GLU Programmer's Guide

Version 0.9 – July 1994

R. Jagannathan
Chris Dodd
Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, California 94025, U.S.A.

email: {jaggan,dodd}@csl.sri.com

Contents

1 Introduction **1**
- 1.1 What is GLU? 1
- 1.2 How GLU compares with PVM and Linda? 1
- 1.3 How to Read this Guide? 2

2 Tutorial Introduction **3**
- 2.1 LUCID – Declarative Nucleus of GLU 3
- 2.2 GLU – A Procedural Extension to LUCID 4

3 LUCID **13**
- 3.1 Types 13
- 3.2 Constants 13
- 3.3 Variables 13
- 3.4 Operators 14
  - 3.4.1 Arithmetic, Relational and Logical Operators 14
  - 3.4.2 Operators `first` and `next` 14
  - 3.4.3 Operator `fby` 15
  - 3.4.4 `@` and `#` Operators 15
  - 3.4.5 `if-then-else` Operator 15
  - 3.4.6 Operators `asa`, `wvr`, `upon` 16
  - 3.4.7 Operator `iseod` 17
  - 3.4.8 Operator `iserror` 17
- 3.5 Extra Operators 18
  - 3.5.1 Operator ‘`,`’ 18
  - 3.5.2 Operator ‘`!`’ 18
  - 3.5.3 Operator ‘`?`’ 18
- 3.6 Expressions 18
- 3.7 User-defined Functions 19
- 3.8 Nested Computations 20
- 3.9 Array Computations 22
- 3.10 Dimensional Abstraction 23
- 3.11 Operator Arity 26
- 3.12 Default Dimension Rule 26

4 From LUCID to GLU **27**
- 4.1 C Data Types 27
- 4.2 C Prototype Function Declarations 28
- 4.3 C Function Definition Conventions 29

4.3.1 Parameter Passing 29
4.3.2 Returning Dynamic Objects 29
4.3.3 Globals 30
4.3.4 Static Variables 30
4.4 Preprocessor 30
4.5 Comments 30
4.6 Annotations 30
4.6.1 Local versus Remote Function Execution 30
4.6.2 Remote Execution Mapping 31

5 Compilation and Execution **32**
5.1 Source to Executables 32
5.2 Running Program Executables 33

Bibliography **35**

A Syntax of GLU Programs **36**
A.1 Scope Rules 36
A.1.1 Program Variables 36
A.1.2 Dimensional Names 36
A.1.3 Precedence and Associativity Rules 36
A.1.4 Reserved Words 37
A.2 Extended BNF Syntax for GLU Programs 37

B Predefined Functions **39**

C Manual Pages for GLU Commands **40**
`gc` – GLU compiler 41
`glu` – GLU runtime system 43

Chapter 1

Introduction

This report is a programmer's guide to GLU (Granular LUCID). It is written for those interested in developing high-performance application software on a variety of platforms. It assumes that readers are well-versed in procedural programming and have a basic understanding of parallel and distributed programming.

1.1 What is GLU?

GLU is a very-high-level system for constructing and executing high-performance applications for diverse platforms ranging from heterogeneous workstation clusters to shared-memory multiprocessors to massively parallel processors.

In GLU, a high-performance application is a LUCID[1] composition of sequential functions expressed in the C language[2] . GLU allows code fragments from existing sequential applications to be reused and only a small amount of new code to be written. Applications expressed in GLU are implicitly parallel and inherently adaptive. In addition, GLU applications are highly portable across multiple platforms.

1.2 How GLU compares with PVM and Linda?

It is useful to briefly compare and contrast GLU with other systems for constructing and executing high-performance applications such as PVM [1] and Linda [2].

While both PVM and Linda offer communication and synchronization support for interacting autonomous processes, the burden of coming up with a process architecture to implement the application in parallel is on the programmer. A process architecture is a set of interacting processes in a particular arrangement that reflects the manner in which the application has been partitioned and the granularity of parallelism. With both PVM and Linda, process architecture for an application is developed in an *ad hoc* manner by the programmer. Whereas with GLU, both the partitioning strategy and granularity of parallelism are integrated into the composition specification from which the process architecture is automatically generated.

With PVM, it is also necessary for the programmer to specify how the process architecture will be *mapped* onto a target system and, when applicable, how *load balancing* will be conducted. This is likely to vary across different target architectures. With Linda, both mapping and load balancing are addressed by the system as the programmer assumes a simple shared-memory programming model.

[1]LUCID is a multidimensional dataflow language.

[2]Whereas C is the procedural language we currently support, we plan to directly support other languages.

With GLU, the mapping and load balancing strategies are integrated into the composition specification itself. Thus, they are automatically incorporated into the application process architecture.

One of the unique natural strengths of GLU is adaptivity . That is, GLU applications can tolerate failure of individual processors or links as well as adapt during execution to using additional processors.

The best way to contrast GLU with PVM and Linda is to think of it as a much higher level programming system. In fact, it is possible, in principle, to implement GLU using PVM or Linda.

1.3 How to Read this Guide?

The readers are encouraged to read Chapter 2 first where a quick tour of GLU programming is given. Chapter 3 describes the language LUCID which is at the heart of GLU. Chapter 4 describes GLU as a hybrid of LUCID and C. Chapter 5 discusses the compilation and execution process for GLU programs. The rest of the guide consists of appendices. Appendix A gives a BNF description of GLU, scope rules, and operator precedence. Appendix B describes standard (predefined) GLU functions that are useful in constructing GLU applications. Appendix C contains "man pages" for the compiler and the runtime system.

Chapter 2

Tutorial Introduction

The purpose of this chapter is to show the essential elements of GLU by example, without getting mired in the precise details of the language. Whereas reading only this chapter will not be sufficient to program effectively in GLU, it will help in understanding the GLU programming model and also will help in writing some simple yet useful programs. The chapters that follow give a more thorough description of GLU.

Programming in GLU is natural, but it requires a shift in the way one thinks of programming, which often is imperative. There are two ways to think of the GLU model of programming. The first way is as a hybrid of declarative (LUCID) programming in which one specifies *what* to compute and imperative (C) programming in which one specifies *how* to compute. The second way to think of GLU is as a procedurally-extended declarative programming language , i.e., LUCID with C extensions. Both views are valid – the first suits those who are familiar with procedural programming while the latter suits those who are familiar with declarative programming . We appeal to both these views, as appropriate, in the rest of the chapter.

2.1 LUCID – Declarative Nucleus of GLU

Let us briefly consider the declarative part of GLU. It corresponds to the language LUCID (circa 1994) which is both a functional and indexical[1] language.

We distinguish between functional and indexical programming by writing two LUCID programs for computing the Fibonacci sequence , which is defined mathematically as follows:

$$\texttt{fib}_i = \begin{cases} i & \text{if } i = 0 \text{ or } i = 1 \\ \texttt{fib}_{i-1} + \texttt{fib}_{i-2} & \text{otherwise} \end{cases}$$

Here is a purely functional program in LUCID to compute `fib`:

```
fib( i ) = if ( i == 0 || i == 1 ) then i
           else fib( i-1 ) + fib( i-2 ) fi;
```

The important thing about this program is that `fib` is a recursively defined function – to compute $\texttt{fib}_3$, function `fib( 3 )` is invoked which requires the computation of `fib( 2 )` and `fib( 1 )` and the computation of `fib( 2 )` would require the computation[2] of `fib( 1 )` and `fib( 0 )`.

Here is a purely indexical program in LUCID to compute `fib`:

[1]Briefly, a language is *indexical* if meanings of expressions depend on a set of implicit contexts each of which vary over an *index set* . Interested readers should consult [3].

[2]Multiple computations of `fib(1)` can be avoided if results of function calls are stored (function memoization).

```
fib = 0 fby 1 fby fib + next fib;
```

The important thing to note here is that `fib` is a nullary (argument-less) function (which we refer to as a variable) which denotes a *sequence* of values over a *time* dimension . The definition of `fib` states that the first element of the sequence is 0, the second element of the sequence is 1 and each subsequent element is computed by adding the preceding two elements. This, in fact, corresponds to the mathematical definition except that it is indexless; rather the index is *implicit* or hidden. To compute $\texttt{fib}_3$, one needs to compute `fib` at time 3 (or $\texttt{fib}_{(\texttt{time=3})}$) which is the sum of $\texttt{fib}_{(\texttt{time=2})}$ and $\texttt{fib}_{(\texttt{time=1})}$. We know that $\texttt{fib}_{(\texttt{time=1})}$ is 1 and $\texttt{fib}_{(\texttt{time=2})}$ is the sum of $\texttt{fib}_{(\texttt{time=0})}$ and $\texttt{fib}_{(\texttt{time=1})}$ (which is already computed.)

Since LUCID is functional, its programs are referentially transparent and can be given meaning mathematically. Since LUCID is also indexical, it offers a fundamentally distributed view of both computation and data structures.

2.2 GLU – A Procedural Extension to LUCID

GLU is a simple yet powerful extension of LUCID: user-defined functions in GLU can be defined procedurally and values they consume and produce can be complex data structures.

This extension makes two things possible: first, conventional parallel computers can be programmed and second, existing sequential applications can be parallelized without having to rewrite them from scratch.

Consider the problem of multiplying two matrices , say A and B, and obtaining a product matrix C. The standard solution is to obtain each element (i, j) of the product matrix C by multiplying the i^{th} row of A with the j^{th} column of B and summing the inner product. A procedural program (in the language C) for multiplying matrices is given below.

```
#include <stdio.h>

struct matrix {
        int nrows, ncols;
        float **v;
        };
typedef struct matrix *MATRIX;


MATRIX block( MATRIX M, int frow, int lrow, int fcol, int lcol )
{
  MATRIX m;
  int i, j;

  m = (MATRIX) malloc( sizeof( struct matrix ) );
  m->nrows = lrow - frow + 1;  m->ncols = lcol - fcol + 1;
  m->v = (float **) malloc( m->nrows * sizeof( float * ) );
  for( i=0; i<m->nrows; i++ )
    m->v[i] = (float *) malloc( m->ncols * sizeof( float ) );

  for( i=0; i<m->nrows; i++ )
    for( j=0; j<m->ncols; j++ )
      m->v[i][j] = M->v[frow+i][fcol+j];

  return( m );
```

```
}

MATRIX mult( MATRIX a, MATRIX b )
{
  MATRIX m;
  int i, j, k;
  float s;

  m = (MATRIX) malloc( sizeof( struct matrix ) );
  m->nrows = a->nrows;  m->ncols = b->ncols;
  m->v = (float **) malloc( m->nrows * sizeof( float * ) );
  for( i=0; i<m->nrows; i++ )
    m->v[i] = (float *) malloc( m->ncols * sizeof( float ) );

  for( i=0; i<m->nrows; i++ )
    for( j=0; j<m->ncols; j++ )
      {
        s = 0;
        for( k=0; k<a->ncols; k++ )
          s = s + a->v[i][k]*b->v[k][j];
        m->v[i][j] = s;
      }

  return( m );

}

MATRIX matmult( MATRIX A, MATRIX B )
{
 int i, j, nr, nc;
 MATRIX C, r;

 nr = A->nrows;
 nc = B->ncols;
 C = (MATRIX) malloc( sizeof( struct matrix ) );
 C->nrows = nr; C->ncols = nc;
 C->v = (float **) malloc( nr*sizeof( float * ) );
 for( i=0; i<nr; i++ )
   C->v[i] = (float *) malloc( nc*sizeof( float ) );

 for( i=0; i<nr; i++ )
   for( j=0; j<nc; j++ )
     {
        r = mult( block(A,i,i,0,nc-1), block(B,0,nr-1,j,j) );
        C->v[i][j] = r->v[0][0];
      }

 return( C );
}

main()
{
  MATRIX A, B, C;
  MATRIX inA(), inB();
```

```
    void out();

    A = inA();
    B = inB();
    C = matmult( A, B );
    out( C );
  }
```

The only user-defined datatype in the above program is `MATRIX`, which is a pointer to the structure `matrix` that consists of the number of rows, number of columns, and a two-dimensional array of `floats`.

