define the search space description to guide MetaLift's synthesis engine (Figure 13). The search space provides the space of possible programs in which the synthesis engine can look for

equivalent program transformations. Using the variable information returned during the analysis phase performed by MetaLift, developers can use MTL to describe the search space. This will be discussed further in Section 3.6.

3. **Code Generation Rules**. The final input that developers provide is the syntax-driven rules to translate the summaries synthesized by MetaLift into executable target DSL code. Note that translating from summaries is much easier than translating directly from source code as the summaries are already encoded using the target DSL's operators. Section 3.9 provides more details about how these rules can be implemented.

Developers then write a transpiler driver that invokes the inputs mentioned above (as shown in Figure 3), using MetaLift's API as follows:

1. **Code Analysis**. MetaLift's Analysis API takes as input the source code to be translated and compiles it into LLVM intermediate representation (IR), as shown in Line 6 in Figure 3. We use LLVM IR because it allows for the compilation of multiple general-purpose languages (e.g., C, C++, Fortran) using various front ends, giving MetaLift the flexibility to support multiple languages on the source side. The goal of the analysis phase is to compute verification conditions from the LLVM IR. Verification conditions (VCs) are logical statements that assert that the program is correct with respect to the given pre-condition and post-condition if they hold. The VCs serve as the specification for the synthesizer. The analysis phase also returns information about the output variables and the variables being modified in the source code. We describe MetaLift's analysis phase in more detail in Section 3.1.
2. **Synthesis and Verification**. Once the analysis phase returns the VCs, developers then invoke MetaLift's synthesis API, as shown in Line 7 in Figure 3. The VCs generated during the analysis phase form the synthesizer's specification. MetaLift then synthesizes program summaries that meet these specifications. To make the search tractable, MetaLift's synthesizer uses the search space description provided by the developers. These summaries are restricted to be expressed using only the operators in the target DSL defined by the developers. Logically, a program summary is the post-condition that captures the program's final states after execution. Finally, MetaLift formally verifies that the generated program summary is semantically equivalent to the input source code using an automated theorem prover. Synthesis and verification phases of MetaLift are discussed in Section 3.7 and Section 3.8, respectively.
3. **Code Generator**. In the final step as shown in Line 8 in Figure 3, MetaLift invokes the user-defined code generation rules to convert the verified summaries into executable DSL code. The compiled code is then returned back to the user.

3 Framework

In this section, we discuss each component of MetaLift in detail by building a transpiler for our example in Figure 2.

3.1 Analysis

The front-end of MetaLift takes as input the source code written in a general-purpose languages which the developers want to transpile to their DSLs. As mentioned, our current prototype supports languages that can be compiled to LLVM IR. Once compiled to the LLVM IR, the next step in the analysis phase is to augment the generated LLVM IR to enable computation of verification conditions.

(a) LLVM code structure with loops.

(b) LLVM code structure modified to remove the loops.

Figure 4 Transformations performed on loop constructs during analysis phase.

Prior work [18, 24] has recently introduced frameworks for automated verification of software systems. These frameworks takes as input the description of the system to be verified, the specification that the system must satisfy, and internally reduces the problem to symbolic constraints that are then discharged to a theorem prover for verification. However, MetaLift cannot reuse any of these front-ends for generating VCs as they rely on the source code already having annotated with invariants and post-conditions. MetaLift instead models invariants and post-conditions as predicates that take in all live variables at the point when they are declared, and leave the body of the predicates to be *synthesized* later on, to be explained in Section 3.7.

Verification Conditions. In Floyd-Hoare logic (FHL) [19], the verification problem is abbreviated using the Hoare Triple $\{A\}\ S\ \{P\}$. To establish the validity of a Hoare triple, we need to prove that for all executions starting from states that satisfy A (pre-condition), after executing statement S, should satisfy P (post-condition). An example of a valid Hoare triple is $\{y \leq x\}\ z := x; z := z + 1\ \{y < z\}$. This problem can be further simplified as finding a Boolean predicate which characterizes the set of pre-conditions from which every execution of S would lead a to state that satisfies P. These Boolean predicates are known as verification conditions. Formally, we can represent this as proving the following logical statement as $A \rightarrow VC(S, P)$.

An additional predicate called loop invariant is required for programs with loop constructs to prove that the post-condition is valid regardless of the number of iterations of the loop. Thanks to the efficient theorem provers [42, 7], such inference rules provided by FHL can be encoded in solvers such that any Hoare triple can be mechanically checked for its validity.

In MetaLift, S corresponds to the program statements in the input code to be transpiled, while A and P correspond to expressions written using MTL to be discussed in Section 3.3. Our goal is to synthesize a post-condition for each input code that are expressed using the target language constructs provided by the user, while ensuring that it forms a valid Hoare triple together with the input code S and pre-condition A.

Transformations for Loop Constructs. For programs that do not have any loop constructs, generating VCs is straightforward. However, programs with looping constructs requires more processing. In Figure 4a, we show a simplified control-flow graph of LLVM basic blocks for a program with loops. For such programs, we first identify the back edges, *i.e,* the edge from the loop body to the loop head, and remove that edge to transform the bitcode into an acyclic graph. To preserve the semantics of the loop after removing the back edge, we annotate the start of the loop head block with *havoc* statement as shown in Figure 4b. The

havoc statement is introduced for all variables modified in the loop to update their contents to fresh symbolic variables. Doing so allow us to mimic the effect of executing an arbitrary iteration of the loop, which will be crucial in generating the correct verification condition.

The acyclic graph now represents any arbitrary iteration of the loop. To prove the validity of the Hoare triple $\{A\}$ **while** G **do** S $\{P\}$, we must identify a *loop invariant* that holds at the beginning of the loop, at every iteration of the loop, and at the end of the loop terminates. As the input code is not annotated with loop variants, to generate VCs which can prove the validity of the described Hoare triple, we add the following annotations (shown in **bold** in Figure 6) to the LLVM code to be transpiled:

1. assert that invariant holds before the entry to the loop (Line 9)
2. havoc the variables being modified inside the loop (Line 13)
3. assume invariant holds before the execution the body (Line 15) and assume loop guard is true (Line 23)
4. assert invariant holds after the execution of the loop body (Line 35)
5. assert invariant holds after the loop exits (Line 38)
6. assume loop guard is false and assert that the post-condition holds before the program exits (Line 40)

3.2 Verification Condition Generation

There are several ways to prove functional equivalence of the source and target program. We adopt VCs as the specification for checking the functional equivalence. VCs enable us to prove complete functional equivalence between the source and generated target code, meaning that we can generate a proof of equivalence for all possible inputs. While there are other potential means to provide specifications, such as using the inputs and outputs from test cases or relying on the equivalence between a general-purpose program and its corresponding DSL program, these approaches have serious drawbacks:

1. Using testing as the specification only guarantees correctness for a finite set of inputs since it's not feasible to generate all possible test cases. Automating test case generation is also not always reliable, as it may not cover all paths of the program, and running these programs might not always be feasible. As a result, the synthesized program will only be semantically equivalent modulo the inputs and outputs from the test cases that were used as specification.
2. Checking equivalence between the source and target programs directly would require encoding the semantics of different constructs appearing in the source, and the most popular symbolic synthesizers [37, 41] or verifiers [42, 7] do not support semantic reasoning of loops. These symbolic synthesizers only reason about loops after they have been unrolled for a finite amount of iterations, effectively converting the loop into straight-line code. Because of this, it is challenging to check equivalence for programs with loops without generating VCs and loop invariants, as doing so only provides guarantees up to a certain bound.

Our VC generation algorithm is inspired from [8]. The LLVM compilation process generates LLVM bitcode which is represented using the LLVM IR. Figure 6 shows the abridged LLVM bitcode for the source code in Figure 2a. LLVM bitcode contains multiple basic blocks, i.e, contiguous sequence of LLVM IR instructions with just one entry and one exit point. A basic block after the transformation in the analysis phase has the general structure shown in Figure 5a.

```
blk: bb_id
  assume e_a;
  %i = ...;
  %i1 = ...;
  assert e_b
  br label %bb_id'
```

(a) Basic Block structure after the analysis phase.

```
blk: bb_id
  assume e_a;
  assume %i = ...;
  assume %i1 = ...;
  assert e_b
  br label %bb_id'
```

(b) Basic Block structure after converting instructions to assumes.

