1. R. Jagannathan and C. Dodd. GLU programmer's guide (version 0.9). Technical Report SRI-CSL-94-06, Computer Science Laboratory, SRI International, Menlo Park, California 94025, U.S.A, July 1994.
2. E.A. Ashcroft and R. Jagannathan. Operator nets. In *Fifth Generation Computer Architectures*, J.V. Woods (editor), pages 177–202, North-Holland, 1986.
3. W.H. Mitchell. Distributed visual dataflow. Ph.D. Thesis, Arizona State University, Computer Science and Engineering Department, Tempe, Arizona, 1991.
4. D. Rajender and A.A. Faustini. VIPER: A visual programming environment for GLU. In Proceedings Seventh International Symposium on Lucid and Intensional Programming (`http://www.csl.sri.com/lucid/ISLIP94/electronic-proceedings.html`), pages 113–123, SRI International, September 1994,
5. W. Du and W.W. Wadge. A 3D spreadsheet based on intensional logic. *IEEE Software*, 7(3):78–89, May 1990.
6. W. Du and W.W. Wadge. The eductive implementation of a 3D spreadsheet. *Software–Practice and Experience*, pages 1097–1114, November 1990.

Both the dependency-graph model and the nested-spreadsheet model represent a program as a set of sheets, with one graphical sheet for each variable or function definition. The structured use of multiple sheets helps better manage visual complexity.

In the dependency-graph model, the expression associated with a variable is represented as a directed acyclic graph with references to variables and functions being denoted symbolically. In the nested-spreadsheet model, the definition of the variable or function is given textually.

In the dependency-graph model, dimensionality is represented using colors with each graphical sheet also showing the relationship between dimensions. In the nested-spreadsheet model, dimensionality is derived from the rank and represented as a spreadsheet with each cell consisting of the defining expression after it has been symbolically reduced based on dimensional position information.

The main advantage of the dependency-graph model is its ability to reveal the static structure of the program in terms of both data and dimensional dependencies. However, the model does not reveal the dynamic structure of the program. This is consistent with the model being intensional.

The main advantage of the nested-spreadsheet model is its ability to reveal the dynamic multidimensional structure of the program through nested spreadsheets. This is consistent with the model being extensional. However, the model relies on a textual description of the static structure of the program, thus making data dependencies less apparent.

5. Conclusions

We have described two different graphical models for GLU programming, namely the dependency-graph model and the nested-spreadsheet model. The intensional dependency-graph model is intended to reveal static dependency and dimensionality information about a program whereas the extensional nested-spreadsheet model is designed to reveal dynamic dependency and dimensionality information.

An effective way to understand the relative strengths and drawbacks of the two models for GLU programming is to implement them as tools and use these tools to build GLU programs. We intend to pursue this in the near future.

6. Acknowledgments

This work was supported by SRI International through its IR&D investment program and by the United States National Science Foundation under grant number CCR-9203732.