In the above program, functions `inA` and `inB` are used to input matrices and `out` is used to output the product matrix; function block is used to extract a submatrix from a matrix; function `mult` is used to multiply compatible submatrices; and function `matmult` uses the above-mentioned functions to multiply two matrices.

We can develop a variant of function `matmult` as given below:

```
  MATRIX lefthalf( MATRIX A )
  {
    return( block( A, 0, A->nrows-1, 0, A->ncols/2 - 1 ) );
  }

  MATRIX righthalf( MATRIX A )
  {
    return( block( A, 0, A->nrows-1, A->ncols/2, A->ncols-1 ) );
  }

  MATRIX tophalf( MATRIX A )
  {
    return( block( A, 0, A->nrows/2 - 1, 0, A->ncols-1 ) );
  }

  MATRIX bottomhalf( MATRIX A )
  {
    return( block( A, A->nrows/2, A->nrows-1, 0, A->ncols-1 ) );
  }

  MATRIX gather( MATRIX tl, MATRIX tr, MATRIX bl, MATRIX br )
  {
    MATRIX m;
    int i, j, hnr, hnc;

    m = (MATRIX ) smalloc( sizeof( struct matrix  ) );
    m->nrows = tl->nrows*2; m->ncols = tl->ncols*2;
    m->v = (float **) malloc(m->nrows*sizeof(float *));
    for(i=0; i<m->nrows; i++)
      m->v[i] = (float *) malloc(m->ncols * sizeof(float));

    hnr = m->nrows / 2;
    hnc = m->ncols / 2;

    for(i=0; i<m->nrows; i++)
      for(j=0; j<m->ncols; j++)
        {
```

```
        if( i < hnr )
          { if( j < hnc )
              m->v[i][j] = tl->v[i][j];
            else
              m->v[i][j] = tr->v[i][j-hnc];
          }
        else
          { if( j < hnc )
              m->v[i][j] = bl->v[i-hnr][j];
            else
              m->v[i][j] = br->v[i-hnr][j-hnc];
          }
      }

  return( m );
}


MATRIX matmult( MATRIX A, MATRIX B )
{
 MATRIX C, TL, TR, BL, BR;

 TL = mult( tophalf( A ), lefthalf( B ) );
 TR = mult( tophalf( A ), righthalf( B ) );
 BL = mult( bottomhalf( A ), lefthalf( B ) );
 BR = mult( bottomhalf( A ), righthalf( B ) );

 C = gather( TL, TR, BL, BR );

 return( C );
}
```

In this program, multiplying a matrix consists of simply gathering the submatrices obtained by multiplying the appropriate half-matrices. The gathering function is performed using function `gather` while selecting the appropriate half-matrices is performed using functions `lefthalf`, `righthalf`, `tophalf`, and `bottomhalf` each of which are defined in terms of `block`.

We now describe how this program can be constructed in GLU.

```
struct matrix {
        int nrows, ncols;
        float v[nrows][ncols];
        };
typedef struct matrix *MATRIX;

int out( MATRIX );
MATRIX inA( );
MATRIX inB( );
MATRIX gather( MATRIX, MATRIX, MATRIX, MATRIX );
MATRIX lefthalf( MATRIX );
MATRIX righthalf( MATRIX );
MATRIX tophalf( MATRIX );
MATRIX bottomhalf( MATRIX );
MATRIX mult( MATRIX, MATRIX );
```

```
out( C ) fby eod where
  A = inA( );
  B = inB( );
  C = matmult( A, B );

  matmult( A, B ) = P where
     P = gather( TL, TR, BL, BR );
     TL = mult( tophalf( A ), lefthalf( B ) );
     TR = mult( tophalf( A ), righthalf( B ) );
     BL = mult( bottomhalf( A ), lefthalf( B ) );
     BR = mult( bottomhalf( A ), righthalf( B ) );
  end;
end
```

The GLU program consists of two parts – the first part consists of user-defined datatypes and user-defined function-prototype declarations and the second part consists of the composition of the program using builtin and user-defined functions. The syntax of the first part is identical to ANSI C , hence needs no further explanation except for the type definition of `struct matrix` – two-dimensional array `v` parameterized using two integer variables, `nrows` and `ncols`. The composition itself is simply an expression, in this case `out( C ) fby eod` where the enclosing *where clause* defines the variable `C`.

The expression `out(C) fby eod` defines a temporal sequence whose first element is the result of invoking `out(C)`, which has the side-effect of displaying matrix `C`, and whose subsequent elements are all `eod`, which is a special value denoting end of data. With most GLU interpreters, the operational effect of displaying an `eod` is termination of program execution.

The definition of `C` is given in the enclosing *where clause* as `matmult( A, B )` where variables `A` and `B` themselves are defined using user-defined function `in`. Unlike `out` and `in` (which are C functions), `matmult` is a user-defined GLU function with two parameters. Function `matmult` is defined to be the variable `P` where `P` itself is the result of applying `gather` to `TL`, `TR`, `BL`, and `BR`. Variable `TL` is the result of multiplying (using `mult`) the left half of matrix `A` (obtained using `lefthalf( A )`) and the top half of matrix `B` (obtained using `tophalf( B )`). Variables `TR`, `BL`, and `BR` are similarly defined.

It is worth talking about how this composition is evaluated given that it is declarative and not procedural. The first thing that happens is that expression `out(C) fby eod` is evaluated. It causes `out(C)` to be evaluated first and `eod` to be evaluated next. Since `out` is a procedural user-defined function , it is only invoked when variable `C` is defined and no sooner. To evaluate `C`, the right-hand-side of its definition, `matmult( A, B )`, is evaluated. Since `matmult` is a composition function and not a procedural function, it is evaluated even before its arguments `A` and `B` are available. In fact, all composition functions are evaluated using "call-by-need" semantics as opposed to "call-by-value" semantics used for C functions.

The main difference between a GLU program and an equivalent procedural program is that the GLU program is inherently parallel unless otherwise constrained, whereas the procedural program is inherently sequential unless parallelism is explicitly identified. In the example above, with the GLU program, the variables `TL`, `TR`, `BL`, and `BR` can be computed simultaneously resulting in four-fold parallelism whereas these variables have to be evaluated sequentially in the procedural program.

Since GLU programs are declarative, they possess the important property of *referential transparency* which means that two occurrences of an expression refer to the same value. Thus, the order of equations in GLU programs is simply a matter of style and do not affect their meanings. One can substitute the right-hand-side of an equation for each occurrence of the left-hand-side and vice versa just as in mathematics.

GLU programs separate composition from computation . This is not only important in various aspects of the program development process such as debugging and modification but it also encourages reuse of existing computational code across applications.

The GLU program described above simply multiplies two matrices then terminates. What if we want to have the program multiply two streams of matrices pairwise? It turns out that doing so in GLU is not that much more work than multiplying just one pair. The GLU composition is shown below:

```
struct matrix {
        int nrows, ncols;
        float **m;
        }
typedef matrix *MATRIX;

int out( MATRIX );
MATRIX inA( int );
MATRIX inB( int );
MATRIX gather( MATRIX, MATRIX, MATRIX, MATRIX );
MATRIX lefthalf( MATRIX );
MATRIX righthalf( MATRIX );
MATRIX tophalf( MATRIX );
MATRIX bottomhalf( MATRIX );
MATRIX mult( MATRIX, MATRIX );

out( C ) where
  A = inA( #.time );
  B = inB( #.time );
  C = matmult( A, B );

  matmult( A, B ) = P where
     P = gather( TL, TR, BL, BR );
     TL = mult( tophalf( A ), lefthalf( B ) );
     TR = mult( tophalf( A ), righthalf( B ) );
     BL = mult( bottomhalf( A ), lefthalf( B ) );
     BR = mult( bottomhalf( A ), righthalf( B ) );
  end;
end
```

This program differs from the previous one in only two respects: Functions `inA` and `inB` now have an argument, `#.time` , and the program expression is simply `out(C)` instead of `out(C) fby eod`.

What this program reveals is the underlying context in which GLU programs are evaluated, namely `time`. That is, each expression (including variables) in the above GLU program actually denotes a temporal stream of values. The value of an expression depends not only on the constituent subexpressions and their binding function but also on the implicit time context. Thus, one can think of the equation `C = matmult( A, B )` as meaning
$\texttt{<C}_0\texttt{,...,C}_t\texttt{,...> = <matmult(A,B)}_0\texttt{,...,matmult(A,B)}_t\texttt{,...>}$
where $\texttt{matmult(A,B)}_t\texttt{ = matmult(A}_t\texttt{, B}_t\texttt{).}$

Consider the evaluation of $\texttt{out(C)}_t$: it causes $\texttt{C}_t$ to be evaluated, which causes $\texttt{matmult(A,B)}_t$ to be evaluated, which, in turn, causes $\texttt{gather(TL}_t\texttt{,TR}_t\texttt{,BL}_t\texttt{,BR}_t\texttt{)}$ to be evaluated. Evaluations of $\texttt{TL}_t$,$\texttt{TR}_t$,$\texttt{BL}_t$, and $\texttt{BR}_t$ eventually cause both $\texttt{A}_t$ and $\texttt{B}_t$ to be evaluated once. Evaluation of $\texttt{A}_t$ invokes `inA` with `#.time` being the current time context, namely, t. (And similarly for $\texttt{B}_t$.) Once $\texttt{A}_t$ and $\texttt{B}_t$

are available, $\mathtt{TL}_t$,$\mathtt{TR}_t$,$\mathtt{BL}_t$, and $\mathtt{BR}_t$ can be generated resulting in $\mathtt{C}_t$ to be generated and function `out` to display the matrix.

Since evaluation of $\mathtt{C}_t$ is independent of $\mathtt{C}_{t+1}$, it is entirely possible for both these computations and any others to proceed simultaneously without interference.

Suppose we wanted to multiply only the first 10 matrix pairs and then terminate – this requires changing only the program expression from `out(C)` to `if #.time < 10 then out(C) else eod fi`.

One of the limitations of `matmult` is that it results in only four-fold parallelism since function `mult` is sequential. It is possible to have `matmult` express more parallelism by making `matmult` recursive as shown below:

```
matmult( A, B ) =
  if smallenough(A,B)
  then mult(A,B)
  else P fi where
   P = gather( TL, TR, BL, BR );
   TL = mult( tophalf( A ), lefthalf( B ) );
   TR = mult( tophalf( A ), righthalf( B ) );
   BL = mult( bottomhalf( A ), lefthalf( B ) );
   BR = mult( bottomhalf( A ), righthalf( B ) );
end;
```

In the above program, we have introduced a function `smallenough` that helps control the granularity of parallelism by deciding when multiplying two matrices cannot be further subdivided. Composition function `matmult` itself is recursive – instead of having `TL`, `TR`, `BL`, and `BR` defined in terms of `mult`, these variables are defined using `matmult`. The parallelism expressed by `matmult` is no longer four-fold – it is actually $O(4^L)$, where L is the number of recursive steps.

