A Compiler Extension for Parallel Matrix Programming

Kevin Williams, Matthew Le, Ted Kaminski and Eric Van Wyk
Department of Computer Science & Engineering
University of Minnesota
Minneapolis, Minnesota
kwill,lematt,tedinski,evw@cs.umn.edu

***Abstract*—This paper describes a compiler extension to our prototype extensible C translator that adds new features for parallel execution of matrix operations and shows their application to problems in spatio-temporal data mining. The extension provides new language features for constructing new matrices, mapping functions over elements of a matrix, and accumulating operations that, for example, can sum values in a matrix. It also provides the appropriate semantic analysis to check for errors before translating the constructs down to parallel C code.**

The extension also provides features that let the programmer indicate how the extension translates these matrix constructs down to C code. Programmers seeking higher levels of performance can specify how the underlying for-loops are structured so that code using, for example, loop-tiling techniques or vector processors, is generated.

In general, compiler extensions supported by our approach allow new domain-specific syntax and semantic analyses to be easily added to the host language. Specifications of the host C language and the extensions are composed to create a custom translator that maps extended C programs down to plain (parallel) C code, checking for domain-specific errors and applying high-level domain-specific optimizations in the process.

I. Introduction and Motivation

Modern computer architectures require parallel programs to make efficient use of their capabilities. But programming for multi-core and many-core processors (which may also have short vector processors in each core) can be a daunting task. Furthermore, scientists and programmers have different performance needs and are willing to put in varying degrees of effort to achieve their desired performance. Some programmers are more casual in their needs; they want reasonable parallel performance but not at the expense of writing parallel code by hand and understanding all the obstacles for creating efficient parallel programs. Others are more aggressive users that demand very high performance. They are often more willing to work quite hard to achieve this higher level of performance by writing and modifying their programs to explicitly take into account processor features such as vector processing capabilities or address issues such as cache performance.

As multi-core and many-core architectures become more prevalent, programming language designers continue to search for useful programming abstractions, analysis, and optimizations that will make programming for these architectures a more productive and less error-prone process for both the casual and more aggressive users. For the casual user, languages strive to relieve the programmer from dealing directly with issues that arise when writing parallel code and instead provide high-level language constructs from which parallel code is automatically generated. For the more aggressive users programming abstractions that explicitly expose parallelism may be required, but these are typically easier to use and less error-prone than basic threads and locks.

There are many domain-specific languages and extensions to general purpose languages that provide language constructs from which parallelism can be easily detected and exploited, as compared with traditional nests of for-loops. These include language such as Single Assignment C (SAC) [1], SISAL [2], and nested data parallelism languages such as NESL [3]. Other languages, such as Cilk [4] and Yada [5], provide abstractions that allow programmers to explicitly express the parallelism in the program, but at a higher and less error-prone level of abstraction than traditional threads and locks.

Of course, no one has found the ideal set of programming abstractions to suit all users, since users needs are quite diverse. The problem is further complicated by the fact that both the architectures used and many of the problem domains using them are rapidly changing. The challenges to programmers are further complicated since it is difficult to evaluate and to experiment with these "whole language" approaches. One cannot easily try one and move to another if it is not sufficient. It is often clumsy to write applications that make use of multiple forms of parallelism as implemented in these different systems. Furthermore, programmers may desire new domain-specific abstractions for other programming problems such as matrix processing. MATLAB, for example, has nice features for this, but using MATLAB code in combination with other systems can also be rather clumsy.

In our view, a potential solution to this problem is extensible languages that allow programmers to customize their languages by easily importing new language features into their compiler/translator. Here, these new features are for parallel programming. Extensible languages allow programmers to more easily experiment and evaluate new language features, packaged as language extensions. Our work has focused on *automatically composable* language extensions so the programmer need not be an expert in programming language design and implementation to use them.

In this paper, we describe our work in developing composable language extensions that we have implemented for a rather complete subset of ANSI C called CMINUS. Some of these extensions are general-purpose in nature; they add a notion of tuples and reference counting pointers that automatically free their data. Others are domain-specific and are directed at matrix processing. Both types of extensions provide new syntax and semantic analyses to the language before translating the added features down to plain C code (for eventual compilation to executable form by a traditional compiler).

Section II provides some background our our approach to extensible compilers and languages. In Section III we discuss a compiler extension that adds MATLAB-like matrices and operations over them to the C host language; Section IV shows how these new language features can be used in a spatio-temporal data mining applications. This is an extension of work previously reported in an exploratory workshop paper [6]. Section V describes a language extension that gives the programmer a great deal of control over the type of C code that is generated from the matrix operations. This extension allows programmers to explicitly transform the generated for-loop structures to, for example, implement a tiling over 2D data. Section VI describes modular analyses that can be applied to the declarative specifications of language extensions, by the extension author, to ensure that independently-developed language extensions will in fact be composable. Section VII describes related work and Section VIII concludes and discusses some future work.

II. Extensible languages and compilers

Classically, languages are constructed by a centralized group of developers so that all components of the language work well together. Programmers can construct libraries which can be used by others to complete tasks in faster or more efficient ways, and, critically, any user is free to include or import any collection of libraries because the language ensures that these libraries do not interfere with one another.

The novelty of this work is that programmers are able to use multiple independently developed *language extensions* in a manner not unlike how they traditionally use multiple independently developed libraries. This is possible because of how the host language and its extensions are implemented and composed. Extensible languages are defined by two primary components: a specification for a host language and a set of specifications for the extensions to that host language. The extensions are constructed by independent parties focusing on their own domain-specific language extensions that may introduce new syntax (notations), semantic analyses (error checking), and optimizations to the host language and its translator. These extension developers need have no knowledge of one another. As with traditional libraries, the programmer using an extensible language is free to choose the set of extensions that fits his or her problem at hand and direct a set of compiler-generating tools to compose the extensions with the host language and construct the compiler for their customized language. The programmer is not required to have any knowledge of the language composition process.