7. References

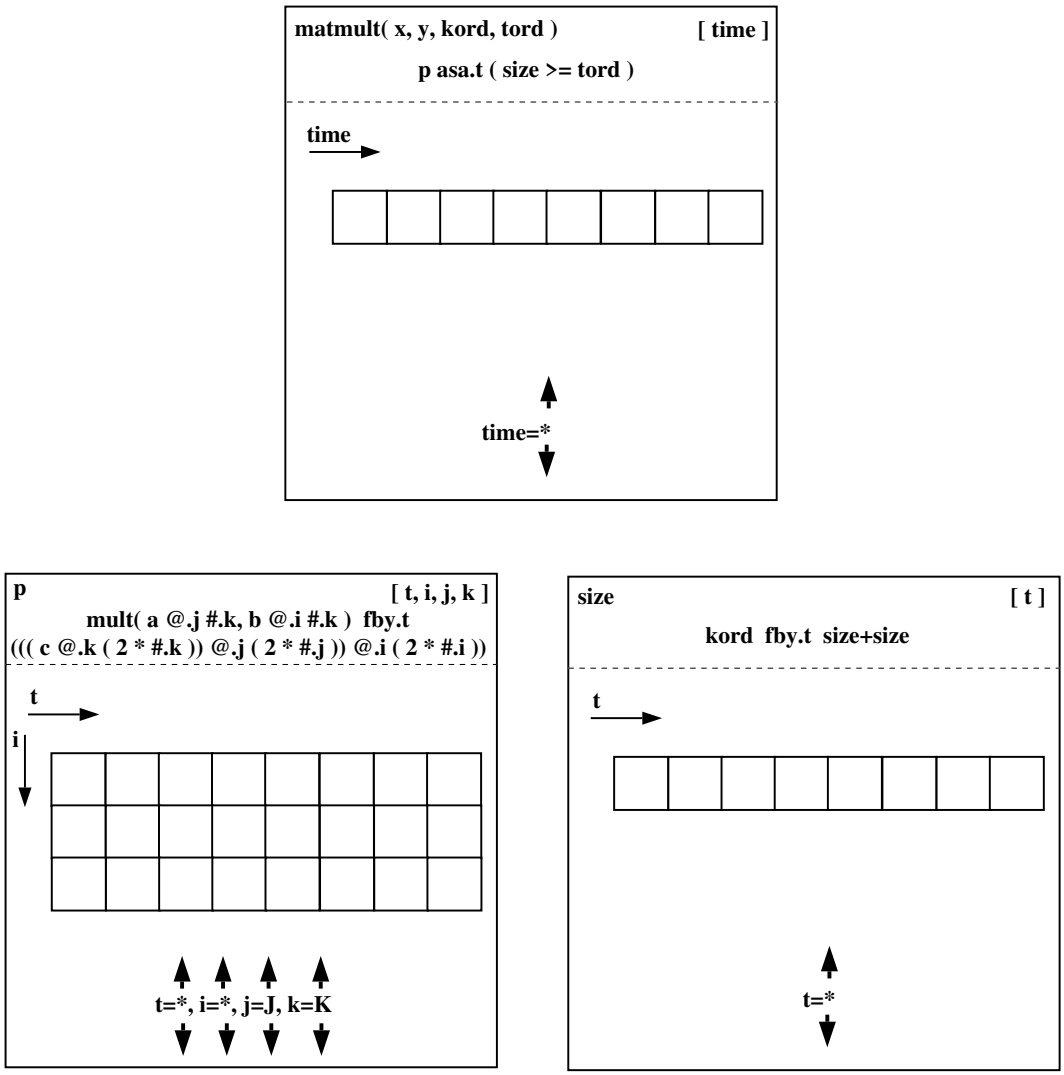


Fig. 5. Nested-Spreadsheet Representation of Matrix Multiplication

denotes the expression `size+size @ t=`i`-1` where i refers to the cell position.

The spreadsheet representation of `p` has a rank of `t`, `i`, `j`, and `k` of which only two, `t` and `i`, are shown; the positions of the other two dimensions being fixed. The expression in the column associated with `t=0` is `mult( a @ [t=0,i=`I`,j=`K`], b @ [t=0,i=`K`, j=`J`] )` where I refers to the row position of each cell in the column. The expression in the subsequent columns (with `t=`T) is `c @ [t=`$T-1$`,i=2*`I`, j=2*`J`, k=2*`K`]` where T refers to the column position and I refers to the row position.

4. Comparison of the Two Models

In this section, we compare and contrast the approaches taken by the two models to reveal dependency and dimensionality information associated with GLU programs.

