AlphaZ and the Polyhedral Equational Model

Tomofumi Yuki
Colorado State University

Sanjay Rajopadhye
Colorado State University

I. Introduction

With the emergence of multi-core processors, parallelism has gone main-stream. However, parallel programming is difficult for many reasons. Programmers now must think about which operations can legally be executed in parallel, when to insert synchronizations, and so on. In addition, parallelism and non-determinism nature of it makes debugging much harder.

One approach to address this problem is automatic parallelization, where programmers still write sequential code, and it is left to the compiler to parallelize the program. However, automatic parallelization is extremely difficult, and even after decades of research it still remains largely unsolved. An alternative approach is to develop parallel programming languages, designed to write parallel programs from the beginning [1], [2], [3], [4], [5], [6].

One area where automatic parallelization has been successful is regular and dense computations that fit the polyhedral model. Although the applicable class of programs are restricted, fully automatic parallelization has been achieved for polyhedral programs. Approaches based on polyhedral analyses are now part of production compilers [7], [8], and many research tools [9], [10], [11], [12], [13], [14] that use the polyhedral model have been developed.

The polyhedral model has its origins in reasoning of systems of equations [15]. When programs are defined as equations, there is no notion of memory or execution order. What needs to be computed are defined purely in terms of values produced by other equations.

The connection between loop programs and equational representations was later made by Feautrier [16] through array dataflow analysis. Array dataflow analysis gives precise value-based dependence information, which can be used to take loop programs into equational view. This equational view is synonimous to the polyhedral representation of programs that the polyhedral compilers manipulate.

By giving concrete syntax to polyhedral representations of programs, we have an equational language to specify polyhedral programs. In this paper, we illustrate some of the benefits of equational programming, from perspectives of both users and compilers. We also present the AlphaZ [17] system that provides polyhedral analyses, transformations, and code generators for such equational language.

II. The Alpha Language

The equational language we use is a variation of the Alpha language [18]. After array dataflow analysis of an imperative program, the polyhedral representation of the flow dependences can be directly translated to an Alpha program. Furthermore, Alpha has reductions as first-class expressions [19] providing a richer representation. The language is mostly mathematical equations, with one key difference: the domains of all equations are explicitly given as polyhedral domains.

Polyhedral domains are how a set of operations (and other things such as memory) are abstracted in the polyhedral model. They are expressed as a polyhedron; a set of points constrained by affine inequalities; or union of finite number of polyhedra.

An Alpha program consists of one or more affine systems. The high-level structure of a system is the following:

```
affine <name> <parameter domain>
   input
      (<type> <name> <domain>;)*
   output
      (<type> <name> <domain>;)*
   local
      (<type> <name> <domain>;)*
   let
      (<name> = <expr>;)*
.
```

Each system corresponds to a System of Affine Recurrence Equations (SARE). The system consists of a name, a parameter domain, variable declarations, and equations that define values of the variables.

A. Parameter Domain

Polyhedral objects may involve program parameters that represent problem size (e.g., size of matrices) as symbolic parameters. Except for where the parameters are defined, Alpha parser treats parameters as implicit indices. For example, a 1D domain of size N is expressed as {i|0<=i<N}, and not {N,i|0<=i<N}.

B. Variable Declarations

Each variable in Alpha is specified a type and its domain. The specified domain should have a distinct point for each value computed throughout the program, including intermediate results. It is important not to confuse variables domains with memory, but rather as simply the set of points where the variable is defined. Some authors my find is useful to view this as single assignment memory allocation, where every memory location can only be written once.[1]

[1]We contend that so called, "single assignment" languages are actually zero-assignment languages. Functional language compilers almost always reuse storage, so nowhere does it make sense to use the term "single" assignment.

C. Reductions

Associative and commutative operators applied to collections of values are explicitly represented as reductions in `Alpha`. Reductions often occur in scientific computations, and have important performance implications. For example, efficient implementations of reductions are available in OpenMP or MPI. Moreover, reductions represent more precise information about the dependences, when compared to chains of dependences.

The reductions are expressed as $\texttt{reduce}(\oplus, f_p, \texttt{Expr})$, where op is the reduction operator, f_p is the projection function, and E is the expressions/values being reduced. The projection function f_p is an affine function that maps points in $\mathbb{Z}^n$ to $\mathbb{Z}^m$, where m is usually smaller than n. When multiple points in $\mathbb{Z}^n$ is mapped to a same point in $\mathbb{Z}^m$, those values are combined using the reduction operator. For example, commonly used mathematical notations such as

$$X_i = \sum_{j=0}^{n} A_{i,j}$$

is expressed as

$$X[i] = \texttt{reduce}(+, (i, j \to i), A[i, j])$$

This is more general than mathematical notations, since reductions with non-canonic projections, such as $(i, j \to i + j)$, require an additional variable to express with mathematical notations. For reductions with canonic projections, we use a shorthand that specify the names of the indices introduced within the reduction:

$$X[i] = \texttt{reduce}(+, [j], A[i, j])$$

III. Benefits of Equational Programming

Programmers benefit from two aspects when programming equationally:

- Separation of Concerns: The equational specification is completely detached from the choice of memory allocations, schedules, and/or other implementation details.
- Equations to Equations: The mathematical equations used to develop new methods can be directly translated to equational programs.

