R·I·T

The GTO Language Manual

Lawrence A. Coon

January, 1994
RIT-TR-94-016

Department of Computer Science
School of Computer Science and Information Technology
Rochester Institute of Technology
Rochester, NY 14623-0887, USA
email: lac@cs.rit.edu

The GTO Language Manual

Lawrence A. Coon

Department of Computer Science
School of Computer Science and Information Technology
Rochester Institute of Technology
Rochester, NY 14623-0887, USA
email: lac@cs.rit.edu

Abstract

This report describes the programming language GTO (*G*raph *T*o *O*ccam) and the associated compiler *gto*. GTO is designed for representing dataflow graphs and *gto* translates that representation into Occam2 for execution on a transputer.

GTO is part of an experimental system that also includes a general purpose high level dataflow language, DIVA, and its compiler, *diva*. The *gto* compiler serves as a backend for the *diva* compiler, and GTO as an intermediate language for the system.

1 Introduction

GTO is a dataflow graph language. Its compiler, *gto*, translates GTO programs into Occam. These are part of an experimental system designed to support dataflow programming on a transputer.[1] The system consists of a general purpose high level dataflow language DIVA, its compiler *diva*, GTO, and its compiler *gto*. Since *diva* translates DIVA programs into GTO, GTO can be considered an intermediate language and *gto* the system backend. However, GTO programs can be written by hand as well as created by the *diva* compiler. See Figure 1.

In this way, support is provided for experimentation with dataflow concepts at both the high and low levels.

This report deals first with the low level programming aspects: the basic dataflow node set, the GTO instruction set, and translating dataflow graphs into GTO. There is then a brief discussion of how a GTO program is translated into Occam, with an emphasis on how deadlock and node input timing problems are handled.

2 The GTO Dataflow Node Set

Dataflow programming is a functional programming methodology. In its most basic form, a dataflow program is represented as a dataflow graph: a directed graph whose nodes are operators and whose arcs carry data from node to node. [3, 4] The arcs carry data from the source node's output port to the destination node's input port.

[1] Multitransputer use is not currently supported.

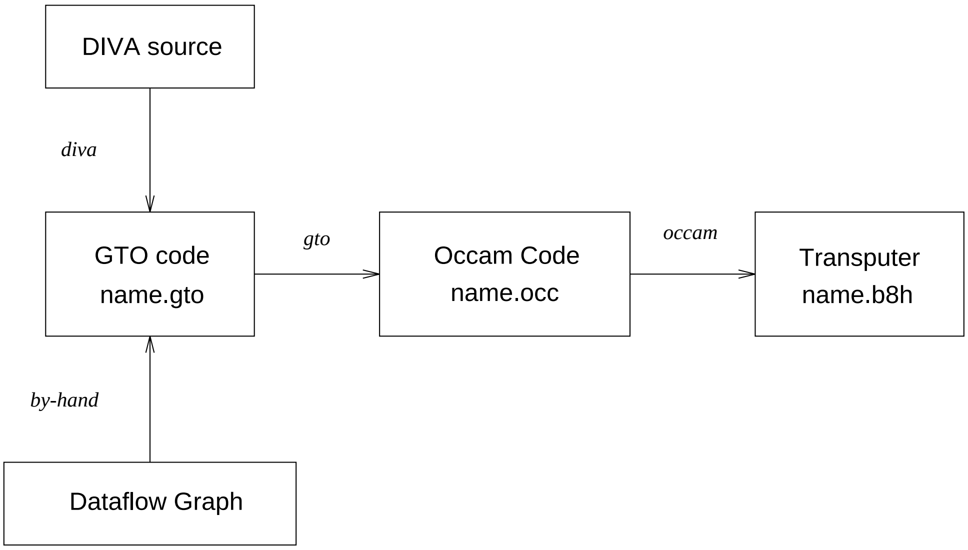


Figure 1: System Overview

In practice only a few node types are needed. These can be divided into three groups:

- Input and Output
- Arithmetic and Boolean
- Routing control

Note that these are not part of the GTO language *per se* but represent a convenient way of constructing a dataflow graph that can then be translated into GTO statements.

2.1 Values

All values are Occam REAL64, a 64 bit floating point number represented according to ANSI/IEEE standard 754-1985. [5] Boolean values are restricted to 1.0 for TRUE and 0.0 for FALSE, *other values are ignored.*

2.2 Input and Output Nodes

It is necessary to be able to insert values into the graph. This can be done either at runtime or compiletime. Runtime specification is done by Input nodes, and compiletime specification by Constant nodes. Both produce a constant value, the difference lies solely in when that value is specified.

