Foundations I

Sanjay Rajopadhye

Contents

1 Introduction & Motivation

These notes describe the mathematical foundations of some of the the important concepts that we encounter in this class. In this first part we will motivate equational programming in the polyhedral model through a number of examples using the Alphabets language. Then, we present the core of Alphabets as systems of recurrence equations and give a taxonomy of such equations. We will use the AlphaZ tool, in particular the WriteC code generator to see how our equations serve as executable, high level specifications of a very important class of computations (four of the thirteen "Berkeley dwarfs").

1.1 Why Equations?

A large class of algorithms, especially those of interest to this class, can be described cleanly and concisely as mathematical equations. Here are a few examples.

Forward Substitution Given an $n \times n$ lower triangular matrix L (with unit diagonals), and an m-vector, b, solve for the vector x in $Lx = b$.

Let us write down the definition of matrix-vector product:

$$b_i = \sum_{j=1}^{n} L_{i,j} x_j$$

But since L is lower triangular, every row "ends" at the diagonal, so the summation can be truncated at i, as follows:

$$b_i = \sum_{j=1}^{i} L_{i,j} x_j$$

Now we take the "last term outside the summation" to get (after some obvious simplification):

$$b_i = \begin{cases} i = 1 & : \quad x_i \\ i > 1 & : \quad x_i + \sum_{j=1}^{i-1} L_{i,j} x_j \end{cases}$$

Now we want to use this equation to "solve" for x. We do this simply by "taking x to the left hand side (lhs)," yielding:

$$x_i = \begin{cases} i = 1 & : \quad b_i \\ i > 1 & : \quad b_i - \sum_{j=1}^{i-1} L_{i,j} x_j \end{cases} \tag{1}$$

Is this a *program* to solve a lower triangular system of equations?

Back Substitution Given an $n \times n$ *upper* triangular matrix U and an m-vector, b, solve for the vector x in $x = b$. We do the same sequence of reasoning:

$$b_i = \sum_{j=1}^{n} U_{i,j} x_j = \sum_{j=i}^{n} U_{i,j} x_j = \begin{cases} i = n & : \quad U_{i,i} x_i \\ i < n & : \quad U_{i,i} x_i + \sum_{j=i+1}^{n} U_{i,j} x_j \end{cases}$$

Hence,

$$x_i = \begin{cases} i = n & : \quad \dfrac{b_i}{U_{i,i}} \\ i < n & : \quad \dfrac{1}{U_{i,i}} \left(b_i - \sum_{j=i+1}^{n} U_{i,j} x_j \right) \end{cases} \tag{2}$$

Is this a program to solve an upper triangular system of equations? Only if U is non-singular, i.e., its diagonal diagonal elements are non-zero.

LU Decomposition Given an $n \times n$ matrix, A, determine two matrices L and U (L is lower triangular with unit diagonal, and U is upper triangular), such that $A = LU$.

By definition,

$$A_{i,j} = \sum_{k=1}^{n} L_{i,k}U_{k,j} \quad = \quad \sum_{k=1}^{\min(i,j)} L_{i,k}U_{k,j} = \begin{cases} i \le j \ : \ \sum_{k=1}^{i} L_{i,k}U_{k,j} \\ i > j \ : \ \sum_{k=1}^{j} L_{i,k}U_{k,j} \end{cases}$$

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

So taking L and U to the left hand side, we can "solve" for these unknowns.

$$U_{i,j} \quad = \quad \begin{cases} 1 = i \le j : A_{i,j} \\ 1 < i \le j : A_{i,j} - \sum_{k=1}^{i-1} L_{i,k}U_{k,j} \end{cases} \tag{3}$$

$$L_{i,j} \quad = \quad \begin{cases} 1 = j < i : \dfrac{A_{i,j}}{U_{j,j}} \\ 1 < j < i : \dfrac{1}{U_{j,j}} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k}U_{k,j} \right) \end{cases} \tag{4}$$

Is this a *program* to factor a matrix into "L-U" factors? Are any additional conditions necessary?

Unbounded Knapsack Dynamic Programming Given an n types of objects, each one with a weight w_i and profit p_i, a knapsack of capacity c, and an unbounded supply of each object type, choose an integer number x_i of each type of object so as to maximize the profit without exceeding the capacity, i.e., maximize $\sum_{j=1}^{n} p_i x_i$ subject to the constraints $\sum_{j=1}^{n} w_i x_i \le c; x_i \ge 0$.