Automatic composition of language extensions raises a number of challenges. We use attribute grammars [7], [8] and context-aware scanners [9] with LR parsers [10] to specify host languages and language extensions since these formalisms naturally compose. Furthermore we have developed modular analyses of these formalisms that provide strong guarantees

```
 1 int main (int argc, char **argv) {
 2   Matrix float <3> mat = readMatrix("ssh.data");
 3   int m = dimSize(mat, 0); //Latitude dimension
 4   int n = dimSize(mat, 1); //Longitude dimension
 5   int p = dimSize(mat, 2); //Time dimension
 6   //Compute temporal means using with-loops
 7   Matrix float <2> means =
 8     with([0,0] <= [i,j] < [m,n]) genarray([m,n],
 9       //Compute temporal mean for location i,j
10       (with([0] <= [k] < [p])
11         fold (+,0.0,mat[i,j,::])/p));
12   writeMatrix("means.data", means);
13   return 0;
14 }
```

Fig. 1: Temporal Mean Algorithm in Extended Cminus

that programmer-selected language extensions will compose with the host language to form a working compiler for their customized language [11], [12]. These formalisms are described in greater detail in Section VI.

Because multiple language extensions can be easily and reliably imported into a general purpose host language such as C, the programmer is much freer to experiment with new language abstractions. He or she can carry out this experimentation and evaluation in their production code instead of on a small toy problem. In some part of their application a new feature from a language extension can be used; the extended translator (generated from the user chosen extension specifications) will check this extended program for errors and translate it down to plain C code, which can then be compiled for execution by a traditional compiler. The extended translator slips into the existing development process as just another step in the compilation process. Based on the results of this effort the programmer can continue to use the extension or easily abandon the extension; either way the cost of the experiment is rather low.

III. Matrix extensions to Cminus

A great deal of scientific code is written in MATLAB due to its expressive language features such as multiple-element indexing and flexible support for matrices of arbitrary rank, which makes the code more concise and easier to both construct and understand than if done in a general purpose language like C or FORTRAN. We provide an implementation of a subset of C as the "host" language, with extensions that support many of the useful features found in MATLAB, Single Assignment C [1], and other general purpose languages.

Our recurring example is shown in Fig. 1, which takes a three-dimensional matrix *mat* composed of sea surface height data split by latitude, longitude, and time, and computes for every measured point on the ocean's surface the average sea height over time. The averages are computed in lines 10-11 via the fold with-loop and every location in *mat* is visited by the *genarray* with-loop, both of which are defined below.

A. Matrix Extension

First, we have a domain specific extension which adds many features that can be found in MATLAB and SAC.

1) Matrix Type: The first thing we have added is a new data type to our host language's type system, which is for matrices. A variable can be declared as a matrix by using the following type expression:

$$TypeExpr \Rightarrow Matrix\ (int \mid bool \mid float)\ \text{‘<’}\ Integer\ \text{‘>’}$$

After the keyword "Matrix" is a type that specifies the type of the matrix elements, followed by an integer literal specifying the number of dimensions. As of now, matrices can only contain integers, booleans, or floating point numbers. Consider line 2 of Fig. 1, where this syntax is used to declare a three-dimensional matrix mat containing floats read from $"ssh.data"$.

2) Arithmetic: Next, we have overloaded the arithmetic and comparison operators in the host language based on argument types, allowing these operators to be used for matrix arithmetic. Our extended type system is able to verify that these operations are only performed on matrices of the same type and rank. Additionally, we are able to perform arithmetic between matrices and scalar values. For the multiplication operator, we have added another operator to denote element wise multiplication as opposed to matrix multiplication in the linear algebra sense. All other operators are all element wise operators. The assignment operator is also overloaded for matrices.

3) Matrix Indexing: We have implemented our matrix indexing in the style of MATLAB. This includes the following variations (where **end** denotes the last element in the given dimension):

(a) Standard matrix indexing which extracts a single element – $data[6, 4, 1]$ returns a scalar.
(b) Range indexing which extracts a contiguous block of cells – $data[0 :: 4, \mathbf{end} - 4 :: \mathbf{end}, 0 :: 4]$ returns a $5 \times 5 \times 5$ matrix, where **end** is the index of the last element in the matrix in that dimension.
(c) Whole dimension indexing, marked by ::, which extracts the entire dimension selected – $data[0, \mathbf{end}, ::]$ returns a vector of size $dimSize(data, 2)$, where $dimSize(m, d)$ is the size of dimension d in matrix m.
(d) Logical indexing which selects a possibly non-contiguous submatrix based on an array of booleans generated by element wise operators – $data[v\%2 == 1, ::, 0]$ returns a $n \times dimSize(data, 1)$ matrix where v is a one-dimensional matrix, $v\%2 == 1$ returns a one-dimensional boolean matrix of the same size as v with n true values such that the i^{th} element is true if and only if $v[i]\%2 == 1$

These various methods of matrix indexing can be used in any combination for a matrix of arbitrary rank. Additionally, this sort of matrix indexing can be used both on the left hand and right hand side of an assignment operator.

Looking back at Fig. 1 we can see in line 11 inside the body of the innermost with-loop that for every point (i, j) we are extracting a chunk of mat along the third dimension. This returns a one-dimensional matrix of length equal to the size of the third dimension of mat.

4) With-Loop: The next feature we have implemented is the with-loop from the Single Assignment C (SAC) language. The with-loop has the syntax specified in Fig. 2.

```
  WithLoop  ::=  with ‘(’ Generator ‘)’ Operation
 Generator  ::=  ‘[’ ExprList‘]’ ROp ‘[’Id (‘,’Id)* ‘]’
                 ROp ‘[’ ExprList ‘]’
       ROp  ::=  ‘ < ’ | ‘ <= ’
 Operation  ::=  genarray ‘(’ ‘[’ ExprList ‘]’ ‘,’ Expr ‘)’
             |   fold ‘(’ FoldOp ‘,’ Expr ‘,’ Expr ‘)’
    FoldOp  ::=  ‘ + ’ | ‘ − ’ | ‘ ∗ ’ | ‘/’ | ‘&&’ | ‘||’ | Id
  ExprList  ::=  Expr ( ‘,’ Expr )*
```

Fig. 2: EBNF With-Loop Syntax

```
Matrix float<2> means;
for (int i = 0; i < m; i++) {
  for (int j = 0; j < n; j++) {
    float _sum = 0.0;
    for (int k = 0; k < p; k++) {
      _sum = sum + mat[i,j,k];
    }
    _sum = _sum / p;
    means[i,j] = _sum;
  }
}
```

Fig. 3: Lines 7-11 of Fig. 1 Expanded With For Loops

The $Generator$ portion specifies a set of indices to operate over, and the $Operation$ portion specifies what is to be done over those indices. The number of expressions in both the upper bound and lower bound should match the number of *Id's* provided, which should also match the number of dimensions provided in the $Operation$. Our extended semantic analysis checks that these criteria are met and can produce error messages if necessary. A with-loop $Operation$ is either a $genarray$ or a $fold$.