The Constant and Input nodes have no input port and one output port. The Constant node has a value associated with it and places one copy of that value on the output port. The Input node has a string associated with it that is used to identify the value being asked for.

The value is read from standard input and one copy is placed on the output port. All input is done before the graph is executed.

Cyclic subgraphs require a new copy of a value for each iteration. Consequently two variations are provided, Mconstant and Minput (M for multiple), that send a new copy of the value each time the previous copy is consumed. These are distinguished graphically from the corresponding single value nodes by the use of a double circle.

Note that input is simply a variation on the constant. In particular, the Minput node only reads in one value, however it sends out as many copies of that value as are needed.[2]

There are two types of output node: Output and Message. These each have one input port and no output ports.

The Output node prints the value on the input port to standard output. Optionally a string may be associated with the node that will be printed along with the value. This is useful for identifying output values, particularly considering the asynchronous nature of execution.

The Message node prints the string associated with it when any input value arrives. The value is not printed, it just serves to trigger the printing of the message.

These are illustrated in Figure 2.

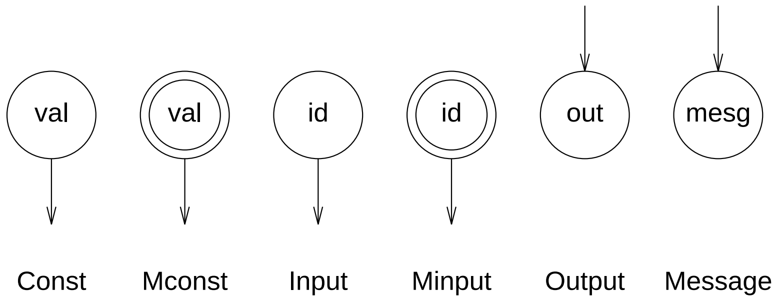


Figure 2: Input, Output and Constant Nodes

2.3 Arithmetic and Boolean Operator Nodes

These nodes are either unary: with one input, or binary: with two inputs. In all cases there is only one output. Should a value need to be sent to more than one node, a Copy node is available that replicates its input value on two output ports. For ease of recognition, the relational nodes are distinguished from the arithmetic by using a diamond shape, and the Copy node is shaded.

These are illustrated in Figure 3.

The arithmetic operations supported are:

[2]It is important not to confuse input values with variables. A basic tenet of dataflow programming is that there are no variables in the traditional imperative programming sense: a reference to a stored value globally accessible within some scope. What can appear to be a variable is really an identifier used as a convenient reference to a value on an arc, not to a place in memory.

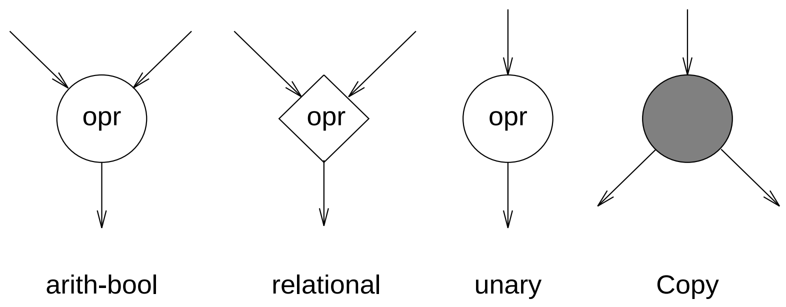


Figure 3: Arithmetic and Boolean Nodes

$+$	addition
$-$	subtraction
$*$	multiplication
$/$	division
$\%$	remainder[3]
$-$	negation

The semantics of these operators is the same as that for the corresponding Occam operator acting on REAL64 data.

The Boolean operators supported are:

$=$	equal
$\neq$	not equal
$<$	less than
$\leq$	less than or equal to
$>$	greater than
$\geq$	greater than or equal to
And	logical And
Or	logical inclusive Or
Not	logical negation

2.4 Routing Control Nodes

Conditional computation involves selective routing of values around the graph.

The basic routing/conditional nodes are Merge and Switch. Two additional nodes, the T-gate (True gate) and F-gate (False gate), are provided for convenience. Each of these nodes has a specialized input port, the control port, that accepts a Boolean value that controls the node's functioning.