Dynamic programming can be used to solve this problem as follows. Let us define a function $F(j)$ that gives the maximum profit that can be obtained with capacity j. Assume that some oracle tells us that the object type i contributes at least once to this optimum. Now, let us leave aside this copy, and consider the way the remainder of the knapsack has been filled up. By the principle of optimality (subproblems of the optimal solution must also be solved optimally), the profit achieved by this must be $F(j - w_i)$. Hence, $F(j) = p_i + F(j - w_i)$.

Unfortunately, we don't have such an oracle, so we don't know whether object type i has contributed to $F(j)$. Hence, we must consider all possible candidates. This leads to the recurrence (we omit the "boundary conditions" for brevity).

$$F(j) = \max_{i=1}^{n} [p_i + F(j - w_i)] \tag{5}$$

Problem: Optimal String Parenthesization We are given a sequence of characters, $S_0 \ldots S_n$, and we want to construct a binary tree with the S_i's as leaves (and hence n internal nodes). Each tree can be viewed as a parenthesization of the string: the root splits the string into a prefix $S_0 \ldots S_k$, and a suffix $S_{k+1} \ldots S_n$ at a point k, and each of the are recursively parenthesized. Consider a substring $S_i \ldots S_j$ parenthesized with the outermost parentheses at k. Let the cost of this outermost set of parentheses be some function $h(i, k, j)$ of the three integers i, j and k that uniquely identify it. Thus the cost of a tree is the cost of these outer parentheses, plus the sum of the costs of its subtrees. Develop an algorithm that determines the the optimal, i.e., minimum cost parenthesization.

Line of Sight This is a (simplified version of) a common problem that occurs in geographical information systems (GIS). Say, you are given altitude data for a rectangular region, i.e., $Z[i, j]$ represents the altitude of point $\langle i, j\rangle$. Assume that the scale of discretization is x. The sun is shining from the west, and subtends an angle θ. We want to determine a Boolean value at each point that determines whether or not it is in the sunlight, i.e., whether or not some point to its west (i.e., a point $\langle i', j\rangle$ with the same j value, and $i' < i$) occludes the sun. The following equation specifies the desired answer.[1]

$$S[i,j] = \begin{cases} i = 1 : \text{true} \\ i > 1 : \bigwedge_{i'=1}^{i-1} \left(\tan\theta > \dfrac{Z[i',j]}{(i-i') * x} \right) \end{cases} \tag{6}$$

1.2 Equations as programs

We have made the case that the mathematical reasoning needed to solve many problems leads to equations, and these equations are in some sense, a very high level specification of an algorithm. The next step would be to actually make these equations the program itself. It is therefore useful to "codify" these equations so that they define a programming language (or rather, a sublanguage, since not everything that we want to program—such as file I/O—can be written as equations). Before we do this however, let us recap some of the advantages and issues that arise when viewing equations as programs.

A primary advantage is that equations are amenable to formal resoning: we can prove properties of equations, analyze the dependence properties, and as we shall see later on, determine how to parallelize them, and how, starting from equations, to generate code (which may be sequential or parallel) or even hardware descriptions (VHDL) of parallel circuits that can be implemented on FPGA platforms. We now illustrate some examples of such reasoning.

Program complexity Consider the example for foward substitution (Eqn. 1). Notice that for an $n \times n$ matrix, L, there are n values of x that need to be determined (for $i = 1 \ldots n$, the subscripts on the lhs). Further, for any given i, the values that j can take can be deduced from the bounds of the summation: $j = 1 \ldots (i-1)$. Hence the set of "index values" where we need to do an "elementary" computation (here a multiply-accumulate) can be viewed as a triangle defined by $\{\langle i,j\rangle \mid 1 \le j < i \le n\}$. There are about $\frac{1}{2}n^2$ points in the triangle, so the complexity of our "equational program" for forward substitution is $\Theta(n^2)$.

Similarly, the complexity of the backward substitution (Eqn. 2) is also $\Theta(n^2)$, and that of LU decomposition (Eqns. 3-4) is $\Theta(n^3)$ (more precisely, there are $\frac{1}{3}n^3$ integer points in its "pyramid shaped domain." Similarly, the complexity of Eqn. 6 is also $\Theta(n^3)$ since there are $\frac{1}{2}n^3$ integer points in its domain.

[1]The equation contains a (deliberate) error.