$fold(foldOp, baseVal, expr)$ extracts the elements specified by the generator and folds them up using the $foldOp$ operator, starting with the value $baseVal$. Again, in Fig. 1 we can see in lines 10-11 we are adding up all elements along the third dimension of mat by folding the operator $+$ over them. We then divide by the total number of elements p to compute the mean.

$genarray(shape, expr)$ generates a new matrix with size and rank according to that of $shape$, where each element in the set of indices specified by the generator is equal to $expr$ and 0 elsewhere. We see an example of this in Fig. 1 in lines 8-11, where we are generating a matrix of size $m \times n$ where each point in the matrix is defined by the inner $fold$ loop for every location $[i, j]$. Note that the shape used in the operation may differ from the indices defined in the generator, however, the shape in the operation must be a superset of the indexes in the generator, which is something that can be checked at runtime. The idea behind this is that the programmer can perform these operations on subsets of a matrix, rather than being forced to fill out the whole matrix.

An approximate translation of the internal expansion of lines 7-11 from Fig. 1 is shown in Fig. 3. By approximate, we mean that non-critical code such as reference counting (described in Section III-B) was omitted for demonstration purposes. Notice that the outer $genarray$ has been replaced

```
Matrix int <2> connComp(Matrix float <2> ssh){
  Matrix int<2> labels =
    init(Matrix int<2>, 721, 1440);
  for(int i = −100; i < 100; i++){
    Matrix bool<2> binary = ssh < i;
    ...//compute connected components
  }
  return labels;
}

int main(int argv, char ** argv) {
  Matrix float <3> ssh =
    readMatrix("ssh.data");
  Matrix int <1> dates =
    readMatrix("dates.data");
  ssh = ssh[::, ::, dates >= 01012000];
  Matrix int <3> labels =
    matrixMap(connComp, ssh, [0,1]);
  writeMatrix("eddyLabels.data", labels);
}
```

Fig. 4: Connected Component Matrix Map Example

with two nested for loops, each iterating over one dimension of mat. The inner fold has been replaced with a loop which adds each sea height for location $[i, j]$, divides it by p (the number of time steps), and copies the value into $means$. A library implementation of this would likely evaluate the result of the with-loops into a temporary variable which is then copied into $means$. Our language extension is able to interact with the assignment and both with-loops and thereby move the assignment and avoid an extraneous copy.

Additionally, the matrix indexing in line 11 of Fig. 1 which originally returned a one-dimensional matrix was removed in the generation of Fig. 3. This was driven by a set of high-level optimizations which observed that the fold iterated across one dimension of mat and there was no need to iterate over a copied slice of mat. This optimization is also not possible via libraries, as high-level and invasive optimizations such as this cannot be applied across separate libraries.

The with-loop provides our extended language a concise way for specifying complex matrix operations. Also, because of the semantics of the generator we have a unique set of elements to access allowing us to perform these operations in parallel, which is discussed in further detail in Section III-C.

5) Matrix Map: Lastly, we describe a matrix-map construct, which is quite similar to the with-loop in that it allows the programmer to map a function over a matrix, but this allows one to compute over ranges of a matrix, rather than element by element. More specifically, the programmer specifies what dimensions he or she would like to apply the function to, and then the rest of the dimensions are implicitly iterated over. The syntax for the matrix map is

Expr ⇒ *matrixMap* '(' *Id* ',' *Expr* ',' '[' *ExprList*']')

Fig. 4 shows an example to help clarify this idea. The goal of this sample program is to take a three dimensional spatio-temporal data set, where for each point in time, we want to label all connected components in space.

```
for(int i = 0; i < dimSize(ssh, 2); i++){
  result[::,::,i] = connComp(ssh[::,::,i]);
}
```

Fig. 5: Semantically equivalent code fragment.

To accomplish this, we have written a function, *connComp*, which takes in a two dimensional matrix and labels all connected components. We then map this function, using the *matrixMap*, over the three dimensional matrix, specifying that we want this function applied to the first and second dimensions (denoted by $[0, 1]$) and then a loop iterating over the third dimension is generated. This can intuitively be thought of as the following few lines of semantically equivalent code in Fig. 5.

This however, becomes much more complicated as the matrix indexing generates more loops, and since this can be run in parallel, we actually lift this out into a new function so that the spawned threads can get direct access to it. One important thing to note here is that when using the matrix-map, the result is always the same size and rank as the matrix getting mapped over, though a generalization of this extension that removes this restriction is being developed.

At first glance, these abstractions may seem general purpose and not specific to the spatio-temporal data mining domain. While this may be true, we designed this language extension with a number of data mining applications in mind. These abstractions have previously been shown to work well in general, however, we found them to be especially useful in our applications.

B. General Purpose Extensions

In addition to these domain-specific extensions, we have also added a few general purpose extensions. The first is a notion of tuples similar to what is done in the functional languages such as ML and Haskell. In MATLAB, one can return multiple arguments from a function, which is a common operation. Tuples give us a way of doing this same thing; however, they are more general and can be used universally, rather than only with functions. The features provided with the tuple extensions are tuple declaration ($(int, float, bool)\ t$;), anonymous creation ($return\ (x, y, z)$;), and tuple assignment ($(a, b, c) = f()$;), along with the expected semantic analyses for each.