Figure 5 Basic block structure of the LLVM bitcode.

```
blk: bb_start
  %i = alloca %list* ;i = input variable words
  %i1 = alloca %dict* ;i1 = output variable counts
  %i2 = alloca i32 ;i2 = loop counter j
  store %list* %arg, %list** %i
  store call %dict* newMap(), %dict** %i1
  store i32 0, i32* %i2
  ;invariant is true before the exectution of the loop
  (assert call inv (load i1) (load i2) arg)
  br label %bb_head

blk: bb_head
  (havoc i1 i2)
  ;invariant is true at the start of the loop body
  (assume call inv ((load i1) (load i2) arg))
  %i7 = load i32, i32* %i2
  %i8 = load %list*, %list** %i
  %i9 = call i32 length(%list* %i8)
  %i10 = icmp slt i32 %i7, %i9 ;j < words.size()
  br i1 %i10, label %bb_body1, label %bb_exit

blk: bb_body1
  (assume i10)  ;loop guard is true
  ;instructions to update the counts in the output map (not shown for brevity)
  %i11 = ...
  %i12 = ...
  br label %bb_body2

;instructions to update the counts in the output map
blk: bb_body2
  %i28 = load i32, i32* %i2, align 4
  %i29 = add nsw i32 %i28, 1
  store i32 %i29, i32* %i2, align 4
  ;invariant is true after executing the body of the loop
  (assert call inv ((load i1) (load i2) arg))

blk: bb_exit
  (assume not(i10))  ;loop guard is false
  %i31 = load %dict*, %dict** %i1
  (assert call ps (i31 arg))  ;post-condition is true
```

Figure 6 LLVM bitcode for the source code in Figure 2a. Bitcode is annotated with assumes, asserts and calls to inv and ps for generating the verification conditions.

Our VC generation algorithm computes an assertion for each of the basic blocks. For each basic block BB in our bitcode we introduce a new Boolean variable BB_{ok} and the VC for that block is expressed as:

$$BB_{ok} = VC(S, \bigwedge_{B \in Succ(A)} B_{ok}) \tag{1}$$

where S represents all the instructions of the block and $Succ(A)$ represents the set of successor basic blocks of block BB.

In addition to the transformations during the analysis phase, we convert all the instructions in the block to assume statements, as illustrated in Figure 5b.

We then symbolically execute all the instructions in the block to generate the VCs. For symbolic execution, we maintain two dictionaries to model the *memory* (m) and *registers* (r). m maps memory cells to their values, while r keeps track of the results of the instructions' execution. The state of the symbolic executor is maintained as a quadruple $< m, r, a, b >$ that represents the state of the memory, registers, assumptions encountered, and assertions encountered thus far during symbolic execution, respectively. a then represents the final VC for the input code fragment after symbolic execution terminates.

STORE

$$\frac{m' = m[r[i] \mapsto v]}{[\![store\ v\ i]\!](< m, r, a, b >) \Rightarrow < m', r, a, b >}$$

MEMORY ALLOCATION

$$\frac{fresh\ l \qquad r' = r[i \mapsto l] \qquad m' = m[l \mapsto \bot]}{[\![i = alloca\ t]\!](< m, r, a, b >) \Rightarrow < m', r', a, b >}$$

LOAD

$$\frac{r' = r[i \mapsto m[i_a]]}{[\![i = load\ i_a]\!](< m, r, a, b >) \Rightarrow < m, r', a \wedge (i = m[i_a]), b >}$$

HAVOC

$$\frac{fresh\ v' \qquad m' = m[v \mapsto v']}{[\![havoc\ v]\!](< m, r, a, b >) \Rightarrow < m', r, a, b >}$$

ARITHMETIC OPERATORS (AOP)

$$\frac{v = aop(r[i_a], r[i_b]) \qquad r' = r[i \mapsto v]}{[\![i = aop\ i_a\ i_b]\!](< m, r, a, b >) \Rightarrow < m, r', a, b >}$$

BINARY COMPARISONS (BOP)

$$\frac{v = bop(r[i_a], r[i_b]) \qquad r' = r[i \mapsto v]}{[\![icmp\ bop\ i_a\ i_b]\!](< m, r, a, b >) \Rightarrow < m, r', a \wedge (i = v), b >}$$

ASSUME

$$\frac{}{[\![assume\ e]\!](< m, r, a, b >) \Rightarrow < m, r, a \wedge [\![e]\!], b >}$$

ASSERT

$$\frac{}{[\![assert\ e]\!](< m, r, a, b >) \Rightarrow < m, r, a, a \rightarrow b \wedge [\![e]\!] >}$$

Figure 7 Computing the VC via symbolic execution on LLVM instructions. The symbol $\wedge$ and $\rightarrow$ denotes logical And and Implies operator respectively.

In Figure 7, we describe how we compute the VC for each type of LLVM opcode, while Table 1 shows our symbolic executor in action using these rules. To illustrate, the STORE rule in Figure 7 states that if the current state of the symbolic executor is $< m, r, a, b >$, to symbolically execute a `store` instruction that stores value v to the cell location stored at register i, we create a new memory m' where the cell $r[i]$ is mapped to the value v, with all other cells and their mappings remain unchanged from m. Similarly, the MEMORY ALLOCATION rule states that given the current symbolic executor state $< m, r, a, b >$, to symbolically execute an `alloca` instruction that allocates new memory pointing to a value of type t at cell l with l to be stored in register i, we create a fresh symbolic cell l representing the pointer to the newly allocated memory, update the register state r to map register i to l instead, and update the memory state m where cell l points to uninitialized value $\perp$. The new symbolic state $< m', r', a, b >$ is then returned.

As a concrete example, we show the state of the memory, registers and assumes expression after the execution of each instruction in the block *bb_head* in Figure 6. For this block, $asserts = null$ and $Succ(A) = \{bb\_body1, bb\_exit\}$. The computed VC according to the Equation (1) and rules in Figure 7 would then be:

$$\begin{aligned} BB\_head_{ok} &= (assumes \rightarrow bb\_body1 \wedge bb\_exit) \\ assumes &= inv(i1\_0, i2\_1, arg) \wedge (i7 = i2\_1) \wedge (i8 = arg) \quad \wedge (i9 = length(arg)) \\ &\quad \wedge (i10 = i2_1 < length(arg)) \end{aligned} \tag{2}$$

Table 1 Symbolic Execution for the block BB_head in Figure 6 using the inference rules defined in Figure 7. Each row depicts an LLVM instruction as well as the resulting *memory* (m) and *registers* (r) state after the instruction is symbolically executed. Due to space constraints, we do not show inv (and True) in all the execution steps.

havoc i1 i2 m= [i1↦ i1_0, i ↦ arg, i2↦i2_1] r= [] assumes = True
assume inv ((load i1) (load i2) arg)) m= [i1↦ i1_0, i2↦ i2_1, i ↦ arg] r= [] assumes = True ∧ inv(i1_0, i2_1, arg)
assume %i7 = load i32, i32* %i2 m= [i1↦ i1_0, i2↦ i2_1, i ↦ arg] r= [i7↦ i2_1] assumes = True ∧ (i7 = i2_1)
assume %i8 = load %list*, %list** %i m= [i1↦ i1_0, i2↦ i2_1, i ↦ arg] r= [i7↦ i2_1, i8↦ arg] assumes = (i7 = i2_1) ∧ (i8 = arg)
assume %i9 = call i32 length(%list* %i8) m= [i1↦ i1_0, i2↦ i2_1, i ↦ arg] r= [i7↦ i2_1, i8↦ arg, i9↦ length(arg)] assumes = (i7 = i2_1) ∧ (i8 = arg) ∧ (i9 = length(arg))
assume %i10 = icmp slt i32 %i7, %i9 m= [i1↦ i1_0, i2↦ i2_1, i ↦ arg] r= [i7↦ i2_1, i8↦ arg, i9↦ length(arg), i10 ↦ (i2_1 < length(arg))] assumes = (i7 = i2_1) ∧ (i8 = arg) ∧ (i9 = length(arg)) ∧ (i10 = i2_1 < length(arg))

Similarly, the VCs for other basic blocks can be constructed. Once the VCs have been generated for the all basic blocks, the VC for the entire program can be expressed as $R \rightarrow BB\_start_{ok}$ where R is the conjunction of VCs for each block and BB_start_{ok} is the Boolean variable introduce for the first block in the LLVM bitcode.