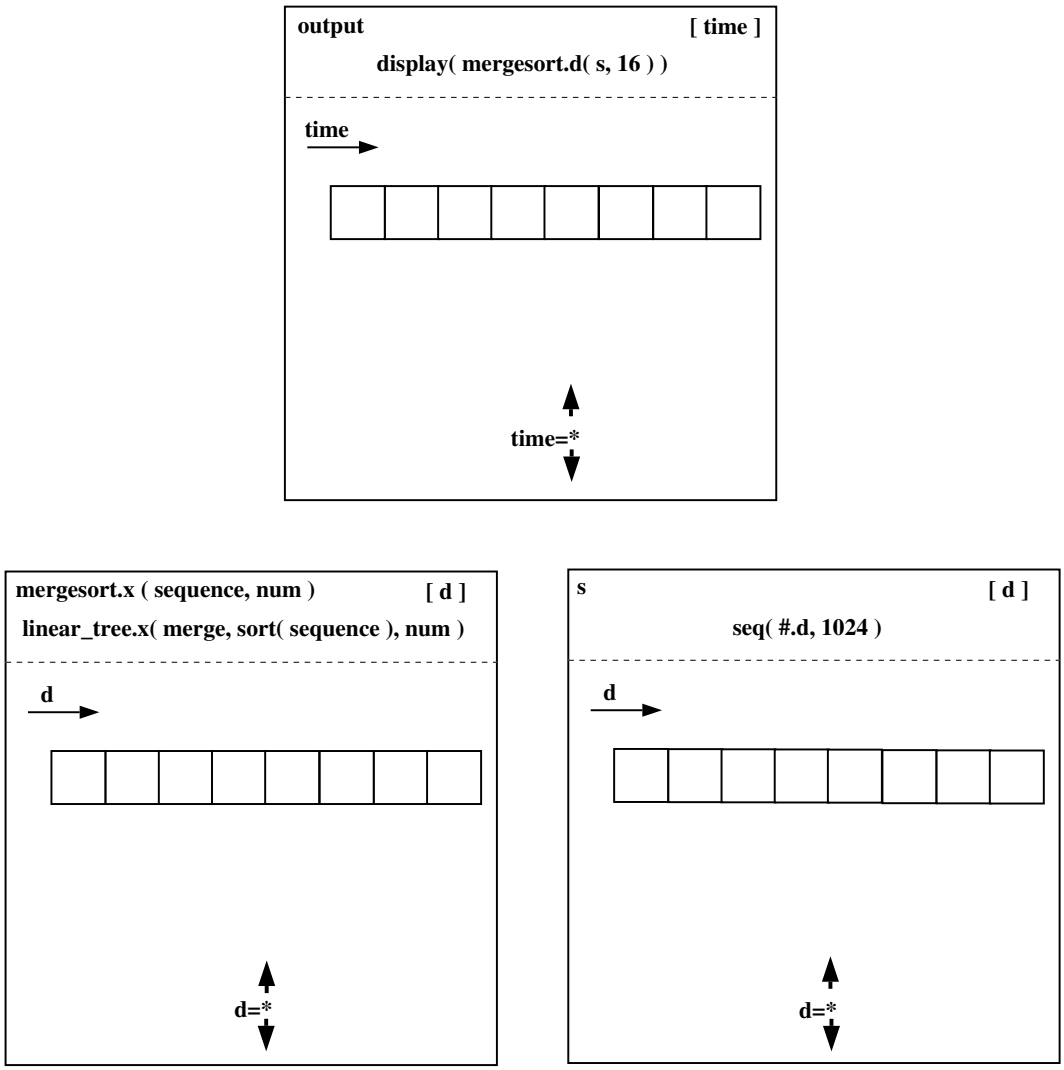


Fig. 4. Nested-Spreadsheet Representation of Mergesort

null.)

The sheet for `s` has a rank of `d`. Its spreadsheet representation is a one-dimensional spreadsheet in the `d` dimension with the i^{th} cell containing the expression `seq( i, 1024 )`.

3.2. Matrix Multiplication Program

The matrix multiplication function under the nested spreadsheet model consists of six graphical sheets, of which three are shown in Figure 5.

The function definition sheet for `matmult(x,y,kord,tord)` denotes it as a `time`-varying sequence although it would equally valid for the rank to be simply *null.*

The spreadsheet representation of `size` is a `t`-varying one-dimensional sequence whose first cell denotes the expression `kord @ t=0` and each of its subsequent cells

grams — one based on spreadsheets. The relationship between flat Lucid programs and spreadsheets is well established.[5] The essential idea is to denote each variable with a spreadsheet, each of whose axes denotes orthogonal dimensions. With this idea, it is possible to represent two (and even three) dimensions. In this section, we propose a spreadsheet model for representing GLU programs that have nested user-defined multidimensional structures.