A second general purpose extension that we have added is the reference counting pointer. Reference counting provides us a way of automatically managing memory. The idea is that we attach an extra 4 bytes to every piece of memory that gets allocated as a reference counting pointer, and use this extra 4 bytes to keep track of how many live references there are to that block of memory. If another variable also becomes a reference for that same piece of data, then we increment this counter by one. Anytime a variable goes out of scope, or gets assigned a new piece of data, then we decrement its reference counter by one. If a reference counter ever reaches zero, then we free that data.

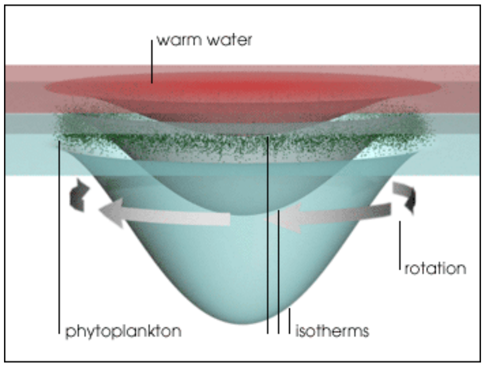

Fig. 6: [13] Rendering of a Cyclonic Ocean Eddy.

C. Parallel Code Generation

The with-loop and matrix-map extensions specify regular computations that can be easily parallelized with the expectations of near linear speedup. Here we discuss an automatic parallelization based on Pthreads. In Section V, we use user-directed transformations to parallelize and optimize with-loops.

A naive translation to Pthreads will not achieve linear speedup. First, there is the issue of thread management overhead. Most multi-threaded programs adopt the fork-join model, where threads are spawned when they are needed, and then destroyed as soon as that parallel computation is done. If there is a lot of disjoint parallel computation to be done, then the program pays the price of creating and destroying threads each time. To mitigate this problem, we have adopted the enhanced fork-join model from SAC [14] where we spawn the necessary number of threads (indicated by a command line argument) at the beginning of program execution and send them straight into a spin lock where they sit idle until some parallel work is to be done. When a parallel construct is encountered by the main thread, it flips the condition that keeps the threads spinning, which releases all of them at once to execute the work in parallel. As soon as each thread is done, it passes through a stop barrier and goes right back into the spin lock. The main thread then waits in the stop barrier until all threads have completed their work.

Another problem that needed to be dealt with was memory management. To take care of this problem we build the underlying implementation of matrices on top of the reference counting pointers described in section III-B. This sort of memory management works well in this setting, as most allocations made are relatively infrequently and are large compared to those made in say a functional programming language. In addition, we were faced with related memory allocation challenges. The issue is that some implementations of malloc are naïvely implemented using a mutex lock to deal with contention over the heap. More recent implementations separate the heap into “arenas” as soon as contention is detected [15], however results shown in [16] show that these “off the shelf” memory allocators do not scale well in this setting. We briefly explored the use of the Hoard allocator; however, as noted in [16], when allocation requests exceed a certain threshold, they are mapped directly to virtual memory using *mmap* and *munmap* which incurs significant overhead.

The basic idea behind our memory allocator is to create a separate heap of a fixed size for each thread at the time of program creation. Each thread only requests memory from its personal heap. If its heap ever becomes depleted, then a new block of memory of the same fixed size is added to its heap's free list, unless the requested size is greater than the fixed size, in which case the requested size is added to the heap and immediately given to that thread.

D. Compiler Extension Implementation

To create a language extension such as this one, the extension developer must define both the concrete syntax and abstract syntax of the constructs as context free grammar rules. The concrete syntax is combined with that of the host language and other user selected extensions and given to our parser generator [9] to build the scanner and parser for the extended language. With the abstract syntax, the extension author also specifies the semantic analysis for the new constructs in the form of attribute grammar rules. These are used to define type checking and other error checking as well as the translation of the construct down to plain C code in the host language.

IV. Ocean Eddy Application

We now shift our focus to a more specific application within spatio-temporal data mining: identifying and tracking ocean eddies. Mesoscale ocean eddies are rotating pools of water in the ocean spanning tens to hundreds of kilometers which can last anywhere from a few days to several months. Ocean eddies are an important phenomenon to monitor as they dominate the ocean's kinetic energy and are responsible for the transport of heat, salt, nutrients, and energy across the oceans [17].

Fig. 6 is a NASA image [13] showing that the rotating nature of ocean eddies makes them identifiable in sea surface height (SSH) data, as it causes the center of the eddy to be lower in height compared to its perimeter. One can identify ocean eddies algorithmically by iteratively thresholding the SSH data and searching for connected components that satisfy certain criteria that are typical of ocean eddies. The problem with this is that it is susceptible to noise in the sea surface height data collected from satellites. One problem with this is that the detection algorithm will miss an eddy for a given time frame, which can have significant impacts on the tracking results [18].

One way to circumvent this issue is to incorporate time into the detection scheme. As ocean eddies travel along a path, they leave a signature in the SSH data over time. In Fig. 7 we see such a signature. The graph shows the SSH time series (solid black line) for a given point on the ocean surface over some number of weeks. What has happened is an eddy traveled through this point, causing the sea surface height to lower, and then as the eddy passed through the point it began to rise again. Both before and after the eddy passed through, there are small “bumps” in the time series, these can be attributed to both the restlessness of the ocean, and inaccurate noisy readings from the satellites. We can quantify these signatures left by eddies in the SSH data by searching for two local maxima in each direction of a local minima, and computing the “area” between that trough and an imaginary line going from peak to peak, as

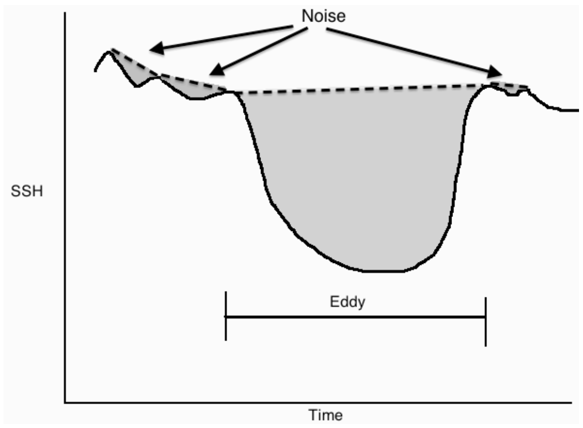

Fig. 7: Sea surface height profile of a single point affected by an ocean eddy over time and computed areas.

seen in Fig. 7. Large areas will then correspond to segments of the time series (troughs), that underwent substantial drops and rises, and those that are shallow, such as the first two and last segments, can be associated with noise. Each point in the trough is then assigned this area, which serves as a way of ranking locations on the map by how likely it is that what is being detected is actually an eddy and not an illusion created by noise.

