Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 19-001

An Investigation of Composable Language Extensions for Parallel Programming

Travis Carlson, Ciaradh Coomey, Aaron Councilman, Patrick Stephen, Eric Van Wyk

January 17, 2019

An Investigation of Composable Language Extensions for Parallel Programming

Travis Carlson, Ciaradh Coomey, Aaron Councilman, Patrick Stephen, and Eric Van Wyk

Department of Computer Science & Engineering
University of Minnesota, Twin Cites
Minneapolis, MN 55455

carl4980,comme043,counc009,steph680,evw@umn.edu

Abstract

This paper demonstrates how parallel programming language features can be specified as composable language extensions to a general-purpose host programming language. To illustrate the expressiveness of language abstractions defined in this way we have re-implemented, as language extensions, various abstractions previously described in the literature that were implemented as part of traditional monolithic programming languages. These include abstractions for spawning parallel tasks, abstractions implementing bounded buffered channels and lattice-based variables to simplify the communication between parallel tasks, and domain-specific abstractions for simple specification of tensor computations backed by a code-generation technique for the efficient storage and processing of tensor computations.

The general-purpose host language is C, as implemented in the AbleC extensible C compiler framework. This system provides certain guarantees of extension composability that ensure that independently-developed language extensions can be automatically composed by programmers that are not experts in language design or implementation. Thus programmers can freely select the abstractions that match their programming problem, their preferred programming style and level of expertise, and their desired performance requirements. This approach also provides benefits to researchers designing and developing new abstractions for parallel programming. It allows them to focus their efforts on the implementation of their new abstractions and re-use the host language implementation of general purpose features such as arithmetic expressions, control-flow statements, type checking, and other basic compiler infrastructure.

1 Introduction

There is a great deal of diversity in parallel programming that is not always well-served by existing parallel programming languages. Different computational problems are amenable to different styles of parallelism. Even within a single application, the style of parallelism that is most appropriate may vary among sub-problems. Experience and expertise in writing parallel programs differs among programmers, as does desired level of parallel performance. Programmers seeking maximal performance on a scientific computation are willing to put in more effort to improve performance than those seeking just some benefit from the multiple cores in their processor on a program that may run only briefly. Programmers also simply have different personal preferences in how programs are written.

Due to these differences, there is no *right* set of language abstractions and no *right* parallel programming language for every situation. Depending on the problem and the programmer, different abstractions may be preferred. In some cases, task-parallelism as found in Cilk-5 [Frigo et al., 1998] may be preferable to data-parallelism as found in parallel implementations of *map* and *reduce* operations. For communication between parallel tasks one may want buffered channels as found in Go [Google, 2018], or lattice variables (called LVars) [Kuper et al., 2014b] as found in LVish [Kuper et al., 2014a]. In other cases, programmer-specified loop transformations as found in Halide [Ragan-Kelley et al., 2013] that deliver higher performance, but with more programmer effort, may be more suitable. Programmers need to be able to freely choose the language abstractions that they prefer for their tasks at hand.

While programmers can choose different parallel programming languages, this approach provides a rather course-grained degree of choice. Languages such as Cilk-5 are *monolithic* in that they have a fixed set of abstractions limiting programmers to one specific view on parallel programming. One cannot, for example, use lattice variables for communicating between tasks when using Cilk. If a problem has a component amenable to Cilk-style task parallelism and another in which the programmer would like to use Halide to explicitly control the parallel code that is generated, one must use two separate languages, which can be cumbersome.

It is the conjecture of this paper that extensible programming language frameworks and compilers provide a potential solution, as these allow programmers, to varying degrees, to freely choose the collection of language abstractions that best fit their task at hand. In these frameworks, programmers import selected language extensions into the framework or compiler to create a domain- or application-specific language with the desired set of language features. Examples of such frameworks include the JastAdd-based extensible Java compiler [Ekman and Hedin, 2007a], SugarJ [Erdweg et al., 2011], and polyglot [Nystrom et al., 2003], all based on Java. Systems that generate C code include xoc [Cox et al., 2008], xtc [Grimm, 2006], and ableC [Kaminski et al., 2017]. The work presented here uses ableC as it supports expressive language extensions that add both new syntax (notations) for new abstractions and static analysis of those abstractions, and it provides certain guarantees of composability that ensure that a programmer can automatically import independently-developed language extensions.

These extensible language systems also provide considerable benefits to researchers developing new parallel programming language abstractions. By adding new language features to a full-featured language such as C, one need not re-implement the common general-purpose features programmers expect (*e.g.* arithmetic expressions, control-flow statements, types and type checking systems). These features are all provided by the *host* language.

The primary contributions in the paper include:

- the specification of existing parallel programming abstractions as composable language extensions: lattice variables [Kuper et al., 2014b], Go language style tasks and channels [Google, 2018], abstractions from the Tensor Algebra Compiler (Taco) [Kjolstad et al., 2017] for specifying efficient computations over sparse and dense tensors
- an illustration of language extensions building on top of others: our tensor extension builds on an extension based on Halide [Ragan-Kelley et al., 2012] to combine Taco-style tensor statements and Halide-style loop scheduling
- organization of language extensions into categories for (*i*) starting and scheduling parallel tasks, (*ii*) communication between parallel tasks, and (*iii*) domain-specific problems, and an illustration of how programmers can pick and choose extensions from these categories, and
- a demonstration of reasonable parallel performance when combining different abstractions for parallel programming.

In Section 2 we give several motivating examples, followed by background on ableC (Section 3) and extension specifications (Sections 4 and 5). (Note that extension specifications are necessarily rather brief, but the associated artifact (if accepted) contains all source material and a guide to them. A graph-traversal example in which the composition of extensions is illustrated is in Section 6, followed by related work (Section 7) and concluding discussion (Section 8).

