JOINT ROYAL MELBOURNE INSTITUTE OF TECHNOLOGY AND
COMMONWEALTH SCIENTIFIC AND INDUSTRIAL RESEARCH ORGANISATION
PARALLEL SYSTEMS ARCHITECTURE PROJECT

GHC on the CSIRAC II Dataflow Computer

TR 118 090 R

M.W. Rawling †

† Division of Information Technology
CSIRO
c/o Department of Communication and Electrical Engineering
Royal Melbourne Institute of Technology
124 La Trobe St
Melbourne 3000

Version 1.0   Original Document 26/9/89

ABSTRACT:

This paper describes a Guarded Horn Clauses (GHC) implementation for the RMIT/CSIRO dataflow machine. Data representations, code templates and theory of operation are all discussed.

GHC on the CSIRAC II Dataflow Computer

TR 118 090 R

M.W. Rawling[1]

Introduction

GHC is a logic programming language developed by Dr. Kazunori Ueda at the University of Tokyo, Graduate School. It benefits from a critical examination of existing logic programming languages, particularly with regards to complexity and parallelism. The result is an elegant and efficient language that is ideally suited to parallel execution in a dataflow environment. GHC augments the dataflow system by providing a natural framework for demand driven computation, buffered communication, mutable data and nondeterminism [1].

This paper discusses some of the important aspects of GHC and its implementation on the CSIRAC II dataflow machine being developed at the Royal Melbourne Institute of Technology (RMIT) in a joint research venture with the Commonwealth Scientific and Industrial Research Organisation of Australia (CSIRO) [2,3]. Data representations, code templates and theory of operation are given to show how naturally GHC maps into the dataflow model.

Guarded Horn Clauses (GHC)

From [3]:

Syntax

A GHC program is a set of guarded Horn clauses of the following form:

$$H :\text{-}\ G_1, \ldots, G_m \mid B_1, \ldots, B_n \qquad (m > 0,\ n > 0).$$

H is called a clause head, G_i's are called guard goals, and B_i's are called body goals. The '|' operator is called the commitment operator and that part of a clause before '|' is called a guard, and the part after '|' is called a body. The guard includes the clause head. A GHC program also includes a goal clause of the form:

$$:\text{-}\ B_1, \ldots, B_n \qquad (n > 0).$$

Semantics

GHC uses a restricted form of parallel input resolution to construct a *non-ordered refutation* of a given goal. A proof tree is constructed in executing a GHC program but instead of using multiple environments (to separate substitutions from different parallel derivation paths) or a complex distributed backtracking mechanism, GHC ensures the determinacy of interprocess communication data by arranging for all variable bindings to be unique and global (within their dynamic scope) throughout the execution of a program. This is achieved through the following rules:

1 Joint RMIT/CSIRO Parallel Systems Architecture Project,
c/o Department of Communication and Electrical Engineering,
Royal Melbourne Institute of Technology,
124 La Trobe St. Melbourne 3000, Australia.

Rules of Suspension

(a) The guard of a clause cannot export bindings to the caller of that clause, and
(b) the body of a clause cannot export bindings to the guard of that clause before that clause is selected for commitment.

Rule of Commitment

To be selected as a substitution clause for a goal *G*, a clause *C* must first succeed in solving its guard (i.e., pass head unification and solve all guard goals subject to the rules of suspension) and then confirm that no other clauses in the program have been selected for *G*. If confirmed, *C* is selected indivisibly, and the execution of *G* is said to be committed to the clause *C*.

The first rule of suspension allows guards to be written in a general manner even though they are restricted to being observers of external bindings. This observer status imposes a dataflow restriction upon guards which ideally suits their implementation on a dataflow machine. The second rule of suspension indicates that a clause body also has observer status until that clause is selected for commitment, after which the body may export bindings.

The rule of commitment guarantees the required uniqueness of bindings by allowing only the one selected clause to export bindings for each goal. There is no backtracking upon failure of a committed clause.