Program Simplification Because equations are mathematical objects, we can use mathematical reasoning about them and either manipulate, and/or transform them. An impressive instance of such reasoning is when we are able to obtain significant savings in the computations: not just by constant factors, but in terms of asymptotic complexity. Consider the equation

$$X_i = \sum_{j=1}^{n}\sum_{k=1}^{n} A_{i,k} * B_{k,j}$$

At first glance, it seeems to have $\Theta(n^3)$ complexity, since we need to compute n answers, each with a double summation. However, if we simply reverse the two summation indices, then note that the first term $A_{i,k}$ is independent of index, j, we can "pull it out" of the inner sumation (because multiplication distributes over addition), giving us:

$$X_i = \sum_{k=1}^{n}\sum_{j=1}^{n} A_{i,k} * B_{k,j} = \sum_{k=1}^{n} A_{i,k} * \left(\sum_{j=1}^{n} B_{k,j}\right)$$

Now, if we let Y_k be the result of the inner summation, we get

$$Y_k = \sum_{j=1}^{n} B_{k,j} \tag{7}$$

$$X_i = \sum_{k=1}^{n} A_{i,k} * Y_k \tag{8}$$

This is a new, equivalent system of equations, each of which can be computed with $\Theta(n^2)$ complexity. We have thus reduced the complexity of our program through "equational reasoning," simply by using a widely known algebraic property, namely distributivity.

Another program simplification technique is through the detection of *scans* (also called *prefix computations*). Consider the equation $Y_i = \sum_{j=1}^{i} X_j$ which describes the computation of n answers, each one requiring the sum of $\Theta(n)$ values, leading to an apparent complexity of $\Theta(n^2)$. However, if we observe closely, each answer, Y_i is just the sum of all the values of X "before" it, i.e., from 1 to i inclusive. Hence, this is a "prefix" computation, and the following equation which has only linear complexity[2] is equivalent.

$$Y_i = \begin{cases} i = 1 & : \quad X_i \\ i > 1 & : \quad Y_{i-1} + X_i \end{cases}$$

These two techniques for equation simplification are extremely powerful: we have just seen simple (you might say trivial) examples.

Before reading any further, please take some time to see if you can simplify the line-of-sight computation above (Eqn. 6) to quadratic complexity (if you try to do it now before looking at the solution below, it will help you on one of the homework problems).

Here's an outline of the solution. Consider the constraint that all the points to the west of $\langle i, j \rangle$ subtend an angle whose tangent is less than $\tan\theta$. We first rewrite it as $ix\tan\theta > Z[i', j] + i'x\tan\theta$ where we have moved all the terms depending on i' to the right of the inequality, and those depending on i to the left. Now, saying that this inequality is true at all i' in a certain range is equavalent to stating that the *maximum* value of the right hand side (rhs) of the inequality is still smaller than $ix\tan\theta$, i.e.,

$$S[i,j] = \begin{cases} i = 1 & : \quad \text{true} \\ i > 1 & : \quad ix\tan\theta > \max_{i'=1}^{i-1}(Z[i', j] + i'x\tan\theta) \end{cases} \tag{9}$$

Now, let us introduce a new variable, $W[i,j] = Z[i,j] + i\tan\theta$. Then we notice that the term inside the $\max$ above is $W[i', j]$. Hence $\max_{i'=1}^{i-1} W[i', j]$, is simply the scan (using $\max$ as the operator) of each row of W (and shifted to the right by one, since the upper bound is $i-1$, rather than i). This can be computed in $\Theta(n)$time for each row, i.e., a total complexity of $\Theta(n^2)$. Using this, $S[i,j]$ can also be computed in $\Theta(n^2)$ time.

PDEs (Heat Equation) The heat equation is an important partial differential equation which describes the variation of temperature in a given region over time. The following special case when the region is one-dimensional is described by the following law.

$$\frac{\partial u}{\partial t} = k\frac{\partial^2 u}{\partial x^2} \tag{10}$$

[2]Although we will not discuss this further here, such computations can also be very effectively parallelized to run in linear time in a scaled manner (this is despite the apparently inherent sequentiality in the computation specified by the new equation)

We shall discretize the x as well as the t dimensions, and view the physical quantity U as being defined over a two dimensional index space, $\{i, t | 0 \le i \le N; 0 \le t \le T\}$ with a discretization of Δt and Δx. Let us write the discretized version of the heat equation, by first defining the discretized approximations of the two derivatives, and substituting them into Eqn.(10). We have some choices in this, and we will how they induce some subtle differences.