2 Motivating examples

In this section we provide a collection of examples to motivate the development and packaging of language abstractions for parallel programming as composable language extensions to a general purpose host language. The first set of examples solve a simple map-reduce style of problem using different combinations of (potentially) independently-developed language extensions. These examples show a few ways in which a programmer can pick the abstractions of their choosing and use the abstractions in different combinations. Here we show language extensions based on Cilk, LVish lattice variables, and task scheduling and message-passing channels as seen in Go. The second example illustrates how a language extension can be dependent on another: we re-implement many of the features of the Tensor Algebra Compiler (Taco) [Kjolstad et al., 2017] and extend it to use Halide-style loop schedules for computations when the tensors are dense. These abstractions are all implemented as language extensions in the ableC extensible language framework and thus the sample programs are essentially C programs with additional language features for parallel programming.

```
1  int map_fold_linear(int *xs, size_t n) {
2    int accum = 0;
3    for (int i=0; i < n; i++) {
4      accum = g(accum, f(xs[i]));
5    }
6    return accum;
7  }
8
9  cilk int fold(int *xs, size_t n) {
10   int res, x, y;
11   if (n > TASK_SIZE) {
12     spawn x = fold(&xs, n/2);
13     spawn y = fold(&xs[n/2], n - n/2);
14     sync;
15     res = g(x, y);
16   } else {
17     res = map_fold_linear(xs, n);
18   }
19   cilk return res;
20 }
21
22 cilk int main (int argc, char **argv) {
23   ... // initialize array ‘arr‘ of size N
24
25   int s;
26   spawn s = fold(arr, N);
27   sync;
28   printf("Result: %d\n", s);
29 }
```

Figure 1: Map-fold solution using the CILK extension.

2.1 Independent language extensions

This section demonstrates how independent language extensions can work together in a single ABLEC program. To illustrate that there is no single best set of abstractions, we present three solutions to a map-fold problem, each using a different set of extensions. Consider the problem of mapping a function `f` over all values in a given integer array and then folding (reducing) the resulting values using a commutative and associative binary function `g`:

```
int f(int i)        { return ...; }
int g(int a, int b) { return ...; }
```

Different performance characteristics of `f` and `g`, different user preferences, and different desires for parallel performance may lead programmers to select different abstractions for specifying their solution to the problem.

Figure 1 shows a solution that uses `spawn` and `sync` constructs from the CILK extension to recursively divide the problem into halves to be computed in parallel. The CILK extension will be presented in more detail in Section 4.1; it is part of the ABLEC ecosystem of extensions and generates code that uses the CILK-5 run-time library [Frigo et al., 1998].

Initially, one task is spawned (line 26) to fold the entire array. If the size of the array is greater than the desired task size (line 11), two recursive tasks are spawned to fold the lower and upper halves of the array; after syncing (line 14), the two resulting values are folded together. When the array passed into the fold function reaches a size that is less than or equal to than the desired task size, the sequential `map_fold_linear` function (line 1) is applied (line 17).

```
1  int leq(int a, int b) {
2    return a <= b;
3  }
4
5  Lvar<int>* accum;
6
7  cilk int task(int *xs) {
8    for (int i=0; i < TASK_SIZE; i++) {
9      put ( f(xs[i]) ) in accum;
10   }
11   cilk return 0;
12 }
13
14 cilk int main(int argc, char **argv) {
15   accum = make_lvar(g, leq);
16   ... // initialize array ‘arr‘ of size NUM_THREADS * TASK_SIZE
17   cilk for (int i=0; i < NUM_THREADS; i++) {
18     task(&arr[i * TASK_SIZE]);
19   }
20   sync;
21   freeze accum;
22   printf("Result: %d\n", get accum);
23 }
```

Figure 2: Map-fold solution using the Cilk and LVars extensions.

Figure 2 shows a solution that uses a composition of the Cilk and LVars extensions. In this solution, the array is split into a fixed number of segments, and a Cilk task is spawned for each segment using a new Cilk for-loop construct (line 17). Each task then iterates over its segment, applying the mapping function `f` to each value and putting each result into the *lattice variable* (LVar) `accum` (line 5). The values in an LVar form a lattice with a least-upper-bound (lub) operator (here `g`) that is used to combine the current value with the value being put into the LVar (as in line 9); LVars are described in detail in Section 4.2. The program initializes `accum` with a lub function `g` and a partial ordering `leq` that may be used to inspect LVars (line 15). In this example, however, we `freeze` the LVar (line 21) to prevent future `put` operations and it is then safe to `get` the value directly (line 22). Putting values into an LVar is a thread safe operation; locks are used that introduce contention between threads. Thus, this solution is likely to be a good choice only if the map function `f` dominates computation time and the fold function `g` is relatively inexpensive. Note that to satisfy the requirements of the least upper bound operation used by LVars, the function `g` must be both associative and commutative.

Figure 3 shows part of a solution that uses a master-worker configuration and two extensions, one for running threads and another for communication via buffered channels. Both extensions use syntax inspired by that in `Go`, but can be used separately. Threads are started (`run` on line 26), each executing the `worker` function (line 12). The master sends new tasks to workers via the `work` channel (line 10), initially these are `MAP` tasks (line 27). Workers read the `work` channel (lines 13 and 18) until a `STOP` task is read (line 14). The details of how the functions `f` and `g` are used to map and reduce the values are elided. Completed task descriptions are sent back to the master on the `done` channel (line 17). The master receives completed tasks and generates more `FOLD` tasks as needed (lines 33–36), though again details are elided in the `get_new_task` function. When a thread returns the final folding task (whose size is equal to `TASK_SIZE`), the master sends a `STOP` task and prints the result. This solution may be a good choice if memory contention for storing results of folding operations needs to be minimized.

Other combinations, such as using Cilk to spawn tasks and Go channels for communication, are also possible, and programmers are free to make these language abstraction choices as they see fit. In Section 6 we present a more significant example and show how programmers select their desired language extensions.

```
1  enum tag { MAP, FOLD, STOP };
2
3  typedef struct {
4    enum tag t;
5    int *addr;
6    int iter_size;
7    int index;
8  } task_def;
9
10 Chan<task_def> work, done;
11
12 void worker() {
13   task <- work;
14   while (task.tag != STOP) {
15     if (task.tag == MAP) { ... } // do map
16     else { ... }                 // do fold
17     done <- task; // send completed task message
18     task <- work; // receive message for new task
19   }
20 }
21
22 int main (int argc, char **argv) {
23   ... // declare and initialize array ‘arr‘
24
25   for (int i=0; i < NUM_THREADS; i++) {
26     run worker();
27     work <- (task_def) {MAP, arr, N, i * TASK_SIZE};
28   }
29   completed <- done;
30
31   while (completed.iter_size > TASK_SIZE) {
32     task_def new_task ;
33     if (get_new_task(completed, &new_task))  {
34       work <- new_task;
35     }
36     completed_task <- done;
37   }
38
39   ... // send STOP task, print result
40 }
```

Figure 3: Map-fold solution using the Go-style task and channels extensions.

```
1  tensor format mat ({dense, dense});
2  indexvar i, j, k;
3
4  int main(int argc, char **argv) {
5    tensor<mat> A = build(tensor<mat>)({M, P});
6    tensor<mat> B = build(tensor<mat>)({P, N});
7    tensor<mat> C = build(tensor<mat>)({M, N});
8    ... // initialize tensors
9    tensor transform {
10     C[i, j] = A[i, k] * B[k, j];
11   } by {
12     order loops i, k, j;
13     parallelize i;
14   }
15   for (unsigned i=0; i<dimenof(C)[0]; i++) {
16     for (unsigned j=0; j<dimenof(C)[1]; j++)
17       printf("\%2.2f ", C[i,j]);
18     printf("\n");
19   }
20 }
```

Figure 4: Dense matrix multiplication using the tensor extension with parallelization by the HALIDE extension.