A GLU program under this model consists of a set of "sheets" with one "main sheet" and several "subsidiary sheets." Associated with each sheet is a name also known as its "folio." There are two kinds of sheets — "graphical" which hold a spreadsheet definition and "textual" which hold a C function definition. (This syntax and the syntax of the data-dependency graph model described earlier have much in common.)

The main sheet graphically describes the program expression whereas variables and functions referred to in the program expression are described in subsidiary graphical sheets, one per sheet.

Each graphical sheet consists of the name of the sheet (folio) on the top left corner and the rank (or dimensionality) of the variable on the top right corner. In the upper middle part of the sheet is the defining expression itself. The spreadsheet representation of the variable or function is in the middle part of the sheet with the dimensionality of the spreadsheet being given by the rank. The bottom part of the sheet holds dimensional position information associated with the variable or function.

In the spreadsheet representation of a variable, each cell contains the symbolically simplified defining expression prior to the cell value being computed, and the value itself after it has been computed. References in a cell expression to other cells of the same spreadsheet are denoted by connecting arcs whereas references to cells of another spreadsheet are referred to symbolically.

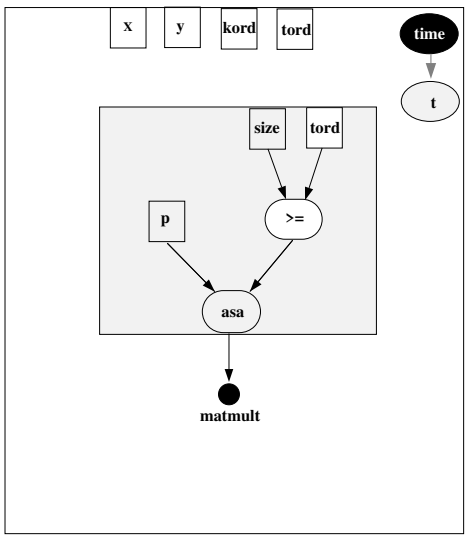
When the rank of a variable exceeds two, the spreadsheet representation is two-dimensional with user-control over the choice of dimensions and over the positions in the unrepresented dimensions. This control is exercised using the dimensional information (in the bottom part of the graphical sheet).

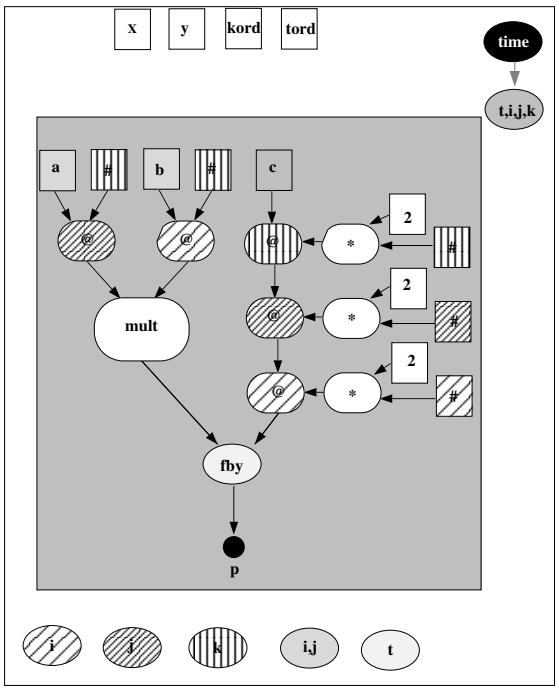
3.1. Mergesort Program

The mergesort program under the nested spreadsheet model is shown in Figure 4. It consists of three graphical sheets one of which is the main sheet.

The main sheet consists of the program expression `display(mergesort.d(s,16))` which varies over the `time` dimension. The program is represented as a one-dimensional spreadsheet that varies in the `time` dimension.