Fig. 8 shows how we can encode such an algorithm using our version of C extended with the language extensions described above. What we have is a function, *scoreTS*, that computes a score for every point in a given time series. In the main function of the program, we map this scoring function over each point in space, applying it to all elements in the third dimension as seen in line 58 (recall dimensions begin at 0, so 2 corresponds to the third dimension).

Within the scoring function, we begin by trimming off the beginning of the time series until we reach our first local maxima. We then continually cut out a chunk of the time series as extracted by the *getTrough* function, and compute the area of that trough using the *computeArea* function. Each point on that trough is then assigned the computed area, and put back into the original time series.

The *getTrough* function simply begins by starting at the first local maxima, and then walks over the time series until it encounters another local maxima, returning this sub-sequence of the data, the beginning index, and the ending index in the form of a tuple.

The *computeArea* function then computes the equation of a line based off of the two local maxima. In line 27 we compute the dotted line seen in Fig. 7 first by creating a one dimensional array containing the elements between zero and the length of *areaOfInterest*, which then gets multiplied by the slope of the line, and added to the y-intercept. In lines 28-32, we then compute the area by subtracting the trough from this line, and sum the differences using a with-loop; lastly, we create an array the same length of the *areaOfInterest* where each point is the computed area.

```
 1 (Matrix float <1>,int , int)
 2 getTrough(Matrix float<1>ts , int i) {
 3   int beginning = i;
 4   int n = dimSize(ts, 0);
 5   //Walk Downwards
 6   while(i+1 < n && ts[i] >= ts[i+1])
 7       i = i+1;
 8   //Walk Upwards
 9   while(i+1 < n && ts[i] < ts[i+1])
10       i = i+1;
11   //Return the trough
12   return (ts[beginning::i], beginning, i);
13 }
14
15 Matrix float<1>
16 computeArea(Matrix float<1> areaOfInterest) {
17   float y1 = areaOfInterest[0];
18   float y2 = areaOfInterest[end];
19   int x1 = 0;
20   int x2=dimSize(areaOfInterest,0)-1;
21   //compute slope
22   float m = (y1-y2) / ((float)(x1-x2));
23   //compute y intercept
24   float b = y1 - m*x1;
25   Matrix float<1> Line = (x1::x2)*m+b;
26
27   float area = with([x1] <= [i] < [x2])
28     fold(+,0.0,line-areaOfInterest);
29
30   return
31     with([0] <= [i] < [dimSize(Line, 0)])
32       genarray([dimSize(Line, 0)], area);
33 }
34
35 Matrix float<1>scoreTS(Matrix float<1>ts) {
36   Matrix float<1>scores =
37     init(Matrix float <1>, dimSize(ts, 0));
38   int i = 0;
39   while(ts[i] < ts[i+1])  // trimming
40       i = i+1;
41   int n = dimSize(ts, 0);
42   int beginning;
43   Matrix float <1> trough;
44   while(i < n-1) {
45     (trough, beginning, i)
46       = getTrough(ts, i);
47     scores[beginning::i]
48       = computeArea(trough);
49   }
50   return scores;
51 }
52
53 main (int argc, char **argv) {
54   //Shape of SSH: 721 x 1440 x 954
55   Matrix float <3> data
56     = readMatrix("ssh.data");
57   Matrix float <3> scores;
58   scores = matrixMap(scoreTS, data, [2]);
59   writeMatrix("temporalScores.data", scores);
60   return 0;
61 }
```

Fig. 8: Ocean eddy scoring implementation.

```
means = with([0,0] <= [i,j] < [m,n])
  genarray([m,n],
    with([0] <= [k] < [p])
      fold (+,0.0,mat[i,j,::])/p)
  transform
    split j by 4, jin, jout,
    vectorize jin,
    parallelize i ;
```

Fig. 9: Temporal mean algorithm written with programmer specified transformations.

V. Explicit program transformation and optimization extension

Certainly, some users may be content with the performance achieved by systems and tools that generate parallel code with little or no explicit specification of the parallelism in their programs. For example, the performance of the parallel code generated from the matrix constructs described above scales nearly linearly with the number of cores on the machine with two 6-core processors on which we have tested it and this may be satisfactory to some users. But there are other users that are looking for a higher level of performance than is typically achieved by such automated approaches. Unfortunately, when these automatic systems do not provide the requisite performance the programmer is typically left to abandon the system entirely and write their parallel code by hand.

Modern compilers are capable of performing sophisticated transformations over nested for-loop computations to expose parallelism, restructure nested loops (*e.g.* to *tile* the data to provide better cache behavior), or to vectorize inner loops to use vector processing units. But the problem is that they often do not perform these transformation in a way the generates the highest performing code. Systems such a Halide [19], [20], CHiLL [21], [22], and CFD Builder [23], [24] allow programmers to indicate what compiler transformations should be performed to direct the compiler in generating the desired code. This gives the sophisticated programmer control of the generated code without having to write all that (often convoluted and intricate) code manually.

The matrix language constructs in the compiler extension described earlier generate the same type of nested for-loops that these compiler transformations typically target, but sometimes fail to optimize as much as desired. We have thus extended the matrix processing constructs to allow the programmer to specify what transformations should be made to the underlying for-loops to maximize performance. Consider again the nested with-constructs in Fig. 1. These with-loops are shown again in Fig. 9 but with sample explicit transformations specified. The language extension uses these specifications to transform the generated for-loops.