2.2 Dependently-developed extensions

In addition to allowing language researchers to build extensions without re-implementing the entire host language, composable language extensions can also build on one another by using abstractions provided by an existing extension or even using the features of existing extensions as the target language for the new extension.

Figure 4 shows parallelization of a dense matrix multiplication using tensor algebra abstractions from the tensor extension (see Section 4.3) and loop transformation abstractions from a HALIDE extension from the ABLEC ecosystem. In contrast to previous examples, the extensions here are not independent; rather, the tensor extension uses syntax specifications from a HALIDE extension. It also generates HALIDE code, which is then translated into plain C.

TACO [Kjolstad et al., 2017] is a C++ library that dynamically generates efficient kernels for tensor expressions in which each dimension may be either sparse or dense. Our extension re-implements TACO as a composable language extension, with additional features from the HALIDE extension to allow programmers that want maximal performance to specify how the compiler should generate the final C code.

Figure 4 demonstrates a matrix multiplication using the combined tensor and HALIDE extensions to take advantage of the ease of writing tensor expressions using the tensor extension and the loop scheduling provided by HALIDE. The first seven lines use tensor extension constructs for defining a dense matrix storage format (`mat`) and for declaring matrices `A`, `B`, and `C`. Line 10 is a tensor assignment that performs the multiplication of `A` and `B`. The tensor assignment is used inside a construct in the tensor extension (lines 9 and 11–14) that builds on top of the HALIDE extension. The `tensor transform` block specifies that the contained (dense) tensor expression (line 10) will be transformed by the HALIDE extension to parallelize the outer loop (while `order loops` appears to be a HALIDE extension transformation, it is from the tensor extension and is discussed in more detail in Section 5). To achieve this transformation, the tensor expression produces loops and evaluations using the abstract productions provided by the HALIDE extension, which is then employed to transform this syntax into standard C code. The HALIDE transformations are provided in the `by` block (lines 11–14).

This motivating example shows that researchers building new parallel programming abstractions can build on the work done by others. As opposed to building a new language, language abstraction developers

can re-use the host language and also abstractions provided by other researchers to create new abstractions for specific kinds of use.

3 Background

Here we briefly describe the ableC [Kaminski et al., 2017] extensible C language framework used in this work. ableC extensions are written in the Silver [Van Wyk et al., 2010] attribute grammar specification language, which uses the Copper [Van Wyk and Schwerdfeger, 2007] parser and context-aware scanner generator. The concrete syntax (notation) of an extension is specified by a context-free grammar consisting of productions, nonterminals, and terminals associated with regular expressions, from which Copper generates a scanner and parser. The abstract syntax and static semantics are specified by an attribute grammar (AG) [Knuth, 1968] that associates nonterminals with attributes and productions with attribute-defining equations. From these specifications, Silver produces a C compiler front-end that scans, parses, constructs the abstract syntax tree (AST) and type checks extended C code, and then generates a translation to plain C code that can then be compiled with a traditional C compiler. Type errors are raised on extensions using an `errors` attribute so that errors are reported in terms of the programmer-written code and *not* the generated plain C code. Extensions define their translation to plain C using a Silver mechanism called *forwarding* [Van Wyk et al., 2002], an important technique for extensibility as it allows an attribute to be computed on this translation if the extension does not define an equation for it. This technique allows the automatic composition of extensions that define new language constructs and others that define new host language static analyses.

The distinguishing characteristics of these tools are the support for expressive language extensions and the modular analyses [Schwerdfeger and Van Wyk, 2009, Kaminski and Van Wyk, 2012] that let language extension developers check that certain composability restrictions are met by their extensions. These assurances mean that programmers can freely pick the independently-developed language extensions they want to use and automatically import the abstractions into ableC with no risk that extensions will conflict. This guarantee maintains a distinction between programmers *using* language extensions and extension developers *writing* language extensions: only the latter need any knowledgeable of language design and the underlying tools. One restriction imposed by the modular analyses for concrete syntax [Schwerdfeger and Van Wyk, 2009] is that extension syntax must begin with a *marking* terminal symbol; for example, the `spawn` terminal on line 12 of Figure 1. This constraint, along with other restrictions, prevents all grammatical conflicts when extension grammars are combined, with one caveat discussed in Section 6. The analysis for attribute grammars [Kaminski and Van Wyk, 2012] ensures there are no missing attribute equations in the combined attribute grammar. One requirement is that all extensions use forwarding to specify their translation down to C code. Kaminski et al. [Kaminski et al., 2017] describe these restrictions in more detail.

4 Independent language extensions

This section describes several parallel programming abstractions that are implemented as independent language extensions. We describe each abstraction's implementation and show that these extensions are expressive enough to closely follow the features of similar abstractions that were previously implemented in separate monolithic languages.

The abstractions are organized into categories from which a programmer can select according to their preferences and the characteristics of the problem they are solving. Section 4.1 presents abstractions for starting and scheduling parallel tasks. Section 4.2 presents abstractions for communicating between parallel tasks. Section 4.3 presents domain-specific abstractions for working with tensors.

4.1 Abstractions for task parallelism

Here we describe two ableC extensions for running parallel tasks. The first provides abstractions for running POSIX threads and avoids much of the error-prone boiler-plate code typically needed with threads. The second adds a new parallel looping construct to the Cilk extension to ableC.

```
1  grammar go_tasks;
2  imports ableC;
3
4  marking terminal Run_t 'run' ;
5
6  concrete production run_c
7  s::Stmt_c ::= 'run' f::Id_t '(' args::Exprs_c ')' {
8    s.ast = run (name(f), args.ast);
9  }
10
11 abstract production run
12 s::Stmt ::= f::Name args::Exprs {
13  s.errors = case f.type of
14             | function_type ( in_Types, _ ) ->
15               ... compare in_Types to args types ...
16             | _ -> err (f.pos, "must be a function")
17  s.lifted = ... AST for declaration of wrapper struct and function ...
18  forwards to ... AST for defining arg struct and pthread create call, e.g.:
19     ({ struct arg_struct_for_f  s;
20        s.arg1 = ...; s.arg2 = ...;
21        pthread_create(f_wrapper, s, ...); })
22 }
```

Figure 5: Specification of the Go-tasks extension.