In order to construct cyclic dataflow graphs it is necessary to have a node that acts as a

[3]The concept of a remainder function acting on floating point numbers may be surprising to some. The formal definition of $x\%y$ is $x-(y*n)$ where n is x/y rounded to the nearest integer.

join, i.e. that has two input ports and and one output port and passes values from the input ports on as soon as they arrive. This is provided by the Xmerge.

These nodes are illustrated in Figure 4.

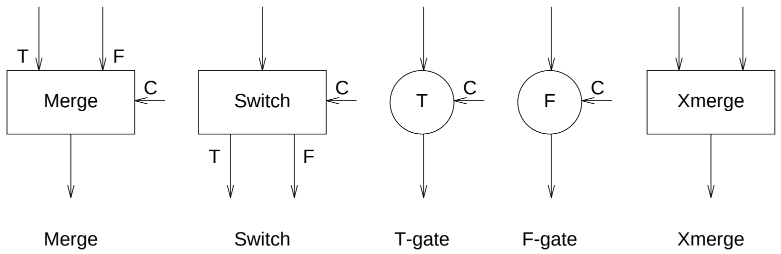


Figure 4: Arithmetic and Boolean Nodes

The Merge node has three input ports, one control and two data, and one output port. The two data ports are labeled T and F (for True and False), and the control port is labeled C. Once values are received on all three input ports, the value received on the control port is used to determine which data port value is passed along to the output port. If the control port value is True, the T data port value is passed on; if False, the F data port value is passed on. The value on the other input port is consumed so it is not there for subsequent uses of the node. Note: values do not queue up on an input port awaiting the appropriate control value to dequeue them. Each execution of an operation requires new values on all input ports.

The Switch node has two input ports, one control and one data, and two output ports. The two output ports are labeled T and F (for True and False), and the control port is labeled C. Once values are received on all three ports, the value received on the control port is used to determine along which output port the input value is to be passed. If the control port value is True, the T output port value gets the input value; if False, the F output port gets the input value. Again, each execution of an operation requires new values on all input ports.

The T-gate and F-gates have two input ports, one control and one data, and one output port. Once the two input values are received, the data value is passed along to the output port if the control value is of the appropriate type, otherwise it is consumed.

The Xmerge gate has two data input ports and one output port. As soon as a value arrives on one of the data ports, it is passed along to the output port.

2.5 Example

To illustrate how a dataflow graph is constructed, consider the problem of printing all the integers from one value to another where the values are specified at runtime.[4] The following steps give a solution:

- Input the two values, say ‘a’ and ‘b’.

[4]For simplicity, assume the values have a zero decimal part.

- Compare a to b.
- If a < b
- Then print a, and increment a and loop back to the comparison.
- Else print the message "done" and stop.

A graph that does this is shown in Figure 5.

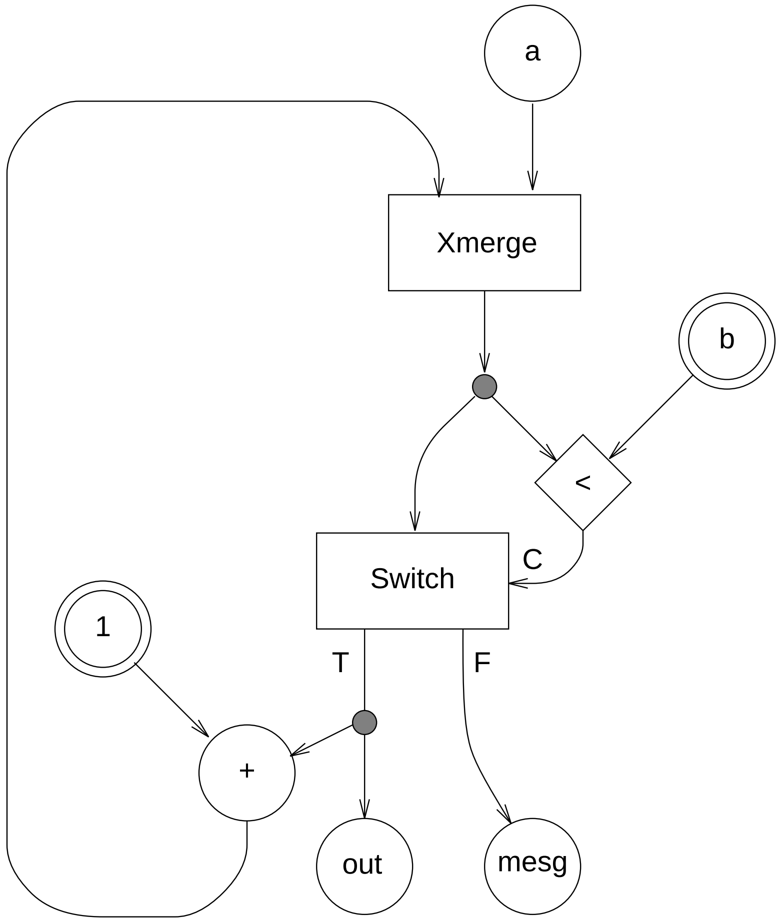


Figure 5: Printing integers from a to b

There are several points that should be noted:

