A Graph Description Language for Coarse-Grained Dataflow Computations

William T. Zaumen
SRI International

March 3, 1988

We are investigating an approach to coarse-grained parallelism that extends an imperative language so that it can specify dataflow graphs and implement operators directly in the base language. This work is motivated by functional programming languages—languages that represent computations by systems of equations. These systems of equations correspond to directed graphs in which each node in the graph has an output that represents the value returned by an operator.\(^1\) The output may, however, be an input to any number of nodes. In stream-oriented systems such as LUCID [1,2], a sequence or stream of values is produced, and primitives exist that manipulate these sequences.

One advantage of using equational languages is that programs written in these languages are relatively easy to analyze compared with conventional imperative languages. This ease of analysis simplifies proofs of correctness, and may simplify programming and debugging for large systems for which formal proofs are either intractable or not desired. The disadvantage of using equational languages is that it may be difficult for a programmer to deal with low-level operations, especially those involving input/output or interrupts, because there is a large "semantic gap" between the behavior of the underlying hardware and the model of computation.

Equational languages are of interest for parallel computation. To exploit these languages fully, one would prefer to use special-purpose hardware [3,4,5]. LUCID\(^2\) is well suited for systems with hardware support for fine-grained parallel computation. However, because most existing hard-

\(^1\)Operators are memoryless functions that return a single value. In the following discussion, we propose an extension to multiple values.

\(^2\)This is also true for other languages such as VAL [6] and SISAL [7].
ware is based on von Neumann processors, coarse-grained parallelism has a pragmatic advantage—complex operators can be written in an imperative language for efficiency, and an equational language can then use these operators. This approach represents an engineering compromise in which one trades-off ease of programming for efficiency.

During the development of a language that supports coarse-grained parallelism, we have discovered that there are advantages to extending equational programming computational models so that nodes can have multiple outputs and can have memory of their previous state. The approach is based on a graph methodology [8]—a computation is viewed as a directed graph with tagged data flowing between nodes that represent operations. Unlike LUCID, the operations are coded directly in an imperative language.

We have chosen C as a base imperative language for pragmatic reasons. The underlying ideas, however, are not specific to C. Unusual features (from a equational programming language viewpoint) include:

- **Typed Arcs.** Arcs are assumed to be typed. An arc may, however, be of type `void`, in which case the arc will match any type. The type of an arc may also be an array of unspecified size.

- **Multiple Output Arcs.** A node may have multiple output arcs, each producing a different value and each of a different type. Because of this, we define *generic nodes* to be essentially operators that return one or more values.

- **Instance Variables.** Instance variables are variables that retain their values from one instance of a node to the next. Each node has its own copy of the instance variables, even if the operations implemented by several nodes are the same. In the subsequent discussion, we will interpret instance variables as special arcs.

- **External Arcs and Instance Variables.** Arcs and instance variables may be declared to be external, so that a generic node function can pass data to other functions without the need to pass it as one of an arguments in a function call. Such arcs are external from the procedural language viewpoint, but not from the dataflow graph’s viewpoint.

All of the above features have a semantic interpretation that is consistent with equational languages. Multiple output arcs are interpreted semantically as a single output arc (producing a data structure), which branches to
several other nodes. Selector nodes then "read" the appropriate part of the structure.

For example, a node implementing integer division might produce two output arcs—one for the dividend and another for the remainder. This can be interpreted as an operator that produces a record containing two integer fields with two selectors—one that reads the first field and another that reads the second

![Diagram of integer division operation]

Similarly, instance variables can be interpreted as arcs that loop back on a node by way of a selector that returns an initial value (as input for the first invocation of the node) and subsequently a previous output of the node

![Diagram of instance variable operation]

Instance variables are specified in the generic node definitions, not in the dataflow graph—the graph shown above is, however, semantically equivalent.

Because nodes that use instance variables and have multiple output arcs can be replaced by an equivalent set of nodes with single outputs and no
instance variables, the formal advantages of using equational languages is maintained. The advantage of instance variables and multiple output arcs becomes apparent when one considers implementation and programming issues:

- Although instance variables can be treated as arcs, they can also be treated specially to improve efficiency. An implementation may, for example, run a node sequentially so that instance variables do not have to be copied. Furthermore, generic nodes that use instance variables can be coded in a way that is “natural” in an imperative language, and programmer-specified output arcs are not needed for a stream that is really private to a node.

- The number and type of instance variables may change as one improves algorithms. From a software reuse standpoint, it is convenient if the dataflow graphs using a generic node do not change if the type or number of instance variables change.

- With type checking, multiple output arcs are very useful. Selector nodes, if actually used, are inefficient (an extra step is added). If nodes must parse their input to obtain particular fields (e.g., a Boolean field used for control), there is a software maintenance problem—many versions of the same generic node may have to be maintained even though they differ only in the way they parse arguments. Multiple output arcs avoid both of these difficulties.

As an example, consider a generic node that produces, as its single output arc, a sum of integers:

```c
    genode int intsum(& result)
    int result;
    {
        int i = 0, sum = 0;
        result = (sum += i);
        i++;
        return;
    }
```

The & indicates that result is an output arc. An alternate implementation of the same generic node is
```c

genode int intsum(& result)
int result;
{
    int i = tag();
    result = i*(i+1) / 2;
    return;
}
```

where `tag()` is assumed to be a primitive returning the tag number for the node. Tag numbers start at zero and are incremented with each instance of a node. In the first implementation of `intsum`, instance variables ensure that all instances of a node with lower tags will execute before the current instance completes. In the second, instances can execute independently because we have made use of a closed-form expression for sums of integers.

The current status of the system is as follows:

- A runtime system using the eduction computational model [10] is being designed [8] but has not yet been implemented.
- A prototype compiler has been partially implemented. The compiler was developed by modifying a compiler for a simulation language called Drama. The Drama compiler is implemented as a preprocessor that runs after the C preprocessor. Output from the Drama compiler, however, maintains enough information so that source-level debugging is possible with standard C debuggers (e.g., dbx or dbxtool on Unix systems). This capability carries over to the dataflow language compiler.
- Code needed to interact with the runtime system has not yet been written, although we do not expect that this will be difficult to do.

Preliminary experience with examples developed manually suggests that multiple output arcs and instance variables are useful, although more work must be done to show that this is indeed the case.

The author would like to thank R. Jagannathan, Alan Downing, Joan Wrabetz, and Lou Schreier for useful discussions.

References