Note that the calls to invariant and post-conditions are just placeholders and they are synthesized during the synthesis phase of MetaLift. At the end of the analysis phase MetaLift would generate the following verification conditions:

1. $\forall \sigma.\ Pre(\sigma) \rightarrow Inv(\sigma)$
2. $\forall \sigma, \sigma'. Inv(\sigma) \wedge Body(\sigma, \sigma') \rightarrow Inv(\sigma')$
3. $\forall \sigma.\ Inv(\sigma) \rightarrow Post(\sigma)$

For simplicity, we do not show the verification conditions using LLVM basic block structure. The verification conditions logically state that

1) invariant must hold before the loop
2) invariant must be inductive i.e. it should be true at every iteration of the loop and
3) invariant must assert the post-condition upon exiting the loop.

Internally, MetaLift represents the VCs using MTL which we describe in the next section.

3.3 MTL

We now describe MTL in detail. MetaLift provides developers with an API to use the constructs in the MTL to define the operators of their target DSL. This is in contrast to the prior standalone transpilers [3, 35, 20] where the semantics of the target DSL are embedded within the transpiler and are not reusable. In Section 3.5, we show how developers can use MTL to describe the semantics of the operators in their target DSL.

$$
\begin{aligned}
e \in expr &:= l \mid var \mid e_1\ bop\ e_2 \mid if\ e_1\ then\ e_2\ else\ e_3 \mid \neg e \mid \\
&\quad f(e_1, e_2, ..., e_n) \mid f_u(e_1, e_2, ..., e_n) \mid e_{list} \mid e_{map} \mid e_{tup} \\
e_{list} \in listExpr &:= empty \mid length(e) \mid get(e, i) \mid append(e, i) \mid \\
&\quad prepend(i, e) \mid concat(e_1, e_2) \mid tail(e, i) \mid take(e, i) \\
e_{map} \in mapExpr &:= empty \mid get(e_{map}, i) \mid insert(e_{map}, e_1, e_2) \\
e_{tup} \in tupExpr &:= make(e_1, e_2, ..., e_n) \mid get(e_{tup}, i) \\
l \in literal &:= True \mid False \mid Integer\ Constant \\
bop \in binaryOp &:= and \mid or \mid\ implies\ \mid = \mid + \mid - \mid * \mid / \mid > \mid <
\end{aligned}
$$

Figure 8 Grammar definition of MTL.

MTL is a strongly-typed functional language that consists of three dialects: one for MetaLift to represent VCs internally, one for users to define their target language operators, and another for users to describe the search space. Figure 8 shows the core grammar of MTL that is shared between the three dialects. Even though MTL is a small language, it is expressive enough that can be used to specify the semantics of real-world DSLs, as we will discuss in Section 4.

As shown in Figure 8, the core expressions can be literals, variables passed in as arguments to the function, conditional expressions, expressions combined using Boolean operators (MTL supports all arithmetic, logical and relational operators). MTL also supports operations over list, tuples, and associative maps. It also supports uninterpreted functions (represented as f_u in Figure 8). Uninterpreted functions have no definition; their defining characteristic is simply

that they are deterministic, *i.e.,* for the same input, they always return the same output. They are useful in modeling certain operators (for example, complex math functions that are used in both the source and target languages) for which we only care about determinism.

MTL also provides support for three native data structures: lists, tuples and associative maps, along with some common operations over these data structures. For lists, in addition to standard list operations such as length, append, and value retrieval, MTL supports list comprehension functions, $tail(lst, index)$, which returns all the elements after the first index elements of the list, and $take(lst, index)$, which returns the first index elements of the list. For associative maps and tuples, MTL provides functions for inserting and retrieving values. Developers can also combine these data structures to construct nested data structures such as a list of tuples, a list of lists and a list of maps.

In summary, the different dialects of MTL are used for the following purposes during program transpilation:

1. To represent the VCs generated during the analysis phase.
2. To specify the semantics of the target DSL.
3. To describe the search space for valid program transformations.

3.4 Expressing Verification Conditions in MTL

$$v \in vcExpr := assert(e) \mid assume(e) \mid havoc(var)$$

Figure 9 Additional constructs in MTL for verification condition generation.

Verification conditions generated during the analysis phase are encoded using the constructs in the MTL. As described in the Section 3.1, VC generation requires some additional constructs. We show these constructs in Figure 9 as *vcExpr*. These constructs include **assume**, **assert**, and **havoc**, which takes core expressions as arguments. Their semantics are defined earlier in Figure 7. *vcExpr* includes these constructs in addition to all the core constructs described in Figure 8. These constructs are useful for annotating the LLVM bitcode with invariant and post-condition placeholders. These are utilized by MetaLift internally and are not accessible to the developers via MTL API.

3.5 Expressing Target DSLs in MTL

As discussed in Section 2, MetaLift builds transpilers by leveraging program synthesis. MetaLift searches for program summaries that are semantically equivalent to the source code. Program summaries capture all the changes to the outputs of the source code and are expressed using the operators in the target DSLs. Once the program summaries are synthesized developers can write simple syntax-driven rules to translate them to the concrete syntax of the target DSL.

In order to synthesize the program summaries, the synthesizer requires the semantics of the DSL operators. Developers can implement their domain-specific operators using MTL as defined in Section 3.3. In Figure 11, we describe how map and reduce functions from the Spark DSL can be defined using MTL.

In MetaLift, each construct in the target language is defined using a dialect of MTL. In Figure 10, we show the two constructs, function declaration (*fnDecl*) and axioms (*axiom*), that are added to the core language described in Figure 8 to form this dialect for defining

$$f \in \mathit{fnDecl} \quad := \quad \mathit{name}(\mathit{arg}_1 : t_1, \mathit{arg}_2 : t_2, ..., \mathit{arg}_n : t_n) : t_r \rightarrow e$$
$$a \in \mathit{axiom} \quad := \quad \mathit{name}(\mathit{arg}_1 : t_1, \mathit{arg}_2 : t_2, ..., \mathit{arg}_n : t_n) \rightarrow e$$

Figure 10 Additional constructs in MTL for defining the operators in the target language.

the operators in the target language. MTL supports recursive and higher-order functions. The target language definition is a collection of function declarations and axioms, where the body of both constructs are defined using *expr*'s in the core MTL.

In addition to the definitions of the operators in the target DSL, developers can provide properties specific to the operators in the DSL. Some of the properties that developers can define for the map and reduce operations are shown in Figure 11. These properties are helpful during the verification phase of MetaLift. We give more details about how these properties are used in Section 3.8.

Modern DSLs contain hundreds of functions and therefore searching for program summaries directly in the DSL APIs is not feasible. To make the synthesis algorithm tractable, MetaLift searches for summaries only using the operators defined by the developers. The high-level nature of MTL enables the developers to succinctly define the operators and abstract out the details of these operators in the DSL API. For example, the map definition in Figure 11 can represent the different variations of the map functions (*map*, *flatMap*, *mapToPair*) available in the Spark DSL.

3.6 Describing Search Space for Synthesis

Developers additionally provide the search space description for the synthesizer. Developers provide this description for program summaries as well as any invariant or functions to be synthesized (for example, in Spark, users can ask the synthesizer to generate the bodies of the λ_m and λ_r functions). In MetaLift, the search space is encoded using a context-free grammar (CFG), with the expression that needs to be synthesized at the top level, and the production rules specifying the possible values that the expression can take.

We use a different dialect of MTL for users to describe the search space. Built on top of the core language and as shown in Figure 12, MTL provides one non-deterministic construct, *Choose*, which the developers can use to describe the search space for the synthesis phase. The search space description is used to guide the synthesis process. Semantically, *Choose* lets the synthesizer return any of expressions in its argument list. It can be recursively nested or evaluate to one of the core expressions.

The search space description impacts the synthesizer's performance. If the grammar is too expressive, the synthesizer may take a long time to synthesize the correct expressions; conversely, if the grammar is too restrictive, the synthesizer may fail to find the correct expressions that satisfy the specification. Prior work [20, 2, 3, 14] specialized the grammar descriptions for the target domains and embedded them in the tools. Unfortunately, developers had no way of controlling the expressiveness of these grammars. MetaLift instead allows developers to programmatically control the grammars and even tune grammars for each benchmark separately.