The sheet for function `mergesort`, which is defined within the scope of the `d` dimension, is simply an invocation of another function `linear_tree`. (As the definition of `mergesort` is invariant with respect to dimension `d`, its rank could just as well be





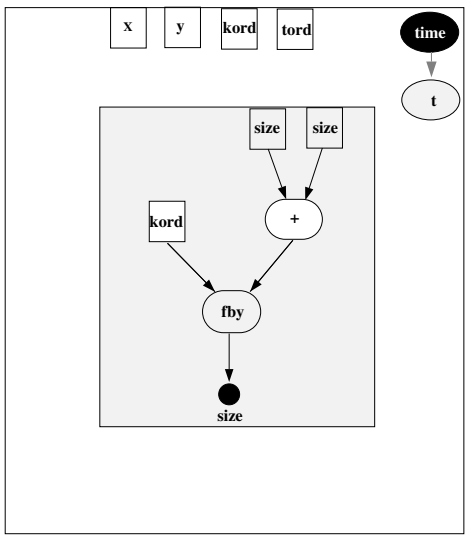


Fig. 3. Dependency-Graph Representation of Matrix Multiplication

plication:

```
matmult( x, y, kord, tord ) = p asa.t ( size >= tord )
  where
    dimension t,i,j,k;
    size = kord fby.t size+size;
    p = mult(a @.j #.k,b @.i #.k)
        fby.t (((c @.k (2 * #.k)) @.j (2 * #.j)) @.i (2 * #.i))
        where
          c = combine( add(p,next.k p),
                       add(next.j p,next.k next.j p),
                       add(next.i p, next.k next.i p),
                       add(next.i next.j p, next.k next.i next.j p)
                     );
          a = sel(x,#.i,#.j,kord);
          b = sel(y,#.i,#.j,kord);
        end;

  end;
```

The function takes four arguments where `x` and `y` are the input matrices, `kord` is the smallest sub-matrix order, and `tord` is the order of the (equal-ordered) input matrices. The function introduces a local dimension `t` in which the matrix product is computed. The result of the function is the value of `p` when `size` is no less than `tord` at some point in the `t` dimension. The definition of `size` is over dimension `t`, starting as `kord` and doubling at each successive point in the `t`-dimension. The definition of `p` uses the three local dimensions `i`, `j`, and `k` over which the cross-sub-matrix-product of sub-matrices of `A` and `B` are defined. The product matrix is assembled from these sub-matrices in dimension `t` using the definition of `c`.

Figure 3 shows the graphical sheets associated with the definitions of `matmult`, `size`, and `p`. While the definitions of `matmult` and `size` are visually simple, the definition of `p` is more complex, particularly with respect to color representation of dimensionality information. Vertices are filled with color according to the associated dimensionality — folio vertices reveal their rank by their colors whereas operation and function vertices reveal their associated dimensions (or absence thereof) by their colors. Also, the nesting of dimensions is revealed on the right top corner of each graphical sheet. The visual repetitiveness of certain subgraphs helps reveal the underlying computational structure. For example, the vertical stack of `@` and the associated vertices on the right correspond to the right-hand-argument of the `fby` operator in the textual program.