A drawback of the recursive `matmult` is the cost associated with repeated division of matrices using `lefthalf`, `righthalf`, `tophalf`, and `bottomhalf`. This not only takes time but also causes unnecessary creation and destruction of intermediate matrices. We devise a nonrecursive version of `matmult` that expresses as much parallelism, without the cost of recursion. To do so, we use the GLU notion of user-defined multidimensionality.

```
    matmult( A, B ) = p asa.t ( (nrows(A) == nrows(p)) && (ncols(B) == ncols(p)) )
      where
        dimension i,j,t;
        a = rowstrip( A, #.i );
        b = colstrip( B, #.j );
        p = mult( a, b ) fby.t
            ( ( gather( p, next.j p, next.i p, next.i next.j p ) @.j (2*#.j) ) @.i (2*#.i)
  );
      end;
```

Composition function `matmult` is defined using a multidimensional computation in dimensions `i`, `j`, and `t`, which are declared using the `dimension` declaration within the *where clause* associated with `matmult`. This multidimensional computation occurs at each `time`-context in the enclosing `time` dimension. It is defined to be the value of variable `p` at context `i=0,j=0,t=T` where `T` is the `t`-context at which the number of rows of the product matrix `p` is the same as the number of rows of matrix `A` and the number of columns is the same as the number of columns of matrix `B`.

Consider the definition of variable `p`:- it says that at any context where `t`-context 0, it is the value of `mult(a,b)` at the same context. Variable `a` which only varies in dimension `i` is defined as function `rowstrip(A, #.i)` where `#.i` refers to the `i`-context from the context of evaluation. (`rowstrip` basically extracts the ith strip of rows from matrix `a`.) Variable `b` which only varies in dimension `j` is defined as function `colstrip(B, #.j)` where `#.j` refers to the `j`-context from the context of evaluation. (It extracts the jth strip of columns from matrix `b`.)

Variable `p` at context `(i=I,j=J,t=T)` where `T>0`, is the result of applying function `gather` to values of variable `p` at contexts `(i=2*I,j=2*J,t=T-1)`, `(i=2*I,j=2*J+1,t=T-1)`, `(i=2*I+1,j=2*J,t=T-1)`, and `(i=2*I+1,j=2*J+1,t=T-1)`. This can be derived as follows:

$$
\begin{aligned}
\texttt{p}_{\texttt{(i=I,j=J,t=T)}} = & \\
& \texttt{(gather(p,next.j p,next.i p,next.i next.j)@.j(2*\#.j))@.i(2*\#.i)}_{\texttt{(i=I,j=J,t=T-1)}} \\
= \; & \texttt{gather(p,next.j p,next.i p,next.i next.j)}_{\texttt{(i=2*I,j=2*J,t=T-1)}} \\
= \; & \texttt{gather(}\texttt{p}_{\texttt{(i=2*I,j=2*J,t=T-1)}}\texttt{, p}_{\texttt{(i=2*I,j=2*J+1,t=T-1)}}\texttt{,} \\
& \quad \texttt{p}_{\texttt{(i=2*I+1,j=2*J,t=T-1)}}\texttt{, p}_{\texttt{(i=2*I+1,j=2*J+1,t=T-1)}}\texttt{)}
\end{aligned}
$$

The best way to visualize the evaluation of `matmult` is to think of each of its variable as representing a series of planes (`i`,`j`) ordered by time `t`. Variable `a` then is a series of identical planes (since the definition of `a` does not depend on time `t`) where each plane consists of identical columns of row strips where each row strip is extracted from matrix `a`. Similarly, variable `b` is a series of identical planes where each plane consists of identical rows of column strips where each column strip is extracted from matrix `b`. (The row strip size and column strip size are predefined.) It is worth noting that any reasonable implementation of GLU would store values of `a` for each `i`-context and values of `b` for each `j`-context, as they are constant in the remaining two dimensions.

Variable `p` consists of a series of planes, where the initial plane consists of matrix products such that the $(i,j)^{th}$ matrix is obtained by multiplying the corresponding (actually i^{th}) row strip of `a` and corresponding (actually j^{th}) column strip of `b`. Each successive plane consists of matrices that are four times as large where the $(i,j)^{th}$ matrix is obtained by coalescing the four matrices using function `gather` in the previous plane at positions $(2i,2j)$, $(2i,2j+1)$, $(2i+1,2j)$ and $(2i+1,2j+1)$.

Assuming that the size of each element matrix in the initial plane is s and the size of the product matrix is S, the $(0,0)^{th}$ element of the $\log_2(S/s)$ plane is the desired product matrix. One way of determining without keeping track of size is to check if the $(0,0)^{th}$ element matrix of a plane has the same number of rows as matrix `a` and the same number of columns as matrix `b` as in `p asa.t ( (nrows(A) == nrows(p)) && (ncols(B) == ncols(p)) )`.

Similar to the recursive version of `matmult`, the parallelism in simultaneous execution of `mult` is 4^L where L is the number of planes. (L is $\log_2(S/s)$.)

It turns out that this kind of computational structure is common to algorithms that solve quite different problems. Therefore, it would be desirable to specify the structure in such a way that it can be not only general but also succinct. This is the idea behind what we call dimensionally-abstract composition functions — functions that abstract not only data but also abstract dimensions. We illustrate one such function, `planar_tree`, in `matmult`.

```
planar_tree.x,y( f, a, size ) = b asa.t s >= size
  where
    dimension t;
    b = a fby.t
        ( ( f( b, next.y b, next.x b, next.y next.x b ) @.x (2 * #.x)) @.y (2 * #.y));
    s = 1 fby.t 4*s;
  end;
matmult( A, B ) = planar_tree.i,j( gather, mult(a, b), n )
  where
    dimension i,j;
```

```
      a = rowstrip( A, ROWSTRIPSIZE, #.i );
      b = colstrip( B, COLSTRIPSIZE, #.j );
      n = ( nrows(A) / ROWSTRIPSIZE ) * ( ncols(B) / COLSTRIPSIZE );
    end;
```

In `matmult`, `planar_tree` is invoked with dimensions `x` and `y` corresponding to `i` and `j`, `f` corresponding to data function `gather`, `a` corresponding to `mult(a,b)`, and `m` corresponding to `n`.

It turns out that `planar_tree` is quite useful in expressing a wide range of computational structures. Hence it has been predefined in a file called `std.g` that can be included using the standard C include facility in a GLU program. (See Appendix B.)

This concludes the tutorial introduction to programming in GLU. The reader should now be able to have a basic understanding of the GLU model of programming and how it differs from purely imperative and purely functional models. The reader should also be able to write simple GLU programs starting from existing C programs.

Chapter 3

LUCID

At the outset we stated that LUCID forms the nucleus of GLU. In this chapter, we describe the ingredients of LUCID in detail. Although LUCID has been around since 1974, what we describe here is the latest incarnation of LUCID (circa 1994), which is substantially different from its predecessors.

3.1 Types

Values in LUCID can be one of two scalar types: integer and floating point. LUCID also provides two special values: `eod` (referred to as 'end of data') which is of a special type that can be only manipulated by a specific operation (`iseod`) and `error` (referred to as 'error object') which is also of a special type that can be only manipulated by a specific operation (`iserror`) . Although there are no booleans, LUCID follows the C convention in that integer 0 is interpreted as *false* and any other integer value is interpreted as *true*. LUCID does not directly provide for any aggregate types such as arrays, structures or linked lists.

There is no explicit declaration for types of values. Value types are implicit and they are determined at compile time using type analysis of the associated expression. For example,

```
x = 3.0 fby 4.0;
y = if x then 3 else 4 fi;
```

Values of `x` are floats because `x` is defined in terms of two floats. Values of `y` are integer because the values that `y` can be assigned are both integers even though the predicate `x` is `float`.

3.2 Constants

LUCID provides for numeric constants (both integers and floats) and for the special constants `eod` and `error`. As mentioned earlier, a constant is not simply a scalar value. Rather, it is an infinite sequence of values where each value is the same. For example, `3` denotes the constant sequence $< 3, 3, \ldots, 3, \ldots >$, `4.0` denotes the constant sequence $< 4.0, 4.0, \ldots, 4.0, \ldots >$, and `eod` denotes the constant sequence $< eod, eod, \ldots, eod, \ldots >$.

3.3 Variables

A variable in LUCID is simply a named expression . A variable name can only be defined once within its scope. Its name can be any alphanumeric identifier starting with a letter and it can include the special character '_ ' (underscore).

A variable is defined using an equation in which the variable is on the left-hand side of the equation and the defining expression is on the right-hand side. The following are valid variable definitions:

```
two = 2;
root = ( -b - s ) / (two*a);
otherroot = (-b + s ) / (two*a);
s = sqrt( b*b - 4*a*c );
```

In each of the examples, the variable on the left-hand side denotes a sequence of values associated with the expression on the right-hand side.

3.4 Operators

LUCID has several predefined operators that can be divided into two general categories: pointwise operators and nonpointwise operators . With pointwise operators, to compute a result value at a certain point in the sequence requires only values at the same point from the operand sequences. All operators for which this is not true are nonpointwise operators. We describe and define these operators assuming that sequences are one-dimensional. Later we extend them to multiple dimensions.

3.4.1 Arithmetic, Relational and Logical Operators

The arithmetic, relational, logical, and bitwise logical operators in LUCID are the same as in C and they are all pointwise. The arithmetic operators are `+` for addition, `-` for subtraction, `*` for multiplication, `/` for division, `%` for modulus, and unary `-` for negative. The relational operators are `>` for greater than, `>=` for not less than, `<` for less than, `<=` for not greater than, `==` for equal to, and `!=` for not equal to. The logical operators are `&&` for conjunction, `||` for disjunction, and unary `!` "!@!!unary for negation. The bitwise logical operators are `&` for "and", `|` for "or", `<<` for left shift, `>>` for right shift, `^` for exclusive-or and `~` for 1's complement.

Here are some examples. (In these and other examples, we use the convention that the first element of a sequence has a box around it.)

`A` *is*	$\boxed{0}$	1	2	3	···
`B` *is*	$\boxed{1}$	3	5	7	···
`A + B` *is*	$\boxed{1}$	4	7	10	···
`A % 2 != 0` *is*	$\boxed{0}$	1	0	1	···
`B >> 1` *is*	$\boxed{0}$	1	2	3	···

3.4.2 Operators `first` and `next`

The `first` and `next` operators are the two simplest nonpointwise unary LUCID operators . Operator `first` returns a sequence each of whose values is the first value of the operand sequence. Operator `next` returns a sequence that consists of all but the first value of its operand sequence.

`A` *is*	$\boxed{0}$	1	2	3	···
`first A` *is*	$\boxed{0}$	0	0	0	···
`next A` *is*	$\boxed{1}$	2	3	4	···
`next next A` *is*	$\boxed{2}$	3	4	5	···
`first next next A` *is*	$\boxed{2}$	2	2	2	···

3.4.3 Operator `fby`

The operator `fby` (pronounced "followed by") is a binary non-pointwise operator that accepts two sequences and produces a result sequence whose first element is from the first operand sequence and whose subsequent elements correspond to the second operand sequence.