Go-style tasks as POSIX threads: Our Go-tasks extension introduces a thin wrapper for running POSIX threads (pthreads). It allows the programmer to spawn tasks using only a single new keyword (`run`) as in the style of the Go language [Google, 2018], avoiding the need for error-prone boiler-plate code. Type-safety problems of `void` pointers are avoided by performing static analyses on the programmer-written code.

Recall line 26 in Figure 3, which illustrated how tasks are started in the Go-task extension using the `run` construct. To understand what a language developer must do to implement these kinds of language extensions, Figure 5 shows the key components of the Silver specification for the Go-task extension. First the grammar is named and host language specification is imported. Note that in practice grammar names are prefixed with the developer's Internet domain name to prevent name clashes (*e.g.*, `edu:myu:ableC`) but this is not shown here due to space and anonymity limitations. To define the new notations (concrete syntax) of an extension, the developer must define new terminal symbols (`run`) and grammar productions that are used in defining the scanner and parser for the programmer-chosen extended language. The `run` marking terminal (line 4) indicates the beginning of new extension syntax. This syntax is specified by declaring a new concrete syntax production (lines 6–9) with the host language `Stmt_c` nonterminal on the left-hand side and the `run` marking terminal followed by function-call syntax on the right-hand side. Terminal symbols that only match a single string (such as `run` or a parenthesis) can be written inside single quotes. The list of expression arguments is recognized by the host language `Exprs_c` nonterminal and is referred to by the name `args` in the Silver code on line 8 that computes the abstract syntax tree (AST) for this construct. When composed with the host language and other extensions by the programmer, a scanner and parser can be generated for the extended language so that programs such as the one in Figure 3 can be recognized.

Concrete syntax trees are decorated by the `ast` attribute which stores their corresponding AST. Attributes are accessed using the dot notation, as seen in line 8 where the AST for the `run_c` concrete production is constructed using the `run` abstract production (line 11). Nonterminals in the concrete syntax typically have a `_c` suffix to distinguish them from their corresponding nonterminals in the abstract syntax that do not have this suffix. This `abstract` production is not used in constructing the parser. During static analysis of extended programs, the attributes on the AST are computed using equations (lines 13–21). Details are elided but this figure gives an outline of what the extension developer can specify. One important attribute

is `errors`, which propagates a list of error messages up the AST to be reported to the programmer. We use pattern matching (line 13) to check if the type of the function named `f` is a function type. If it is, we extract the types of the inputs (`in_Types`) and check if these match the types of the expressions in `args`; otherwise, we generate an appropriate error message, referring to the position (`f.pos`) of the function name in the programmer-written code, describing the error (line 16).

The PTHREADS library presents an API that requires spawned functions to accept a single `void` pointer argument that is interpreted as a pointer to a `struct` containing the actual function arguments. Thus, each `run` construct generates the declaration of 1) a C `struct` containing a field for each function argument and 2) a wrapper function that casts its single `void` pointer argument to the type of this new struct and calls the original function `f` with the appropriate arguments. This code is sketched in lines 18–21. To accomplish this task, the *lifting* mechanisms of ABLEC are used to raise these declarations to global scope, as sketched in line 17. The extension then uses forwarding (lines 18–21) to generate code that initializes this newly-declared struct and calls `pthread_create` to create a pthread for the newly-declared wrapper function. The SILVER code for this process is not shown, just a comment illustrating the result.

Cilk-style tasks: Here we briefly describe an extension for ABLEC based on CILK [Frigo et al., 1998] and a new addition that adds a parallel loop construct. The specification of the CILK extension is significantly more sophisticated than the GO-task extension described above. The CILK extension defines several marking terminals and concrete syntax productions that use them. These define the constructs to `spawn` new functions, a `sync` statement to wait for the completion of all functions spawned by that thread, and a `cilk` function declaration construct. These were used in the example in Figure 1. The specifications for concrete syntax closely resemble those for the GO-tasks as seen in Figure 5 and are thus not shown here.

The complexity of the CILK extension comes from the translation of these new constructs into plain C code that makes calls to the CILK-5 run-time task scheduling library. Each `cilk` function is translated into two C functions called clones: a fast sequential clone and a slower clone used in the work-stealing process to execute tasks in parallel. Each `spawn` construct is translated into C code using the run-time library. The extension also performs type checking on the extension constructs so that type errors are reported on the code the programmer wrote and not on the C code generated from those constructs.

In the CILK-5 system, these parsing, error checking, and code generation tasks were performed by a Cilk-to-C translator. In addition to the tasks performed by the CILK extension to ABLEC, that translator also had to implement these tasks for the rest of the C language. This endeavor represents a tremendous amount of work that can be avoided when language abstractions are implemented as language extensions instead of as part of a monolithic programming language.

To this extension we have added a new parallel `for` loop construct that lifts the body of the loop into a new `cilk` function and generates a standard `for` loop that uses the `spawn` construct to run multiple instances of that lifted function in parallel. This is used in the graph-processing example in Section 6 on lines 10 and 15 of Figure 7. Since the specification of this new loop uses similar ABLEC and SILVER features as seen in Figure 5, the specification is not shown here.

4.2 Abstractions for communicating between tasks

When tasks run in parallel and communicate through shared memory, there is the potential of non-deterministic behavior. To avoid this, programmers pay careful attention to detecting and preventing conflicting reads and writes. This is often done using low-level locks, whose use can be tedious and error-prone. In Section 2, we introduced two language extensions that provide abstractions that are intended to be easier to correctly use than locks. The channel extension used in Figure 3 introduces a new infix operator, `<-`, but otherwise uses techniques similar to those used in the `run` extension from Section 4.1. Due to space limitations, we do not describe the extension here. Instead, we discuss the implementation of Kuper's LVARS (lattice variables) [Kuper et al., 2014b, Kuper, 2015]. Neither of these extensions *prevents* data-races or deadlock, but the abstractions make writing deterministic programs easier and allow programmers to avoid many of the pitfalls of traditional languages that lead to these problems.