3. Nested Spreadsheet Model

We now consider an extensional approach for graphically representing GLU pro-

```
#include <mergesort.h>

extern int display( SEQUENCE );

int display( SEQUENCE s )
{ int i;
  for( i=0; i<s->length; i++ )
      fprintf( stdout, "%s\n", s->str[i] );
  return( s->length );
}
```

```
#include <mergesort.h>

extern SEQUENCE seq( int, int );

SEQUENCE seq( int which, int size )
{ SEQUENCE s;
  int i;

  s = (SEQUENCE) malloc( sizeof( *SEQUENCE ) );
  s->length = size;
  for( i=0; i<s->length; i++ ) s->str[i] = malloc( MAXLEN );

  for( i=0; i<s->length; i++ )
      fscanf( stdin, " %s", &s->str[i] );
  return( s );
}
```

```
#include <mergesort.h>

extern SEQUENCE sort( SEQUENCE );

SEQUENCE sort( SEQUENCE s )
{
  qsort( &s->str[0], s->length, MAXLEN, strcmp )
  return( s );
}
```

```
#include <mergesort.h>

extern SEQUENCE merge( SEQUENCE, SEQUENCE );

SEQUENCE merge( SEQUENCE u, SEQUENCE v )
{ SEQUENCE s;
  int i, ui, vi;

  s = (SEQUENCE) malloc( sizeof( *SEQUENCE ) );
  s->length = u->length + v->length;
  for( i=0; i<s->length; i++ ) s->str[i] = malloc( MAXLEN );

  ui =0; vi = 0; i=0;
  while( (ui<u->length) && vi<v->length) ){
  if( strcmp( u->str[i], v->str[vi] ) <= 0 )
    { strcpy( s->str[i], u->str[ui]; ui++ }
  else
    { strcpy( s->str[i], v->str[vi]; vi++ );
   i++;
  }
  if( ui < u->length )
    for( ;ui<u->length;ui++,i++ ) strcpy( s->str[i], u->str[ui] );
  if( vi < v->length )
    for( ;vi<v->length;vi++,i++ ) strcpy( s->str[i], v->str[vi]);
  return( s );
}
```

