

Portfolio — A Graphical System for GLU Programming

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

email: jaggan@csl.sri.com

Abstract

We describe a graphical system for developing GLU programs called *Portfolio*. We show how the rich notion of graphical abstraction in Portfolio results in programs that are easy to develop, easy to understand, and easy to manipulate. We show that there is a direct mapping between the graphical and textual GLU programs making existing textual GLU code and associated tools easily usable under Portfolio. We also describe an interactive model for sequentially debugging GLU programs under Portfolio that uses probes to visualize demand and data propagation.

1. Introduction

GLU is a hybrid intensional language for programming conventional parallel computers¹. A GLU application 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 cryptic textual syntax is usually their first glimpse of a non-procedural 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 textual nature of Lucid — one which makes notions of dependency and dimensionality opaque to the novice.

It has long been recognized that one way to ease Lucid and GLU programming is to do so graphically using data-dependency graphs. This has been the basis for graphical languages such as Operator Nets², vLucid³, and VIPER⁴. Although these languages prescribe a graphical syntax, they employ a limited notion of graphical abstraction — one that applies to user-defined functions only. As a result programs can become visually complex, and consequently opaque to the programmer. The objective of this paper is to extend the notion of graphical abstraction so that programs are visually simple, thus more expressive and easier to manipulate.

The paper is organized as follows. First, we describe a graphical system for developing GLU programs called *Portfolio*. Second, we show how Portfolio can be used to graphically express GLU programs that are easy to understand and easy to manipulate. Third, we show how textual and graphical GLU programs can be derived from each other and the benefits therein. Last, we outline how Portfolio supports a dataflow-oriented model for debugging GLU programs.

2. PortFolio

A GLU program in Portfolio 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 is either an operation vertex (that denotes a predefined Lucid operation) or 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 is 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 grayscales in the monochrome world) 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 the folio's textual definition in C. (It does not contain any dimension parameters).

3. An Example

The following example shows a graphical description in Portfolio of a GLU program to perform parallel mergesort on a sequence of strings. The Portfolio 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 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 pattern), 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 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 this example, we can see that graphical programming in Portfolio is both simple and convenient. 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 or patterns, dimensional information is represented in a manner that is orthogonal to dependency information. Each textual sheet defines exactly one C function and its associated prototype definition. Thus