Like Kuper's LVARS [Kuper et al., 2014b] model, our extension allows communication between threads through shared variables whose possible values form a lattice. A lattice as used here has a partial ordering and a least-upper-bound (lub) operator. For example, a lattice whose values are sets may use set inclusion

for the partial ordering and set union as the lub. A lattice here also has a bottom value that is smaller than all other values, w.r.t. the partial ordering, and a top value representing an erroneous state of the variable.

The `put ( value ) in lvar` construct, as seen on line 9 in Figure 2, is used to write new values into an LVar. The construct uses the lub to combine the current value with the new value, which ensures that the lattice variable's value only grows monotonically. A restriction of the modular analysis for concrete syntax prevents adding new terminals to the follow-sets [Aho et al., 1986] of host language nonterminals, and thus the parentheses around the value are needed if the expression is not just a simple variable. This construct does the expected type checking so that helpful error messages are reported on the code the programmer wrote in terms of the `get`, `put` and `freeze` operations and not the generated C code. This C code includes a call to a library template function, written using an ableC templating extension that provides a limited version of C++-style templates. One advantage of the extension syntax is that the type arguments to template functions (also used in the C implementation of `get` and `freeze`) are inferred and generated automatically, lessening the syntactic burden on the programmer.

LVars provide two mechanisms for reading their values. The first is shown on lines 21–22 in Figure 2 in which the LVar is frozen (using `freeze`) to prevent any future `put` operations. After freezing, the value can be read directly using the `get` operation. Any future `put` operation generates a run-time error. A second mechanism (not shown due to space constraints) uses a blocking `get` operation which waits for the value of the LVar to move above some threshold in the lattice of the LVar. The threshold is defined as a set of *activation set* values in which the lub of any values in different activation sets is the top erroneous value of the lattice. A sequence of `put` operations may result in the LVar value being at or above (w.r.t. the partial ordering) the values in just one activation set. This activation set is returned by `get`. These restrictions ensure that the order in which different tasks read from or write to an LVar does not affect the final result of that LVar. This is an important safety guarantee of this programming abstraction.

LVars provide a convenient abstraction for writing deterministic parallel programs. While the implementation in LVish [Kuper and Newton, 2013] ensures the program is deterministic, our implementation does not. In the same spirit as Cilk, the extension makes it easier for programmers to write deterministic programs by avoiding more error-prone low-level constructs. A more complete description of LVars and the utility in parallel programming can be found in Kuper's original work [Kuper and Newton, 2013, Kuper et al., 2014b, Kuper et al., 2014a, Kuper, 2015].

4.3 Abstractions for specific domains, tensors

This section describes a new language extension for solving domain-specific problems using multi-dimensional arrays, or tensors. Tensor algebra is an increasingly important tool in fields such as data analysis, scientific modeling, and machine learning. Of particular interest are approaches for efficiently storing dense and sparse tensors and performing computations over them. The tensor extension discussed here is a re-implementation of the Tensor Algebra Compiler (Taco) [Kjolstad et al., 2017] as a composable ableC language extension with some additional abstractions.

Historically, tensor algebra systems consisted of a set of hand-written kernels designed to compute specific tensor computations given specific storage formats, where a format indicates if the data of each dimension is sparse or dense. Taco is a C++ library that dynamically generates efficient kernels for programmer-written tensor expressions with arbitrary formats. Kjolstad et al. [Kjolstad et al., 2017] demonstrated that the code generated by their system is capable of performance that is competitive with hand-written tensor algebra kernels, while providing much greater flexibility to programmers since their programs are not limited to a predetermined set of kernels. The Taco code-generation process uses only statically available information such as tensor sparsity/density formats, not dynamic information such as values stored in tensors; thus, we can generate the same code statically.

The implementation of declarations of tensors formats and tensors (as seen in lines 1–2 and 5–7 in Figure 4) use techniques already described and thus we focus here on the compilation of tensor statements and a new looping construct in the extension that is not part of the Taco system.

The process of compiling a tensor statement such as `C[i,j] = A[i,k] * B[k,j]` relies first on the ableC capability for extensions to overload operators such as array access, multiplication, and assignment to transform (via forwarding) the original AST constructed of host-language overloadable productions into an AST constructed with productions from the extensions that are specific to tensor implementations of the

operators. Second, it relies on the re-implementation of the sophisticated Taco code-generation algorithm in Silver.

Operator Overloading in ableC: No new concrete syntax is specified in the tensor extension to parse tensor statements like the one above; the host-language syntax for array access, multiplication, and assignment are used. These generate an AST with hooks that extensions can use to specify how these overloaded ASTs are transformed, via forwarding, into ASTs constructed using productions from a language extension that implement these operations for a new type.

The overloadable production for array access checks the type of the expression being accessed; in `C[i,j]` on both lines 10 and 17 in Figure 4, this is the tensor type. Using forwarding, this AST is transformed into an AST constructed by the tensor access production defined in the tensor extension. This production checks the types of the index expressions. If they are all integers (as on line 17) the access transforms into an expression for extracting a `double` value from a tensor. If any index expression has type `indexvar` (as in all accesses on line 10) then the expression transforms into an expression of type `tensor_accessor`. Overloading of multiplication and assignment are similar. For multiplication, if the operands have type `double`, the expression transforms into a standard C host language multiplication, but if either operand is a `tensor_accessor`, the expression transforms into a tensor-accessor multiplication. This process has the effect, via forwarding transformations, of generating an AST that is similar in structure to the internal ASTs of Taco, constructed of productions that implement the tensor-statement code generation algorithm.

Code-generation for tensor statements: The extension productions implement a code-generation algorithm for tensor statements that is nearly the same as in the Taco system. A slight difference in our algorithm, however, resolves a known bug (see `https://github.com/tensor-compiler/taco/issues/121`) in their system for tensor multiplication with both sparse and dense dimensions. The important point is that sophisticated code-generation can be carried out by language extensions in this approach and is demonstrated by our tensor extension.

Iterating through sparse tensors: Our extension also introduces a new loop construct for iterating over the values stored in tensors, especially useful for tensors with a sparse final dimension. This construct is used in Section 6, where sparse tensors are used as a convenient and efficient way to represent edges in a large graph:

```
foreach (double v :  edges[n, i]) { ..  i ..  }
```

Here `n` is an `int` and `i` is an `indexvar`. The code generated from this construct iterates though the sparse vector of edges leaving node `n` and declares an `int` variable `i` inside the loop body that shadows the outer `indexvar` with the same name so that the index (in this case the number of the node connected to `n`) can be used. The implementation of this construct uses the same techniques as seen in Section 4.1 and is thus not discussed here.