AND Parallelism

GHC supports AND parallel execution of guard and body goals (as opposed to say a left to right linear approach, although that would also be valid). In fact, head unification (parameter passing) and execution of both guard and body goals may all be done in parallel provided the rules of suspension are adhered to. In particular, it is possible to begin body execution eagerly (i.e., before commitment is achieved).

OR Parallelism

Limited OR parallelism is exploited by trying guard evaluation (including head unification) of all potential substitution clauses (i.e., those with matching head functors and arities) for a given goal in parallel. An eager GHC implementation will also include parallel execution of body goals from different substitution clauses.

Otherwise Goals

GHC allows the use of the special guard goal 'otherwise'. This goal succeeds when all other guards in a predicate definition have failed. Ueda gives a general definition of otherwise in which normal guard goals can appear along with an otherwise goal, and more than one clause in a predicate definition can include an otherwise goal (presumably, such clauses would be mutually exclusive due to head unification or other guard goals).

Flat GHC

Flat GHC is a restricted form that does not allow user defined predicates to appear as guard goals. This restriction allows a compile time analysis to determine which guard goals may lead to suspension [1,6]. This allows for a more efficient implementation since special forms of unification, that do not require the transmission of variable/clause 'threshold' information can be used.

Implementation

The CSIRAC II 'hybrid' dataflow architecture combines the features of the static/queued and dynamic/tagged dataflow models. Some architectural features to bear in mind when considering the GHC implementation details are as follows:

- Data tokens are of variable length and strongly typed, type checking occurs at run time.
- Short vector and compound (recursive) token types are supported.
- Tokens can queue on input arcs to machine nodes in FIFO order.
- Tokens carry a colour/tag which is involved in both matching and routing. Routing uses token colour as well as static addressing (node/processing element numbers) to distribute work load.
- Primitive nodes have independent match and evaluation functions. Matching functions are special operations performed prior to the evaluation of nodes and include a one input bypass function, two input associative match, deferred storage, etc. [2,5]. Relevant matching functions and node functions are described in a brief glossary at the end of this paper.

Data Representation

The GHC implementation uses four main data types:

- *constants* Integers, reals and atoms represented by numeric and string type tokens.
- *structures* Represented by compound tokens that are of variable length and recursive e.g., the structure

  fred(1,john(2,abc))

  is represented by the compound token

  cm {cv "fred" i8 2 i8 1 cm { cv "john" i8 2 i8 2 cv "abc" }},

  where the token types cm, cv and i8 stand for compound, character vector and eight bit integer respectively. The second component in each compound data field is the integer arity of the corresponding GHC structure.
- *variables* Represented by environment tokens which are the dynamic addresses of variable manager graphs. Requests are made to variable managers through these addresses.
- *results* Success/failure represented by boolean tokens.

Lists are simply structures with a '.' functor and appropriate sub-structure. This leads to a strict implementation of lists that limits concurrency and requires potentially very long compound tokens to be transmitted between nodes in the dataflow graph. Future versions will overcome both of these problems by implementing lists as pointers to stored cons cells each of which has a head and tail part.

Predicate Translation

Figure 1 shows the general dataflow graph for a GHC predicate made up of several clause definitions. All clauses are treated equally with no special significance given to their ordering within a program. Clauses within a predicate definition share access to a common commit graph which arbitrates between them at run time. The result of a predicate will either come from a committed sub-clause or the commit operator itself (should all clause guards fail). These mutually exclusive result sources are all merged in the predicate graph to define the output of that predicate.