The *Choose* function in Figure 12 is the basic construct that developers can use to describe the search space. $\mathit{Choose}(e_c, e_c)$ allows the users to specify the set of candidate values for a particular expression. The candidate values (e in Figure 12) are described using the constructs in MTL (Figure 8). The synthesizer then selects from this set of possible

```
def targetLang():
    fnDecl("map", lst, λm) =
        if length(lst) == 0 then empty
        else concat(λm(get(ls, 0)), map(tail(lst, 1)))

    fnDecl("reduce", lst, λr) =
        if length(lst) == 0 then 0
        else λr(get(lst, 0), reduce(tail(lst ,1)))

    fnDecl("reduceByKey", lst, λr) =
        reduceByKeyHelper(lst, getkeys(lst), {}, λr)

    fnDecl("redcueByKeyHelper",lst, keys, outMap, λr) =
        if length(lst) == 0 then outMap
        else reduceByKeyHelper(lst, tail(keys, 1),
            insert(outMap, get(keys, 0), reduce(getVals(outMap, get(keys,0)), λr)), λr)

    fnDecl(getKeys, lst): ...

    fnDecl(getVals, lst): ...

# operator specific properties
    axiom("distributiveMapLemma", lst1, lst2) =
        map(concat(lst1, lst2), λm) = concat(map(lst1, λm), map(lst2, λm))

    axiom("distributiveReduceLemma", lst1, lst2, key) =
        get(reduceByKey(concat(lst1, lst2), λr), key) =
               get(reduceByKey(lst1, λr), key) +
                   get(reduceByKey(lst2, λr), key)

    axiom("inductiveMapLemma", lst, index) =
        implies(and((index ≥ 0), (index < length(lst))),
                   map(tail(lst, index), λm) = concat(λm(get(lst,index)),
                                                      map(tail(lst, index + 1), λm)))

    axiom("inductiveMapReduceLemma", lst, index) =
        implies(and((index ≥ 0), (index < length(lst))),
               reduce(map(take(lst, index + 1), λm), λr) =
                   λr(reduce(map(take(lst, index), λm), λr), λm(get(lst, index))))
```

Figure 11 Semantics of the operators in Spark DSL defined using constructs in MTL.

$$e_c \in \mathit{chooseExpr} \quad := \quad \mathit{Choose}(e_c, e_c) \mid e$$

Figure 12 Additional constructs in MTL for description of the search space.

candidates, an expression that meets the specification. In terms of the CFG, *Choose* describes the production rules, i.e., the expansion rules for a non-terminal in the grammar. These grammars can potentially be recursive in nature. We provide a "bound" parameter in our API to control the depth of these grammar. This provides additional flexibility to the developers to programmatically define the depth of unrolling of their search space.

The VCs mentioned in Section 3.1 can be trivially satisfied by setting the invariant and post-condition to be True. The search space description helps prevents the synthesizer from generating such trivial solutions. At a high-level, MetaLift requires the post-condition search space to have the following structure:

$$\forall v_o \in outputVars.\ v_o = e_c \in chooseExpr$$

where *outputVars* is the set of all output variables in the source code. *e* in *chooseExpr* are expressions described using the core constructs in MTL using set of all the *input arguments* and the variables modified in the program. For the example in Figure 2, $outputVars = \{count\}$ and $modVars = \{j, word\}$. Logically, it states that the output variables in the program should be expressed as an expression over the operators in the target DSL. MetaLift uses standard static analysis techniques to infer these variables during the analysis phase. In addition to restricting the synthesizer from generating trivial solutions, the search space description helps in scaling the synthesis problem.

Figure 13 shows one possible grammar description for the post-condition, invariant, mapper, and the reducer function for the translation problem described in Figure 2. The grammar for the program summary asserts that the output variable equals some MapReduce expression over the input data. The grammar for the invariant reasons about the bounds of the loop counter and how the count variable is modified in each iteration of the loop. The mapper function can return an empty list or a list of key-value pair and the reducer function can choose to reduce the input values using one of the arithmetic operators. In this grammar we have restricted the output to be a map, reduce or map followed by a reduce operation.

As mentioned earlier, developers can programmatically control the grammar structure. In the Spark transpiler, a few possible search strategies include incrementally increasing the number of map reduce operations that can be used to express the output variable or incrementally increasing the number of emit statements that the mapper function can use. Figure 14 shows how the driver code in Figure 3 can be modified (less than 20 LOC) to implement the incremental search for number of emit statements. In Figure 14, each iteration of loop increments the number of emits by 1 until the synthesizer finds the semantically equivalent program summaries. This programmatic definition of grammar allows developers to experiment with the synthesis engine without needing to be synthesis experts. Some of these strategies were encoded in the previous tools [3] to make the search tractable.

MetaLift automatically generates the program summaries once the developers have provided the semantics of their operators and the search space description.

3.7 Synthesis

We now discuss the synthesis phase of MetaLift. A typical program synthesis problem is characterized by three parameters: the specification, space of possible programs and the search techniques used by the synthesizer to search for candidate solutions. In MetaLift, the specification are the VCs generated from the source code during the analysis phase and the space of possible programs is represented by the operators and search space specified by the developer. During the synthesis phase, MetaLift uses this information to synthesize the program summaries and any necessary invariants. Essentially, program summaries are logical statements asserting what should be true if the program terminates and it should hold for all possible executions of the program starting from a state that satisfies the pre-condition.

Formally, the synthesis problem can be stated as

$$\exists ps, inv_1, inv_2, ..., inv_n.\ \forall \sigma.\ VC(S,\ ps,\ inv_1,\ inv_2,\ ...,\ inv_n, \sigma) \tag{3}$$

The goal of the synthesizer is to infer the definitions of *ps* and *inv* (in case of programs with loops) such that for all program states σ, the verification conditions (generated during the analysis phase) for a given input source code S is true.

```
def grammar(j, counts, words):
    def psGrammar(counts, words):
        MR = Choose(map(words, λm), reduce(words, λr), reduceByKey(map(words, λm), λr))
        ps = Choose(counts = MR)
        return ps

    def invGrammar(j, counts, words):
        intConst = Choose(0,1)
        listChoice = Choose(words, take(words,j), tail(words,j)
        counterExp1 = Choose((j ≤ intConst), (j ≥ intConst),
                             (j < intConst), (j > intConst))
        counterExp2 = Choose((j ≤ length(words)), (j ≥ length(words)),
                              (j < length(words)), (j > length(words)))
        outExp = Choose(counts = MR)
        MR = Choose(map(listChoice, λm), reduce(listChoice, λr),
                  reduceByKey(map(listChoice, λm), λr))
        inv = Choose(And(counterExp1, counterExp2, outExp))
        return inv

    def mapperGrammar(v):
        litChoice = Choose(0,1,v)
        Emit = Choose(Tuple(litChoice,litChoice),litChoice)
        λm = Choose(append(empty, Emit))
        return λm

    def reducerGrammar(v1, v2):
        λr = Choose((v1 + v2), (v1 - v2), (v1 * v2), (v1 / v2))
        return λr
```

Figure 13 Search space description for the example in Figure 2 using MTL.

```
def searchSpace(liveVars, modVars, numEmits):
    def generateEmits(numEmits):
        if numEmits == 1: return append([], Emit)
        else:
            return append(Emit, generateEmits(numEmits-1))

    def mapperGrammar(v, numEdits):
        litChoice = Choose(0,1,v)
        Emit = Choose(Tuple(litChoice,litChoice),litChoice)
        λm = Choose(generateEmits(numEmits))
        return λm

def transpiler(source): # incremental search driver program
    liveVars, modVars, VC = analyze(source)
    numEmits = 1
    isSynthesized = False
    while(isSynthesized == False):
        grammar = searchSpace(liveVars, modVars, numEmits)
        verifiedSummaries = synthesize(VC, targetLang(), grammar)
        if verifiedSummaries != None:
            isSynthesized = True
        else:
            numEmits += 1
    transpiledCode = codeGen(verifiedSummaries)
    return transpiledCode
```

Figure 14 Modified driver code to implement incremental grammar search for controlling the number of emits in the λ_m function.

Many search techniques have been proposed to solve the problem stated in Equation (3). These include enumerative search [5], deductive search [30], constraint solving [38], statistical [34] and neural approaches [26]. Currently, MetaLift relies on Rosette, an off-the shelf synthesis solver [41] to perform this search; in principle, it could use any synthesis solver supporting the needed theories. To leverage Rosette as the synthesizer, we need to translate the MTL descriptions in the Rosette language. Due to the highly-syntactic nature of the Rosette language, this translation is straightforward. Synthesizer enumerates candidates for ps (and $invs$) from the search space described by the developer until it finds one that satisfies the specification.