- The Xmerge is needed to prime the loop.
- Since an Input node was used to get the value of a, only one copy of that value is available and the incremented value will be passed on by the Xmerge as soon as it arrives.
- It is necessary to use an Minput node to get the value of b, since each time a comparison is made the value is consumed and a new copy needs to be provided.

- Synchronization is not a problem since the node operation does not occur until all input ports have their values. For instance, in this example the successive values of the Mconstant node that produces 1 are obviously ready for use by the addition node before the value from the copy node is available and it seems that some data would be lost.[5] However, this is not the case since neither of the input values to the addition node are sent until both are available because of the use of Occam channels to implement dataflow arcs.

3 The GTO Instruction Set

A *gto* program consists of a set of instructions that represent a dataflow diagram. Each instruction represents a node in the diagram and contains information as to the destination of the value produced by the node. The program is, in effect, a linearization of the dataflow graph.

Instructions are represented as n-tuples of the following form:

instruction_number opcode [literal] {destination_tuples}*

where:

instruction_number	identifies the node
opcode	the node operation symbol
literal	floating point value for a constant node, or output tag string for an output node, or variable tag string for an input node
destination_tuples	output arc pairs

The optional string field is provided for input nodes so the user can be prompted for input and for output nodes so the output can be identified. Also, a value field is required for constant nodes. All arithmetic is assumed to be floating point.

Output arcs are specified by tuples consisting of the destination node's instruction_number and the appropriate input port number. It is not necessary to specify input arcs since each input arc is the output arc of some other node and will be specified when that node is processed.

Since input ports will be referred to by number, it is important to have a consistent numbering convention. Ports will be referenced from left to right, with any control port assumed to be at the far right. For example, the input ports of a merge node should be numbered: True, 1; False, 2; control, 3.

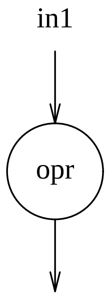
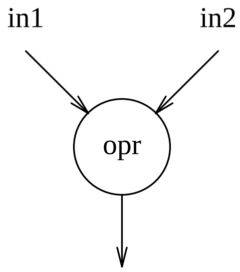
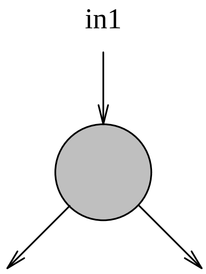
Following is a summary of the syntax and semantics of all instructions currently supported (Table 1), and graph conventions for the corresponding nodes (Table 2).

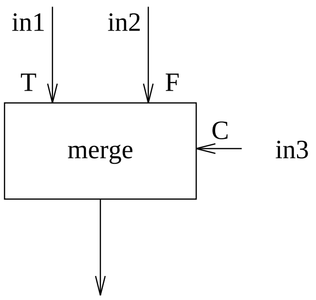
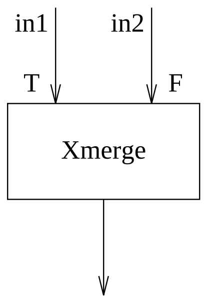
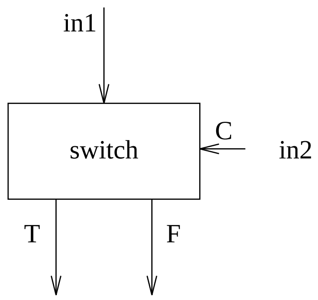
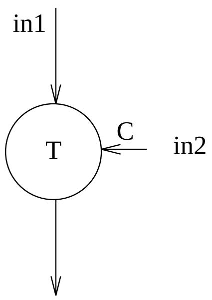
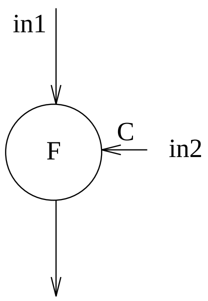
[5]The problem would be clearer if the input value varied rather than always being 1.