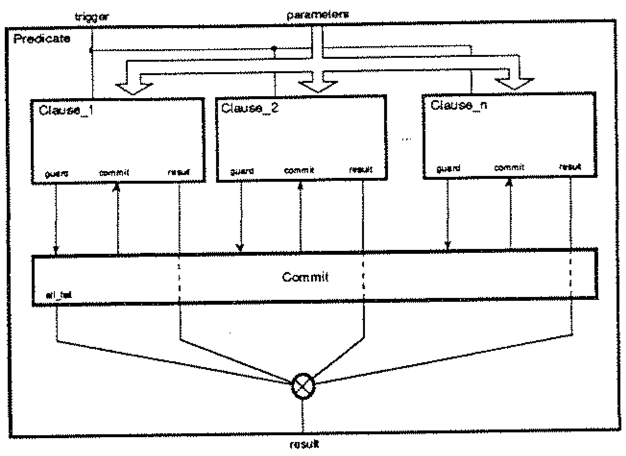

FIGURE 1 General Predicate Dataflow Graph

Clause Translation

Each clause within a predicate definition shares that predicate's trigger and actual parameters. A typical clause translation is shown in figure 2. The clause graph uses the trigger input to generate local variables (i.e., variables not appearing as formal parameters). A commit gate controls execution of the clause body by making the passage of trigger and body variables conditional upon clause commitment. This scheme represents a lazy implementation of clause bodies which overcomes any potential problems with run away recursions, etc.. Of course, as stated above, an eager implementation is still valid in principle.

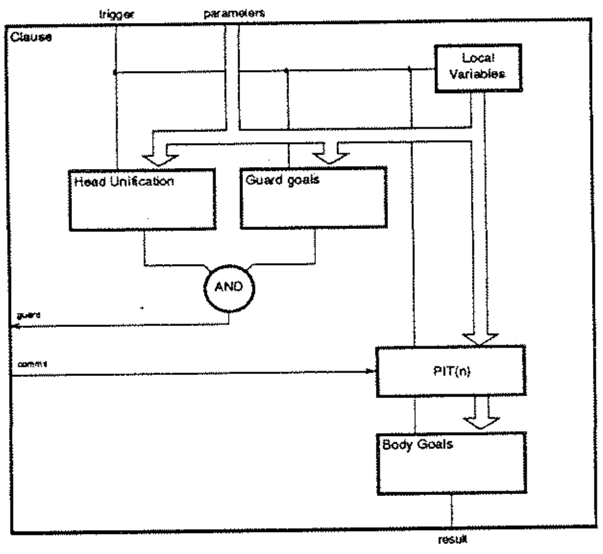

FIGURE 2 General Clause Dataflow Graph

Suspension

As required by GHC's first rule of suspension, guard evaluation suspends whenever it would otherwise cause the binding of a non-local variable. Figure 3 shows a general 'wait' graph that accepts any data type and returns a non-variable type. In the case of variables, 'wait' recurses down binding chains repeatedly calling the special variable manager function *Var_Wait* until a non-variable is found on the chain (see the section on variables for more detail). The wait graph is also used as a primitive form of 'eval' for arithmetic predicates and also by the data display routines.

