Building parallel programming language constructs in the AbleC extensible C compiler framework

A PPoPP Tutorial

Travis Carlson and Eric Van Wyk
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN, USA
travis.carlson@cs.umn.edu,evw@umn.edu

CCS Concepts • **Software and its engineering → Extensible languages**; **Translator writing systems and compiler generators**; • **Computing methodologies → Parallel programming languages**.

1 Introduction

In this tutorial participants learn how to build their own parallel programming language features by developing them as language extensions in the ableC [4] extensible C compiler framework. By implementing new parallel programming abstractions as language extensions one can build on an existing host language and thus avoid re-implementing common language features such as the type checking and code generation of arithmetic expressions and control flow statements. Using ableC, one can build expressive language features that fit seamlessly into the C11 host language.

A distinguishing characteristic of ableC is that the underlying tools can ensure that independently-developed extensions can all be safely and automatically composed with the host language. This allows programmers (that are not knowledgeable about the techniques used to implement language extensions) to pick and choose the collections of language extensions that they would like to use with the assurance that the tools will generate a working compiler for their custom language. Certain composability constraints must be satisfied by the language extensions to ensure this composability guarantee, but they still allow for expressive language features to be specified.

In Section 2 we discuss a few sample ableC language extensions; the tutorial focuses on how to build language extensions of this type in ableC. These extensions add new concrete syntax, static analyses, overloading of existing C operators, and code transformation and generation techniques to the host C language. Section 3 discusses the ableC framework and how language extensions are implemented. Section 4 describes the tutorial goals and gives links to ableC, sample extensions, and the online resources for the tutorial.

PPoPP '19, February 16–20, 2019, Washington, DC, USA
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6225-2/19/02.
https://doi.org/10.1145/3293883.3302574

2 Motivation and Examples

The significant amount of diversity in parallel programming is not always well-served by existing parallel programming languages. Different computational problems are amenable to different styles of parallelism and applications may be composed of sub-problems that are more amenable to different styles of parallelism. Furthermore, experience and expertise in writing parallel programs differs among programmers, as does desired level of parallel performance. Programmers seeking maximal performance on scientific computations may be willing to put in more effort to improve performance than those just seeking 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 their programs are written.

With all of these differences, there is no *right* set of language abstractions and no *right* parallel programming language for every situation. Depending on the problem, the situation, and the programmer, different abstractions may be preferred. In some cases, task-parallelism as found in Cilk-5 [1] 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 [2], or lattice variables (called LVars) [8] as found in LVish [7]. In other cases, the programmer-specified loop transformations that are found in Halide [9] and deliver higher performance, but with more programmer effort, may be more suitable. Programmers should 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 somewhat cumbersome.

Although a main thrust of our research has been to develop the tools and techniques that support the composition of independently-developed language extensions, we have done so while still allowing quite expressive language features to be developed as composable language extensions. If language researchers could only introduce quite limited forms of new syntax, static analysis, or code generation in language extensions then the cost of composability would be too high. We have thus implemented a number of extensions for parallel programming, including features from the parallel languages and techniques described above. The examples described below demonstrate the kinds of composable language features that can be developed in ableC. The tutorial guides participants through the process of developing language extensions similar in complexity to these.

Cilk, algebraic data types, and regular expressions: As a first example, consider the extended C program in Figure 1 that comes from our previous work on ableC [4]. This program uses three different extensions that provide (*i*) Cilk-5 [1] style constructs for specifying parallel computations, (*ii*) inductive/algebraic data types as found in many functional languages, and (*iii*) regular expression literals and matching operators. This program runs in parallel to count the number of strings in a binary tree that match a regular expression. The first token in each extension-introduced construct is underlined in the figures to help identify the extensions.

On lines 3-7 is a declaration of a `Tree` datatype with `Fork` and `Leaf` constructors; a fork node contains pointers to two sub-trees and a string while a leaf node contains just a string. In the `count_matches` function we see, on line 10, a use of the pattern-matching `match` statement that matches the tree `t` against one of three patterns to determine what branch of the `match` to execute. These three patterns, on line 11, 23, and 25, specify the cases in which the tree `t` is a fork node, a leaf node whose string consists of positive digits, or a leaf node that does not match the pattern above. In the fork case on line 11 the components of the tree are identified as `t1`, `t2`, and `str` and referenced in the associated code.

This program also uses constructs from an extension providing Cilk-5 style task-based parallelism. Lines 13 and 14 spawn new parallel tasks to count the number of matches in the subtress `t1` and `t2` while the main thread checks the string `str` on the current node. Line 19 is a Cilk `sync` command that waits for the spawned tasks to complete. These extension constructs are translated to plain C code that uses the original MIT Cilk-5 runtime as the implementation the work-stealing techniques [1].