First, we could write $\left.\frac{\partial u}{\partial t}\right|_{(i,t)} \approx \frac{U[i,t] - U[i,t-1]}{\Delta t}$ using what is called as the "backward-looking" approximation, or as $\left.\frac{\partial u}{\partial t}\right|_{(i,t)} \approx \frac{U[i,t+1] - U[i,t]}{\Delta t}$. Similarly, the partial derivative with respect to space could be written as either $\left.\frac{\partial u}{\partial x}\right|_{(i,t)} \approx \frac{U[i,t] - U[i-1,t]}{\Delta x}$ or as $\left.\frac{\partial u}{\partial x}\right|_{(i,t)} \approx \frac{U[i+1,t] - U[i,t]}{\Delta x}$. The second spatial derivative is given by $\left.\frac{\partial^2 u}{\partial x^2}\right|_{(i,t)} \approx \frac{\left.\frac{\partial u}{\partial x}\right|_{(i,t)} - \left.\frac{\partial u}{\partial x}\right|_{(i-1,t)}}{\Delta x}$ (backward looking) or $\left.\frac{\partial^2 u}{\partial x^2}\right|_{(i,t)} \approx \frac{\left.\frac{\partial u}{\partial x}\right|_{(i+1,t)} - \left.\frac{\partial u}{\partial x}\right|_{(i,t)}}{\Delta x}$ (forward) with the caveat that the first derivative used *inside* this formula should be the *opposite* of the choice of (forward or backward) for the second derivative[3]. Thus we have two alternate formulas for each of $\frac{\partial^2 u}{\partial x^2}(i,j)$ and $\frac{\partial u}{\partial t}(i,j)$. With one choice, $\frac{\partial^2 u}{\partial x^2}(i,j)$ is as follows[4]:

$$\begin{aligned} \left.\frac{\partial^2 u}{\partial x^2}\right|_{(i,t)} &= \frac{\frac{U[i+1,t]-U[i,t]}{\Delta x} - \frac{U[i,t]-U[i-1,t]}{\Delta x}}{\Delta x} \\ &= \frac{1}{\Delta x^2}\left(U[i+1,t] - 2U[i,t] + U[i-1,t]\right) \qquad (11) \end{aligned}$$

Plugging this and the forward looking approximation for $\left.\frac{\partial u}{\partial t}\right|_{(i,j)}$ into Eqn. 10 we obtain, after a bit of rearrangement, and using $\rho = \frac{k\Delta t}{\Delta x^2}$

$$U[i,t+1] = \rho U[i+1,t] - (2\rho - 1)U[i,t] + \rho U[i-1,t] \qquad (12)$$

[3]This may seem counter-intuitive at first glance, but it ensures that the resolution of the discretization is maintained. Exercise: Show that if both the derivatives are backward looking, then $\left.\frac{\partial^2 u}{\partial x^2}\right|_{(i,j)}$ will depend on $u[i-2,t]$ and if both are forward looking, it will depend on $u[i+2,t]$.

[4]The other choice is left as an exercise.

We find it convenient to replace the $t+1$ and the t indices by, respectively, t and $t-1$, yielding

$$U[i,t] = \rho U[i+1,t-1] - (2\rho-1)U[i,t-1] + \rho U[i-1,t-1] \tag{13}$$

In practice, the heat equation and many other partial differential equations are the basis of numerical simulations: the *initial conditions* (i.e., the values of $U[i,t]$ for $t=0$) and *boundary conditions* (values of $U[i,t]$ at $i=0$ and at $i=N$) are given, and we desire to compute the *final conditions* (values of $U[i,T]$ for $i=1\ldots N$), and possibly, the values of $U[i,t]$ at some, or all, intermediate time steps.

Note that since the time index on the lhs of Eqn. 13 is t and on the rhs it is $t-1$, we can directly use this equation to specify a program: the expression on the rhs is viewed as the rule to determine the value on the lhs. Effectively, we treat the rhs as "known" values, and the lhs as the unknowns. In fact, this equation specifies the well known "explicit method" for solving PDEs.

On the other hand, let us see what happens if we combine our formula for the second spatial partial derivative, with the *backward looking* approximation for the temporal partial derivative, i.e., combining Eqn. 11 with $\left.\frac{\partial u}{\partial t}\right|_{(i,t)} \approx \frac{U[i,t]-U[i,t-1]}{\Delta t}$. We obtain, after the usual simplification,

$$\rho U[i+1,t] - (2\rho+1)U[i,t] + \rho U[i-1,t] = -U[i,t-1] \tag{14}$$

where we have again attempted to take the terms involving t on the lhs and those involving $t-1$ to the rhs.Since the lhs is not a single term but an *expression*, this equation cannot be directly used as a program. But maybe, if we simply rewrote the equation leaving only one term on the lhs, say as

$$U[i+1,t] = \frac{2\rho+1}{\rho}U[i,t] - U[i-1,t] - \frac{1}{\rho}U[i,t-1] \tag{15}$$

could we now treat this as an equation. Do you see any problems? Hold on to these and other questions. We will come back to them later on.

2 Recurrence Equations

In what follows, $\mathcal{Z}$ denotes the set of integers, and $\mathcal{N}$ the set of natural numbers.