The process of transforming this code fragment to plain C code begins by expanding the with-loops into the for-loops shown in Fig. 3 and then applying the transformations in the order in which they appear. The result of the expansion and *split* transformation is shown in Fig. 10. Here we see that the loop indexed by j has been split into two loops. The transformation also replaces instances of j with the appropriate

```
for (i=0; i<m; ++i) {
  for (jout=0; jout*4<n; ++jout) {
    for (jin=0; jin<4; ++jin) {
      _sum = 0.0;
      for (k=0; k<p; ++k) {
        _sum = _sum + mat[i,jout*4+jin,k];
      }
      means[i,jout*4+jin] = _sum / p;
    }
  }
} transform
    vectorize jin,
    parallelize i ;
```

Fig. 10: Temporal mean algorithm after expansion and application of *split* transformation.

```
float* _stage_p =
  (float*)_mm_malloc(4 * sizeof(float), 16);
for(i=0; i<4; ++i) { _stage_p[i] = (float)p; }
const __m128 _p = _mm_load_ps(_stage_p);

float* _zeros =
  (float*)_mm_malloc(4 * sizeof(float), 16);
for(i=0; i<4; ++i) { _zeros[i] = 0.0; }

#pragma omp parallel for private(i,j,k,_q)
for(i=0; i<m; ++i) {
  for(jout=0; jout*4<n; ++jout) {
    __m128 _sum = _mm_load_ps(_zeros);
    for(k=0; k<p; ++k) {
      __m128 _temp =
        _mm_load_ps(&(mat[i,jout*4,k]));
      _sum = _mm_add_ps(_temp,_sum);
    }
    _sum = _mm_div_ps(_sum,_p);
    float _q[4] __attribute__((aligned(16)));
    _mm_store_ps(_q,_sum);
    means[i,jout*4,0] = _q[0];
    means[i,jout*4,1] = _q[1];
    means[i,jout*4,2] = _q[2];
    means[i,jout*4,3] = _q[3];
  }
}
```

Fig. 11: Temporal mean algorithm after all transformations.

expression $jout * 4 + jin$. (To keep the example simple we have assumed that the dimension n is a multiple of 4.)

Next, the *jin* loop is vectorized and the outer loop parallelized using an OpenMP pragma, as shown in Fig. 11. To vectorize this code, we use Intel's SSE which uses 128 byte vectors. We fill each vector with 4 32-bit single-precision floating point numbers. These parameters can be set differently for different systems. Note the addition of many new variables involved in loading data into vectors before the averaging computation and receiving stores afterwards. These have been floated above the outermost for loop because they are unchanged by the loops. It should be clear that the transformation between Fig. 10 and Fig. 11 is a complex one, especially compared to adding the programmer specified transformation.

This extension, as in Halide, CHiLL, and CFD Builder, gives the programmer a great deal of control over the generated code. Users of it are expected to be somewhat more sophisticated developers than those using the earlier extensions that automatically generate parallel C code. This control allows programmers to experiment with different loop structures in their search for higher performance. They can more easily experiment with different tile sizes, loop vectorizations, or approaches to parallelism without having to manually rewrite their code for each configuration that they would like to try. Note that we intentionally do not provide any performance numbers here since the resulting performance is really up to the programmer to choose the appropriate set of transformations and any performance numbers are more of an evaluation of that particular set of transformations as opposed to the general approach. The papers on Halide [19], [20], CHiLL [21], [22], and CFD Builder [23], [24] provide a more complete discussion on how this technique can be used to improve code performance and provide several performance evaluations.

The implementation of this language extension is similar to the implementation of the auto-parallelizing matrix construct extension in that the extension developer must specify the concrete and abstract syntax of the new constructs, as well as the attribute grammar specifications that perform basic semantic analysis for error checking. In this case to detect, for example, that the loop indices in the transformations correspond to loops in the code being transformed.

This extension makes use of *higher-order* [25] attributes in Silver to perform the various transformation. These are attributes that allow code fragments (syntax trees) to be manipulated and propagated around the syntax tree just like other attributes. The *split* transformation, for example, uses these to extract the body of the loop, modify the appropriate index variables, and generate the two nested loops that replace the one being split.

An important feature here is that new transformation specifications can be easily added, in the same way in which new independently-developed language extensions are added to the host language. For example, a transformation specification to *tile* two nested loops indexed by x and y can be specified as two splits and a reorder. After splitting x into xin and $xout$ and y into yin and $yout$, a reorder transformation rearranges the loops into the following order from outermost to innermost: $xout$, $yout$, xin, yin.

VI. Building Extensible Languages and Composable Language Extensions with Silver

Language translation/compilation can be split roughly into two steps. First, the text is converted into a syntax tree by using a scanner and parser. Second, semantic analysis is performed on the abstract syntax tree to check for semantic errors, and if there are none, translate the syntax tree into the required output (e.g. an executable or a host language program). Generating these tools from a composition of a host language specification and several independently-developed language extension specifications presents a number of interesting challenges.

A. Scanning and Parsing

A scanner converts text into a stream of tokens based on a set of regular expressions for terminal symbols (keywords, literal integers, variables, etc). This stream of tokens is supplied to the parser which generates the syntax tree according to a context-free grammar specification that defines the language syntax. Each extension specifies its own regular expressions for terminals and context free grammar for its added syntax, the grammar specifications in Silver are similar to, but more verbose, those shown in Fig. 2.

These syntax specifications for the host language and the user selected extensions naturally compose, but conflicts may arise. It is possible that two different languages will want to use the same keyword (such as "with" in the with-loop construct). To solve this, Copper, our parser and scanner generator, constructs a context-aware scanner [9]. Such a scanner uses the "context" of the parser to determine which of the overlapping keywords is to be recognized. In cases where context is not enough, it allows the programmer to add a simple annotation to determine which extension the keyword is to belong to. Details of this process can be found in a previous paper [9].

There can also be problems in the parsing of extended languages in which the composed context-free grammar is ambiguous. To avoid this problem we require that the composed grammar be in the LALR(1) [10] class of deterministic (and thus unambiguous) grammars. The restrictions of LALR(1) are significantly eased by using a context aware scanner since overlapping keywords, and more generally, overlapping regular expressions for terminals, are allowed.