Currently available synthesizers have limited support for handling recursive functions in the search space. Internally, synthesizers use theorem provers to determine whether a candidate is valid, i.e., whether it meets the specification. However, while MTL supports list and other data structures, the theory of lists in theorem provers is incomplete and thus insufficient to verify all possible summaries the synthesizer might generate. As a result, the synthesizers struggle to solve the synthesis problem described in Equation (3). To make synthesis tractable, we simplify the problem by first performing bounded synthesis, and subsequently sending candidate summaries that pass bounded synthesis to a general theorem prover to validate. In Equation (3), rather than searching for program summaries that satisfy the VCs for all program states (σ), we search only for a finite set of program states. For example, we limit all list data structures to lengths of up to size 2, and all integers to bit-widths of up to 7 bits. These parameters can be changed by the developers using MetaLift's API. Note that we bound the program states only during the synthesis phase, and during verification we check if the synthesis phase output is true for all program states.

The summaries returned by the synthesizer using the target language and search space defined in Figure 11 and Figure 13, respectively, are shown in Figure 15. The output variable $count$ is synthesized as a series of map and reduce operations. In the mapper phase, λ_m maps each word w in the input list to the key-value pair $(w, 1)$. In the reducer phase, $reduceByKey$ groups all the unique keys in the map output, reduces each group using the add operator and returns a map containing the key-value pairs $(w, frequency)$.

After bounded synthesizer generates the summaries, they are parsed and represented using the constructs in MTL. Note that the generated summaries are not in the concrete syntax of Spark yet but it is expressed in the operators defined by the developers. Developers can easily convert these summaries using simple syntax-driven rules which we describe in Section 3.9. In the next section, we provide details about the verification of the synthesized summaries and invariants, as well as proving that the inferred invariant is indeed sound.

3.8 Verification

Given candidate summaries that are generated by the bounded synthesizer, MetaLift next automatically performs the full verification of the synthesized summaries and invariants. As previously stated, the synthesized summaries and invariants generated during the synthesis phase satisfy the verification conditions only for a finite set of program inputs. For instance, in our running example, the synthesis phase returns a solution that is only verified for lists with lengths up to 2 and, integers with bit-width up to 7. The goal of the verifier is to prove that the program summaries are valid for all the program states (all integers and lists sizes). MetaLift uses *satisfiability modulo theories* (SMT) solvers [10, 42, 7] to solve this verification problem. Formally, the theorem prover checks for the satisfiability of the following problem

$$\forall \sigma \; . \; \neg VC(S,\ ps,\ inv_1,\ inv_2,\ \ldots,\ inv_n,\ \sigma) \tag{4}$$

```
def ps(counts, words):
     counts = reduceByKey(map(words, λm)), λr))
     return counts

def inv(j, count, words):
    return And((j ≥ 0), (j ≤ length(words)),
           (count = reduceByKey(map(take(words, j) λm), λr)))

def λm(v):
    return append(empty, Tuple(v,1)))

def λr(v1, v2):
    return (v1 + v2)
```

Figure 15 Program summaries and invariants generated by the synthesizer for the source code in Figure 2a.

where the definitions inferred during the synthesis phase are substituted for the placeholders for *ps* and *inv*'s in the VCs. Negating the assertion and checking if there exists some program state that satisfies the assertions is standard practice in program verification: if the theorem prover discovers a program state that satisfies the negated assertion then the generated program summaries or the invariants are incorrect. However, if there exists no such state then the inferred summaries and the invariants prove the validity of the verification conditions and thus the semantic equivalence of the summaries to the source code.

As mentioned in Section 3.3, MTL supports various data structures. MetaLift models list and tuples in MTL using the SMT solver's built-in functionality of algebraic data types. Algebraic data type definition requires the user to declare the data type and associate a sort (type) with the declaration. Following that, users declare the accessors and constructors for data retrieval and creation of new data structures, respectively. Associative maps in MTL are modeled using the the SMT solvers built-in theory of arrays. MetaLift generates the verification problem automatically by translating MTL to SMT-Lib format [9]. As the SMT-Lib format does not support higher order functions, we inline them while converting.

Figure 16 shows the simplified version of the verification condition generated during the analysis phase for the source code in Figure 2a. We now show that the invariant described in Figure 15 is necessary and sufficient to prove the verification conditions.

Initial Condition. The initial condition asserts that the loop invariant holds immediately before the loop. Before the loop executes, $j = 0$ and *counts* is an empty map. The invariant expresses *counts* as a series of map and reduce operation applied to the first j elements of the input words list. Since $j = 0$, the map reduce operation will be applied to an empty list, returning an empty map according to the definitions in Figure 11. Hence, the invariant holds in the initial state.

Preservation. The loop preservation VC asserts that if the loop invariant hold at any arbitrary iteration, j, of the loop then it should also hold in the next iteration, $j + 1$, of the loop. The notation $counts[w \mapsto e]$ denotes the assignments statement, i.e., the key w in *counts* gets assigned the value e. This is proved by induction. We first prove that the invariant holds at the initial condition, we assume that the invariant holds at iteration j, i.e., the map reduce operation has already computed the frequency of first j words in the list. We need to show that the invariant holds after one more execution of the loop. This is

true since in the $j+1^{th}$ iteration of the loop, the map reduce operation would compute the frequency of the first $j+1$ words from the input list, thereby incrementing the count of the $j+1^{th}$ word in the output map *counts* by 1.

Termination. Finally, the termination condition states that if the loop terminates, the invariant should imply the post condition. This is true because at the end of the loop $j = size(words)$ and the map reduce operation would have computed the frequencies for all the words in the list which is the same expression as the program summary.

Initial Condition	$Inv(j = 0, counts = \{\}, words)$
Preservation	$Inv(j,\ counts,\ words) \wedge (j\ <\ size(words)) \rightarrow$ $Inv((j\ +\ 1)\ , counts[words[j] \mapsto words[j] + 1]\ , words)$
Termination	$Inv(j,\ counts,\ words) \wedge \neg\ (j\ <\ size(words)) \rightarrow PS(count\ , words)$

Figure 16 Verification conditions for the source code in Figure 2a.

3.8.1 Leveraging Additional Axioms

Verifying loop invariants in general is undecidable, as MTL supports recursion and higher-order functions. To aid in verification, developer can provide additional *axioms* to the theorem prover to prove the validity of loop invariants and program summaries for VCs such as in Figure 16. Figure 11 shows how developers can use MTL to provide operator specific axioms to MetaLift. These axioms are translated from MTL to SMT-Lib format and included in the verification problem by MetaLift. These axioms from our experience are simple properties like associativity, commutativity, and distributivity of the operators in the target language. If developers only need to ensure bounded correctness for the translation, they don't need to define these axioms.

Figure 17 shows the SMT-Lib translation of the map operator axioms described in Figure 11, where map_λ_m is the inlined definition of the map operator. The first axiom states that the map operators is distributive over two input lists. While the second one asserts the inductive property of the map operator. Other properties described in Figure 11 can be similarly translated to SMT-Lib format.

```
;distributive map lemma
(assert (forall ((lst1 (List T)) (lst2 (List T)) )
(= (map_λm (concat lst1 lst2)) (concat (map_λm lst1) (map_λm lst2)))))

;inductive map lemma
(assert (forall ( (lst (List T)) (index U) )
(=> (and (>= index 0) (< index (length lst)))
(= (map_λm (tail lst index)) (concat (λm (get lst index))
         (map_λm (tail lst (+ index 1))))))))
```

Figure 17 Translation of the axioms defined using MTL to SMT-Lib format.

3.9 Code Generation

The program summaries verified by MetaLift are expressed using the high-level operators defined by the developers. The final step is to implement syntax-driven rules to translate these summaries to the concrete syntax of the target DSL. This translation is much easier than the general translation from source language to target DSL API because the developers
1) only need to write rules for the operator they defined in the target language and
2) do not need to worry about semantic equivalence to the source code as MetaLift automatically verifies the program summaries.