Fig. 2. Textual Sheets for Mergesort

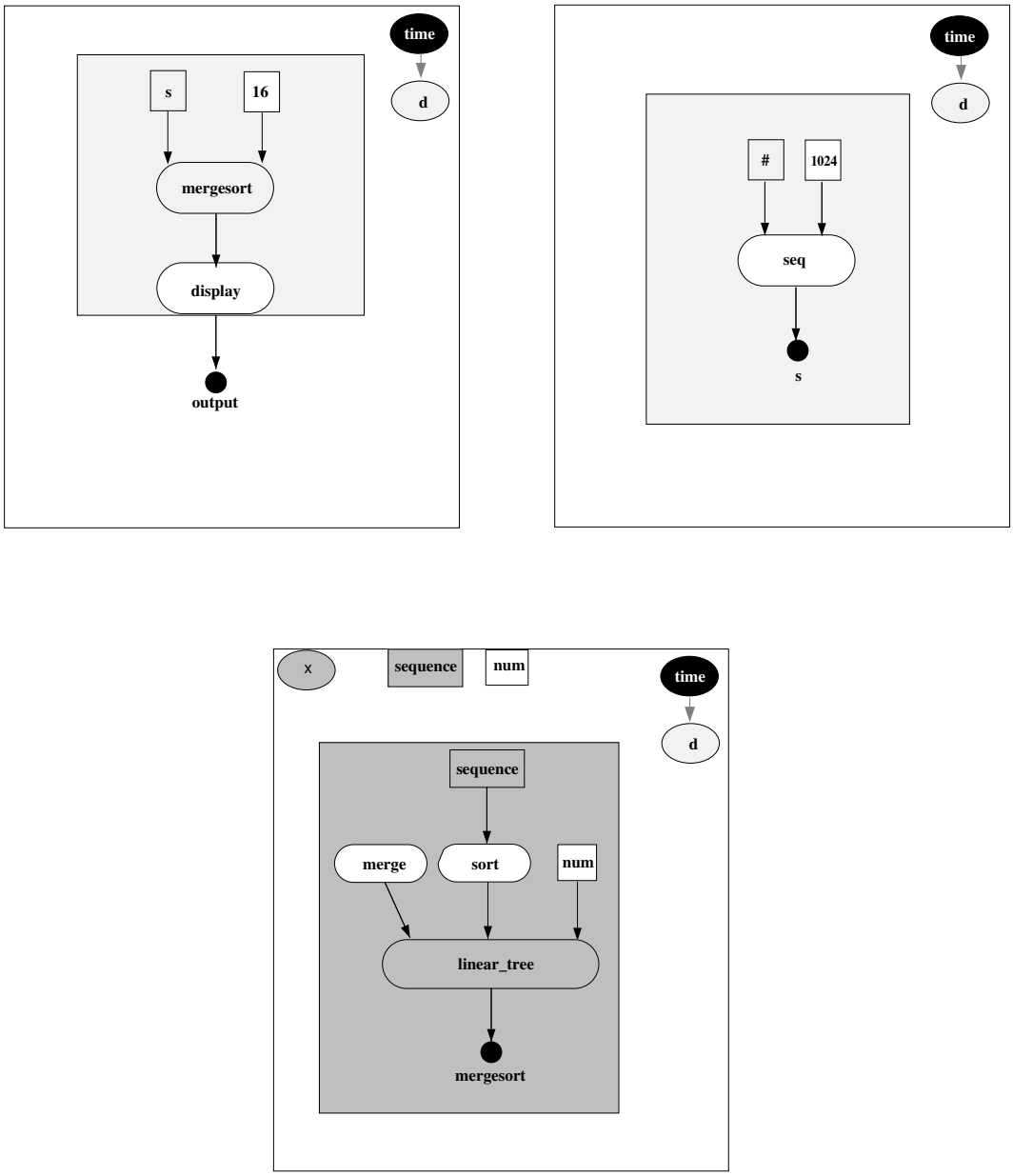


Fig. 1. Dependency-Graph Representation of Mergesort

the folio's textual definition in C. (It does not contain any dimension parameters.)

2.1. Mergesort Program

The following example shows a dependency-graph description of a GLU program that performs parallel mergesort on a sequence of strings. The program consists of one main sheet, three subsidiary graphical sheets and four subsidiary textual sheets.

The graphical sheets are shown in Figure 1 and the textual sheets are shown in Figure 2. The main graphical sheet (the one that defines `output`) consists of function `mergesort` being applied to a folio vertex labeled `s` and a constant vertex `16`. The function vertex is dimensionally qualified by `d` (as denoted by the associated color), with `d` being introduced in the context of this definition. The result of `mergesort` is the only argument of the vertex labeled `display`, whose result is the desired output. (Function `display` is defined in a textual sheet.)

The graphical sheet associated with folio `s` consists of a simple definition: C function vertex `seq` being invoked with two arguments, one of which is the current position in the `d` dimension and the other the length of the list of strings. (Function `seq` is defined in a textual sheet.)

The graphical sheet for function `mergesort` has two data parameters (`s` and `n`) and one dimensional parameter (`x`). Its definition is the application of `linear_tree`, a predefined function, with three arguments, `merge` (which is a function name), `sort` applied to parameter folios `s` and `n`. The dimensional argument to `linear_tree` is the dimensional parameter to `mergesort` itself. The result of `linear_tree` is also the result of `mergesort`. (Functions `sort` and `merge` are defined in textual sheets.)

From the example above, we can observe the following points about the dependency-graph model. Each graphical sheet can have only one definition. The definition itself is a tree which means there can be no cycles. Additionally, folio vertices abstract other definitions and interior vertices abstract functions. By using colors, dimensional information is represented in a manner that is orthogonal to dependency information. Operator vertices denoting intensional operators are filled appropriately to denote the dimensions associated with them as with `#` in definition of `s` and `linear_tree` in the definition of `mergesort`. Folio vertices are filled with the color associated with ranks of the associated variable as in the definition of `s` in the main program and the definition of formal parameter, `sequence`, in the definition of `mergesort`. Each textual sheet defines exactly one C function and its associated prototype definition. Thus C functions and Lucid user-defined functions can be viewed in uniform manner from graphical sheets.

2.2. Matrix Multiplication Program

Consider the following GLU function `matmult` to perform parallel matrix multi-

express nested multidimensional computations expressible using Lucid circa 1995.

2. Structured Data-Dependency Graph Model

A GLU program under the intensional data-dependency graph model consists of a set of "sheets," one of which is referred to as the "main sheet" and the rest of which are referred to as "subsidiary sheets." Each sheet has a unique name associated with it which we refer to as its "folio." (The folio of the main sheet is by convention referred to as `output`.)