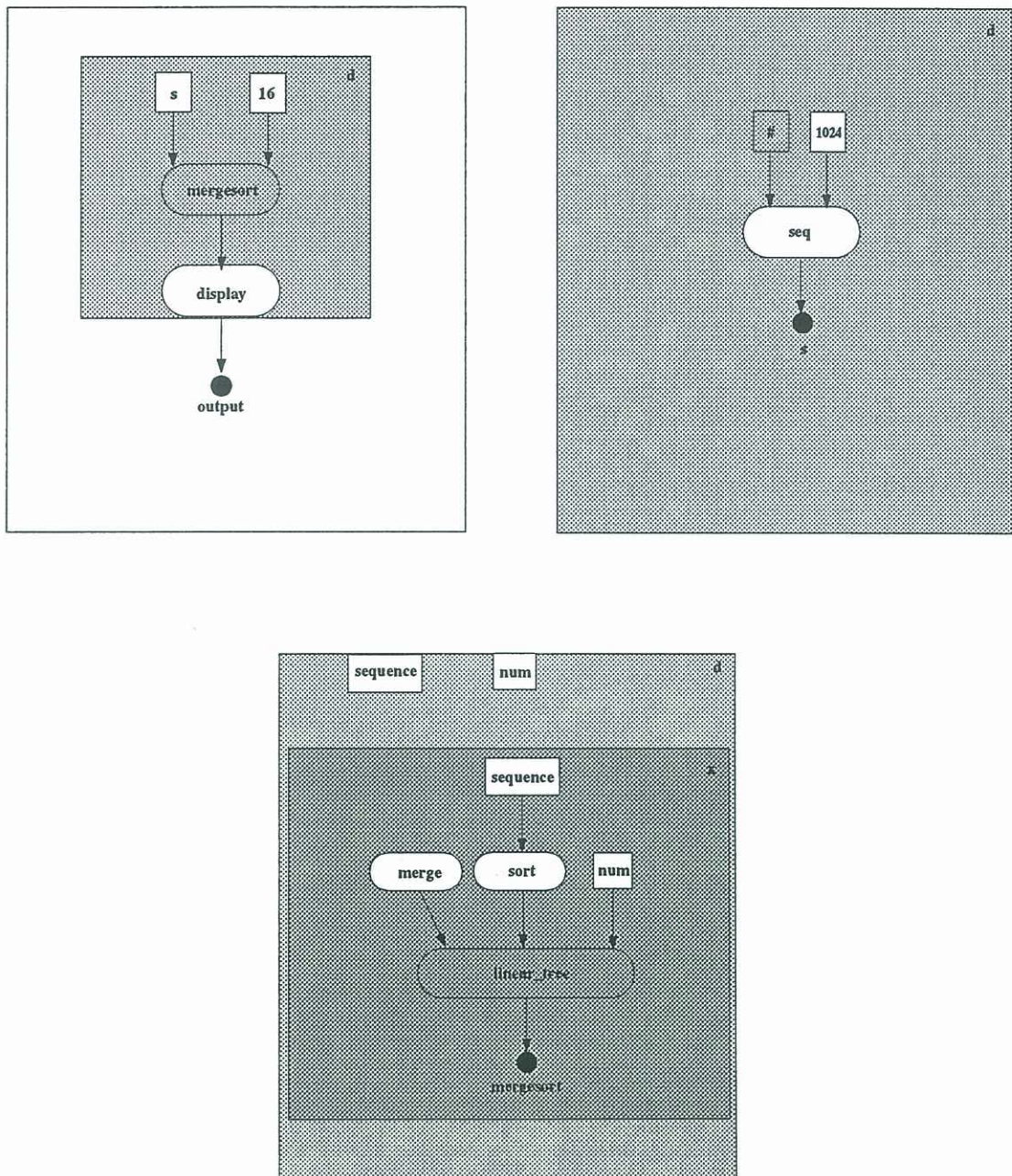


Fig. 1. Graphical Sheets for Mergesort Program

```
#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 Program

C functions and Lucid user-defined functions can be viewed in an uniform manner from graphical sheets.

4. Textual-Graphical Correspondence

There is a one-to-one correspondence between textual GLU programs and their graphical equivalents under Portfolio. That is, given a textual GLU program it is possible to construct its equivalent in Portfolio and given a Portfolio program it is possible to construct its textual equivalent.

For example, the mergesort program described earlier can be converted to the following textual form.

```
display( mergesort.d( s, 16 ) ) where
  dimension d;
  s = seq( #.d, 1024 );
  mergesort.x( sequence, num ) =
    linear_tree.x( merge, sort( sequence ), num );
end
```

The expression

```
display( mergesort.d( s, 16 ) where dimension d;
```

is represented in the main graphical sheet.

The definition

```
s = seq( #.d, 1024 );
```

is represented in the graphical sheet for s.

And the definition for mergesort is in yet another graphical sheet.

The direct correspondence between graphical and textual programs is useful in many ways. Existing textual applications and textual library functions in GLU can be viewed and manipulated graphically under Portfolio. Also, interfaces to existing tools for textual programs (such as compilers, analysis, and runtime tools) can be easily converted to work with graphical program equivalents. And Portfolio programs can be represented internally as their textual equivalents. For example, the above program by simple substitution can be converted to a single expression —