In MetaLift, we conduct synthesis in the high-level MTL rather than searching through the concrete syntax of the DSL. This approach is used to make the search process more tractable during synthesis. A DSL may include multiple variations of the same operator, with each having minor differences while they are functionally equivalent. For instance, the Spark DSL contains multiple variations of the map operator (*map, flatMap, mapToPair*). Rather than searching through these different concrete implementations, we conduct the search using a single implementation that captures the different operators' high-level semantics. Once a solution is synthesized in the MTL, converting it to concrete syntax becomes straightforward. The developer can decide which variation to use on basis of the return value of the λ_m function. For example, if the λ_m function returns a key-value pair or list with single key-value pair, the developers can use *mapToPair* from the DSL. If the λ_m function returns a list containing multiple key-value pairs, the developers can then use *flatMapToPair*. Table 2 shows an excerpt of the translation rules for the Spark DSL. Applying the translation rules to the summary generated in Figure 15 would result in `words.mapToPair(v -> (v,1)).reduceByKey((v1,v2) -> v1 + v2)`. Another advantage is that if the same operator can exists in multiple DSL (e.g. convolution in tensor processing libraries), we only need to perform the search once and then code generation rules can translate the summary into concrete syntax of any library. In Figure 18, we show the implementation for some of these rules for the Spark DSL.

Table 2 Translation rules for converting program summaries in MTL to Spark DSL.

Pattern	Translation Rule
$[\![map(I, \lambda_m \rightarrow list(Pairs))]\!]$	l.flatMapToPair($[\![\lambda_m]\!]$)
$[\![map(I, \lambda_m \rightarrow (Pair\ or\ list(Pair)))]\!]$	l.mapToPair($[\![\lambda_m]\!]$)
$[\![map(I, \lambda_m \rightarrow T)]\!]$	l.map($[\![\lambda_m]\!]$)
$[\![reduce(I, \lambda_r)]\!]$	l.reduce($[\![\lambda_r]\!]$)
$[\![reduceByKey(I, \lambda_r)]\!]$	l.reduceByKey($[\![\lambda_r]\!]$)
$[\![\lambda_m(v) \rightarrow e)]\!]$	(v -> $[\![e]\!]$)
$[\![\lambda_r(v1, v2) \rightarrow e)]\!]$	((v1,v2) -> $[\![e]\!]$)
$[\![e_1\ aop\ e_2]\!]$	$[\![e_1]\!]$ *aop* $[\![e_1]\!]$

4 Evaluation

We now describe our experience in building synthesis-driven transpilers using MetaLift. We have implemented the core of MetaLift in Python. We provide developers with a Python API for using the constructs in our MTL for defining the semantics of their DSL operators and search space description. MetaLift uses Rosette [41] as its synthesis back-end engine, and supports Z3 [42] and CVC5 [7] for verification.

In this section, we show that our MTL is general enough to be used to create compilers for different DSLs. We demonstrate this by creating compilers using our framework for two DSLS that target very different applications:

```
def codeGen(verifiedSummaries):
    ps = verifiedSummaries['ps']
    def eval(expr):
        if expr[0] == "reducebykey":
            return "%s.reducebykey(%s)"%(eval(expr[1]), eval(expr[2]))
        elif expr[0] == "map":
            if len(lm) == 1: map_func = "mapToPair"
            else: map_func = "flatMaptoPair"
            return "%s.%s(%s)"%(expr[1],map_func,eval(expr[2]))
        ...
    return eval(ps)
```

Figure 18 Implementation of syntax-driven rules for translating summaries to executable code in target DSL.

1. **Distributed Computing DSL**. In this case study, we build a compiler that translates sequential Java code to Spark DSL [43]. Spark provides users with APIs to perform large-scale data processing efficiently by distributing the computations across multiple clusters.
2. **Hardware DSL**. In this case study, we build a compiler that translates Domino [35], which allows users to implement data-plane algorithms (such as congestion control and load balancing) for network switches to Banzai atoms, which represent atomic operations typically available in network switches. By combining these atoms, users can implement various programmable network switches with Banzai.
3. **Vector Operation DSL**. In this case study, we build a compiler that translates sequential C++ code to vector operations. For loops are generally slower compared to their vectorized operations on large datasets and libraries such as Scipy, Pytorch and Tensorflow provide efficient implementations of these vectorized operators.

These case studies differ in the program structure of the source code, in addition to targeting different domains. The Spark and vector case study contains programs with loops, whereas Domino contains only straight line programs with no looping constructs. Prior work [3, 35] implemented these case studies as two separate specialized compilers. We demonstrate that MetaLift can be used to create all these compilers, and that MetaLift simplifies the process of building DSL compilers that leverage program synthesis.

4.1 Case Study: Spark

MapReduce [15] is a popular programming paradigm which enables users to write compute intensive-applications capable of processing massive datasets by distributing the computations across multiple clusters. Mapreduce programming model decomposes the processing into two primitives *map* and *reduce*. MapReduce first breaks down the dataset into multiple independent chunks and then uses the following three stages to process the data:

1. **map phase**: each node in the cluster receives a small chunk of the data. Each node then locally processes the data by applying the mapper function and produces a set of key-value pairs.
2. **shuffle phase**: all the key-value pairs from different mapper functions are then sorted, grouped together by key and then redistributed to different nodes such that each node receives values belonging to the same key.
3. **reduce phase**: each node then aggregates the values using the reducer function.

Spark [43] is an open-source analytics framework. Spark provides users with highly-efficient implementations of map and reduce operations via its APIs in high-level languages such as Python and Java.

To leverage the optimizations provided by the MapReduce paradigm, Casper [3] and Mold [31] are two compilers that automatically translate legacy code written in Java to sequence of map and reduce operations. Casper used program synthesis to perform the transformations, whereas Mold used a syntax-driven rule based approach. We demonstrate that using MetaLift developers can build the core capabilities of the Casper toolchain by easily defining the semantics of the map and reduce using our MTL.

MetaLift's implementation of Casper. We manually rewrote the Casper benchmarks from Java to C++, as MetaLift's frontend currently do not support Java. We then define the semantics of the map and reduce operations using our MTL, as illustrated in Figure 11. We also provide additional axioms to assist the verifier in proving validity of the synthesized program summaries and loop invariants (Lines 23-39 in Figure 11). As shown in Figure 13, using MTL we provide the description of the search space for the program summaries, invariant, mapper and reducer functions. Casper implemented incremental grammar search to make the search tractable, and a cost-based evaluation model to select from different semantically equivalent program summaries. In our MetaLift implementation, we use incremental search and in Figure 14 we describe how developers can easily modify the driver code to implement such search strategies.

Results. We evaluate our implementation on the benchmarks [1] that were successfully translated by Casper. Our implementation translates **44** benchmarks of the **49** benchmarks that Casper translated. Synthesizer times out on the five benchmarks which our implementation failed to translate. We believe the reason for these failures is that MetaLift uses LLVM to generate verification conditions, whereas Casper implemented a specialized verification condition generator for Java source code directly, resulting in considerably more optimized VCs that synthesizers can easily solve. We use a timeout (synthesis + verification) of 60 minutes for all the benchmarks. The average running time for our implementation of the Casper benchmarks was $\approx$ 3mins. We observed that MetaLift-generated code had the same structure to the ones which Casper synthesized (i.e., same number of map reduce stages and same implementation of the mapper and reducer functions), so we expect the output code to have similar performance as that generated by Casper. In contrast to Casper, which required **25578** lines of code, our implementation requires less than **1000** lines.

4.2 Case Study: Domino

Domino [35] is a domain-specific language for data-plane algorithms that run on programmable line-rate switches. The Domino DSL provides a C-like interface to define a "packet transaction" on a stream of network packets. While the syntax of Domino is a subset of C, there are a number of restrictions on memory allocation, loops, and control flow structure to prevent writing Domino programs which cannot be executed at "line-rate" (the speed with which packets arrive on a programmable switch). A particularly relevant restriction is that Domino is loopless, obviating the need to synthesize loop invariants as required in Spark. Figure 21 contains an example of a packet transaction written in Domino which implements a simple Rate-Control Protocol (RCP) [40] by accumulating the sum of packet round-trip-times (RTTs) for which the RTT is under the maximum allowable value.

```
StateResult atom_template(int state_1, int state_2, int pkt_1, int pkt_2) {
  if (rel_op(Opt(state_1), Mux3(pkt_1, pkt_2, C()))) {
    state_1 = Opt(state_1) + Mux3(pkt_1, pkt_2, C());
  }
  StateResult ret = new StateResult();
  ret.result_state_1 = state_1;
  ret.result_state_2 = state_2;
  return ret;
}
```

Figure 19 An example of a Banzai atom used in the Domino compiler [36].