Because the composition of LALR(1) grammars does not always result in an LALR(1) grammar, an analysis has been developed that imposes further restrictions on the extension grammars to ensure that the composition of these restricted grammars is LALR(1) [11]. The analysis is run by the extension developer on his or her extension to check that it is in this restricted class. Formally, this is stated as follows:

$$\begin{aligned}(\textbf{for each } i \in [1,n].\ & \mathit{isLALR}(CFG^H \cup CFG_i^E) \wedge \\ & \mathit{isComposable}(CFG^H,\ CFG_i^E)\ ) \\ \Rightarrow \mathit{isLALR}(CFG^H \cup \{CFG_1^E, \ldots, CFG_n^E\})\end{aligned}$$

It is the *isComposable* check that determines if the extension grammars are in the more restricted set. The important point about this analysis is that if the programmer chooses only extensions that have passed the analysis, then they have a strong guarantee that the generated scanner and parser for their chosen set of extensions will be LALR(1) and thus a working and correct scanner and parser can be generated.

The domain-specific matrix extension does pass this test. The tuples extension does not, however, since the initial symbol for tuple expressions is a left-paren, "(", which violates the restriction that a unique initial terminal symbol is needed on extension syntax. The "with" terminal, for example, is such an initial terminal. Thus the tuples extension will be packaged as part of the host language. One could modify the tuple terminals to be "(." and ".)" and thus be distinguishable from other symbols in the host language and thus pass this analysis. For a complete discussion of this analysis and the restrictions it imposes please see the original paper [11].

B. Semantic Analysis and Translation

The semantic analysis phase performs type checking, uses these types to resolve the overloading of operators such as addition (+) and assignment (=), find and report semantic errors, and drive the translation of extension constructs down to plain C code. The language extension described in this paper perform the same semantic analysis on their added syntax, for example checking that number of expressions in the upper and lower bounds of a with-loop are the same.

These analyses are specified by extension developers as attribute grammar specification in Silver [8], our attribute grammar system. Attribute grammars (AG) [7] provide a means for decorating program syntax trees with attributes describing, for example, the type of an expression, or the C translation of a statement. These values are computed based on a set of equations that define attribute values based on other attributes in the tree.

A challenge arises in that the composition of the extension AG specifications may not be *well-defined* (meaning some attributes do not have defining equations). Silver has a modular well-definedness analysis, similar in format to the modular determinism analysis for Copper described above, that extension designers can run on their extension. It guarantees that if only extensions that pass this analysis are chosen, then the composition of them will be well defined. All extensions described above pass this analysis. Full details of this analysis can be found in a previous paper [26].

VII. Related Work

There have been many efforts to improve the practice of scientific software development. These include the development of new programming languages and language constructs allowing one to write programs at a high-level level of abstraction, as compared to low-level constructs in languages such as C and FORTRAN. For example, constructs such as *nested* data parallel arrays can be found in domain-specific languages such as NESL [3] that improve on performance and the ease of use of *flat* data parallel arrays found in versions of FORTRAN. Fortress [27], X10 [28], and Chapel [29] are entirely new languages that have also been developed. However, many of these developments have not found as wide of an audience as they might warrant, for various technical and non-technical reasons. 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.

Thus an *extensible* approach in which new domain-specific features, as chosen by the application developer in the same way that he or she chooses to use different libraries in traditional development, may be a viable alternative to these *monolithic* approaches described above. We are not the only ones interested in extensible languages and tools. One approach is to use domain-specific "embedded" languages, in which domain-specific features are added as libraries but to host languages that can give these features the same look and feel of host language features. The DeLite project is a prime example of this in which the host language is Scala; of note is the use of a meta-programming technique called lightweight modular staging [30] which allows the Scala code to analyze and optimize code. There are also approaches, similar to ours, in which compilers and translators are generated from some (composed) declarative specifications of language syntax and semantics. JastAdd [31], also based on attribute grammars has been used to develop an extensible Java compiler [32] and the Spoofax [33] system, based on the Stratego term-rewriting system [34] has also been used to add domain-specific and general purpose extensions to Java [35]. These approaches are similar in spirit to ours, however these tools do not offer the modular analysis that ensure that the composition will actually result in a working compiler or translator. They often do work just fine, though someone with knowledge of language design is often needed to be involved in the composition process. This differs from our approach in which we expect non-language experts to simply pick the extensions that they want to use and have a guarantee that they will, in fact, work together.

The matrix extension with explicit transformation specifications in Section V is based on work in Halide [19], [20], CHiLL [21], [22], and CFD Builder [23], [24]. But these systems have limited capabilities to be easily extended. Our aim is to generalize these techniques and implement them in an extensible manner so new transformation specifications can be easily added.

VIII. Conclusion

Many components of the matrix language extension shown above are based on MATLAB and the SAC language. Our aim here is not to claim that these language features are of our creation or that these are the best ones for this application, but to demonstrate that such features can be provided to programmers as modular, "plug-able", additions to their programming language. This is not possible with MATLAB and SAC as they are monolithic languages not designed to be extended.

Our hypothesis is that by giving programmers the freedom to choose the (set of) domain-specific features that fit their task at hand will provide a better solution than requiring programmers to pick the combination of monolithic, stand-alone languages that happen to have the language features that they feel that they need. Our work in progress reported here shows that interesting language features from existing scientific and high-performance languages can be added as composable language extensions.

Our future work is to add to this set of features in this extension with additional constructs useful in scientific and high-performance computing, and to provide additional optimizations of program computations and to the generated code. We are also interested in language extensions for parallel programming more generally, beyond matrix programming. To this end we are also developing a extension that adds Cilk [4] style parallelism constructs to C. The goal is to determine how sophisticated run-times, like in Cilk, can be delivered as a pluggable language extension.

We are now completing ABLEC, a specification of the full C11 language as a Silver attribute grammar (available at http://melt.cs.umn.edu/ableC). We will be porting these language extensions to use that host language specification. This will enable programmers to experiment with these kinds of language extensions on existing C applications. This allows for an incremental approach to using these tools. Programmers can easily experiment with an extension or two on an existing program without having to rewrite the entire program into another language. This ease of use is a significant advantage for extensible approaches and one we expect to leverage once the full C host language is complete and a rich set of extensions are available for use.