5 Dependent extensions: TACO + Halide

As in the Taco system [Kjolstad et al., 2017], our extension can generate parallel code using OpenMP directives when the dimensions of the output tensor are dense. But for dense tensors there are existing systems that can generate more efficient code. For example, the Halide system [Ragan-Kelley et al., 2012] allows programmers to write loop schedules that specify how loops are to be transformed to generate efficient code. These schedules can be used to reorder, unroll, split, tile, parallelize, and vectorize loops. Separating the computation (written as standard loops over dense tensors) from the schedules frees programmers to more easily experiment with different transformations without having to write the transformed code themselves, a time-consuming and error-prone process. In this section we show how an extension can build on top of another to provide interesting combinations of parallel programming abstractions, specifically we show how our tensor extension can build on an existing extension providing Halide-like language features so that the convenient tensor statement notation (line 10 in Figure 4) can be combined with Halide-style loop transformations (lines 11–14) to generate efficient dense-tensor code. We introduce `order loops` which is

```
1  grammar tensors;
2  imports ableC, halide;
3
4  concrete production halideTensorComp_c
5  s::Stmt_c ::= 'tensor' 'transform' '{'
6    tn::Expr_c '[' ix::Expr_c ']' '=' v::Expr_c
7    '}' 'by' '{' ts::Transformations_c '}'
8  {
9    s.ast = halideTensorComp(tn.ast, ix.ast,
10   v.ast, ts.ast);
11 }
12
13 abstract production halideTensorComp
14 s::Stmt ::= tn::Expr ix::Expr v::Expr ts::Transformations {
15   s.errors = ...
16   forwards to ... AST using halide productions
17 }
```

Figure 6: A portion of the SILVER specification of the tensor extension which builds on top of the HALIDE extension.

not part of the HALIDE extension; instead it can be used as the first transformation to specify the order of the loops for the tensor statement to generate, which directly affects the code generation algorithm.

Figure 6 shows part of the tensor extension specification that imports both the ABLEC specification and also the HALIDE extension specification (line 2) to provide the combined constructs demonstrated in Figure 4. The HALIDE extension provides the nonterminal `Transformations_c` used on line 7 in the concrete syntax of a `tensor transform` construct; it derives the various kinds of loop schedules one can write in Halide, as seen on lines 12–13 in Figure 4. This production uses host language expression nonterminal and terminal symbols for the syntax of the tensor statement. The ASTs of these parse trees are used to construct the AST of the Halide-tensor computation using the abstract production `halideTensorComp` (line 13). This production uses the host language `Stmt` and `Expr` nonterminals and the HALIDE nonterminal `Transformations`.

The (elided) equation for `errors` checks for errors on the tensor statement and transformations. This production uses forwarding to transform the `tensor transform` construct into a HALIDE-style `transform` construct with explicit loops using abstract productions from the HALIDE extension. This is a relatively straightforward process of processing the tensor statement components and is thus elided here.

Frameworks in which language extensions can build on top of other extensions allow for interesting combinations of features and can save language abstraction developers a considerable amount of time and effort.

6 Composition of independent extensions

Here we demonstrate how the independent extensions of Section 4 can work together. The example uses the tensor, CILK, and LVARS extensions, and shows that this particular set of abstractions provides reasonable performance benefits for a graph-processing problem even though the extensions operate independently. Although some programmers may prefer this particular combination of abstractions, it is not the best fit for all problems, and not all programmers will have the same preferences; other programmers may choose other combinations of abstractions for solving this problem.

To generate a custom ABLEC compiler, the programmer specifies the name of the extended compiler to be generated, the host language to be extended (ABLEC), and the list of extensions to be used. The specification for this section's example is written as:

```
construct MyAbleC as ableC translator
  using tensors; cilk; lvars;
```

```
1  cilk int main(int argc, char **argv) {
2    Lvar<int> *accum = make_lvar(g, leq);
3    tensor format ds ({dense, sparse});
4    tensor<ds> edges;
5    ... // initialize edges
6    IntSet *new = singleton_set(START);
7    IntSet *seen = empty_set();
8
9    while (!is_empty(new)) {
10     cilk for (IntSetItr *itr = begin(new); !at_end(itr); move_next(itr)) {
11         put ( f(get_next(itr)) ) in accum;
12     }
13     seen = union(seen, new);
14     Lvar<IntSet*> *nbrs = make_lvar(union, subset);
15     cilk for (IntSetItr *itr = begin(new); !at_end(itr); move_next(itr)) {
16       indexvar i;
17       int n = get_next(itr);
18       foreach (double v : edges[n, i]) {
19         put i in nbrs;
20       }
21     }
22     sync;
23     freeze nbrs;
24     new = set_minus(get nbrs, seen);
25   }
26   freeze accum;
27   printf("\%d\n", get accum);
28 }
```

Figure 7: Map-fold over a connected component of a graph, using the tensor, Cilk, and LVars extensions.

It is possible that the marking terminals of different extensions will overlap and this kind of lexical ambiguity must be resolved by the programmer. This disambiguation is quite straightforward and amounts to choosing a prefix that the programmer writes just before each conflicting marking terminal in their program, as described more fully in work on ableC by Kaminski et al. [Kaminski et al., 2017].

Figure 7 shows an extended ableC program using the tensor, Cilk, and LVars extensions to solve a graph processing problem. A function `f` is mapped over all nodes connected to an initial node (`START`) and then a function `g` is used to fold up the map results. This is similar to earlier examples in that multiple calls to `f` and `g` execute in parallel. The lattice variable `accum` (line 2) stores the final result and performs the fold function `g` when values are `put` into it. The tensor extension enables the programmer to naturally represent the edges of the graph by an adjacency matrix `edges` (line 4) in which rows and columns are indexed by node identifiers and non-zero values signify the existence of edges. Any position of the first dimension may be accessed to look up the neighbors of a node, while the second dimension will only be used for iteration over its non-zero elements; thus, the dimensions are specified to be dense and sparse, respectively (line 3). Traversal of a connected component of the graph is performed in stages; each iteration of the main loop (line 9) computes over one level of newly-found neighboring nodes, and repeats until no new neighbors are found. The Cilk parallel loop construct (line 10) is used to spawn, for each new node, a task executing the loop body that applies the map function `f` and folds the result into the LVar `accum` (line 11). Note that there is no immediate `sync` and thus the code proceeds in parallel to find the next level of neighboring nodes, again using the Cilk parallel loop construct (line 15) to spawn a task for each new node. These new tasks use the tensor `foreach` loop to iterate over all neighbors (line 18) and `put` the neighbors into the LVar `nbrs` (line 19), which is configured to use set union as its least upper bound and the subset function as its partial order (line 14). The LVar must be frozen (line 23) prior to reading the set of neighbors (line 24).