Domino compiles to Banzai, a machine model for programmable line-rate switches. The Banzai target encodes hardware constraints fundamental to these switches, the most important of which is *atomicity*, thereby guaranteeing that packet operations occur transactionally. Banzai provides an abstraction over programmable switch architectures with the notion of an *atom*, a stateful processing unit that contains atomic operations which can be used to implement data-plane algorithms. An example of a Banzai atom is given in Figure 19.

Compilation to Banzai. The Domino compiler [35] has a three-stage pipeline to compile a feasible Domino program to Banzai. In the first stage, the compiler preprocesses the code by

1) recursively transforming branches into conditional assignments
2) rewriting operations on state variables to occur on temporary packet fields instead
3) converting to static single-assignment (SSA) form and
4) flattening to a three-address code representation.

In the second stage, the Domino compiler decomposes the input code into a sequence of "codelets", a smaller block containing three-address code. To do so, the compiler executes dependency analysis on the input code to form a dependency graph. The compiler splits the input code into strongly-connected component blocks ("codelets"), and forms a meta-directed acyclic graph of these codelets, thereby capturing block-scale dependencies. The compiler then schedules the codelets in topological order to ensure that all dependencies are satisfied.

Finally, the Domino compiler performs code generation by

1) distributing work throughout codelets to ensure that no block takes too long (the "pipeline width" constraint) and
2) using the SKETCH [37] program synthesizer to map each codelet to one of a set of Banzai atoms which are feasible in hardware.

MetaLift implementation of Domino. We demonstrate how MetaLift can be used to significantly simplify the synthesis of Domino benchmarks to Banzai atoms. First, we encode the Banzai target language and grammar as a set of stateless operations, specified in Figure 20. By ensuring our target language and grammar contains only stateless operations, we guarantee atomicity. We then decompose the Domino benchmarks into codelets – which we represent as C++ functions – as in the second stage of Domino compiler. However, we make two key assumptions:

1) we assume that array reads and writes happen in between codelets to ensure our target language and grammar does not need stateful operations

```
def targetLang():
    ## Variable and Constant Getters
    def const(): return Choose(*CONSTANTS) # CONSTANTS is a list of allowed constants
    def var(): return Choose(*VARS) # VARS contains the input to the codelet
    def var_or_const(): return Choose(var(), const())
    def opt(n): return Choose(n, 0)

    ## Banzai Atoms
    def arith_op(a, b): return Choose(a + b, a - b, a * b)
    def rel_op(a, b): return Choose(a == b, a != b, a <= b, a < b, a > b, a >= b)
    def raw(): return opt(var()) + var_or_const()
    def rw(): return var_or_const()
    def mul_acc(): return opt(var()) * var_or_const() + var() + var()

    def pred_raw():
        if rel_op(opt(var()), var_or_const()): return opt(var()) + var_or_const()
        else: return var()

    def if_else_raw():
        if rel_op(opt(var()), var_or_const()): return opt(var()) + var_or_const()
        else: return opt(var()) + var_or_const()

    def sub(): ...

    def nested_ifs(): ...
```

Figure 20 The semantics of the target Banzai atoms defined using MTL.

```
#define MAX_ALLOWABLE_RTT 30
struct Packet { int size_bytes; int rtt; };

/* State variables */
int input_traffic_Bytes = 0;
int sum_rtt_Tr = 0;
int num_pkts_with_rtt = 0;

/* Transaction code */
void func(struct Packet pkt) {
  input_traffic_Bytes += pkt.size_bytes;
  if (pkt.rtt < MAX_ALLOWABLE_RTT) {
    sum_rtt_Tr += pkt.rtt;
    num_pkts_with_rtt += 1;
  }
}
```

```
AddStateRet3(
    # _input_traffic_Bytes
    Add(_input_traffic_Bytes,
        size_bytes),
    # _sum_rtt_Tr
    Add(rtt, _sum_rtt_Tr)
        if Not(Ge(rtt, 30))
        else _sum_rtt_Tr,
    # _num_pkts_with_rtt
    Add(_num_pkts_with_rtt, 1)
        if Not(Ge(rtt, 30))
        else Add(0, _num_pkts_with_rtt
)
```

Figure 21 End-to-end synthesis of Domino DSL to Banzai via MetaLift.

2) and we assume that each codelet has up to three outputs, all stored in temporary packet fields, and that each later codelet in a "pipeline" receives the set of all relevant outputs from prior codelets.

Finally, we compile the decomposed Domino benchmark to LLVM IR using Clang, and then programmatically synthesize each codelet to the target language and grammar.

We show an end-to-end example of synthesizing a Domino benchmark in Figure 21. The MetaLift synthesized summary contains three atoms, two of which are `pred_raw` (see Figure 20 for atom definitions) and one is a stateless arithmetic operation. `AddStateRet3` is not an atom, but rather the language primitive to output three return values (the new state and packet variables) due to how it is represented the C++ Domino benchmark.

Table 3 Comparison of *maximum* atoms per stage used in the synthesized output of the Domino compiler and MetaLift. The maximum atoms per stage for each benchmark are taken from the Domino paper [35].

Benchmark name	MetaLift atoms/stage	Domino atoms/stage
Bloom filter	3	3
Heavy Hitters	3	9
Flowlets	2	2
RCP	2	3
Sampled NetFlow	2	2
HULL	3	1
Adaptive Virtual Queue	3	3
Queueing priority computation	2	2
DNS TTL change tracking	2	3
CONGA	2	2

Results. We successfully synthesized all ten Domino benchmarks with our solution described above, which took a total of **1052** un-minified lines of code. For domino, the evaluation metric stated in the prior work [35] was not performance but rather feasibility, i.e., if we could transpile a program to a banzai atom then these are feasible to run on a programmable switch device. The average compilation time for these 10 benchmarks was $\approx$ 6secs. The Domino compiler, measuring all C++ source files and headers, totals **4036** lines of code, so our solution leveraging MetaLift requires **74%** fewer lines of code. Table 3 shows that the synthesized output maximum atom count per stage from MetaLift is under the maximum number of atoms per stage that the Domino compiler used for all but one of the ten benchmarks. Differences can be partially attributed to different benchmark decompositions (and therefore, different numbers of stages): for example, in the "HULL" benchmark, the Domino compiler output had 7 stages with maximum 1 atom/stage while MetaLift had 5 stages with maximum 3 atoms/stage. The other primary source of variance is that the Domino compiler optimized for surface area and speed at the hardware level, while MetaLift only optimized for the number of atoms. As a result, certain benchmarks like "Heavy Hitters" were synthesized in far fewer atoms by MetaLift, but those atoms were slower at a hardware level. Nonetheless, the success of this case study further demonstrates the capability of MetaLift to decrease the complexity of verified lifting solutions while simultaneously raising the level of abstraction.

4.3 Case Study: Vector Operations

In machine learning worflows, one crucial step is to pre-process datasets, but writing processing pipelines using loops can be computationally expensive due to the large size of the datasets. To improve efficiency, libraries like Pytorch, Tensorflow, and Scipy offer highly optimized vector or matrix operations for performing these operations faster. To take advantage of the optimizations, we build a transpiler with MetaLift to translate loopy array processing programs to vectorized operations. For example, consider the program in Figure 23 that computes the sum of consecutive elements in an array. This sequential program can be implemented using a convolution operation with a kernel of $[1, 1]$ and a stride of 1. For this transpiler, we encode the semantics of the operators such as 1D convolution, element-wise vector (matrix) multiplication and dot-product using MTL. In Figure 23, we show the

definition of the 1D convolution operation in our MTL. Note that if we were to search for an equivalent convolution program using the concrete syntax of tensor libraries we will have to search and verify for each individual library. Instead, by lifting the semantics to our MTL, we need to perform this search only once and then using simple syntax-driven rules we can translate this to any of the tensor libraries such as Tensorflow or Pytorch.

Results. We evaluate our implementation on **5** array processing benchmarks which can represented using the vector operations described above. These benchmarks are a combination of the stencil kernels introduced in prior work [20] and C++ kernels scraped from the web. We use the same 60 minutes timeout as the previous case studies. Since all these are loopy programs, MetaLift synthesizes any additional invariants also required to prove the functional equivalence. We can translate all the benchmarks with an average running time of $\approx$ 2mins. Once we have the summaries synthesized in our MTL, we write code generation rules to translate them to PyTorch, Tensorflow and Scipy. Our implementation requires less than **500** lines of code.

```
vector<int> program(vector<int> data){
    vector<int> result;
    for (int i = 0; i < data.size() - 1; i++)
        result.push_back(data[i] + data[i + 1]);
    return result;
}
```

Figure 22 Sequential C++ array processing program.

```
def conv(data, kernel, stride):
    if length(lst) == 0 then empty
    return prepend(dot_product(data, kernel), conv(tail(data,stride), kernel, stride))
```

Figure 23 Semantics of the convolution operator using MTL.