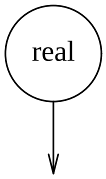
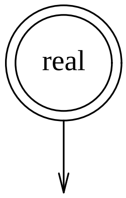
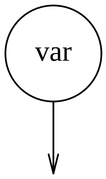
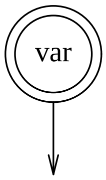
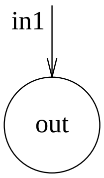
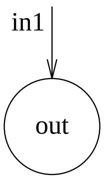
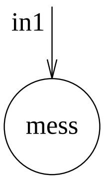
Table 1: Supported Instructions and Their Semantics

Dataflow Operator Symbol		
Graph Symbol	GTO Symbol	Functionality
$+$	Add	in1 $+$ in2 $\rightarrow$ out1
$-$	Sub	in1 $-$ in2 $\rightarrow$ out1
$*$	Mul	in1 $*$ in2 $\rightarrow$ out1
$/$	Div	in1 $/$ in2 $\rightarrow$ out1
%	Mod	in1 % in2 $\rightarrow$ out1
$-$	Neg	$-1 *$ in1 $\rightarrow$ out1
$=$	Eq	in1 $=$ in2 $\rightarrow$ out1
$\neq$	Ne	in1 $\neq$ in2 $\rightarrow$ out1
$<$	Lt	in1 $<$ in2 $\rightarrow$ out1
$\leq$	Le	in1 $\leq$ in2 $\rightarrow$ out1
$>$	Gt	in1 $>$ in2 $\rightarrow$ out1
$\geq$	Ge	in1 $\geq$ in2 $\rightarrow$ out1
And	And	in1 And in2 $\rightarrow$ out1
Or	Or	in1 Or in2 $\rightarrow$ out1
Not	Not	Not(in1) $\rightarrow$ out1
gray circle	Copy	in1 $\rightarrow$ out1 and out2
T	Tgate	if in2 is True then in1 $\rightarrow$ out1 otherwise nothing is output
F	Fgate	if in2 is False then in1 $\rightarrow$ out1 otherwise nothing is output
switch	Switch	if in2 is True then in1 $\rightarrow$ out1 otherwise in1 $\rightarrow$ out2
merge	Merge	if in3 is True then in1 $\rightarrow$ out1 otherwise in2 $\rightarrow$ out1
Xmerge	Xmerge	value on in1 or in2 $\rightarrow$ out1 whichever is ready first
circled real	Const	value $\rightarrow$ out1 (once only)
double circled real	Mconst	value $\rightarrow$ out1 (initially and whenever the previous output is consumed)
name	Input "string"	prints the string and reads a real from stdin value $\rightarrow$ out1 (once only)
name	Minput "string"	prints the string and reads a real from stdin value $\rightarrow$ out1 (initially and whenever the previous output is consumed)
out	Output	prints the value on in1 to stdout
out	Output "string"	prints the string and value on in1 to stdout
mesg	Message "string"	prints the string to stdout

Table 2: Node Conventions

Arithmetic and Boolean Expressions		
Operator Name	Graph Symbol	Example
Unary		Neg 4 2
Binary		Add 4 2
Copy		Copy 4 1 4 2

<table>
<tr><th colspan="3">Control of Routing</th></tr>
<tr><th>Operator Name</th><th>Graph Symbol</th><th>Example</th></tr>
<tr><td>Merge</td><td>
</td><td>Merge 4 1</td></tr>
<tr><td>Xmerge</td><td>
</td><td>Xmerge 4 1</td></tr>
<tr><td>Switch</td><td>
</td><td>Switch 4 1 5 1</td></tr>
<tr><td>Tgate</td><td>
</td><td>Tgate 4 1</td></tr>
<tr><td>Fgate</td><td>
</td><td>Fgate 4 1</td></tr>
</table>

Input and Output		
Operator Name	Graph Symbol	Example
Const		Const 5.0 4 1
Mconst		Mconst 5.0 4 1
Input		Input “a” 4 1
Minput		Minput 4 1
Output		Output
Output		Output “string”
Message		Message “string”

4 Programming in GTO

Programming consists of creating a dataflow graph using the GTO node set, then linearizing it using the GTO instruction set.

Linearization can be done in three steps:

1. Systematically numbering each node. Top to bottom and right to left will be used in this module.
2. Labeling each arc with the number of the destination node and the input port (its destination tuple).
3. Constructing an instruction for each node consisting of its number, opcode, a literal if needed, and the destination tuples.