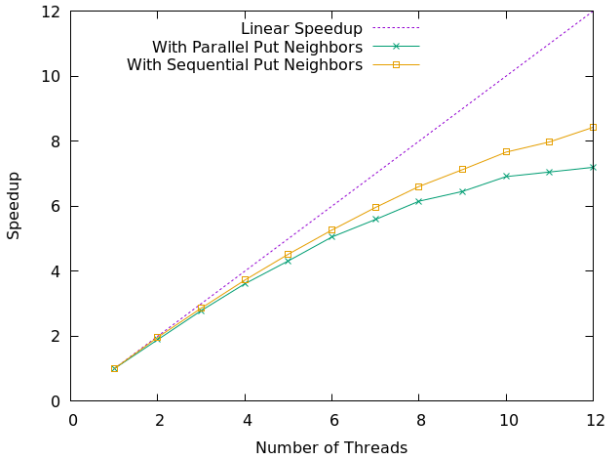

Figure 8: Speedup of graph traversal example of Figure 7.

We tested the Figure 7 example on a two-processor, 12-core 2.00GHz Intel E5-2620 machine with 16GB of main memory, using a graph dataset with edges representing Facebook friend relations [Leskovec and Sosič, 2016]. Again, we do not claim that these abstractions are the best choice for this problem, only that the programmer has considerable freedom in choosing their abstractions and that the resulting performance is not hurt by the composition. In fact, our experiments indicate that the parallelization of the loop on line 15 actually negatively affected performance. Figure 8 shows the resulting speedup as the number of threads was increased up to the number of physical cores, both for the code as shown in Figure 7 and for a modification in which the Cilk for loop on line 15 was replaced with a sequential loop. We observe nearly-linear speedup for a small number of threads and then moderate improvements. We conclude that reasonable performance gains are possible using parallel programming abstractions that are implemented as composable language extensions, though the performance of these extensions could be improved.

7 Related work

There are many ways for parallel programming abstractions to be developed and deployed to programmers. The most common is traditional libraries; these do provide the composability we seek in that the programmer can pick and choose various independently-developed abstractions as they see fit. For example, the Microsoft Task Parallel Library (TPL) provides abstractions for parallel programming such as the following parallel for-loop from the TPL documentation [Corp., 2017]:

```
Parallel.For (0, N, i => ... computation using i
                      in range from 0 to  N-1 ...)
```

This runs a computation (a lambda-expression) for each value in the range (here from `0` to `N - 1`) concurrently. While functional, libraries cannot provide the same syntactic richness and static analysis that language extensions can. The Taco [Kjolstad et al., 2017] library uses overloading to provide quite nice syntax for tensor computations, but some abstractions, such as those in Cilk cannot be effectively implemented this way.

New programming languages also provide new abstractions. The so-called high productivity languages of X10 [Charles et al., 2005], Chapel [Cray, 2007], and Fortress [Sun-Microsystems, 2007] provide a rich collection of parallel programming abstractions that aim to target a wide range of problems and programming styles. Other language such as Cilk-5 [Frigo et al., 1998], Yada [Gay et al., 2011], and Single Assignment C [Grelck and Scholz, 2006], and Deterministic Parallel Java [Bocchino et al., 2009] all provide useful features. But each of these monolithic languages provides a specific vision of parallel programming and they do not support extension with domain-specific features such as the tensor expressions and code generation as seen in Section 4.3.

Besides ableC [Kaminski et al., 2017], other extensible frameworks could have been chosen. JastAdd [Ekman and Hedin, 2007b] and Kiama [Sloane, 2011] are attribute grammar systems, like our system Silver [Van Wyk et al., 2010]. SugarJ [Erdweg et al., 2011] and polyglot [Nystrom et al., 2003] also support the specification of extensible languages. As do xoc [Cox et al., 2008] and xtc [Grimm, 2006], which like ableC, are extensible specifications of C. But all, even telescoping languages [Kennedy, 2000] (but not ableC) lack modular analyses of Silver [Kaminski and Van Wyk, 2012] and our scanner and parser generator Copper [Schwerdfeger and Van Wyk, 2009] that guarantee language extensions composability. One exception is typed syntax macros in Wyvern [Omar et al., 2014] and VerseML [Omar, 2017] which provide syntactic composition guarantees like the modular analysis in Copper. These are macro systems, however, and thus do not support the static analysis of new language constructs for error checking as we saw in the task `run` extension in Figure 5.

8 Discussion and future work

We have developed several parallel programming abstractions as composable language extensions and shown that they work well together. This approach gives the programmer the freedom to pick the abstractions from the different categories that best suit their preferences, experience, and task at hand when writing parallel programs. The ableC framework allows expressive abstractions to be specified and provides the benefit of guarantees of language extension composability - something not found in other approaches. This approach also provides benefits for language researchers; when their new ideas can be implemented as language extensions they need not implement all the general-purpose features of the host language and can build on top of other extensions to provide interesting combinations of features, as seen in the tensor and Halide combination.

We continue to develop the extensions described here and improve their performance. Our LVars extension does not provide the determinism guarantee found in LVish. A possible solution is a static analysis that checks that all parallel tasks only communicate through LVars. A challenge to this is that different extensions introduce different task-creating abstractions and thus the analysis would need to take place on the C code these extensions translate to. The Go-style `run` extension is implemented using pthreads instead of the more efficient user-space mechanisms used in Go. We are investigating what is required to implement something similar as a language extension. Another area of investigation is in the tensor extension to allow Halide-style loop schedules even when some of the tensor dimensions are sparse.

Another category to explore are abstractions for data parallelism. Parallel map and reduce array operations or constructs that implement Google's MapReduce [Dean and Ghemawat, 2004] systems as a language extension are potential opportunities.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1628929. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF.