Compiler benefit from "cleaner" representation of the program. Polyhedral representations extracted from loop programs often have a number of complex boundary cases, influenced by implementation decisions of the loop programs. In addition, explicit representation of reductions can benefit both users and compilers, by providing higher abstraction.

A. Separation of Concerns

One strong benefit to programmers come from the equational language not having any notion of memory or execution order. The AlphaZ system accepts the equational language, and a separate specification for memory allocations, schedules, and other choices of implementation. Then it is the code generator's job to produce an efficient implementation that reflects these choices. Furthermore, since the equational view is identical to polyhedral representation of programs, some of these choices may be made using analyses from polyhedral compilation.

Thus, the equational specification is completely separated from implementation details. When programmers implement algorithms in C or FORTRAN, the algorithm quickly becomes entangled with implementation details: the choice of memory allocation, the schedule and so on. After spending some time in performance tuning, it requires a huge effort to change the algorithm itself.

Moreover, a specific implementation may not work for different platforms. For example, programs must be largely re-written if the target platform changes from multi-core processors to GPGPUs.

B. Equations to Equations

When the scientists develop new method for computing or modeling, it is usually developed in terms of mathematical equations. Our conjecture is that it is then straight forward to convert such equations into equational programs.

For example, take LU decomposition. Given an $n \times n$ matrix, A, we want to find two matrices L and U, lower and upper triangular respectively, such that $A = LU$. By definition, we have the following:

$$\begin{aligned} A_{i,j} &= \sum_{k=1}^{n} L_{i,k} U_{k,j} = \sum_{k=1}^{\min(i,j)} L_{i,k} U_{k,j} \\ &= \begin{cases} 1 = i \le j: & U_{i,j} \\ 1 < i \le j: & U_{i,j} + \sum_{k=1}^{i-1} L_{i,k} U_{k,j} \\ 1 = j < i: & L_{i,j} U_{j,j} \\ 1 < j < i: & L_{i,j} U_{j,j} + \sum_{k=1}^{j-1} L_{i,k} U_{k,j} \end{cases} \end{aligned}$$

Then, we apply standard algebra to "solve" for L and U to derive an algorithm:

$$L_{i,j} = \begin{cases} 1 = j < i: & \frac{A_{i,j}}{U_{j,j}} \\ 1 < j < i: & \frac{1}{U_{j,j}} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} U_{k,j} \right) \end{cases}$$

$$U_{i,j} = \begin{cases} 1 = i \le j: & A_{i,j} \\ 1 < i \le j: & A_{i,j} - \sum_{k=1}^{i-1} L_{i,k} U_{k,j} \end{cases}$$

The above can (almost) directly be translated into a corresponding `Alpha` program:

```
affine LUD {N|N>0}
input
   float A {i,j|1<=(i,j)<=N};
output
   float L {i,j|1<i<=N  && 1<=j<i};
   float U {i,j|1<=j<=N && 1<=i<=j};
let
   L[i,j] = case
      {|1==j} : A[i,j] / U[j,j];
      {|1<i}  : A -
         reduce(+, [k], L[i,k]*U[k,j]);
   esac;
   U[i,j] = case
      {|1==i} : A[i,j];
      {|1<i} : A[i,j] -
         reduce(+, [k], L[i,k]*U[k,j]);
   esac;
.
```

Note that the equations are identical to the mathematical equivalent, except that they are in a different syntax. The user is required to specify domains of variables. In the above, the matrix A is specified an $N \times N$ domain, and L and U are specified triangular domains.

Similar steps can be taken for more complicated algorithms. For example, RNA secondary structure prediction algorithm used in the UNAfold software package [20], can be specified in the `Alpha` language.

Although the class of programs that can be expressed as equations are limited, we believe that for those it makes sense to express as equations, equational language is a viable option.

C. Cleaner Representations

When polyhedral representations are extracted from loop programs, it may not be as "clean" as the equivalent specification written directly as equations. For example, Jacobi-style stencils are regular programs that can easily be represented in the polyhedral model. It is a type of stencil computations where an n-dimensional data is updated iteratively over time, and each update uses values from the previous time step.