When drawing graphs it will often be more convenient to change the position of the ports. In that case care should be taken to clearly identify the ports, and to use the proper numbers when referring to them. Otherwise, the program could end up calculating $b - a$ instead of $a - b$ or sending a result down the wrong arc from a merge node.

To illustrate, consider the example discussed in Section 2, printing all the integers from one value to another. A graphical solution is shown in Figure 5. Numbering the nodes and labeling the arcs gives the graph shown in Figure 6. The resulting GTO program is shown in Figure 7.

For more details and examples see "A GTO Tutorial" [1].

5 Running the GTO Compiler

The *gto* compiler takes as its input a dataflow program written in GTO and produces an equivalent Occam program that runs under Inmos's Toolset software.

The source file must be of the form "name.g". The library containing the node procedures, DF.lib, must be accessible. The compiler creates an Occam file with the same name but with the ".g" replaced by ".occ" (as this is what is expected by the Occam compiler). The resulting Occam2 program is compiled and executed in the normal fashion.

gto name.g

imakef name.b8h

make -f name

iserver -sb name.b8h

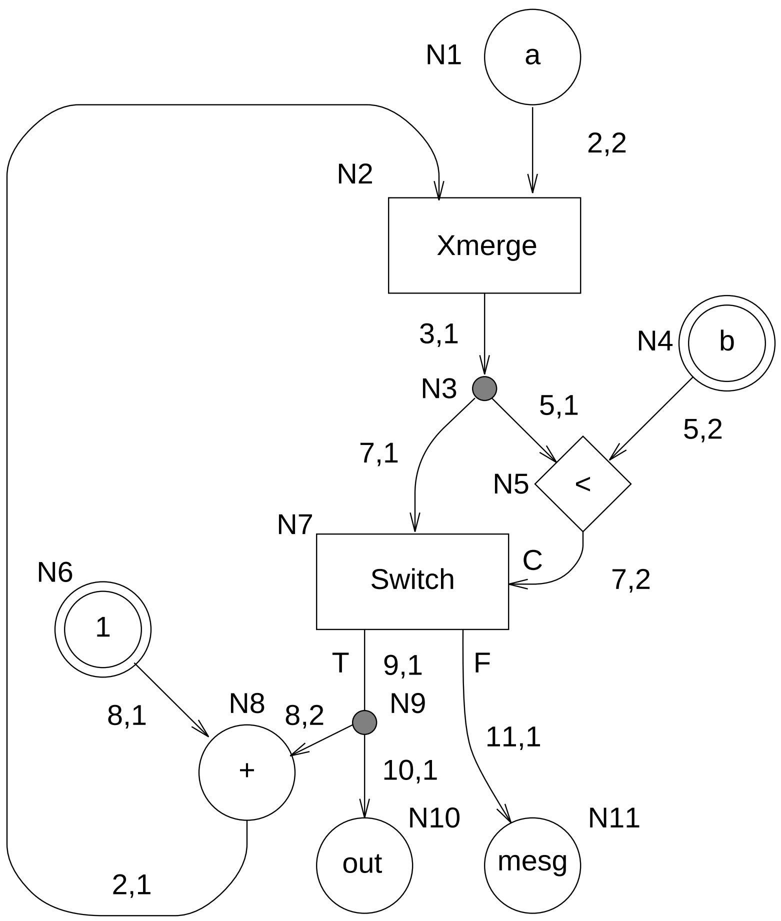


Figure 6: Labeled graph for printing Integers from a to b

In order to fully simulate the dataflow paradigm, it was necessary to create infinite loops in all but the simplest cases. Consequently, to halt execution it is usually necessary to hit the break key to exit the iserver.

6 Translating into Occam

6.1 Basic Strategy

Translation is based on the functional similarity between Occam processes with channel parameters and dataflow nodes and arcs. The basic strategy is to represent nodes as Occam processes, and arcs as channels that pass data between those processes [2]. Joining the channel definitions to a PAR containing the procedures allows for parallel execution.

There are two major problems with this simplistic approach:

— parallel access to IO channels.

— graph cycles can require more than one value from a node.

1	Input	"a"	2 2	
2	Xmerge		3 1	
3	Copy		7 1	5 1
4	Minput	"b"	5 2	
5	Lt		7 2	
6	Mconst	1.0	8 1	
7	Switch		9 1	11 1
8	Add		2 1	
9	Copy		8 2	10 1
10	Output			
11	Message	"done"		

Figure 7: GTO program for printing Integers from a to b

The solutions to these problems are sketched out below and discussed in detail in the next section.