References

[Aho et al., 1986] Aho, A., Sethi, R., and Ullman, J. (1986). *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

[Bocchino et al., 2009] Bocchino, R. L., Adve, V. S., Adve, S. V., and Snir, M. (2009). Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 4–4. USENIX Association.

[Charles et al., 2005] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. (2005). X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Notices*, 40(10):519–538.

[Corp., 2017] Corp., M. (2017). Task-based asynchronous programming. `https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming`.

[Cox et al., 2008] Cox, R., Bergany, T., Clements, A., Kaashoek, F., and Kohlery, E. (2008). Xoc, an extension-oriented compiler for systems programming. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 244–254.

[Cray, 2007] Cray (2007). Chapel language specification 0.750. http://chapel.cs.washington.edu/spec-0.750.pdf.

[Dean and Ghemawat, 2004] Dean, J. and Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. In *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*.

[Ekman and Hedin, 2007a] Ekman, T. and Hedin, G. (2007a). The JastAdd extensible Java compiler. In *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA)*, pages 1–18. ACM.

[Ekman and Hedin, 2007b] Ekman, T. and Hedin, G. (2007b). The JastAdd system - modular extensible compiler construction. *Science of Computer Programming*, 69:14–26.

[Erdweg et al., 2011] Erdweg, S., Rendel, T., Kastner, C., and Ostermann, K. (2011). SugarJ: Library-based syntactic language extensibility. In *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA)*, pages 391–406. ACM.

[Frigo et al., 1998] Frigo, M., Leiserson, C. E., and Randall, K. H. (1998). The implementation of the Cilk-5 multithreaded language. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 212–223, New York, NY, USA. ACM.

[Gay et al., 2011] Gay, D., Galenson, J., Naik, M., and Yelick, K. (2011). Yada: Straightforward parallel programming. *Parallel Computing*, 37(9):592–609.

[Google, 2018] Google (2018). The go programming language. `https://golang.org`.

[Grelck and Scholz, 2006] Grelck, C. and Scholz, S.-B. (2006). SAC: a functional array language for efficient multi-threaded execution. *International Journal of Parallel Programing*, 34(4):383–427.

[Grimm, 2006] Grimm, R. (2006). Better extensibility through modular syntax. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 38–51. ACM.

[Kaminski et al., 2017] Kaminski, T., Kramer, L., Carlson, T., and Van Wyk, E. (2017). Reliable and automatic composition of language extensions to C: The ableC extensible language framework. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):98:1–98:29.

[Kaminski and Van Wyk, 2012] Kaminski, T. and Van Wyk, E. (2012). Modular well-definedness analysis for attribute grammars. In *Proceedings of the International Conference on Software Language Engineering (SLE)*, volume 7745 of *LNCS*, pages 352–371, Berlin, Germany. Springer.

[Kennedy, 2000] Kennedy, K. (2000). Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *Proc. of the 14th International Parallel and Distributed Processing Symposium (IPDPS00)*.

[Kjolstad et al., 2017] Kjolstad, F., Kamil, S., Chou, S., Lugato, D., and Amarasinghe, S. (2017). The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77:1–77:29.

[Knuth, 1968] Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145. Corrections in **5**(1971) pp. 95–96.

[Kuper, 2015] Kuper, L. (2015). *Lattice-Based Data Structures for Deterministic Parallel and Distributed Programming*. PhD thesis, Indiana University.

[Kuper and Newton, 2013] Kuper, L. and Newton, R. R. (2013). LVars: lattice-based data structures for deterministic parallelism. In *Proceedings of ACM SIGPLAN Workshop onf Function High Peformance Computing, FHPC 2013*.

[Kuper et al., 2014a] Kuper, L., Todd, A., Tobin-Hochstadt, S., and Newton, R. R. (2014a). Taming the parallel effect zoo: Extensible deterministic parallelism with lvish. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 2–14, New York, NY, USA. ACM.

[Kuper et al., 2014b] Kuper, L., Turon, A., Krishnaswami, N. R., and Newton, R. R. (2014b). Freeze after writing: Quasi-deterministic parallel programming with lvars. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 257–270, New York, NY, USA. ACM.

[Leskovec and Sosič, 2016] Leskovec, J. and Sosič, R. (2016). Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1. Data used available at `http://snap.stanford.edu/data/ego-Facebook.html`.

[Nystrom et al., 2003] Nystrom, N., Clarkson, M. R., and Myer, A. C. (2003). Polyglot: An extensible compiler framework for Java. In *Proceedings of the Conference on Compiler Construction (CC)*, volume 2622 of *LNCS*, pages 138–152. Springer.

[Omar, 2017] Omar, C. (2017). *Reasonably Programmable Syntax*. PhD thesis, Carnegie Mellon University, Pittsburgh, USA.

[Omar et al., 2014] Omar, C., Kurilova, D., Nistor, L., Chung, B., Potanin, A., and Aldrich, J. (2014). Safely composable type-specific languages. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP)*, pages 105–130. Springer.

[Ragan-Kelley et al., 2012] Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S., and Durand, F. (2012). Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4):32:1–32:12.

[Ragan-Kelley et al., 2013] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. (2013). Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 519–530, New York, NY, USA. ACM.

[Schwerdfeger and Van Wyk, 2009] Schwerdfeger, A. and Van Wyk, E. (2009). Verifiable composition of deterministic grammars. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 199–210, New York, NY, USA. ACM.

[Sloane, 2011] Sloane, A. M. (2011). Lightweight language processing in Kiama. In *Proceedings of the 3rd summer school on Generative and Transformational Techniques in Software Engineering III (GTTSE '09)*, volume 6491 of *LNCS*, pages 408–425. Springer.

[Sun-Microsystems, 2007] Sun-Microsystems (2007). The Fortress Language Specification, version 1.0. http://research.sun.com/projects/plrg/fortress.pdf.

[Van Wyk et al., 2010] Van Wyk, E., Bodin, D., Gao, J., and Krishnan, L. (2010). Silver: an extensible attribute grammar system. *Science of Computer Programming*, 75(1–2):39–54.

[Van Wyk et al., 2002] Van Wyk, E., de Moor, O., Backhouse, K., and Kwiatkowski, P. (2002). Forwarding in attribute grammars for modular language design. In *Proceedings of the Conference on Compiler Construction (CC)*, volume 2304 of *LNCS*, pages 128–142. Springer-Verlag.

[Van Wyk and Schwerdfeger, 2007] Van Wyk, E. and Schwerdfeger, A. (2007). Context-aware scanning for parsing extensible languages. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 63–72, New York, NY, USA. ACM.