Definition 1 *A **Recurrence Equation** defining a function (variable) X at all points, z, in a domain, D, is an equation of the form*

$$X[z] = D^X \quad : \quad g(\ldots X[f(z)]\ldots) \tag{16}$$

where

- *z is an n-dimensional **index variable**.*
- *X is an "n-dimensional" **data variable**. There a couple of equivalent alternative ways to view X. It can be thought of as an n-dimensional array whose values at all $z \in D^X$ are implicitly defined by the equation; it may also be seen as a function of n integer arguments.*
- *$f(z)$ is a **dependency function** (also called an **index** or **access** function), $f : \mathcal{Z}^n \to \mathcal{Z}^n$;*
- *the "$\ldots$" indicate that g may have other arguments, each with the same syntax;*
- *g is a strict, single-valued function; it is often written implicitly as an* expression *involving operands of the form $X[f(z)]$ combined with basic operators and parentheses. Note that for analysis purposes, g is considered atomic (i.e., executing in a single step) unless it has a* reduction *(as defined later). If it has a reduction it may or may not be considered atomic, depending on the assumptions of the machine model used for the analysis.*
- *D^X is a set of points in $\mathcal{Z}^n$ and is called the **domain** of the equation. Domains are often polyhedral index spaces, parameterized with one or more, say s size parameters. The parameters are viewed as an s-dimensional vector p.*

A variable may be defined by more than one equation. In this case, we use the syntax shown below:

$$X[z] = \begin{cases} & \vdots \\ D_i & : \quad g_i(\ldots X[f(z)]\ldots) \\ & \vdots \end{cases} \tag{17}$$

*Each line is called a **case**, and the domain of X is the union of the (disjoint) domains of all the cases, $D^X = \bigcup_i D_i$.*

Definition 2 *A recurrence equation (16) as defined above, is called an* ***Affine Recurrence Equation*** *(*ARE*) if every dependence function is of the form,* $f(z) = Az+Bp+a$*, where* A *(respectively* B*) is a constant* $n \times n$ *(respectively,* $n \times l$*) matrix and* a *is a constant* n*-vector. It is said to be a* ***Uniform Recurrence Equation*** *(*URE*) if it is of the form,* $f(z) = z + a$*, where* a *is a constant* n*-dimensional vector, called the dependence vector.* URE*s are a proper subset of* ARE*s, where* A *is the identity matrix and* $B = 0$*.*

Definition 3 *A* ***system*** *of recurrence equations (*SRE*) is a set of* m *such equations, defining the data variables* $X_1 \ldots X_m$*. Each variable,* X_i *is of dimension* n_i*, and since the equations may now be mutually recursive, the dependence functions* f *must now have the appropriate type.*

Reductions

The key difficulty you must have encountered is that we have no syntax for the *reduction* operations: associative and commutative operators like addition, multiplication, max, etc., applied to a collection of values.

We will now introduce a simple, yet powerful syntax for this. We simply allow the function g to have the form, $\texttt{reduce}(\texttt{op}, f', \texttt{expr})$. Here,

- op is an associative and commutative operator;
- expr is an expression (it is most convenient to assume that the expression is just a new variable, Y, and to assume that there is an equation $Y = \texttt{expr}$ defined over an appropriate domain D^Y);
- f' is a many-to-one mapping from indices to indices, usually it maps $\mathcal{Z}^n$ to $\mathcal{Z}^{n-k}$ (where the expr is n-dimensional).

Consider a reduction equation as follows.

$$X(z) = \texttt{reduce}(\texttt{op}, f', Y)$$

Its semantics can be explained as follows. X is defined over a domain D^X which is the image of D^Y by the function f' (this implies D^Y is $n-k$-dimensional). Because f' is many-to-one, each $z \in D^X$ is the image of many points $z' \in D^Y$. The reduce expression states that the value of X at any point z is obtained by applying op to the values of Y at all the z' that are mapped by

f' to z (this is a mouthful; please read each word carefully to make sure you understand what this says).

With this explanation, we can now write an SRE for the forward substitution example:

$$x[i] = \begin{cases} i = 1 & : \quad b[i] \\ i > 1 & : \quad b[i] - \texttt{reduce}(+, (i,j \rightarrow i), T[i,j]) \end{cases} \tag{18}$$

$$T[i,j] = \{i,j \mid 1 \leq j < i \leq n\} : L_{i,j}x_j \tag{19}$$

Taxonomy of Recurrence Equations

As we have seen above, recurrence equations may be classified along many aspects:

- single or system;
- class of dependence functions: arbitrary, affine or uniform;
- parameterized domains or single domain;
- class of domains over which they are defined.