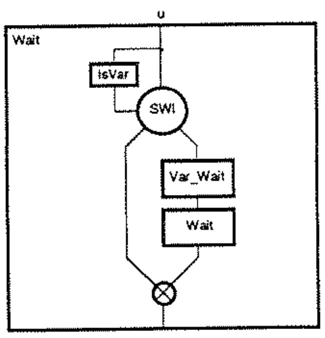

FIGURE 3 Wait Graph

Commitment

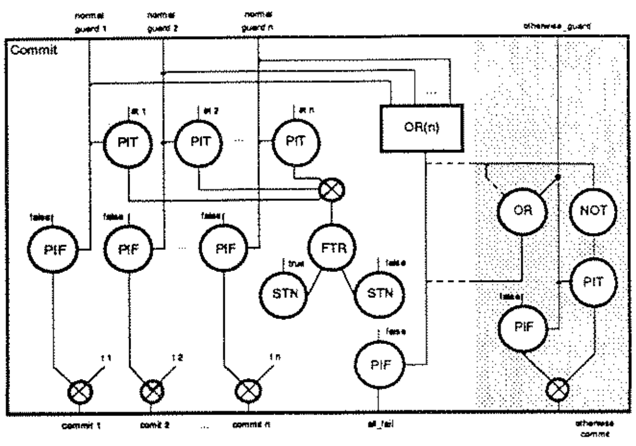

FIGURE 4 Commit Graph

Commitment is achieved by a nondeterministic merge of commit requests from all candidate clauses with true guards (should all guards fail then failure of the predicate is reported immediately). In the current implementation (figure 4) it is the first candidate clause to report a true guard that is selected for commitment, although any other selection process (e.g., random) is equally valid. Thus the selection process is biased in favour of candidate clauses with 'fast' guards, all else being equal. A single FIRST THEN REST node uses a special matching function to create a dynamic branch in the dataflow graph which results in a true token being sent to the successful candidate clause and a false token being sent to all others.

An OR tree detects the failure of all candidate clauses and generates the 'all_fail' result for the commit graph. A restricted version of 'otherwise', which allows only one otherwise clause per predicate, uses the all_fail result from normal clause guards as a success signal for the otherwise goal. Of course, an otherwise guard can still fail if other goals in the same guard should fail.

In the case of more than one otherwise clause, it is not clear what should happen when more than one potentially successful otherwise goal is present. A potentially successful otherwise goal being one for which all other head and guard unification is already successful. Ueda's suggestion, which calls for an otherwise goal to succeed only after all other guards in that predicate have failed, leads to deadlock. This can be avoided however by allowing potentially successful otherwise goals to compete for selection nondeterministically. This is in keeping with GHC's commitment process since the success of one guard implies failure of all others in that predicate. This approach can be implemented by merging otherwise requests in a manner similar to normal clause commitment.

Variables

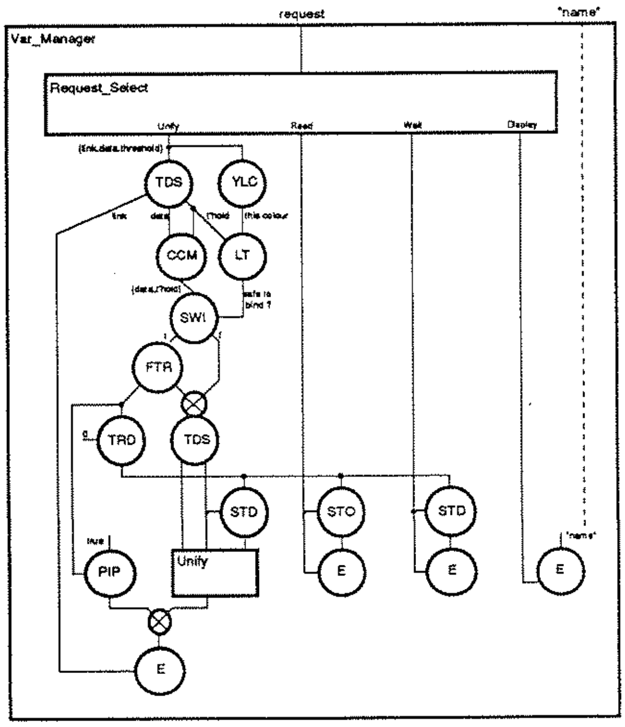

FIGURE 5 Variable Manager Graph

Variables are supported by statically assigning a manager graph for each variable that is not a formal parameter of a clause (figure 5). All requests for action upon a variable are handled by the reentrant manager graph in the same colour (being the colour of the owning clause), aided by the queueing ability of graph arcs. Care must be taken to ensure the determinacy of reentrant graphs like this, compound tokens are used here to limit critical merges on parallel data paths.