`A` *is*	$\boxed{0}$	1	2	3	⋯
`B` *is*	$\boxed{11}$	21	31	41	⋯
`A fby B` *is*	$\boxed{0}$	11	21	31	⋯
`B fby A` *is*	$\boxed{11}$	0	1	2	⋯
`A fby B fby A+B` *is*	$\boxed{0}$	11	11	22	33 ⋯
`A fby next B` *is*	$\boxed{0}$	21	31	41	⋯

The following are mathematical definitions of some of the operators we have informally described. Note that the subscript refers to the implicit context or index.

$$\begin{aligned}
[\texttt{0}]_k &= 0 \\
[A\texttt{+}B]_k &= [A]_k + [B]_k \\
[A\texttt{!=}B]_k &= [A]_k \neq [B]_k \\
[\texttt{!}A]_k &= \texttt{!}\,[A]_k \\
[\texttt{first}A]_k &= [A]_0 \\
[\texttt{next}A]_k &= [A]_{k+1} \\
[A\ \texttt{fby}\ B]_k &= \text{if } k \leq 0 \text{ then } [A]_k \text{ else } [B]_{k-1} \text{ fi}
\end{aligned}$$

3.4.4 `@` and `#` Operators

The `@` (at) operator is the most primitive LUCID operator using which most of the other LUCID operators can be defined. The `@` operator takes two sequences, a base sequence and an index sequence, and creates a new sequence by using each value of the index stream as an index or a context into the base sequence.

The `#` operator (hash) operator makes explicit the underlying implicit index in a particular dimension. In particular, `#.dim` gives the current underlying `dim`-context where `dim` is the name of a dimension such as `time`.

A	=	$\boxed{1}$	1	2	3	5	8	13	21	34	55	⋯
B	=	$\boxed{1}$	2	4	8	7	0	5	2	3	9	⋯
A `@` B	=	$\boxed{1}$	2	5	34	21	1	8	2	3	55	⋯
`#.time`	=	$\boxed{0}$	1	2	3	4	5	6	7	8	⋯	

Denotationally, the `@` and `#` operators can be defined as follows.

$$\begin{aligned}
[\texttt{A @B}]_k &= [\texttt{A}]_{[\texttt{B}]_k} \\
[\texttt{\#.time}]_k &= k
\end{aligned}$$

3.4.5 `if-then-else` Operator

The `if-then-else` operator takes three sequences, a boolean sequence and two base sequences, and creates a new sequence by using each value of the boolean sequence to select the corresponding value from the first base sequence if `true` and the corresponding value from the second base sequence if `false`.

P	=	···	$\boxed{1}$	0	0	1	1	1	1	0	1	0	···
A	=	···	$\boxed{10}$	1	2	3	5	8	13	21	34	55	···
B	=	···	$\boxed{21}$	2	4	8	7	0	5	2	3	9	···
`if` P `then` A `else` B `fi`	=	···	$\boxed{10}$	2	4	3	5	8	13	2	34	9	···

Operators `first`, `next`, and `fby` can be defined in terms of `@` and `of-then-else` as shown below.

```
first A = A @ 0
next A = A @ (#.time + 1)
A fby B = if (#.time)==0 then A @ 0 else B @ (#.time - 1) fi
```

3.4.6 Operators `asa`, `wvr`, `upon`

The LUCID operator `asa` (pronounced "as soon as") accepts two operand sequences and produces a result sequence. This operator can be thought of as producing an operand dependent constant sequence. The value of this constant is the k^{th} value of the first operand sequence such that the k^{th} value of the second operand sequence is *true* (non-zero) and all values prior to it are *false* (zero).

P	=	$\boxed{0}$	0	0	1	1	1	1	0	1	0	···
A	=	$\boxed{10}$	11	29	23	5	8	13	21	34	55	···
A asa P	=	$\boxed{23}$	23	23	23	23	23	23	23	23	23	···

Operator `asa` can be defined in terms of `first`, `next`, and `if-then-else`:

```
x asa p  =  y where y = first ( if p then x else next y fi ) end
```

As with the operator `asa`, the operator `wvr` (pronounced "whenever") has two operand sequences and one result sequence. This operator selects those values from the first operand sequence whose corresponding values in the second operand sequence are *true*. The result sequence, in a sense, is a filtered version of the first operand sequence where the extent of filtering is determined by the second operand sequence.

P	=	$\boxed{0}$	0	0	1	1	1	1	0	1	0	···
A	=	$\boxed{10}$	11	29	23	5	8	13	21	34	···	
A `wvr` P	=	$\boxed{23}$	5	8	13	34	···					

The operator `asa` is frequently used in expressing conditional iteration as in the following program which computes `N!`:

```
fac asa n == N
  where
    n = 1 fby n+1;
    fac = 1 fby fac*n;
  end
```

Variable `n` is the sequence of natural numbers and variable `fac` is the sequence of factorials. Expression `fac asa n == N` selects the value of `fac` when the value of variable `n` is `N`.

Operator `wvr` can be defined in terms of `@`, `fby`, and `if-then-else`:

```
x wvr p  =  x @ t2  where
                      t1 = if p then #.time else next t1 fi;
                      t2 = t1 fby (t1 @ t2+1);
                    end
```

Like operator `wvr`, the operator `upon` is binary. This operator `upon` 'stretches' the first operand sequence based on the second operand sequence. Specifically, the k^{th} value of the result sequence is the p^{th} ($p \leq k$) value of the first operand sequence such that p of the first k values of the second operand sequence are *true*.

P	=	$\boxed{0}$	0	0	1	1	1	0	1	0	$\cdots$
A	=	$\boxed{10}$	11	29	23	5	8	13	21	34	$\cdots$
A upon P	=	$\boxed{10}$	10	10	11	29	23	23	5	5	$\cdots$

Operator `upon` can be defined in terms of `@`, `fby`, and `if-then-else`:

```
x upon p  =  x @ t1 where
                  t1 = 0 fby if p then t1+1 else t1 fi;
                end;
```

The `upon` operator is useful when you wish to merge two streams without losing any elements of either:

```
sort(A, B) = if dA < dB then dA else dB fi
    where
        dA = A upon dA > dB;
        dB = B upon dA <= dB;
    end
```

Here is an example of `sort(A, B)`:

A	=	$\cdots$	$\boxed{1}$	2	4	7	10	12	13	20	21	$\cdots$
B	=	$\cdots$	$\boxed{3}$	5	6	8	9	11	14	15	16	$\cdots$
dA	=	$\cdots$	$\boxed{1}$	2	4	4	7	7	7	10	10	$\cdots$
dB	=	$\cdots$	$\boxed{3}$	3	3	5	5	6	8	8	9	$\cdots$

3.4.7 Operator `iseod`

Operator `iseod` tests if its argument is the special value `eod`. It returns 1 if it is and returns 0 otherwise.

X	=	$\boxed{1}$	4	eod	9	eod	16	$\cdots$
iseod X	=	$\boxed{0}$	0	1	0	1	0	$\cdots$

3.4.8 Operator `iserror`

Operator `iserror` tests if its argument is the special value `error`. It returns 1 if it is an error object, otherwise it returns 0.

X	=	$\boxed{1}$	4	error	9	error	16	$\cdots$
iserror X	=	$\boxed{0}$	0	1	0	1	0	$\cdots$

The error object, `error`, is produced by an operation under the following conditions:

1. Operation execution results in an arithemtic exception (argument domain exception, argument singularity, overflow range exception, underflow range exception).

2. Operation has mismatched argument types (such as float as second argument of the `@` operator).

3. One or more of the operands are themselves error objects.

3.5 Extra Operators

We introduce three operators that are not part of standard LUCID. These operators are used to produce the appropriate side-effects.

3.5.1 Operator ‘`,`’

The ‘`,`’ (comma) operator corresponds to the value of the second argument produced after the first argument has been evaluated. `X,Y` is equivalent to `if iseod X then Y else Y fi`.

$$\begin{array}{lcccccccc} X & = & \boxed{1} & 4 & 9 & 16 & 25 & 36 & \cdots \\ Y & = & \boxed{-1} & -4 & -9 & -16 & -25 & -36 & \cdots \\ X,Y & = & \boxed{-1} & -4 & -9 & -16 & -25 & -36 & \cdots \end{array}$$

3.5.2 Operator ‘`!`’

The ‘`!`’ (question mark) operator corresponds to the value of the second argument produced at the same time as the value of the first argument is been produced which is discarded. The difference between the ‘`,`’ and the ‘`!`’ operator is that the former is sequential whereas the latter is not. Thus, ‘`!`’ will produce a result as long as the second argument is defined regardless of whether the first argument is defined or undefined ($\perp$).

$$\begin{array}{lcccccccc} X & = & \boxed{1} & \perp & 9 & \perp & 25 & 36 & \cdots \\ Y & = & \boxed{-1} & -4 & -9 & -16 & -25 & -36 & \cdots \\ X!Y & = & \boxed{-1} & -4 & -9 & -16 & -25 & -36 & \cdots \end{array}$$

3.5.3 Operator ‘?’

The ‘?’ (question mark) operator evaluates both its arguments and produces the argument that evaluates first. It corresponds to a parallel or non-deterministic merge or choice operator which is not sequential. The output depends only on the relative rates of the two inputs and not on their values. For example, `1 ? 2` can be either `1` or `2` with equal probability.

3.6 Expressions

A LUCID expression can be as simple as a constant or a variable.

```
pi = 3.14159267;
another_pi = pi;
```

Or it can be an operation symbol together with operands, which are expressions themselves.

```
two_pi = 2 * pi;
circumference = 2 * pi * radius;
odds = 1 fby odds + 2;
powers_of_two = 1 fby 2*powers_of_two;
```

Or it can be a function symbol together with actual parameters, which are expressions.

```
n = 1 fby add( n, 1 );
powers_of_two = pow( 2, n );
```

Or it can be a where-clause .

```
running_average where
  running_average = X fby running_average + diff
                        where
                          diff = (next X - running_average ) / n
                                 where
                                   n = 2 fby n+1;
                                 end;
                        end;
end;
```

Here is an example of the computation of `running_average`.

X	=	$\boxed{1}$	4	9	16	⋯
running_average	=	$\boxed{1}$	2.5	4.67	7.5	⋯
diff	=	$\boxed{1.5}$	1.17	2.83	⋯	
n	=	$\boxed{2}$	3	4	5	⋯

3.7 User-defined Functions

It is possible to define functions just like in other languages. What is unusual about LUCID functions is that they are not necessarily pointwise. Thus, it is better to think of them as functions whose parameters denote sequences rather than values. This is consistent with how one thinks of LUCID operators. Here is a simple example of a LUCID function, `avg3`, which given a sequence `s` returns a constant sequence consisting of the average of the next three elements of `s`.

```
avg3( s ) = ( s + next s + next next s ) / 3;
```

Here is a an example of an evaluation of `avg3`.

s	=	$\boxed{1}$	2	4	7	10	12	13	20	21	⋯
next s	=	$\boxed{2}$	4	7	10	12	13	20	21	⋯	
next next s	=	$\boxed{4}$	7	10	12	13	20	21	⋯		
avg3(s)	=	$\boxed{2.33}$	4.33	7	9.67	11.67	15	18	⋯		

The parameter `s` should be thought of as a sequence because even though `avg3( s )` is evaluated at time t, the values of `s` that are needed is at time t, $t + 1$, and $t + 2$. For this reason, LUCID functions are always evaluated lazily – their parameters are evaluated "call-by-need" (as opposed to call-by-value parameter passing of C). Basically, call-by-need evaluation means that a parameter is evaluated only when it is needed in the function body *in the context in which it is needed.*

Here is another example of a LUCID function that computes the running average of a sequence of numbers `x`.

```
ravg(x) = running_average where
  running_average = x fby running_average + diff
    where
      diff = (next x - running_average ) / n
             where
               n = 2 fby n+1;
             end;
    end;
```

Here is an example of a recursive function in LUCID to determine the factorial of a sequence of numbers denoted by `n`.

```
fac(n) = if n == 0 then 1 else n*fac(n-1) fi;
```

If `n` denotes the sequence of natural numbers, here is what gets evaluated:

$$\begin{array}{lcccccccc} \text{n} & = & \boxed{1} & 2 & 3 & 4 & 5 & 6 & \cdots \\ \text{fac(n)} & = & \boxed{1} & 2 & 6 & 24 & 120 & 720 & \cdots \end{array}$$

LUCID functions can not only accept expressions as arguments but also function names. However, LUCID functions cannot return functions. Thus, they are not higher-order in the classical sense. Here is an example of a function `arith` that applies the binary arithmetic function `f` to the two parameter sequences of `arith`:

```
arith(f,a,b) = f(a,b);
```

For example, `arith(add,a,b)` adds each value of `a` with the corresponding value of `b`. And `arith(pow,a,b)` computes the $\mathtt{b}^{th}$ power of `a`.