When a programmer implements Jacobi stencils as loop programs, even the most naïve implementation would only use two copies of the data array. Then the array being read alternates each iteration, and so as the array being written. Such alternation may be implemented in many different ways, such as pointer swaps, unrolling of the time loop, explicit copying, and so on. Apart from some of them that make polyhedral analysis not applicable (e.g., pointer swaps), all of them complicate its corresponding polyhedral representations.

We may avoid such unnecessary complications, if the program is directly specified in equations. Extracted representations often reflect pre-mature optimizations, which sometimes can be harmful to the compiler and to the final performance.

D. Reductions

Explicit representation of reductions is beneficial for both users and compilers. For users, reductions commonly occur in mathematical equations, which can be directly represented in `Alpha`.

For compilers reductions, and its algebraic properties, can enable otherwise impossible optimizations. Algebraic properties can sometimes be used to perform extremely powerful optimizations, such as complexity reduction [17], [21]. Moreover, reductions are supported by commonly used parallelization methods, namely OpenMP and MPI, and compilers can take advantage of their efficient implementations.

IV. The AlphaZ System

The AlphaZ system is a system for exploring new approaches in polyhedral compilation [17], [22]. It accepts `Alpha` programs and also can process loop programs by first translating into `Alpha` via dataflow analysis.

AlphaZ aims to provide maximum control to the user, as opposed to other polyhedral tools that focus on full automation. Therefore, the user may specify a number of execution strategies, such as:

- Schedule: affine schedules
- Memory Allocation: pseudo-projective mapping (affine mapping + element-wise modulos)
- Parallel Loops: dimensions of schedules can be annotated as parallel dimensions
- Tiling: dimensions of schedules can be flagged to be tiled after code generation (post-processing is necessary for parametric tiling [23], [24].)

All of the above may be found through existing analysis on polyhedral programs, or be specified by the user.

These specifications are completely detached from the equational specification itself, and the user may try a number of different choices of the implementations by giving different specifications. We have code generators for C and C+OpenMP parallelization, and code generators for C+CUDA and C+MPI are currently being developed.

In addition to the code generators that require implementation details to be specified, we have a special code generator that generates code only from the `Alpha` program. The produced code executes in a demand-driven fashion, and is memory inefficient. However, this code generator may be used to test the program before figuring out the implementation choices.

A. Human in the Loop

In addition to specifying the execution strategies, AlphaZ provides a number of semantic preserving transformations for manipulating equations. By exposing such control to the user, domain specific knowledges and/or decisions guided by human analyses can be reflected, instead of relying on the compiler to do what the user wants.

For example, inlining of a reference to a value computed by another equation increases the amount of computation,

but may lead to better performance, by exposing additional parallelism for instance.

Such decisions are difficult to make for compilers, and AlphaZ encourages users to interact with the system to guide the compilation.

V. Sloppy Equations

One of the challenges in using the `Alpha` language is its preciseness. Contrary to our thought that the scientists would prefer an equational language over FORTRAN, some of the feedbacks we got were different. For the last few decades, the equations had to be implemented in C or FORTRAN in the end, and therefore, detailed boundary cases are often only worked out during the implementation.

When the equations are discussed in a paper, boundary conditions are omitted from the equations, since it usually does not help explaining the method. Thus, equations including all the boundary cases are sometimes not available.

The `Alpha` language requires every point in the domain of variables to be precisely defined, including all of the boundaries. Therefore, direct translation to `Alpha` is sometimes not possible.

For example, consider a Gauss-Seidel stencil computation over 1D data:

```
for (t=0; t < T; t++)
   for (i=1; i < N; i++)
      A[i] = foo(A[i-1], A[i], A[i+1]);
```

The corresponding precise equational specification is the following:

$$A_{t,i} = \begin{cases} t = 0 & : Ain_i \\ t > 0 \wedge 0 < i < N & : foo(A_{i-1}, A_i, A_{i+1}) \\ t > 0 \wedge i = 0 & : Ain_i \\ t > 0 \wedge i = N & : Ain_i \end{cases}$$

Note the case branches for initialization and boundaries on the data. Since the loop program does not update the boundaries, the corresponding cases of the equation always depend on the input. With a one-dimensional domain, there are only two boundaries, but with higher dimensions and more complicated dependences, the number of boundary cases can quickly grow. Specifying all of such boundaries is tedious and error prone, and we would like to relieve the user from such process.

One of the ideas we have in response to this challenge is called sloppy equations. The equations in `Alpha` are required to be precise, but we would like to allow *sloppy* specifications, and then deduce the precise specification using polyhedral analysis.