The environment tokens that represent variables participate in unification and clause substitution (parameter passing) in the same way as other data types. These tokens can be thought of as pointers to the dynamic address of the manager graph where the variable's current binding is stored. Variable managers support unification, waiting, reading (same as waiting but does not wait for the variable to be bound) and display of variables.

In GHC many unifications may attempt to bind a variable at the same time, so there is a need to prevent races and/or contention during variable binding. Various methods are available including semaphores, consistency checks, etc., however the current implementation uses a single FIRST THEN REST node in a manner similar to clause commitment. The region of the manager graph that does the actual binding is executed once only by the first Var_Unify request. Later requests read the original binding and call unify recursively.

In the full GHC implementation, the variable manager contains code to check the current binding threshold during variable unify requests. Any attempt to bind above this threshold leads to suspension, as if the variable was already bound. A single machine node (STORE DEFERRED) queues suspended wait requests for a variable and automatically transmits a non-variable binding when it occurs. The simple STORAGE node used in the 'read' section of the manager does not defer read requests and is used by the display routines to determine if a variable has been bound at all. If a variable has no binding, the STORAGE node will return a 'null' token and the variable will displayed in the traditional 'A = A' style.

At this stage there is no automatic 'dereferencing' of variables so that long variable to variable binding chains can build up, adversely effecting efficiency. In future versions these long binding chains will shortened by asynchronous dereferencing processes spawned each time a variable is interrogated, thus the chain A > B > C > 123 will be shortened to A > 123, B >123, C > 123 upon a wait or read call to A.

Unification

The unification algorithm is made up of a type selector and five subfunctions for handling different argument type combinations (figure 6). The type selector (figure 7) uses a combination of tests, switches and merges to route the arguments to one of five different output port pairs.