3.8 Nested Computations

A common computational structure is a nested computation such as nested `do` or `for` loops . We show how nested computations can be expressed in LUCID.

Suppose we wish to write a function `pow` which repeatedly takes pairs of non-negative integers and raises the first to the power of the second; e.g., 2 to the power of 3 is 8. Here is a LUCID program that works for the first pair of values.

```
pow( x, n ) = p asa i == first n
      where
            i = 1 fby i + 1;
            p = 1 fby p * first x;
      end
```

In function `pow`, for each successive `time` context, `p` is multiplied by the first value of `x`. This continues until a variable that we have chosen to act as the counter, namely `i`, equals the first value of `n`, after which the value of `p` is the first value of `x` raised to the power of the first value of `n`. Function `pow` can only raise the first value of `x` to the first value of `n` and not any of the subsequent pairs. The `first` in `first x` and `first n` is needed because these values *have* to be invariant as successive values of `p` are computed.

For this function to work on successive pairs, what we really need is a way to "freeze" each pair while the value of `x` is raised to the power of the corresponding value of `n` using repeated multiplications. This is possible in LUCID using dimensional `where` clauses .

A dimensional `where` clause is a `where` clause that contains a declaration of one or more dimensions that are local to the scope of the `where` clause. These dimensions are nested within any enclosing dimensions and they are orthogonal to each other.

Consider the following version of function `pow`:

```
pow( x, n ) = p asa.inner i == n
      where
        dimension inner;
        i = 1 fby.inner i + 1;
        p = x fby.inner p * x;
      end
```

First of all, observe the `dimension` declaration which declares `inner` to be a dimension local to the `where` clause The `inner` dimension is nested within the `time` dimension and this is what allows us to express nested iteration – for each point or context in the `time` dimension, a subcomputation can proceed along the `inner` dimension. Second, observe that operators `asa` and `fby` within the `where` clause have been qualified by the name of the dimension as in `asa.inner` and `fby.inner`. What this means is that these operators work in the qualified dimension instead of the default `time` dimension. That is, `i` varies in the `inner` dimension as does `p`. However, the parameters to function `pow`, `x` and `n`, do not vary in the `inner` dimension because they are defined outside the scope of the `where` clause; they only vary in dimension `time`.

Let us now see how the function works. Assume both `x` and `n` vary in dimension `time` and assume that function `pow` is invoked for each successive pair of values. Each invocation of `pow` causes a subcomputation to be started. Variable `p` defines a sequence of successive powers of `x` in the `inner` dimension and variable `i` defines a counter sequence. The head of the `where` clause determines when the subcomputation terminates – it terminates when the value of `i` in the `inner` dimension is equal to the value of `n` which is invariant in the `inner` dimension. The function defines a sequence in the `inner` dimension where each value of the sequence is the value of `p` in the `inner` dimension when `i` equals `n`. Since the use of `pow` is from outside the scope of the `inner` dimension, only the first value of the result sequence is visible outside the scope of the function definition. Thus, successive invocations of `pow` using successive pairs of values of `x` and `n` will return the appropriate results (x^n).

Here is another example of nested computation : compute the square root of a sequence of values `a` that vary in dimension `time` using Newton's iterative algorithm .

```
sqroot( a ) = next.inner approx asa.inner ( abs( next.inner approx - approx ) < 0.001 )
  where
    dimension inner;
    approx = 1 fby.inner (approx+a/approx)/2;
  end
```

As with function `pow`, function `sqroot` defines a nested dimension called `inner`. Variable `approx` captures the Newton's iterative algorithm starting with 1 as the initial approximation for the square root. The termination condition for the subcomputation specified by the function is the value of `approx` in the `inner` dimension when the absolute difference between it and the previous value of `approx` in the `inner` dimension is less than 0.001. That is the value that is "returned" by function `sqroot`.

Now suppose one wanted to find the square root of a sequence of powers x^n, we can combine the two functions in the obvious way:

```
sqroot( pow( x, n ) )
```

Note that the `inner` dimensions in `pow` and `sqroot` are distinct as per the scope rules.

We now illustrate how dimensional `where` clauses can be used to express doubly nested iteration to implement the formula $c = \Sigma_{i=1}^{n} \Sigma_{j=1}^{m} a_i b_j$.

Assuming `a` and `b` are two sequences in the `time` dimension and `n` and `m` are constants, we can write the following definition for `c`:

```
c = s asa.i i == n
    where
      dimension i;
      s = 0 fby.i s + t
      t = v asa.j j == m
          where
```

```
                dimension j;
                v = 0 fby v + (a @.time i) * (b @.time j);
              end;
        end
```

The definition of c is a doubly nested iteration in terms of dimensions i and j as per the formula given above. The definition says that the value of c is the value of s when the current context in the i dimension is n. (Note the use of a dimension name, i, as a variable – it returns the current i dimension context.) Variable s starts off as 0 and at each successive i dimension context, it accumulates the value of t that for each i dimension context is defined using a nested iteration in dimension j. In particular, the value t at some i dimension context is the value of v when the j dimension context is m. And variable v is defined in the j dimension as the sum of the products of i^{th} value of a and j^{th} value of b. (Note that we have extracted the i^{th} value of a using a qualified @ operator – a @.time i or variable a at time dimension context given by i, and the j^{th} value of b using the same – b @.time j.)

3.9 Array Computations

Array computations refer to nested computations that are used to manipulate multidimensional arrays . In the previous section, we have shown how the dimensional where clause can be used to express nested computations. Here we show how they also can be used to express multidimensional array computations.

Consider the problem of computing distribution of electric potential on a two-dimensional grid over time with fixed potential at the electrodes. The usual numerical solution to this problem is successive over-relaxation where the potential at each point in the grid is simply the average of the potentials at the neighboring points at the previous time step. This is captured quite succinctly in the following LUCID definition:

```
grid where
  dimension v, h;
  grid = if ELECTRODE then POTENTIAL else 0 fby avg( grid )
    where
      avg( g ) = ( prev.v prev.h g + prev.v next.h g +
                   next.v prev.h g + next.v next.h g ) / 4;
    end;
end
```

The definition of grid consists of a two-dimensional where clause, the dimensions being v and h. If the v and h contexts denote an electrode then the value of grid at that point is always POTENTIAL. Otherwise, it starts off as 0 and at each successive time dimension context, it is the average of the four neighboring values of grid at the previous time dimension context. The operator prev is simply the dual of next except that it is only defined for positive contexts: prev.v x = x @.v (v-1) and prev.h = x @.h (h-1).

This is a classic example of an array computation in the sense that grid denotes a two-dimensional surface in dimensions v and h and operators prev and next are used to navigate this surface in all four directions.

Another example of an array computation is matrix transposition . Here is a LUCID function to transpose a matrix in dimensions i and j. It uses the realign operator that is defined as realign.a,b x = x @.a b meaning rotate the a-th dimension of x into the b-th dimension.

```
transpose( a ) = realign.z,j realign.j,i realign.i,z a
  where
    dimension z;
  end
```

To understand how function `transpose` works, consider what happens when `transpose` is evaluated at context (i=I,j=J).

$$\begin{aligned}
\texttt{transpose(a)}_{(\texttt{i=I,j=J,z=0})} = {} & \\
& \texttt{realign.z,j realign.j,i realign.i,z a}_{(i=I,j=J,z=0)} \\
= {} & \texttt{realign.j,i realign.i,z a}_{(i=I,j=J,z=J)} \\
= {} & \texttt{realign.i,z a}_{(i=I,j=I,z=J)} \\
= {} & \texttt{a}_{(i=J,j=I,z=J)} \\
= {} & \texttt{a}_{(i=J,j=I)}
\end{aligned}$$

We can also use `realign` in a succinct expression of matrix multiplication of two matrices, `a` and `b`, in dimensions `i` and `j` of order n.

```
sum( ( realign.j,k a ) * ( realign.i,k b ), n )
  where
    dimension k;
    Sum( x, n ) = s asa.k ( #.k == n )
      where
        s = 0 fby.k s + x;
      end;
  end
```

3.10 Dimensional Abstraction

A simple yet powerful feature of LUCID is dimensional abstraction – functions and variables can be defined using dimension parameters. Consider a simple example:

```
sum.z( x, n ) = s asa.z ( #.z == n )
  where
    s = 0 fby.z s + x;
  end
```

We can use `sum` to add all the elements (n^2) elements of a matrix (`a`) by first adding each column (along dimension `i`) and then adding the resulting vector (along dimension `j`).

```
sum.j( sum.i( a, n ), n )
```

Here we are passing to function `sum` the dimension along which it should operate – first `i` and then `j`. Since addition is commutative, we can switch the order i.e., add the rows first and then the resulting column vector.

```
sum.i( sum.j( a, n ), n )
```

We could also define a function `sum2d` that accepts the two dimensions as dimensional parameters and adds the matrix:

```
sum2d.u,v( x, n ) = s asa.u ( #.u == n )
  where
    s = 0 fby.u s + p;
    p = q asa.v (#.v == n )
        where
          q = 0 fby.v x + q;
        end;
  end
```

We can use function `sum2d` as shown below to add the elements of `a`.

```
sum2d.i,j( a, n )
```

Not only are functions dimensionally abstract but so are variables (since they are nullary functions). Here is an example of a "generic" two-dimensional variable `unit` that has the value 1 along the diagonal and 0 elsewhere:

```
unit.i,j = if #.i == #.j then 1 else 0 fi;
```

Now suppose we wanted to define a variable `w` that has the same value as `unit` along dimensions `x` and `y` and a variable `v` that has the same value along dimensions `z` and `x`, we can define them as follows:

```
w = unit.x.y;
v = unit.z.x;
```

Let us consider the definition of function `sum` again. It is easy to see that it is inherently sequential as elements of a vector are added one at a time even though they could be added in parallel (as addition is associative). The idea of a parallel `sum` is to add elements of a vector pairwise and then add the resulting elements pairwise and so on until there is a single element that is the sum of the vector. This can be expressed as follows:

```
psum.z( x, n ) = s asa.t ( m == n )
  where
    index t;
    s = x fby.t ( s + next.z s ) @.z (2*#.z);
    m = 1 << t;
  end
```

Let us examine `psum` in detail. It is a dimensional function that accepts one dimension parameter `z` and two data parameters `x` and `n`. It introduces a local dimension `t`. It is defined as the value of `s` when `m` equals `n` along dimension `t`. Variable `s` that is defined in dimensions `z` and `t` initially corresponds to all the elements of `x` along dimension `z`. At each subsequent `t` dimension context, `s` is the sum of pairs of elements (along dimension `z`) at the previous `t` dimension context. We can write this as follows:-

$$\mathtt{s}_{(\mathtt{z=Z,t=T})} = \begin{cases} \mathtt{x}_{(\mathtt{z=Z,t=0})} & \mathtt{if\ t\ =\ 0} \\ \mathtt{s}_{(\mathtt{z=2Z,t=T-1})} + \mathtt{s}_{(\mathtt{z=2Z,t=T-1})} & \mathtt{otherwise} \end{cases}$$