5 Related Work

Program Synthesis-Based Compilers. Many tools have been developed previously which leverage program synthesis to translate code written in general-purpose programming languages to DSLs while preserving semantics. Examples of such compilers include STNG [20], which converts Fortran stencil computations to the Halide [32] DSL; QBS [14], which translates sequential Java database processing queries to SQL; Casper [3], which converts sequential Java code to map-reduce operations; and Domino [35], a compiler that translates network packet processing algorithms for programmable switches. All of these compilers are fully automated, but they are tailored to a specific DSL and cannot be reused to build a compiler for a new DSL. Implementing these tools required extensive knowledge of program synthesis and verification, making it difficult for developers to leverage the underlying approach of synthesis-based transformation for their own DSLs. MetaLift, on the other hand, uses MTL to provide a very high-level abstraction for developers to specify the semantics of their DSLs and automatically build compilers that can perform semantic preserving transformations. Other prior work [34] leverages program synthesis to perform superoptimization for the x86

instruction set and [23] employs a data-driven approach for verifying peephole optimization in LLVM. These tools perform transformations for low-level languages while MetaLift builds compilers for DSLs.

Syntax-Driven Transpilation. The traditional solution to the translation problem is to create a syntax-driven compiler. These rule-based systems rely on users to create rules that pattern match in order to perform the required translation. An example of one such compiler is [31], which translates sequential Java code to into MapReduce operations. As such, these rule-based systems tend to be difficult to design and are vulnerable to translation errors. Perhaps even more problematic is that such approach is brittle to changes in DSL semantics.

Neural Approaches for Transpiling. Recently, there has been lot of interest in using neural machine translation to perform source-to-source translations. A number of approaches, including supervised [25, 13] and unsupervised [33] learning based techniques have been proposed for translating between general-purpose programming languages. These models do not require any input from developers, but require massive amounts of data for training. For instance, [33] required over 100 million functions from C++, Python and Java to train their model. MetaLift targets DSLs for which such huge amounts of data may not be available always. Furthermore, the transformations performed by these neural models are not verified, which may provide an opportunity to introduce bugs in the code.

6 Conclusion and Future Work

In this paper we described MetaLift, a unified platform for building DSL compilers. MetaLift allows developers to build compilers for their own DSLs by utilizing the synthesis-based program transformation approach that has been the underlying technique for many previous DSL compilers [2, 20, 35]. MetaLift achieves this with its design of a specification language called MTL. Using MTL developers can express the semantics of their target DSL and search space description to guide the synthesis engine. This programmatic approach to describing DSLs allows MetaLift to build compilers for various DSLs. We described our experience in building synthesis-driven transpilers by using MetaLift to build three transpilers targeting very different application domains. We demonstrate that MetaLift significantly reduces the effort required to create specialized implementations of these three compilers. With a unified framework, MetaLift opens up new research directions for automated transpilation, such as leveraging dynamic execution traces to automatically infer loop invariants, and improving synthesis with neuro-symbolic methods, which can learn effectively search strategies from similar programs, as well as oracle-guided synthesis methods [28, 29], which integrate verification with synthesis and extend the expressive power of synthesis queries beyond SMT. MetaLift is modular to easily add any of these optimizations. As a framework, MetaLift has the potential of dramatically lowering the barrier for both research into synthesis driven transpilation and adoption of new specialized high performance hardware and DSLs.

References

1 Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. `https://github.com/uwplse/Casper/tree/master/bin/benchmarks`, Accessed: 2022-03-14.

2 Maaz Bin Safeer Ahmad and Alvin Cheung. Leveraging parallel data processing frameworks with verified lifting. In *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016.*, pages 67–83, 2016.

3 Maaz Bin Safeer Ahmad and Alvin Cheung. Automatically leveraging mapreduce frameworks for data-intensive applications. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1205–1220. ACM, 2018.

4 Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 2006.

5 Takuya Akiba, Kentaro Imajo, Hiroaki Iwami, Yoichi Iwata, Toshiki Kataoka, Naohiro Takahashi, Michal Moskal, and Nikhil Swamy. Calibrating research in program synthesis using 72,000 hours of programmer time, 2013.

6 Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1216–1225, New York, NY, USA, 2012. ACM. doi:10.1145/2228360.2228584.

7 Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength smt solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing.

8 Mike Barnett and Rustan Leino. Weakest-precondition of unstructured programs. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87. ACM Press, September 2005. URL: https://www.microsoft.com/en-us/research/publication/weakest-precondition-of-unstructured-programs/.

9 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

10 Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 33, pages 1267–1329. IOS Press, second edition, 2021.

11 Rastislav Bodík and Barbara Jobstmann. Algorithmic program synthesis: introduction. *STTT*, 15(5-6):397–411, 2013.

12 Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 363–375, 2010.

13 Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 2552–2562, Red Hook, NY, USA, 2018. Curran Associates Inc.

14 Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 3–14, New York, NY, USA, 2013. ACM. doi:10.1145/2491956.2462180.

15 Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. doi:10.1145/1327452.1327492.

16 Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *IEEE Micro*, 28(4):13–27, July 2008. doi:10.1109/MM.2008.57.

17 Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 26-28, 2010, Hagenberg, Austria*, pages 13–24, 2010.

18 Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 343–361, Cham, 2015. Springer International Publishing.

19 C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969. doi:10.1145/363235.363259.

20 Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 711–726, New York, NY, USA, 2016. ACM. doi:10.1145/2908080.2908117.

21 David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators, 2018.

22 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

23 David Menendez and Santosh Nagarakatte. Alive-Infer: Data-driven Precondition Inference for Peephole Optimizations in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 49–63, New York, NY, USA, 2017. ACM. doi:10.1145/3062341.3062372.

24 Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 225–242, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3341301.3359641.

25 Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 651–654, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2491411.2494584.

26 Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=rJ0JwFcex.

27 Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 396–407, New York, NY, USA, 2014. ACM. doi:10.1145/2594291.2594339.

28 Elizabeth Polgreen, Kevin Cheang, Pranav Gaddamadugu, Adwait Godbole, Kevin Laeufer, Shaokai Lin, Yatin A. Manerkar, Federico Mora, and Sanjit A. Seshia. UCLID5: multi-modal formal modeling, verification, and synthesis. In *34th International Conference on Computer Aided Verification (CAV)*, volume 13371 of *Lecture Notes in Computer Science*, pages 538–551. Springer, 2022.

29 Elizabeth Polgreen, Andrew Reynolds, and Sanjit A. Seshia. Satisfiability and synthesis modulo oracles. In *Proceedings of the 23rd International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, January 2022.

30 Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *OOPSLA 2015 Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126, October 2015. URL: https://www.microsoft.com/en-us/research/publication/flashmeta-framework-inductive-program-synthesis/.

31 Cosmin Radoi, Stephen J. Fink, Rodric Rabbah, and Manu Sridharan. Translating imperative code to mapreduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 909–927, New York, NY, USA, 2014. ACM. doi:10.1145/2660193.2660228.

32 Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM. doi:10.1145/2491956.2462176.

33 Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html.

34 Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA – March 16 – 20, 2013*, pages 305–316, 2013.

35 Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 15–28, 2016.

36 Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Domino examples, October 2018. URL: https://github.com/packet-transactions/domino-examples/blob/master/banzai_atoms/pred_raw.sk.

37 Sketch. https://people.csail.mit.edu/asolar/, 2016. Accessed: 2016-05-01.

38 Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2008. AAI3353225.

39 Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, April 2014. doi:10.1145/2584665.

40 C.-H. Tai, J. Zhu, and N. Dukkipati. Making large scale deployment of rcp practical for real networks. In *IEEE INFOCOM 2008 – The 27th Conference on Computer Communications*, pages 2180–2188, 2008. doi:10.1109/INFOCOM.2008.285.

41 Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, New York, NY, USA, 2013. ACM. doi:10.1145/2509578.2509586.

42 The z3 theorem prover. https://github.com/Z3Prover/z3, 2017.

43 Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.