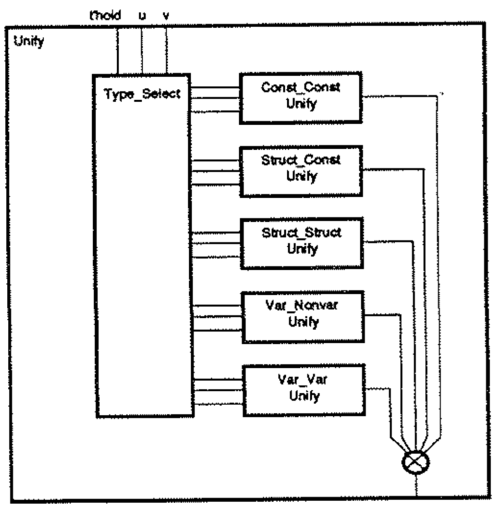

FIGURE 6 Unify Graph

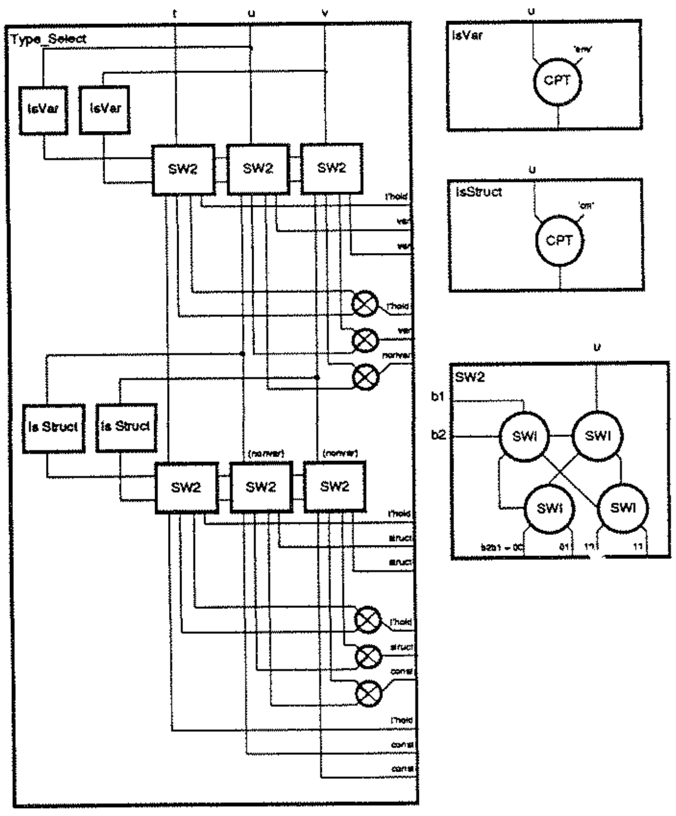

FIGURE 7 Unify Type Selector Graphs

Figure 8 gives details of the subgraphs called by unify to handle each of the different type combinations. The threshold input is not used by these routines directly but must be maintained for use during variable binding.

Const_Const_Unify uses a single low level node (EQUALS) to unify two constants by direct comparison.

A structured item can never unify with a constant and so Struct_Const_Unify always fails immediately. A PASS IF PRESENT node emits a FALSE token when fired by the unification trigger.

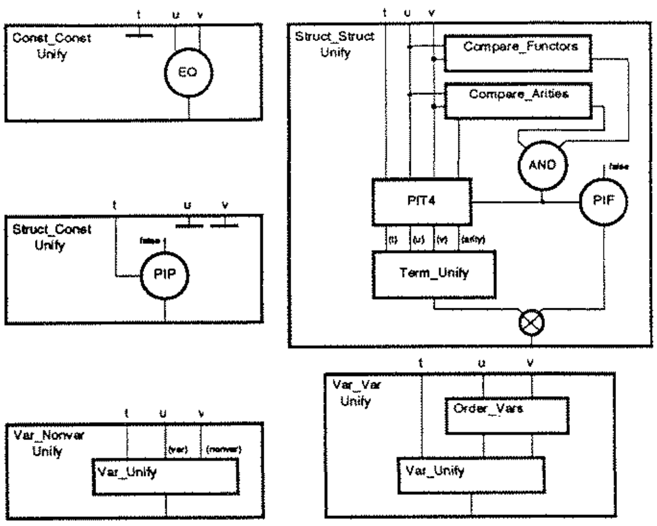

FIGURE 8 Unify Subgraphs

In Struct_Struct_Unify, two structured terms are first compared for functor and arity and if these are equal then unification recurses over pairs of corresponding structure elements in parallel. Figure 9 details the subgraphs called by Struct_Struct_Unify.

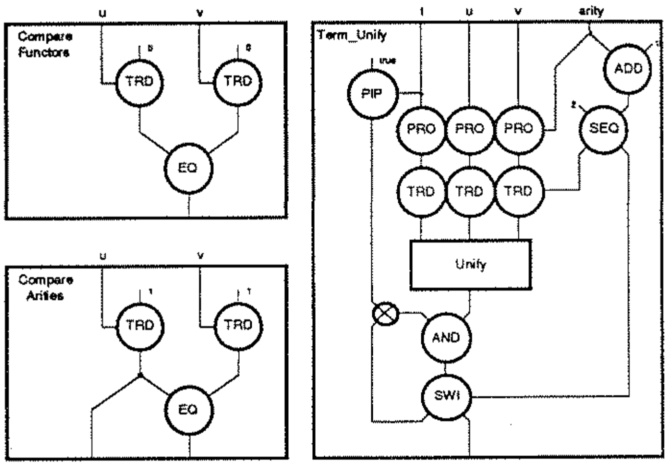

FIGURE 9 Struct_Struct_Unify Subgraphs

The Term_Unify subgraph uses PROLIFERATE, SEQUENCE and TRANSMITTED READ nodes to generate queues of corresponding element pairs. Tightly pipelined unifications of these pairs then proceeds in parallel because of the run-time distribution of function calls used in the hybrid dataflow model. Element unification results are conjoined in a tight loop under control of a SWITCH node and a boolean stream generated by the SEQUENCE node.