Acknowledgment

We thank Vipin Kumar, Micheal Steinbach, James Faghmous and the rest of the University of Minnesota Climate Data Mining group for their collaboration. This work is partially supported by NSF Awards #0905581 and #1047961.

References

[1] C. Grelck and S.-B. Scholz, "SAC: a functional array language for efficient multi-threaded execution," *Int. J. Parallel Programing*, vol. 34, no. 4, pp. 383–427, Aug. 2006.

[2] D. Cann and J. Feo, "SISAL versus FORTRAN: A comparison using the livermore loops," in *Proc. of ACM/IEEE Conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, pp. 626–636.

[3] G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein, and M. Zagha, "Implementation of a portable nested data-parallel language," in *Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*. ACM, 1993, pp. 102–111.

[4] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 1998, pp. 212–223.

[5] D. Gay, J. Galenson, M. Naik, and K. Yelick, "Yada: Straightforward parallel programming," *Parallel Comput.*, vol. 37, no. 9, pp. 592–609, Sep. 2011.

[6] M. Le, K. Williams, and E. Van Wyk, "A composable domain specic language extension for spatio-temporal data mining," in *Presented at the Workshop on Domain-Specific Languages and High-Level Frameworks for High-Performance Computing (WOLFHPC)*, 2013.

[7] D. E. Knuth, "Semantics of context-free languages," *Mathematical Systems Theory*, vol. 2, no. 2, pp. 127–145, 1968, corrections in **5**(1971) pp. 95–96.

[8] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan, "Silver: an extensible attribute grammar system," *Science of Computer Programming*, vol. 75, no. 1–2, pp. 39–54, January 2010.

[9] E. Van Wyk and A. Schwerdfeger, "Context-aware scanning for parsing extensible languages," in *Intl. Conf. on Generative Programming and Component Engineering, (GPCE)*. ACM, 2007, pp. 63–72.

[10] A. Aho, R. Sethi, and J. Ullman, *Compilers – Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[11] A. Schwerdfeger and E. Van Wyk, "Verifiable composition of deterministic grammars," in *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM, June 2009, pp. 199–210.

[12] T. Kaminski and E. Van Wyk, "Integrating attribute grammar and functional programming language features," in *Proc. of Intl. Conf. on Software Language Engineering (SLE)*, ser. LNCS, vol. 6940. Springer-Verlag, July 2011, pp. 263–282.

[13] Nasa earth observatory. [Online]. Available: http://earthobservatory.nasa.gov/Features/Eddies

[14] C. Grelck, "Shared memory multiprocessor support for functional array processing in sac," *J. Functional Programming*, vol. 15, no. 3, pp. 353–401, May 2005.

[15] Linux programmer's manual. [Online]. Available: http://man7.org/linux/man-pages/man3/malloc.3.html

[16] C. Grelck and S.-B. Scholz, "Efficient Heap Management for Declarative Data Parallel Programming on Multicores," in *3rd Workshop on Declarative Aspects of Multicore Programming (DAMP'08), San Francisco, CA, USA*. ACM Press, 2008, pp. 17–31.

[17] L. Fu, D. Chelton, P. L. Traon, and R. Marrow, "Eddy dynmaics from satellite altimetry," *Oceanography*, vol. 23, pp. 14–25, Feb 2010.

[18] J. Faghmous, M. Uluyol, L. Styles, M. Le, V. Mithal, S. Boriah, and V. Kumar, "Multiple hypothesis object tracking for unsupervised self-learning: An ocean eddy tracking application," in *Proc. 27th AAAI Conference on Artificial Intelligence*, 2013.

[19] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 2013, pp. 519–530.

[20] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 32:1–32:12, Jul. 2012.

[21] G. Rudy, M. Khan, M. Hall, C. Chen, and J. Chame, "A programming language interface to describe transformations and code generation," in *Proc. of Conf. on Languages and Compilers for Parallel Computing (LCPC)*. Springer-Verlag, 2010, pp. 136–150.

[22] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame, "A script-based autotuning compiler system to generate high-performance cuda code," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 31:1–31:25, Jan. 2013.

[23] J. Jayaraj, "A strategy for high performance in computational fluid dynamics," Ph.D. dissertation, University of Minnesota, Minneapolis, Minnesota, USA, 2013, available at http://purl.umn.edu/158483.

[24] P.-H. Lin, "Performance portability strategies for computational fluid dynamics (CFD) applications on hpc systems," Ph.D. dissertation, University of Minnesota, Minneapolis, Minnesota, USA, 2013, available at http://purl.umn.edu/155965.

[25] H. Vogt, S. D. Swierstra, and M. F. Kuiper, "Higher-order attribute grammars," in *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 1989, pp. 131–145.

[26] T. Kaminski and E. Van Wyk, "Modular well-definedness analysis for attribute grammars," in *Proc. of Intl. Conf. on Software Language Engineering (SLE)*, ser. LNCS, vol. 7745. Springer-Verlag, September 2012, pp. 352–371.

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

[28] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *SIGPLAN Notices*, vol. 40, no. 10, pp. 519–538, 2005.

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

[30] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls," in *Proc of ACM Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 2010, pp. 127–136.

[31] T. Ekman and G. Hedin, "The JastAdd system - modular extensible compiler construction," *Science of Computer Programming*, vol. 69, pp. 14–26, December 2007.

[32] ——, "The JastAdd extensible Java compiler," in *Proc. of ACM Conf. on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA)*. ACM, 2007, pp. 1–18.

[33] L. C. L. Kats and E. Visser, "The Spoofax language workbench. Rules for declarative specification of languages and IDEs," in *Proc. of ACM Conf. on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA)*. ACM, 2010.

[34] E. Visser, "Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9," in *Domain-Specific Program Generation*, ser. LNCS, no. 3061. Spinger-Verlag, June 2004, pp. 216–238.

[35] M. Bravenboer and E. Visser, "Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions," in *Proc. of ACM Conf. on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA)*. ACM, 2004, pp. 365–383.