The third extension adds regular expression literals and an new infix matching operator, =~, to check if a string matches a regular expression, as seen on line 15. This extension also extends the algebraic data type extension to add an new form of pattern; this regular expression pattern matches when, as expected, a string matches the regular expression. This is seen on line 23.

```
1.  #include <stdio.h>
2.
3.  typedef datatype Tree  Tree;
4.  datatype Tree {
5.    Fork (Tree*, Tree*, const char*);
6.    Leaf (const char*);
7.  };
8.
9.  cilk int count_matches (Tree *t) {
10.   match (t) {
11.     Fork(t1,t2,str) -> {
12.       int res_t1, res_t2, res_str;
13.       spawn res_t1 = count_matches(t1);
14.       spawn res_t2 = count_matches(t2);
15.       if ( str =~ /[1-9]*/ )
16.         res_str = 1;
17.       else
18.         res_str = 0;
19.       sync;
20.       cilk return res_t1 + res_t2 +
21.                   res_str;
22.     }
23.     Leaf(/[1-9]*/) -> {
24.       cilk return 1; }
25.     _ -> { cilk return 0; }
26.   } ;
27. }
28.
29. cilk int main (int argc, char **argv) {
30.   int count;
31.   Tree *t = initialize_tree();
32.   spawn count = count_matches(t);
33.   sync;
34.   printf ("number of matches = %d\n", count);
35.   cilk return 0;
36. }
```

Figure 1. An example ableC program using extensions that provide Cilk-style parallel programming abstractions, algebraic data types with extensible pattern matching, and regular expressions.

This extension demonstrates a few different capabilities of ableC: notably the mechanisms for defining new concrete syntax, static analysis of them (*i.e.* type checking), and translation of the constructs down to plain C code.

Halide-style loop transformations: When writing code where performance is a critical aspect, there are many optimizations and transformations that a compiler can conceivably apply to generate highly efficient code. However, compilers do not have the same insight into the problem as the developer and do not always apply the set of transformations that result in the most efficient code. Halide [9] is a system in which developers can write performance critical code in a clear and easy to read manner and then explicitly define the loop transformations that they want to be applied.

Figure 2 demonstrates a Halide-inspired ableC extension used in a function to multiply matrices. This example comes from the supplementary material of the ableC paper [4]. In this function the `transform` construct consists of two parts, first a loop-based computation, here for matrix multiply, and second a sequence of loop transformations to apply. These transformations split and tile the loops, parallelize the outer loop, and vectorize an inner loop. The Halide system, and our extension-based re-implementation of it, allow a programmer to experiment with different types and combinations of transformations without having to actually write the transformed code.

```
void matmul (unsigned m, unsigned n, unsigned p,
             float a[m][p], float b[p][n],
             float c[m][n]) {
  transform {
    for (unsigned i : m, unsigned j : n) {
      c[i][j] = 0;
        for (unsigned k : p) {
          c[i][j] += a[i][k] * b[k][j];
        }
      }
  } by {
    split i into (unsigned i_outer,
                  unsigned i_inner :
                  (m - 1) / NUM_THREADS + 1);
    parallelize i_outer
         into NUM_THREADS threads;
    tile i_inner, j
         into (TILE_DIM, TILE_DIM);
    split k into
         (unsigned k_outer
          unsigned k_unroll : UNROLL_SIZE,
          unsigned k_vector : VECTOR_SIZE);
    unroll k_unroll;
    vectorize k_vector;
  }
}
```

Figure 2. An example program segment for matrix multiplication using a composable language extension providing Halide-like separate specification and transformation of iterative constructs.

Additional extensions: The ableC paper and supplemental material describe several other extensions providing language features for writing and statically type checking SQL queries, Go-style task and message-passing constructs, and using lambda-expressions and closures.

A more recent extension developed in our group is a re-implementation of much of Kjolstad's tensor algebra compiler [6] (TACO) as an ableC extension. TACO and our extension allow one to specify which dimensions of a tensor are to be stored in a dense or sparse manner, thus allowing for space-efficient representations of large mostly-sparse tensors. These systems generate efficient code to perform general sparse tensor operations. For example, when `i`, `j`, and `k`, have been declared as *index variable*, statements like

```
    c[i, j] = a[i, k] * b[k, j];
```

are automatically translated into efficient implementations of the operations, in this case a matrix multiplication of sparse matrices. In this example, no new syntax is added by the extension. Instead the operator overloading features of ableC are used. The overloaded assignment (=), multiplication (*), and array access (`[ ]`) operators inspect the types of their arguments to determine what abstract syntax to translate to. In this case, it is abstract syntax for this extensions sparse tensor operations. These, in turn, generate the same kind of efficient code produced by TACO. That system is written as a C++ library and relies on operator overloading for writing computations like the one above. Since ableC also supports the specification of new syntax it avoids a few issues with C++ operator overloading to provide, in a few instances, a bit more natural means of specifying tensor computations.