There are two kinds of sheets: "graphical sheets" and "textual sheets." A graphical sheet holds a graphical Lucid definition while a textual sheet holds a textual C definition. (The main sheet by convention is a graphical sheet.)

The main (graphical) sheet consists of zero or more dimensions (with associated colors) in addition to the default dimension for `time`, exactly one output vertex (named `output`) and its definition tree.

A subsidiary graphical sheet consists of zero or more named function or data parameter vertices, zero or more dimension parameters (with associated colors), exactly one named data output vertex with the name being known as the sheet's folio, and the folio's definition tree.

The definition tree (of a graphical sheet) consists of source and sink vertices as well as interior vertices, and edges connecting them. Its source vertices (with no incoming edges and one outgoing edge) are either folio vertices or constant vertices. Its sink vertex (with one incoming edge and no outgoing edge) corresponds to the sheet's output vertex. Each interior vertex has several incoming edges but exactly one outgoing edge. It may be an operation vertex (that denotes a predefined Lucid operation), a Lucid function vertex (that denotes a user-defined Lucid function), or a C function vertex (that denotes a user-defined C function). The edges incident with an incoming vertex are outgoing from a folio vertex, a constant vertex, or another interior vertex. The edge outgoing from an interior vertex goes either to another interior vertex or to an output vertex.

Interior vertices have zero or more dimensionality. Vertex dimensionality is denoted by filling the vertex with the colors associated with the appropriate dimensions. An interior vertex with zero dimensionality is colorless, an interior vertex with dimensionality of one is filled with the associated color, and an interior vertex with dimensionality of more than one is filled with the appropriate colors. The use of colors (or gray-scale patterns) to denote dimensions is intended to complement the use of vertex and edges to denote dependencies.

A subsidiary textual sheet consists of zero or more named data parameter vertices, exactly one named data output vertex (the name being known as the sheet's folio) and

Intensional and Extensional Graphical Models for GLU Programming*

R. Jagannathan
SRI International
Menlo Park, California 94025, U.S.A.
E-mail: jagan@csl.sri.com
WWW: http://www.csl.sri.com/jagan

ABSTRACT

GLU is a text-based hybrid intensional language for programming conventional parallel computers whose cryptic, linear syntax often obscures static data and dimensional dependencies, thus making program functionality less apparent. We describe two different graphical models for GLU programming — an intensional model based on dependency graphs and an extensional model based on spreadsheets — designed to better present data and dimensional dependencies in GLU programs. We compare the two models in their respective approaches for revealing latent data and dimensional dependencies in GLU programs.

1. Introduction

GLU is a hybrid intensional language for programming conventional parallel computers.[1] A GLU program consists of a Lucid program augmented with user-defined C functions and C types. Typically, the Lucid program is used to describe the application in terms of data dependencies between C functions where most of the processing occurs.

Novice GLU programmers while familiar with C and its textual syntax, are quite unfamiliar with Lucid. Their introduction to Lucid through its textual syntax is usually their first glimpse of a nonprocedural language and almost certainly their first exposure to intensionality. For these programmers, Lucid programs are often hard to understand, and even harder to express. One reason for this is the cryptic, linear nature of Lucid, which hides data and dimensional dependencies in Lucid programs.

One approach for easing Lucid and GLU programming is by the use of data-dependency graphs. This approach forms the basis for graphical languages such as Operator Nets[2], vLucid[3], and VIPER[4]. While these languages prescribe a graphical syntax, the syntax can often result in flat, unstructured programs that tend to be visually complex. We develop a model based on data-dependency graphs that permits the construction of structured graphical programs.

Another approach for easing Lucid and GLU programming is by representing programs as spreadsheets, as proposed by Du and Wadge.[5,6] The Du/Wadge approach assumes a flat three-dimensional space. We extend the spreadsheet-based model to

*To appear in *Proceedings of ISLIP'95 – Eighth International Symposium on Languages for Intensional Programming, M.A. Orgun and E.A. Ashcroft Editors, World-Scientific, 1995.*