We can think of `s` as consisting of n elements (along dimension `z`) when `t = 0` and half-as-many elements at each successive `t` dimension context until there is only one element. And this is when `t` dimension context is $\log_2(n)$ or when $2^t == n$ which is captured in `s asa.t m == n` where `m = 1 << t`. Function `psum` is equivalent to `sum` and can be used in its place in adding matrix `a` of order n.

```
psum.i( psum.j( a, n ), n )
```

The computational structure of psum, that of an inverted binary tree, is widely applicable. For example, it can be used to find the maximum of a set of numbers or it can be used to sort elements using mergesort. Instead of instantiating psum for each different use, we can specify the structure as a dimensionally abstract function where the specific use of it (captured in a function) is passed as an argument.

```
linear_tree.X(f, data, size) = tot asa.t tsize <= 1
    where
      index t;
      tsize = size fby.t (tsize+1)/2;
      tot = data fby.t
                 if X >= tsize/2
                    then tot @.X (2 * #.X)
                    else f(tot, next.X tot) @.X (2 * #.X);
                    fi;
    end
```

Function linear_tree not only works when the number of elements is a power of 2 but for any positive number.
The following defines psum in terms of linear_tree:

```
psum.z( x, n ) = linear_tree.z( add, x, n )
                   where add( a, b ) = a + b; end
```

The following defines a function pmax2d that finds the maximum of a two-dimensional matrix a in dimensions i and j in terms of linear_tree:

```
pmax2d.i,j( x, n ) = linear_tree.j( max, linear_tree.i( max, x, n ), n )
                       where max( a, b ) = if a > b then a else b fi; end
```

We can also use linear_tree in matrix multiplication as described on page 23.

```
linear_tree.k(add, (realign.j,k a) * ( realign.i,k b ), n)
  where
    dimension k;
    add( a, b ) = a + b;
  end
```

Another useful computational structure, that we call planar_tree, works just like linear_tree except in two dimensions.

```
planar_tree.X,Y(f, data, size) = tot asa.t tsize <= 1
    where
      dimension t;
      tsize = size fby.t tsize/4;
      tot = data fby.t ( f( tot, next.X tot, next.Y tot, next.X next.Y tot )
                        @.X (2 * #.X) ) @.Y (2 * #.Y);
    end
```

We can redine pmax2d as follows:

```
pmax2d.i,j( a, n ) = planar_tree.i,j( max4, x, n*n )
                       where
                             max4( a, b, c, d ) = max( max(a,b), max(c,d) );
                             max( a, b ) = if a > b then a else b fi;
                       end
```

3.11 Operator Arity

There are two kinds of operator arity: roman arity , which refers to the number of operands and greek arity , which refers to the number of qualifying dimensions.

LUCID operators are either unary , binary , or ternary . For example, `!` and `first` are unary operators whereas `*` and `asa` are binary operators and `if-then-else` is the (only) ternary operator.

LUCID operators are one of nulladic , monadic , dyadic , or polyadic . For example, all arithmetic, relational and logical operators are nulladic as are `iseod` and `iserror` . Monadic operators include `#` , `fby` , `asa` , `wvr` , `upon` , and `before` . The only dyadic operator is `realign` . Operators `first` , `next` , `prev` , and `@` are polyadic.

3.12 Default Dimension Rule

The default dimension rule applies only to monadic operators , i.e., operators which are qualified with a single dimension name. It is as follows.

A monadic operator without a dimensional qualifier is equivalent to a monadic operator with the `time` dimension qualifier unless it appears in a dimensional where-clause in which case the operator is equivalent to the operator qualified by the innermost (or last declared) dimension in the where-clause.

The following definition of `linear_tree` is equivalent to the one given earlier since `asa` refers to `asa.t` and `fby` refers to `fby.t` because of the default dimension rule.

```
linear_tree.X(f, data, size) = tot asa tsize <= 1
    where
      index t;
      tsize = size fby (tsize+1)/2;
      tot = data fby
                 if X >= tsize/2
                    then tot @.X (2 * #.X)
                    else f(tot, next.X tot) @.X (2 * #.X);
                    fi;
    end
```

Chapter 4

From LUCID to GLU

LUCID is purely a dataflow language in which a program is expressed as a structured set of multidimensional data dependencies. Whereas the declarative nature of LUCID results in concise and natural expressions of problem solutions, LUCID programs are incompatible with the conventional imperative model of programming . In particular, LUCID programs do not execute efficiently on control-driven processors and it is difficult to perform imperative activities such as I/O .

GLU, which stands for Granular Lucid, is a hybrid of LUCID and the language C . The basic idea is quite simple – it is to use LUCID as a language for composing applications from C functions. A GLU program is a LUCID program in which user-defined functions can be C functions and values can be (almost) any valid C object such as chars , ints , floats , doubles , fixed-length arrays , structures , and pointers to any of these. Basically, LUCID deals with C functions and C datatypes as uninterpreted entities. C functions are called by value as they are assumed to be pointwise and strict. The only LUCID operators that can be meaningfully applied to C objects are `iseod` which returns 1 if the object is actually `eod` and 0 otherwise and `iserror` which returns 1 if the object corresponds to `error` and 0 otherwise.

A GLU program consists of two parts:

1. C data types and prototype functions declarations
2. LUCID program

4.1 C Data Types

All of the ANSI C type declarations are allowed except for variable-length (incomplete) arrays and unions some of whose members are pointers .

Examples of some C type declarations are given below.

1.
```
struct complex
{
  double re;
  double im;
};
typedef struct complex *COMPLEX;
```
This defines `complex` to be a C type that consists of two floating point members, `re` and `im`. It also defines `COMPLEX` to be an C type that denotes a pointer to object of C type `complex`.

2.
```
struct list
```

```
{
  COMPLEX z;
  string id;
  struct list *next;
}
typedef struct list *LIST;
```

This declares `list` as a C type with `z` being a member of type `complex` and `next` being a pointer (holder of address) to another object of type `list`. Note that `id` is of a predefined type `string` which is equivalent to `char *` of C. It also defines `LIST` to be a C type that is a pointer to structure of C type `struct list`.

3.
```
typedef char STR[128];
```
`STR` denotes a C type that is a fixed-length array of 128 chars.

4. Earlier we indicated that a C object in GLU cannot be of an incomplete array type, i.e., one whose length is not known. GLU provides a way of specifying variable length arrays – the array is defined in a structure with its length being specified by a variable that is also a member of the structure.

```
struct matrix
{
  int x, y;
  COMPLEX m[x][y];
}
typedef struct matrix *MATRIX;
```

This declares `matrix` as C structure type whose members are two integers denoted by `x` and `y` and a variable-sized matrix `m` of elements of type `COMPLEX` with the number of rows denoted by `x` and the number of columns denoted by `y`. It also defines `MATRIX` as a pointer to structure of type `matrix`.

5. GLU also excludes C objects to be of a union type where one (or more) of its members is a pointer. In GLU, any such union has to be discriminated in that an additional field `t` has to be used to identify the current type of the value.

```
union multi switch( int t ) {
  case 0: int i;
  case 1: COMPLEX v;
  case 2: MATRIX m;
}
```

The C type `multi` has two members: a union of `int i` and pointers types `COMPLEX v` and `MATRIX m`; and `t` that indicates which of the three types in the `union` is currently applicable.

4.2 C Prototype Function Declarations

In a GLU program, the prototype of each C function that is used is given – the function definition itself is given elsewhere. A prototype function declaration specifies the parameter types and the result type of each function. It is of the form:

```
C-TYPE C-function-name( C-TYPE, ..., C-TYPE );
```

Here are some examples:

```
COMPLEX add( COMPLEX, COMPLEX );
struct matrix in();
int display( STR );
COMPLEX kth( LIST, int );
MATRIX mult( MATRIX, MATRIX );
```

Function `add` accepts pointers to two objects of type `complex` and returns a pointer to the result that is of type `complex`. Function `in` is nullary but returns an object of type `struct complex`. Function `display` accepts one argument, a fixed-size array of `char` and returns an integer. Function `kth` returns a pointer to a `complex` object and accepts a pointer to a `list` object and an integer as its arguments. Finally, function `mult` accepts two pointers to matrices and returns a pointer to a matrix.

4.3 C Function Definition Conventions

A C function definition needs to be consistent with the prototype definition of that function in its arity and types of parameters and result. For example, consider the function `add` whose prototype declaration was `COMPLEX add( COMPLEX, COMPLEX )`. Its definition would be as follows:

```
COMPLEX add( COMPLEX a, COMPLEX b )
{
  COMPLEX c;

  c = (COMPLEX) malloc( 1, sizeof(*COMPLEX) );

  c->re = a->re + b->re;
  c->im = a->im + b->im;

  return( c );
}
```

The parameters and the result are of type `COMPLEX` and the C function creates the object (using `malloc`) that `add` returns.

4.3.1 Parameter Passing

When a C function is used in a GLU program, parameter passing is call-by-dereferenced-value — all values when applicable are dereferenced. Unlike in C, it is not possible for a C function to modify the object pointed to by an actual parameter – only the copy of the object would be modified.

In the above example, setting `a->re` to 0.00 would not cause the `re` member of the actual parameter to be modified.

4.3.2 Returning Dynamic Objects

When a C function needs to return a pointer to an object, it has to create the object (using `malloc`) and return the pointer to the object. This is shown in the example above with `c`. When a parameter or result is a pointer, the C function is *not* responsible for freeing the objects pointed by the parameter or result.

4.3.3 Globals

In general, C functions should avoid sharing any data using global variables since it undercuts the high-level view of the GLU model of programming. In addition, global variables have implementation-specific semantics making them a "use at your own risk" feature.

Global variables can be used to share common data across functions provided the functions execute in the *same* address space. (This can be specified using annotations – see Section 4.6.)

Global variables can also be used to retain data across function invocations. Again, successive invocations of the function are expected to be in the *same* address space.

When using global variables, the programmer is completely responsible for the management of the global variables. As far as GLU is concerned, global variables are part of the external world over which it has no control.

4.3.4 Static Variables

Like global variables, static variables do not necessarily have the semantics as in sequential C programs. Whenever such a variable is used in GLU programs, it is the programmer's responsibility to ensure that the "right" static variable is being accessed as there are likely to be multiple copies, one per each address space.

4.4 Preprocessor

GLU supports C's preprocessing capability including macro substitution, conditional compilation, and inclusion of named files.

4.5 Comments

Any line starting with `//` is treated as a comment.

4.6 Annotations

GLU allows two types of annotations – one for specifying whether functions are to be executed locally or remotely and another for specifying a particular remote location for function execution.

Both these annotations enable the programmer to exercise control over the mapping of computations to processors. While this is counter to GLU, it is necessary because the current GLU compiler assumes that the underlying concrete architecture is an embodiment of a PRAM. Consequently, the compiler does not attempt to intelligently distribute data and its processing as there is no benefit to doing so. With annotations, it is possible for a programmer to explicitly manage the distribution of data and processing although such management is likely to be ad hoc.

4.6.1 Local versus Remote Function Execution

In GLU, each function prototype function definition can optionally be prefixed by the keywords `local` or `remote`. (The default is `remote`.)

When a function prototype declaration is prefixed as `local`, it is executed in the same address space where it was invoked.

When a function prototype declaration is prefixed as `remote` or not prefixed, it is executed remotely on another address space.