These examples illustrate some ableC capabilities for developing composable and expressive language extensions. The tutorial leads participants through the process of developing language extensions of similar complexity.

3 AbleC, an extensible C specification

The ableC extensible C framework is a specification of C11 written using the Silver attribute grammar system [11] and the Copper LR-parser and context-aware scanner generator [12]. Language extensions are written using the same tools. From a specification to compose a set of extensions with the host language, Silver carries out the composition to effectively create a specification of the custom, extended language. From this specification, concrete syntax specifications for the a scanner and parser are extracted and provided to the Copper parser/scanner generator. The abstract syntax and attribute grammar specifications define the static analyses and code generation aspects of the host language and the extensions. An attribute grammar evaluator is generated and integrated into a implementation that performs many of the steps in a traditional compiler. The scanner and parser read an extended program, such as those in the figures above, and construct an abstract syntax tree (AST) representation. On this AST various static analysis are performed, the most important being the type checking of the program. For example, the algebraic data type extension will raise a type error if a pattern does not match the type of the data it is to be matched against. This allows type errors to be detected on programmer written code. Overloaded operators, such tensor multiply and array accesses above are translated to the implementations as specified by the extension.

Extension language constructs also specify how they translate down to plain C code. The ableC framework generates a plain C program from these translation, which is given to a traditional C compiler to create an executable program. Many of these translations are straightforward; for example the algebraic data type declarations translate into the expected collection of C `struct` and `union` declarations. But some translations are more complex. For example, A `cilk` function is translated into two functions, a fast and slow clone that make calls to the MIT Cilk-5 runtime. The extension performs the essentially the same translation as the Cilk-to-C translator that was part of the Cilk-5 system, but without the need to implement all the C-language features since these are provided by the host language.

4 Goals and materials

The primary goal of this tutorial is to introduce participants to the ableC framework and work through some example extension specifications to learn how to develop new language extensions of interest to them. This will cover the underlying tools, Silver and Copper, and how concrete syntax, abstract syntax, static analyses, and code-generation are specified using them. Part of this will cover the modular analyses that Copper uses [10] to ensure that deterministic scanners and parsers are generated from extension compositions. We also discuss the modular analysis used by Silver to ensure that composed attribute grammars are well-defined [3, 5].

The ableC specification, the underlying tools Silver and Copper, and the example extensions discussed above and used in the tutorial are all open-source and freely available online at the URLs below:

- ableC is available at http://melt.cs.umn.edu/ableC, archived at https://doi.org/10.13020/D6VQ25
- Silver is available at http://melt.cs.umn.edu/silver, archived at https://doi.org/10.13020/D6QX07
- Example extensions can be found on GitHub at https://github.com/melt-umn/
- tutorial materials are all available online at http://melt.cs.umn.edu/tutorials

Acknowledgments

This material is partially based upon work supported by the National Science Foundation (NSF) 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. We also thank Ted Kaminski, Lucas Kramer, and Aaron Councilman in their work in developing ableC and some of the above extensions.

References

[1] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 212–223.

[2] Google. 2018. The Go Programming Language. (2018). https://golang.org.

[3] Ted Kaminski. 2017. *Reliably Composable Language Extensions.* Ph.D. Dissertation. University of Minnesota, Minneapolis, Minnesota, USA. Available at http://hdl.handle.net/11299/188954.

[4] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. Reliable and Automatic Composition of Language Extensions to C: The ableC Extensible Language Framework. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 98 (Oct. 2017), 29 pages.

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

[6] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages.

[7] Lindsey Kuper, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R. Newton. 2014. Taming the Parallel Effect Zoo: Extensible Deterministic Parallelism with LVish. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 2–14.

[8] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. 2014. Freeze After Writing: Quasi-deterministic Parallel Programming with LVars. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 257–270.

[9] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 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)*. ACM, New York, NY, USA, 519–530.

[10] August Schwerdfeger and Eric Van Wyk. 2009. Verifiable Composition of Deterministic Grammars. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 199–210.

[11] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: an Extensible Attribute Grammar System. *Science of Computer Programming* 75, 1–2 (January 2010), 39–54.

[12] Eric Van Wyk and August Schwerdfeger. 2007. Context-Aware Scanning for Parsing Extensible Languages. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, New York, NY, USA, 63–72.