6.1.1 I/O Problems

Occam's restriction on parallel access to channels causes problems when there is more than one Input node or Output node.

The solution for input was to treat them as runtime constants. All actual input is done "up front" (before the PAR) and the Input nodes in the PAR refer to these values. In effect, an Input node is treated as a Constant node whose value is determined at runtime, before execution of the graph begins.[6]

The solution for output was to have parallel access to the output channel handled by ending the PAR with an infinitely looping WHILE containing an ALT whose alternatives correspond to the Output nodes in the graph. When an output request occurs, the appropriate alternative is triggered and the procedure loops back and waits for the next output to arrive. Since output channel access is synchronized by the ALT, there is no problem with parallel access.

6.1.2 Graph Cycle Problems

Any node other than the Input and Constant nodes can be part of a graph cycle. Consequently, such nodes need to be able to accept a new input value and produce a new value for each cycle. This is done by making the corresponding processes loop infinitely, accepting input, processing it and then looping back to wait for more input.

[6]This means that input requests cannot be based on the results of program execution.

6.2 Translation Details

Translation basically consists of filling in the following program template.

heading
    — housekeeping
    — link to process library DF.lib
variable declarations for input values
channel declarations for arcs
input template for each input node
a PAR containing
    — a process call for each node (except Output)
    — the output loop containing an ALT for each output node

6.2.1 Process Library

A library was created containing a PROC for each of the dataflow operations. These have channel parameters that provide input and output to the process, and code that provides the required functionality. Translation consists of mapping nodes to PROC instances from the library, mapping arcs to channels, then placing the resulting code in a PAR for parallel execution.

Examples of the PROCs that implement Add, Mconst, and Switch nodes follow. Note the use of the "WHILE TRUE" to create infinite loops. This is needed in the Mconst PROC to provide for a new copy of the value each time a value is read off the output channel, and in the Add and Switch PROCs so they can function as part of a cycle.

```
PROC Add (CHAN OF REAL64 in.a, in.b, out)
  REAL64 op1, op2 :
  WHILE TRUE
    SEQ
      PAR
        in.a ? op1
        in.b ? op2
        out ! op1 + op2
:


PROC Const (VAL REAL64 value, CHAN OF REAL64 out)
    out ! value  -- output value once only
:


PROC Mconst (VAL REAL64 value, CHAN OF REAL64 out)
  -- Multiple Constant
```

```
  WHILE TRUE
    out ! value  -- output value each time previous is read
:

PROC Switch (CHAN OF REAL64 in.a, in.control, out.T, out.F)
  REAL64 op1, op2 :
  WHILE TRUE
    SEQ
      PAR
        in.a ? op1
in.control ? op2
IF
        op2 = 1.0(REAL64)  --if control is true
  out.T ! op1      --put token onto true output arc
op2 = 0.0(REAL64)  --if control is false
  out.F ! op1      --put token onto false output arc
TRUE
          SKIP
:
```

6.2.2 I/O Details

As mentioned above, input is done in two stages: first the value is read into a variable, then the value is used in a Const PROC placed inside the PAR. For example, the following is based on the code generated for the loop example discussed in Section 4. The three I/O instructions print a prompt, read a value and then place it in var1. The Const then accesses that value and sends it off on channel/arc C2.2 to instruction/node 2.

```
REAL64  var1, :
 CHAN OF REAL64 C2.2 :
 SEQ
   so.write.string(keyboard,screen,"a")
   so.read.echo.real64(keyboard,screen,var1,error)
   so.write.nl(keyboard,screen)
   PAR
     Const(var1, C2.2)
     ...
```

The output code looks like the following (also based on the loop example). The Channel C10.1 alternative is triggered by the integer to be printed (it is generated by instruction 10 and waits for the value sent by instruction/node 9). The channel C11.1 alternative carries a value (send by instruction/node 7) whose sole purpose is to trigger the “done” message of instruction 11.

```
REAL64 outvar :
    WHILE TRUE
      ALT
        C10.1 ? outvar
          SEQ
            so.write.real64(keyboard,screen,outvar,15,3)
            so.write.nl(keyboard,screen)
        C11.1 ? outvar
          so.write.string.nl(keyboard,screen,"done")
```

6.2.3 Putting It All Together

There are four parts to the program template that are filled in by the compiler when translating a program.

The declaration section for input variables and channels
The input code section
The process invocations for each node
The output process

Since each of these depends on information that can be anywhere in the program it seems that four passes are required. However, as these are physically separate, only one pass is needed during which information is written to four different files, one for each section, that are concatenated to form the final Occam program.