Here are some examples:

```
local COMPLEX add( COMPLEX, COMPLEX );
local struct matrix in();
local int display( STR );
COMPLEX kth( LIST, int );
remote MATRIX mult( MATRIX, MATRIX );
```

Functions `add`, `in`, and `display` are executed in the same address space from where they are invoked and functions `kth` and `mult` are executed in a remote address space.

4.6.2 Remote Execution Mapping

Each function invocation when suffixed by a non-negative integer within square brackets specifies that the function should be executed in the virtual address space identified by the integer. (In actuality, multiple virtual address spaces would be mapped to a single physical address space.) The integer could be a constant or it could be any valid GLU expression.

Here are some example expressions:

```
mult[time](a,b)
kth[f(i)]( l, i )
mult[1](a,b)
```

In the first example, `mult` is executed in the virtual address space specified by `time`. In the next example, `kth` is executed in the virtual address space derived by applying function `f` to `i` in the current context. In the last example, `mult` is always executed in virtual address space `1`.

Chapter 5

Compilation and Execution

This chapter describes the process of compiling GLU programs to machine-specific executables. It also describes how these executables can be run with various options.

5.1 Source to Executables

GLU compilation occurs in two phases. In the first phase, the GLU program with the appropriate C functions are converted to a generator or master program and an executor or worker program. In the second phase, the master program and the worker program are compiled to machine-specific executables.

Assume we are given a GLU program in a file named `foo.g` and the accompanying C functions in a separate file named `foo.c`. To compile this program, we issue the following command:

```
gc foo.g foo.c
```

This results in two executables to be generated – `foo` and `foo-wrk` corresponding to the generator (or master or server) and executors (or worker or client).

If `foo.o` is already available, it is possible to avoid the compilation of `foo.c` and directly use `foo.o`:

```
gc foo.g foo.o
```

During the compilation of `foo.g`, a header file `foo_glu.h` is automatically generated which should be included in `foo.c`. This header file consists of all data types that are defined in `foo.g` and used in `foo.c`.

It is possible to prevent generation of the header file (in case you already have one that you want to preserve) by using the `-nohdr` option.

```
gc -nohdr foo.g foo.c
```

It is possible to indicate that only a sequential executable is to be generated using the `-seq` option. The default is to generate executables for parallel execution (which can be explicitly indicated with a `-par`) option.

```
gc -seq foo.g foo.c
```

In this case, only `foo` is generated as the executable.

When parallel executables are being generated, it is possible to indicate that the architecture is homogeneous to enable faster communication. This is done with the `-hom` option. The default is to assume a heterogeneous architecture.

```
gc -hom foo.g foo.c
```

It is also possible to specify directories to search for including header files and linking with libraries. Assume that in addition to the current directory, it is necessary to include header files in a subdirectory called `include`. This can be indicated as follows:

```
gc -I./include foo.g foo.c
```

Assume that two standard libraries `libX11.a` and `libm.a` need to be used at link time along with library `libfoo.a` under `./lib`. This can be indicated as follows:

```
gc foo.g foo.c -L./lib -lX11 -lm -lfoo
```

All the rules for linking from C compilation apply.

It is also possible to compile the program so that it can be symbolically debugged:

```
gc -g foo.g foo.c
```

Some of the other options include `-v` for verbose compilation; `-pre` for preserving the source files generated in the first phase of the compilation: `foo-gen.c`, `foo-com.c`, and `foo-wrk.c`; and `-O` for optimized code generation.

5.2 Running Program Executables

When you execute a GLU program, by default you get an output stream in the `time` dimension starting with `time=0`. The program terminates when the output (at some `time` context) is `eod` after all outputs at previous `time` contexts have been generated. What is displayed is the scalar version of the object specified at the head of the program's `where` clause.

Suppose that we compile and execute the following program:

```
fib where
  fib = 1 fby 1 fby fib + next fib;
end
```

The output would be:

```
[0]: 1
[1]: 1
[2]: 2
[3]: 3
[4]: 5
[5]: 8
[6]: 13
...
```

If the head of the program above is `fib @ 6 fby eod`, the output would simply be:

```
[6]: 13
```

Let us find out how we run the executables generated by GLU programs. If the executable is the result of a standard GLU compilation, you have to run the master program and one or more worker programs. In the example below, the master is being run with one worker.

```
foo &
foo-wrk &
```

It is possible to have multiple workers, and it is up to the programmer to run them on different processors (which we will assume are workstations):

```
foo &                                      % master on workstation wkstn0
rsh wkstn1 <FULL_PATH_NAME>/foo-wrk wkstn0 &  % worker on workstation wkstn1
rsh wkstn2 <FULL_PATH_NAME>/foo-wrk wkstn0 &  % worker on workstation wkstn2
...
rsh wkstnN <FULL_PATH_NAME>/foo-wrk wkstn0 &  % worker on workstation wkstnN
```

Note that `<FULL_PATH_NAME>` of `foo-wrk` is needed unless it is at the top level of the user directory. Also note that workers can be added and deleted during execution without affecting program functionality.

If the executable is the result of a sequential GLU compilation, running the program is exactly as in `C`. You simply enter the name of the program:

```
foo
```

If you want to generate traces of program execution, you would use the `-debug` option with the master and with the worker:

```
foo -debug &
foo-wrk -debug &
```

The option `-debug` gives minimal tracing – when functions become executable and when they return their results. It is possible to get a more detailed trace of the `master` by using `-debug=2` and `-debug=3`.

If you want to control garbage collection of a GLU program, you can do so using the `-sweep` and `-age` options.

```
foo -sweep=2 -age=10
```

The above command says that garbage collection sweeps should occur every 2 seconds of the program's virtual execution time (not elapsed time) and that a value can be discarded if it has been around for 10 sweeps without being used.

If you want to override any of the `local` or `remote` annotations at runtime, you can do so:

```
foo -local f1 -remote f2 -remote f3 -local f4
```

The above command says that functions `f1` and `f4` will be treated as `local` and functions `f2` and `f3` will be treated as `remote`.

The `-w` option is necessary when remote execution mapping annotation is used in the GLU program. You can prespecify the number of workers that the master will expect with the `-w` option:

```
foo -w10
```

This command says that precisely 10 workers are expected by the master.

Another useful option is the logging option (`-log`) that causes a logfile containing execution traces to be generated each time the program is executed that can be analyzed in post-mortem mode.

```
foo -log=foo.log
```

This command says that logging information should be written to file `foo.log`. The default is `logfile`. Note that logging information is complete and accurate only when the program terminates normally.

Bibliography

[1] V. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

[2] N. Carriero and D. Gelernter. How to write parallel programs: A Guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[3] E.A. Ashcroft, A.A. Faustini, and R. Jagannathan. An Intensional Parallel Processing Language for Applications Programming. In B.K. Szymanski, editor, *Parallel Functional Programming Languages and Environments*, chapter 2. ACM Press, 1991.

Appendix A

Syntax of GLU Programs

A.1 Scope Rules

A.1.1 Program Variables

The scope of an occurrence of a variable used in a GLU `where` clause is either *local* or *global*. It is local if it is defined in that clause. If it is not local, it must be global and defined as the closest lexically enclosing `where` clause in which the variable is defined. There are no free variables in GLU.

A.1.2 Dimensional Names

The scope of dimensional names is defined in the same way as the scope of program variables. The exception to this rule is the implicit dimensional name `time`, which is predefined.

A.1.3 Precedence and Associativity Rules

We list the hierarchy of precedences amongst LUCID syntactic units. Syntactic units with highest precedences are at the top of the list, and ones with lowest precedences are at the bottom.

Precedence	Syntactic Unit	Associativity
Highest	`()`, ‘.’, ‘,’ (delimiter)	left to right
	`first`, `next`, `prev`, `iseod`, `iserror`, unary `!`, `verb+ +`, `+`, `-`	right to left
	`@`	right to left
	`*`, `div`, `/`, `%`	left to right
	`+`, `-`	left to right
	<<, >>	left to right
	<, <=, > , >=	left to right
	`==`, `!=`	left to right
	`&`	left to right
	^	left to right
	|	left to right
	`&&`	left to right
	||	left to right
	‘,’, `!`, `?`	left to right
	`asa` , `upon` , `wvr`	left to right
Lowest	`fby` , `before`	right to left

A.1.4 Reserved Words

These identifiers in addition to those of ANSI C are reserved words in GLU .

```
if
then
else
fi
dimension
index
where
end
first
next
prev
fby
before
asa
wvr
upon
time
div
true
false
eod
iseod
realign
```

A.2 Extended BNF Syntax for GLU Programs

```
<program>              ::=   { <procedural definition> | <non-procedural definition> }*
```

```
                               <expression>
<procedural definition> ::=    <type declaration> |
                               <prototype declaration>
<type declaration>      ::=    SAME AS IN ANSI C
<prototype declaration> ::=    SAME AS IN ANSI C
<non-procedural definition>
                        ::=    <dimension definition> |
                               <variable definition>  |
                               <function definition>
<dimension definition>  ::=    dimension <dimension list> ; |
                               index <dimension list> ;
<dimension list>        ::=    <dimension name> { , <dimension name> }
<dimension name>        ::=    <identifier>
<variable definition>   ::=    <variable name> { . <dimension list>  } = <expression> ;
<variable name>         ::=    <identifier>
<function definition>   ::=    <function name> { . <dimension list> }
                               ( <formal parameters> ) = <expression> ;
<function name>         ::=    <identifier>
<formal parameters>     ::=    <identifier> {, <identifier> }
<expression>            ::=    <constant> |
                               <variable name> |
                               <dimension index> |
                               <fcall> |
                               <uop> <expression> |
                               <expression> <bop> <expression> |
                               if <expression> then <expression> else <expression> fi |
                               ( <expression> ) |
                               <where expression>
<constant>              ::=    eod | error | true | false | <integer> | <real>
<dimension index>       ::=    # . <dimension name> | <dimension name>
<fcall>                 ::=    <function name> { . <dimension list> }
                               ( <expression list> )
<expression list>       ::=    <expression> { , <expression> }
<uop>                   ::=    <simple uop> | <complex uop> | iseod | iserror
<simple uop>            ::=    ! | ˜ | - |
<complex uop>           ::=    <lucid uop> { . <dimension list> }
<lucid uop>             ::=    first | next | prev | realign
<bop>                   ::=    <simple bop> | <complex bop>
<simple bop>            ::=    <arith bop>  |
                               <relational bop> |
                               <logical bop> |
                               <bitwise bop>
<arith bop>             ::=    + | - | * | / | div | %
<relational bop>        ::=    < | > | <= | >= | == | !=
<logical bop>           ::=    && | ||
<bitwise bop>           ::=    << | >> | ‘|’ | \& | \^
<complex bop>           ::=    <lucid bop> { . <dimension list> }
<lucid bop>             ::=    fby | before | asa | wvr | upon | ‘@’ | ‘?’ | ‘!’ | ‘,’
<identifier >           ::=    <letter> { < alpha_numeric> }
<alpha_numeric >        ::=    <letter> |  _ | <digit>
<where expression>      ::=    <expression> where
                               { <procedural definition> | <non-procedural definition> }*
                               end
```

Appendix B

Predefined Functions

GLU provides a standard set of predefined dimensionally-abstract functions that can be used as templates to construct GLU programs. The functions reside in a file called `std.g` which can be included in any GLU program.
The contents of `std.g` are as follows:

```
lparent.X(f) = f @.X (2 * #.X);
rparent.X(f) = f @.X (2 * #.X + 1);

linear_tree.X(f, data, size) = first.X(tot asa.t tsize <= 1)
    where
        index t;
        tsize = size fby.t (tsize+1)/2;
        tot = data fby.t
                if X >= tsize/2
                    then tot @.X (2 * #.X)
                    else f(tot, next.X tot) @.X (2 * #.X)
                    fi;
    end;

planar_tree.X,Y(f, data, size) = first.X,Y(tot asa.t tsize == 1)
    where
        index t;
        tsize = size fby.t tsize/4;
        tot = data fby.t (f(tot, next.X tot, next.Y tot, next.X,Y tot)
                    @.X (2 * #.X)) @.Y (2 * #.Y);
    end;

cubic_tree.X,Y,Z(f, data, size) = first.X,Y,Z(tot asa.t tsize == 1)
    where
        index t;
        tsize = size fby.t tsize/8;
        tot = data fby.t ((f(tot, next.Z tot, next.Y tot, next.Y,Z tot,
                        next.X tot, next.X,Z tot, next.X,Y tot,
                        next.X,Y,Z tot)
                      @.X (2 * #.X)) @.Y (2 * #.Y)) @.Z (2 * #.Z);
    end;
```

Appendix C

Manual Pages for GLU Commands

NAME

gc – GLU compiler

SYNOPSIS

gc [**–com** | **–gen** | **–seq** | **–wrk**] [**–G**] [**–nohdr**] [**–pre**] [**–rec**] [**–hom**] [**–v**] [*gcc options*] *sourcefiles* . . .

