Extensible Languages for Sustainable Development of High Performance Software in Materials Science

Eric Van Wyk[†], Yousef Saad[†], and James R. Chelikowsky[⋆]

[†]Department of Computer Science and Engineering, University of Minnesota
[⋆]Institute for Computational Engineering and Sciences, University of Texas at Austin

Traditional Parallel Languages

There has been much research aimed at improving the state of the art of scientific software development. New programming languages and language constructs have been developed that allow one to write programs at a *high-level level of abstraction*, as compared to low-level constructs in languages such as C and FORTRAN.
However, sometimes the new constructs that may be useful to a developer cannot be used because they are spread across multiple incompatible languages or the domain-specific language that might be used does not fit into the existing development process for an existing application. A new approach to developing scientific software is needed.

Extensible Languages

Extensible programming languages allow programmers to mix and match new language features and thus provide a promising alternative. When these new features introduce new mechanism for expressing and exploiting parallelism they enable developers to freely experiment with different notions of parallelism before selecting the one that is the best fit for the task at hand.
Language features, packaged as *composable* language extensions, extend the *host* programming with

- new domain-specific syntax or notations
- new semantic analysis for error detection or detecting opportunities for optimizations
- new code transformations that optimize the domain-specific language constructs.

An example problem

Consider the very simple problem of determining the maximum value in a sub range of the third dimension of a 3D matrix.
Below are three code snippets using different composable language extensions [1]. These extensions add:

1. MATLAB inspired matrix indexing
2. Single Assignment C (SAC) inspired matrix computation operations
3. CFD-Builder and Halide inspired manually-specified compiler transformations to performed on *for* loops.

ANSI C + MATLAB

In this example, we extend ANSI C with syntax for accessing and slicing matrices based on MATLAB. The code snippet below first extracts the desired sub-region of the matrix and then, using standard C for-loops, computes the maximum.

```
Matrix region = mat[0::100,end-100::end,3];
int maximum = -200;
for (int i = 0; i < 100; i++) {
  for (int j = 0; j < 100; j++) {
    if (region[i,j] > maximum) {
      maximum = region[i,j];
    }
  }
}
```

ANSI C + SAC

Here, concepts from *Single Assignment C* (which are based on ideas from functional programming) are used to "fold" the `max` operation across all the elements in the sub-matrix `region`.

```
int maximum =
  with([0,0] <= [i,j] < [100,100])
  fold( max, -200.0, region[i,j]);
```

ANSI C + Halide

```
int maximum = -200;
transform
  for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 100; j++) {
      if (region[i,j] > maximum) {
        maximum = region[i,j];
      }
    }
  }
  using split i by 4, iin, iout,
        split j by 4, jin, jout,
        parallelize iout,
        vectorize jin ;
```

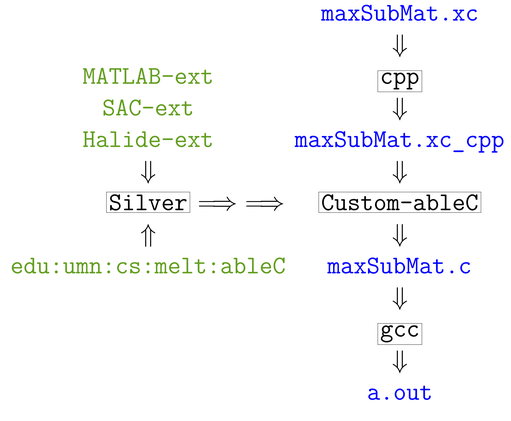
In the code snippet above, ideas from CFD-Builder and Halide are realized as a language extension that applies a series of programmer-specified transformations to the preceding for-loops. In this case, the loops are tiled, the outermost one is parallelized, and the inner most vectorized.
Unlike the SAC extension, which can be automatically parallelized and is quite easy for novices to use, this extension is for the demanding user who is willing to hand craft the transformations that will deliver the highest performance.

ableC

ableC is an extensible specification of ANSI C to which language extensions defining new features can be easily added by a programmer.
What distinguishes ableC is that language extensions are *composable*. This means that a collection of language extensions that are developed independently can be chosen by a programmer and imported into his or her ableC compiler.
Thus programmers with no knowledge of language design and implementation can easily extend their language with the features that they need for the task at hand.

Building extended compilers

The process of creating and using an extended compiler is diagrammed below. Due to the modular analyses described in the next column, the programmer can simply select the desired set of extension that they want their custom ableC compiler to use. Silver, our attribute grammar system, constructs the custom extended compiler that can translate the extended program, after type checking and optimizing it, down to plain ANSI C code that *gcc* then translates to executable form.



Challenges

- Challenge: Automatically composing concrete syntax specifications.
  Solution: the context-aware scanning and modular composability analysis in Copper [2].
- Challenge: automatically composing language extension semantics.
  Solution: attribute grammars with forwarding and modular well-definedness analysis as found in Silver [3, 4].
- Generating debuggers, interpreters, and IDEs for extended languages.
  Solution: additional descriptions in host language and language extensions specifications from which Silver generates debugging tools.

References

[1] Kevin Williams, Matthew Le, Ted Kaminski, and Eric Van Wyk. "A compiler extension for parallel matrix programming," In *Proc. of the International Conf. on Parallel Programming*, (ICPP), September 2014.

[2] August Schwerdfeger and Eric Van Wyk. "Verifiable composition of deterministic grammars," In *Proc. of ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 199–210. ACM, June 2009.

[3] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. "Silver: an extensible attribute grammar system," *Science of Computer Programming*, 75(1–2):39–54, January 2010.

[4] Ted Kaminski and Eric Van Wyk. "Modular well-definedness analysis for attribute grammars," In *Proc. of Intl. Conf. on Software Language Engineering (SLE)*, volume 7745 of *LNCS*, pages 352–371. Springer, September 2012.

Contact Information

- Web: http://www.melt.cs.umn.edu
- Email: Eric Van Wyk, evw@umn.edu