One example of sloppiness is default behavior. We allow one of the case branches to have a keyword `default` instead of constraints, and it becomes the default behavior for the set of points with no other matching case. Then the 1D Gauss-Seidel can be written as:

$$A_{t,i} = \begin{cases} t > 0 \wedge 0 < i < N & : foo(A_{i-1}, A_i, A_{i+1}) \\ \texttt{default} & : Ain_i \end{cases}$$

The precise domain of `default` can be analyzed from the domain of A and other case branches. The analysis result itself may be useful for the scientists in implementing in other languages, even if they choose not to use AlphaZ generated code.

Although the addition of sloppiness to `Alpha` language is still experimental, and is currently not implemented, we believe that such addition will significantly increase the usability of our equational language.

VI. Conclusions

In this paper, we have presented the equational view of the polyhedral model. It is essentially intermediate representations of polyhedral compilers with an equational concrete syntax. There are benefits to having such an equational programming language, both for programmers and for compilers.

By having an equational specification that only specify what to compute, without any implementation details, complete separation of the algorithm and implementation is achieved. Moreover, convenient abstractions, such as reductions, can easily be specified and utilized.

We believe that equational languages can be a viable DSL for specifying polyhedral programs, such as stencils.

References

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. Maessen, S. Ryu, G. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund *et al.*, "The Fortress Language Specification," *Sun Microsystems*, vol. 139, p. 140, 2005.

[2] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[3] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, Aug. 1998.

[4] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove, "X10 language specification version 2.2," Mar. 2012, x10.sourceforge.net/documentation/languagespec/x10-latest.pdf.

[5] UPC Consortium et al., "UPC language specifications," *Lawrence Berkeley National Lab Tech Report LBNL–59208*, 2005.

[6] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella *et al.*, "Titanium: A high-performance Java dialect," *Concurrency Practice and Experience*, vol. 10, no. 11-13, pp. 825–836, 1998.

[7] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L. Pouchet, "Polly–Polyhedral optimization in LLVM," in *1st International Workshop on Polyhedral Compilation Techniques*, 2011.

[8] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G. Silber, and N. Vasilache, "GRAPHITE: Polyhedral analyses and optimizations for GCC," in *Proceedings of the 2006 GCC Developers Summit*, 2006.

[9] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2008, pp. 101–113.

[10] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," Technical Report, University of Southern California, Tech. Rep., 2008.

[11] F. Irigoin, P. Jouvelot, and R. Triolet, "Semantical interprocedural parallelization: An overview of the pips project," in *Proceedings of the 5th International Conference on Supercomputing*. ACM, 1991, pp. 244–251.

[12] C. IRISA, "The MMAlpha environment."
[13] B. Meister, A. Leung, N. Vasilache, D. Wohlford, C. Bastoul, and R. Lethin, "Productivity via automatic code generation for PGAS platforms with the R-Stream compiler," in *Proceedings of the Workshop on Asynchrony in the PGAS Programming Model*, 2009.
[14] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan, "Hybrid iterative and model-driven optimization in the polyhedral model," INRIA Research Report, Tech. Rep. 6962, June 2009.
[15] R. Karp, R. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *Journal of the ACM*, vol. 14, no. 3, pp. 563–590, 1967.
[16] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.
[17] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye, "Alphaz: A system for design space exploration in the polyhedral model," in *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing*, 2012.
[18] C. Mauras, "ALPHA: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones," Ph.D. dissertation, L'Université de Rennes I, IRISA, Campus de Beaulieu, Rennes, France, December 1989.
[19] H. Le Verge, "Reduction operators in alpha," in *Parallel Algorithms and Architectures, Europe*, ser. LNCS, D. Etiemble and J.-C. Syre, Eds. Paris: Springer Verlag, June 1992, pp. 397–411, see also, Le Verge Thesis (in French).
[20] N. Markham and M. Zuker, "Software for nucleic acid folding and hybridization," *Methods in Molecular Biology*, vol. 453, pp. 3–31, 2008.
[21] T. Yuki, G. Gupta, T. Pathan, and S. Rajopadhye, "Systematic implementation of fast-i-loop in UNAfold using AlphaZ," Technical Report CS-12-102, Colorado State University, Tech. Rep., 2012.
[22] T. Yuki, V. Basupalli, G. Gupta, G. Iooss, D. Kim, T. Pathan, P. Srinivasa, Y. Zou, and S. Rajopadhye, "Alphaz: A system for analysis, transformation, and code generation in the polyhedral equational model," Technical Report CS-12-101, Colorado State University, Tech. Rep., 2012.
[23] A. Hartono, M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Dyntile: Parametric tiled loop generation for parallel execution on multicore processors," in *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2010, pp. 1–12.
[24] D. Kim, "Parameterized and multi-level tiled loop generation," Ph.D. dissertation, Fort Collins, CO, USA, 2010.