```
display( linear_tree.d( merge, sort( seq( #.d, 1024 ) ), 16 ) ) where
  dimension d;
end
```

The Portfolio equivalent of this program is shown in Figure 3.

5. Interactive Debugging in Portfolio

One of the benefits of parallel programming in GLU is that debugging is much simpler than with explicit approaches. Basically, a GLU program is debugged for its functionality entirely in a sequential mode. It is guaranteed to exhibit the same functionality when run in parallel provided C functions of the GLU program are indeed functions, i.e., their results only depend on their parameters. Thus, there is no need to be concerned with race conditions and synchronization points — a common concern while debugging explicitly-parallel programs.

In debugging a GLU program in sequential mode, the programmer is interested in determining if the program has correctly realized the underlying algorithm. With the Lucid part of the GLU program, the focus is on whether the data dependencies are correct while with the C part, the focus is on whether the functions behave as expected. Since the Lucid part is evaluated eductively, it is natural and sufficient to only visualize demand and data flow when debugging. Since the C functions are evaluated as imperative functions, it is necessary to be able to understand their behavior in terms of control flow.

A GLU programmer typically debugs the Lucid part by generating runtime traces of *all* demands being propagated and *all* values being produced to verify that the Lucid part is correct. This is typically used in a “batch” mode. The C functions themselves are debugged interactively by running the GLU program in sequential mode under a conventional debugger such as gdb or in batch mode by using printf calls at appropriate places in the C code. For the most part, GLU debugging is non-interactive and not exclusively at source-level (with the Lucid part).

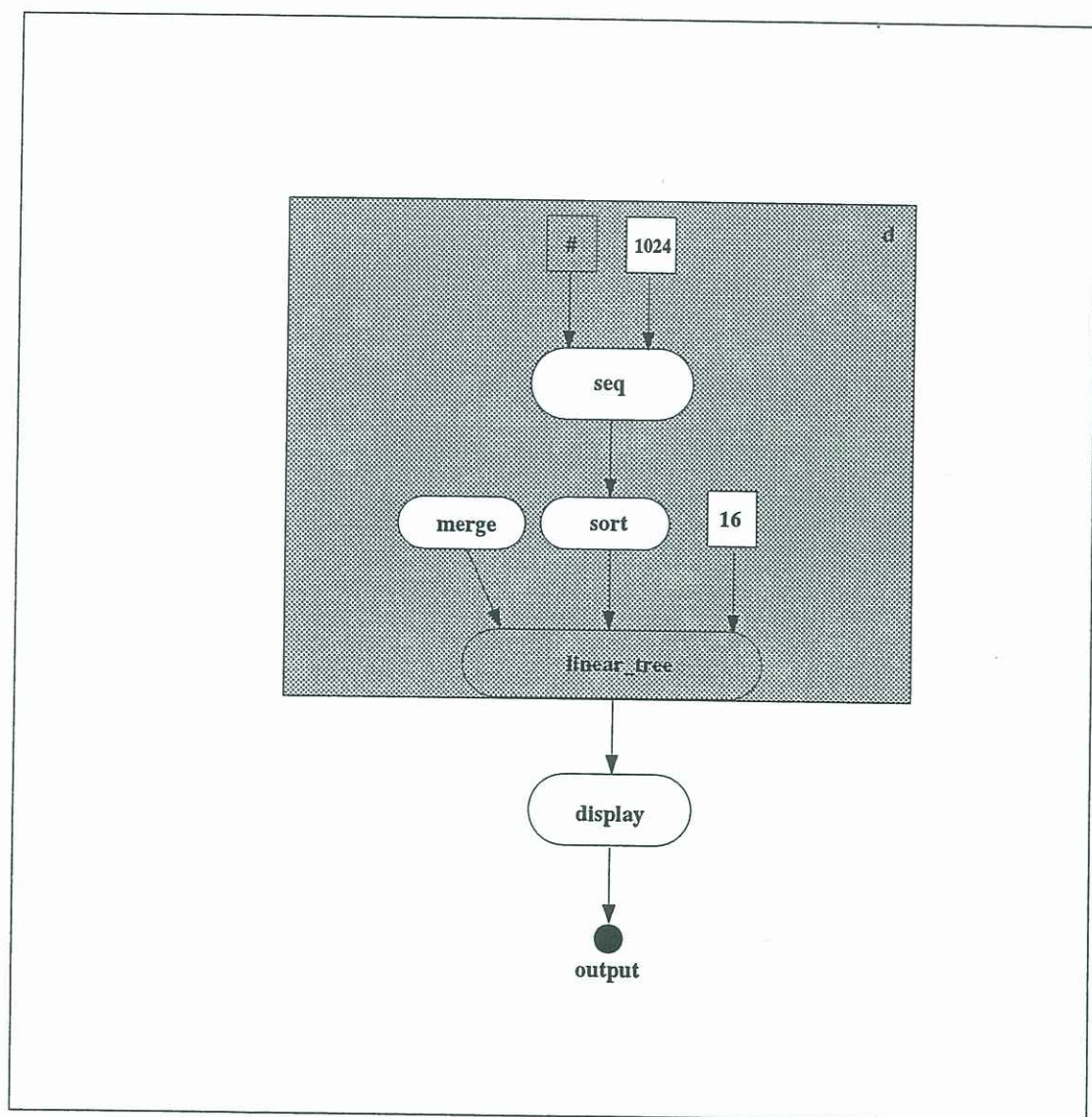


Fig. 3. Graphical Sheets for the Transformed Mergesort Program

Portfolio facilitates a simple interactive model of debugging GLU programs at the “source” (graphical and textual) level. The model is based on the notion of a *probe*. There are three kinds of probes:

1. *Edge probe*, which reveals the flow of demands and values on an edge.
2. *Folio probe*, which reveals the demand and value history for a folio.
3. *Function probe*, which reveals the propagation of demands and availability of values at a function/operation vertex.

An edge probe in demand mode will show information regarding each demand such as context, source of demand, and the demand tree for the demand. An edge probe in value mode will show information regarding each value such as context, associated demand, and value tree for the value.

A folio probe in demand mode will show the demand history associated with the folio including demands that have been satisfied and demands that are outstanding. In value mode, it will show the value history associated with the folio including values that have been produced and values that are being computed with their associated usage counts.

A function probe will show the propagation of demands each time the function (or operation) is demanded, and the availability of values each time a demanded value arrives. If the function is external (i.e., C function), it will invoke a conventional debugger (like gdb) when the function is to be applied on its values. The internals of the function itself has to be debugged conventionally, by analyzing control flow.

The debugging model enables GLU programs to be debugged interactively in the same framework in which these programs are understood — in terms demands and values associated with edges (expressions) and folios (variables), and in terms of demand generation and value availability at functions and operations.

6. Conclusions

We have described a graphical system for developing GLU programs called *Portfolio*. With Portfolio, it is possible to graphically construct GLU programs in a well-structured manner using intuitive notions of data dependency and dimensionality. We have shown that there is a direct mapping between the graphical and textual GLU programs which makes existing textual GLU code and associated tools easily usable under Portfolio. We have also described an interactive model for sequentially debugging GLU programs under Portfolio that uses probes to visualize demand and data propagation.

Our plan is to implement Portfolio and evaluate its benefits by studying the effectiveness with which novice GLU programmers can develop and refine GLU applications.

References

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*, ed. J. V. Woods, 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 ISLIP 94*, pages 113–123, SRI International, Menlo Park, California, 1994.