As an example of a complete program, the Occam code generated for the looping example follows.

```
#INCLUDE "hostio.inc"
PROC main(CHAN OF SP keyboard, screen, [] INT memory)
  #USE "convert.lib"
  #USE "hostio.lib"
  #USE "DF.lib"
  BOOL error :
  REAL64  var1, var4 :
  CHAN OF REAL64 C10.1, C11.1, C2.1, C2.2, C3.1, C5.1, C5.2,
                C7.1, C7.2, C8.1, C8.2, C9.1 :
  SEQ
    so.write.string(keyboard,screen,"a")
    so.read.echo.real64(keyboard,screen,var1,error)
    so.write.nl(keyboard,screen)
    so.write.string(keyboard,screen,"b")
    so.read.echo.real64(keyboard,screen,var4,error)
    so.write.nl(keyboard,screen)
```

```
    PAR
      Const(var1, C2.2)
      Xmerge(C2.1, C2.2, C3.1)
      Copy(C3.1, C7.1, C5.1)
      Mconst(var4, C5.2)
      Lt(C5.1, C5.2, C7.2)
      Mconst(1.000000(REAL64), C8.1)
      Switch(C7.1, C7.2, C9.1, C11.1)
      Add(C8.1, C8.2, C2.1)
      Copy(C9.1, C8.2, C10.1)
      REAL64 outvar :
      WHILE TRUE
        ALT
          C10.1 ? outvar
            SEQ
              so.write.real64(keyboard,screen,outvar,15,3)
              so.write.nl(keyboard,screen)
          C11.1 ? outvar
            so.write.string.nl(keyboard,screen,"done")
:
```

7 System Requirements and Availability

The *diva* and /gto compilers are written in C using Lex and Yacc and currently run under SunOS 4.1.1 on Sun3 workstations and under SunOS 4.1.2 on Sun Sparc workstations. No special features were used, so porting to other UNIX [7] environments should be straightforward.

The Occam code produced by the *gto* compiler runs on a configuration of Transtech MCP1000 boards running Inmos's Toolset software. Porting to other transputer systems such as TDS should also be straightforward but tedious.

The system is available by anonymous ftp at ftp.cs.rit.edu/pub/lac in two formats: gto.tar.gz or gto.Z. It is also available on the World Wide Web at http://cs.rit.edu/ lac/gto.html.

This includes the following:

- source code for the *gto* compiler
- GTO test suite
- "The GTO Language Manual"
- "A GTO Tutorial" [1]

[7] UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

8 Acknowledgements

Current work on the DIVA/GTO system is a team effort with Edith Lawson, School of Computer Science and Information Technology, Rochester Institute of Technology, Rochester, N.Y.

Early work on the dataflow node set design and translation to Occam involved important contributions by Carol Torsone, Department of Computer Science, St. John Fisher College, Rochester, N.Y. and Fereydoun Kazemian, School of Computer Science and Information Technology, Rochester Institute of Technology, Rochester, N.Y.

References

[1] L. A. Coon, "A GTO Tutorial," Computer Science Department (RIT-TR-93-017) Rochester Institute of Technology, February 1994.

[2] L. A. Coon and E. A. Lawson, "DIVA, a Dataflow Language for Transputers," *Transputer Research and Applications 6, Proceedings of the Sixth Conference of the North American Transputer Users Group,* IOS Press, Washington, 1993.

[3] Alan L. Davis and Robert M. Keller, "Data Flow Program Graphs," *Computer*, February 1982, pp. 26-41.

[4] Chris Hankin, David Till, and High Glaser, "Applicative Languages and Data Flow," *Functional Programming: Languages, Tools and Architectures*, Halstead Press: John Wiley & Sons, 1987.

[5] INMOS Ltd., "Occam2 Reference Manual", Prentice-Hall, 1987.

[6] E. A. Lawson, L. A. Coon, and C. M. Torsone, "Data Flow Graphs and Their Linearization," Computer Science Department (RIT-TR-92-004), Rochester Institute of Technology, March 1992.

[7] Howard E. Motteler, "Occam and Dataflow," *Proceedings of the Second Conference of The North American Transputer Users Group*, October 18-19, 1989.

[8] Lan Yang and Lan Jin, "A Dataflow Implementation of Functional Language FP on Transputers," *Transputer Research and Applications 4, Proceedings of the Fourth Conference of the North American Transputer Users Group*, October 11-12, 1990.