DESCRIPTION

gc is the GLU compiler front end. It invokes the GLU compiler to translate programs written in GLU into C programs and invokes *gcc* to compile and link them into an executable image. Most of the command line options are simply passed on to *gcc* or *ld* as appropriate. *gc* also understands the various filename suffixes understood by *gcc* and handles them appropriately. Files ending in *.g* are taken to be GLU source programs and will sent to the GLU compiler before being passed on to *gcc.*

OPTIONS

–com Generate code that is common to both the generator and worker. This option is meaningful only when combined with **–c** or **–G.**

–gen Generate code for the generator. When not combined with **–c** or **–G**, it will also produce the common code – which is required to produce an executable generator.

–seq Generate code for a sequential program. This is much like the **–gen** option, except it links with a special library that will treat all C functions as local functions and that does not use any workers.

–wrk Generate code for the worker. When not combined with **–c** or **–G**, this option will also produce the common code – which is required to produce an executable worker.

–G Produce C code. Doesn't call *gcc* on the results of the GLU to C translator. The C code will be left in files with suffixes *com.c,* gen.c, and *wrk.c.*

–nohdr Do not produce a header file. The default is to produce a header file with the suffix *glu.h,* which will contain all the C declarations from the GLU source file.

–pre Preserve intermediate files. Normally, intermediate files (generated C code and object files) are placed in */tmp* and are removed when compilation is complete. With this option, they will be placed in the current directory and will not be removed.

–rec This special flag is required to get the communications code to work correctly if you have defined recursive structures (structures that contain pointers to themselves, directly or indirectly). It results in less efficient code if its not needed. It has no effect on sequential programs.

–hom This flag makes the communications code assume that the workers and generator are running on compatible machines. It results in more efficient code which will fail if run on different machines. It has no effect on sequential programs.

−v Verbose. Show the commands constructed by the compilation driver before executing them.

EXAMPLES

The following rules can be useful in Makefiles

```
%_com.c %_gen.c %_wrk.c %_glu.h: %.g
                                        gc -G -hdr $<

prog: prog_gen.o prog_com.o $(OTHERS)
                                        gc -gen -o $@ $<

prog-wrk: prog_gen.o prog_com.o $(OTHERS)
                                        gc -wrk -o $@ $<
```

ENVIRONMENT

CC C compiler to use. Defaults to *gcc.*

CPP C preprocessor to use. Defaults to *gcc* -E -x c.

GLUHOME Directory containing the various GLU compiler passes, include files, and libraries. Default value set on installation.

TMPDIR Directory to use for intermediate files. Defaults to */tmp.*

FILES

file.**g**	GLU source file
*file***gen.c**	Compiled code for generator
*file***wrk.c**	Compiled code for worker
*file***com.c**	Compiled code common to both worker and generator
*file***glu.h**	C declarations extracted from GLU source file
file	Executable generator or sequential program
file–**wrk**	Executable worker program

SEE ALSO

glu(1), gcc(1).

BUGS

Please report bugs to **glu-bugs@csl.sri.com**

NAME

GLU runtime system and general information

SYNOPSIS

name [**–age=***nn*] [**–bs=***size*] [**–debug**[=*nn*]] [**–quiet**] [**–local** *function*] [**–log**[=*file*]] [**–random**[=*seed*]] [**–remote** *function*] [**–sweep=***nn*] [**–w=***nn*]

name-**wrk** [**–bs=***size*] [**–debug**] [*hostname*]

DESCRIPTION

The Glu compiler, when producing parallel code, produces two executables, a generator and a worker. To execute in parallel, you need to invoke one copy of the generator and one or more copies of the worker.

The generator runs the main LUCID program, executing all lucid code and all local C functions. Remote C functions will be farmed off to workers. The system can detect and survive the failure of any worker, and will redo any uncompleted job of the failed worker elsewhere.

GENERATOR OPTIONS

–age=*nn* Sets the number of value cache sweeps which must pass for an object to be thrown out of the value cache, since the object's last reference. Default is 4.

–bs=*size* Set the buffer size for all communications with workers. Large buffers are more efficient if you pass around large amounts of data, but require more memory. You may specify a suffix of **k** or **K** to specify kilobytes instead of bytes. Default is 512.

–debug[=*nn*] Set or increment the debugging level. This controls the amount and detail of debugging information that is printed. Each level outputs the information of all lower levels.

1	Information about C functions as they are invoked and complete
2	Information about each demand and value that is cached.
3	Information about each demand and value of a named object.
4	Information about every demand and value.

–local *function* Treat *function* as a local function (execute in the generator).

–log[=*file*] Log various useful statistics to *file*. If *file* is unspecified, logs to **logfile**.

–quiet Suppress then normal display of each top level value as it is produced. If this option is supplied and **-debug** is not, the only output produced will be from local C functions.

–random[=*seed*] Normally, remote C functions are assigned to workers on a FCFS basis. This option causes them to be assigned in a random fashion.

–remote *function* Treat *function* as a remote funtion (pass of to a worker for execution).

–sweep=*nn* Set the time between value cache sweeps. Along with **-age** this controls how quickly old objects will be flushed from the cache.

–w=*nn* Specify how many workers there will be for use with the worker-spec syntax in the input file. If you don't use worker-specs, this will have no effect.

WORKER OPTIONS

–bs=*size* Set the buffer size for all communications with the generator.

–debug Turn on worker debugging information. This is the same information as level 1 on the generator, but on a per-worker basis.

hostname Specifies which machine the generator is running on. Defaults to **localhost.**

SEE ALSO

gc(1).

BUGS

Please report bugs to **glu-bugs@csl.sri.com**

Index

!, 26
!, 18
    =, 14
#, 15, 26
%, 14
&, 14
&&, 14
*, 14, 26
+, 14
,, 18
-
    binary, 14
    unary, 14
/, 14
//, 30
<, 14
<<, 14
<=, 14
==, 14
>, 14
>=, 14
>>, 14
?, 18
@, 15, 26
#.time, 9
^, 14
|, 14
||, 14
~, 14

adaptivity, 2
address space, 30
    physical, 31
    virtual, 31
annotation, 30
    local, 30, 34
    mapping, 34
    remote, 30, 34
    remote execution mapping, 31
architecture
    heterogeneous, 33
    homogeneous, 33
arity
    greek, 26
    roman, 26
array
    fixed length, 27
    variable length, 27
asa, 16, 26

before, 26

C, 1, 3, 27
    ANSI, 8, 27
    reserved words, 37
C function
    global variables, 30
    parameter passing, 29
    returning result, 29
    static variables, 30
call-by-dereferenced-value, 29
call-by-need, 8, 19
call-by-value, 8, 19, 27
char, 27
comments, 30
compilation, 32
composition, 8, 9
composition function, 8
    dimensionally abstract, 11
computation, 9
    array, 22
    doubly nested, 21
    multidimensional, 10
    nested, 20, 21
constant
    in LUCID, 13
context, 3, 9, 19
convention
    C function definition, 29
cubic_tree, 39

dataflow
    multidimensional, 1
debugging, 33, 34
declaration
    data type, 8, 27
    fixed length, 28
    function prototype, 8, 27, 28
    recursive structure, 28
    structure, 27
    union, 28
    variable length, 28
default dimension rule, 26
dimension
    nested, 21
`dimension`, 21
dimensional abstraction, 23
dimensions, 20
`double`, 27

`eod`, 13, 27, 33
`error`, 13, 17, 27
executable, 32
    parallel, 32
    sequential, 32
execution
    parallel, 34
    sequential, 34
expression, 13
    compound, 18
    constant, 18
    function, 18
    operator, 18
    variable, 18

`fby`, 15, 26
fibonacci sequence, 3
`first`, 14, 26
`float`, 27
function
    dimensionally abstract, 24, 39
    GLU, 8
    higher-order, 20
    LUCID, 19
    nonrecursive, 10
    nullary, 24
    predefined, 39
    procedural, 8
    recursive, 3, 10, 19

global variables, 30
GLU
    comparison, 1
    compiler command, 43
    EBNF syntax, 37
    execution command, 45
    extension of LUCID, 4
    manual pages, 40
    overview, 1
    reserved words, 37
grabage collection, 34
granularity, 1, 10

header file
    generation, 32
    inclusion, 33

I/O, 27
`if-then-else`, 15, 26
`int`, 27
`iseod`, 13, 17, 26, 27
`iserror`, 13, 17, 26, 27

language
    dataflow, 27
    functional, 3
    indexical, 3
    procedural, 3
lazy evaluation, 19
libraries
    standard, 33
Linda, 1
`linear_tree`, 25, 26, 39
load balancing, 1
logging, 34
LUCID, 1, 3, 27
    as a functional language, 3
    as an indexical language, 3
    circa 1994, 13
    nucleus of GLU, 3, 13

mapping, 1
master, 32
matrix multiplication, 4, 23
matrix transposition, 22
model of programming
    declarative, 3
    GLU, 3
    imperative, 27

multidimensional array, 22

Newton's square root algorithm, 21
`next`, 14, 26

operator
- arithmetic, 14
- associativity rules, 36
- binary, 26
- bitwise logical, 14
- dyadic, 26
- logical, 14
- monadic, 26
- nonpointwise, 14
- nulladic, 26
- pointwise, 14
- polyadic, 26
- precedence rules, 36
- predefined, 14
- relational, 14
- ternary, 26
- unary, 26

operator arity, 26

parallel computers, 4
partitioning, 1
`planar_tree`, 25, 39
pointer, 27
- in union, 27

preprocessor, 30
`prev`, 26
process architecture, 1
program
- GLU, 8
- LUCID, 27
- master, 32, 34
- procedural, 4, 8
- worker, 32, 34

program execution, 33
programming
- declarative, 3
- imperative, 3
- procedural, 3

PVM, 1

`realign`, 22, 23, 26
referential transparency, 4, 8
reserved words, 37

scope
- dimensional names, 36
- global, 36
- local, 36

sequence
- constant, 13
- value, 4

static variable, 30
structure, 27
successive over-relaxation, 22

`time` dimension, 4, 21, 33, 36
`time` context, 33
trace generation, 34
types
- in GLU, 27
- in LUCID, 13

union, 27
`upon`, 17, 26

variable, 4, 13

`where` clause, 18, 36
- dimensional, 20

`where` clause, 21
`where` clause
- dimensional, 21, 22

worker, 32
workstation, 34
`wvr`, 16, 26