A variable is unified with a non-variable using the variable manager call Var_Unify. Two variables are handled by Var_Var_Unify which first imposes a partial order on the variables and then unifies the first variable to the second by calling Var_Unify. The ordering of variables is such that unbound variables in a clause always bind towards variables in an ancestor clause. Actually, it is possible to bind variables in the opposite direction, i.e., ancestor variables bind towards child variables, these two directions have different consequences for both debugging and efficiency. For two different variables in the same clause, the ordering is arbitrary bar that it must be consistent (to avoid loops). The ordering is achieved by a single node (GREATER THAN SWAP) which directly compares the variables' environments.

Theory of Operation

Translation to Intermediate Form

The intermediate form/assembly language used by the CSIRAC II system is a block structured dataflow language called 'i2' [4]. Each i2 function call is either a call to a user defined function or a primitive machine node. Nodes with special matching functions are indicated as mf.nf where mf is the matching function and nf is the node function. The following i2 translation of a clause from a GHC prime number generator will be used to illustrate the operation of the GHC system in more detail. The corresponding dataflow graph is given in figure 10.

Example of clause translation:

GHC clause:

filter(P, [X|Xs], Ys) :- X mod P =\= 0 | Ys = [X|Ys1], filter(P, Xs, Ys1).

i2 translation:

```
1       define filter_3_2(trig p1 p2 p3 commit) -> guard result;
2           begin
3             |** HEAD **
4             ylc(trig) -> thold;
5             (p1) -> P;
6             wait(p2) -> wp2;
7             cap(X 'cm {cv 1 "." i8 2}') -> lit_1;
8             cap(Xs lit_1) -> lit_2;
9             unify(thold wp2 lit_2) -> head;
10            (p3) -> Ys;
11            |** GUARD **
12            wait(X) -> sys_arg_1;
13            wait(P) -> sys_arg_2;
14            mod(sys_arg_1 sys_arg_2) -> sys_arg_3;
15            ne(sys_arg_3 'i32 0') -> sub_guard;
16            |** BODY **
17            cap(X` 'cm {cv 1 "." i8 2}') -> lit_4;
18            cap(Ys1 lit_4) -> sys_arg_4;
19            Var_Unify(trig` Ys` sys_arg_4) -> sys_arg_5;
20            filter_3(trig` P` Xs` Ys1) -> sys_arg_6;
21            and(sys_arg_5 sys_arg_6) -> result;
22            |** COMMIT **
23            and(head sub_guard) -> guard;
24            pit(trig commit) -> trig`;
25            (trig`) -> thold`;
26            pit(P commit) -> P`;
27            pit(X commit) -> X`;
28            pit(Xs commit) -> Xs`;
29            pit(Ys commit) -> Ys`;
30            |** LOCAL VARIABLES **
31            make_var(X,'cv 1 "X"',thold);
32            make_var(Xs,'cv 2 "Xs"',thold);
33            make_var(Ys1,'cv 3 "Ys1"',thold`);
34          end;
```

Each clause operates in its own colour (which is determined dynamically as part of the standard function call mechanism) and all variables created in that clause also have that colour. Variable parameters do not exist as clause owned variables but are passed by data flow directly through the call interface (e.g., in line 5 parameter p1 is aliased directly to the local variable P – it would also be correct, but far less efficient, to create P as a local variable and unify p1 with P). Non-variable formal parameters are created as literals and unified with their corresponding actuals (e.g., the structured term [X|Xs] is created in lines 7 and 8, and then unified with wp2 in line 9). In this case suspension is mandatory should the actual parameter be a variable and so the parameter is 'waited on' until it is non-variable (line 6). This is more efficient than calling unify directly, although that would also succeed since an attempt to bind a variable parameter would defer due to suspension and the variable would be waited on at that stage.

All clauses, goals and some system predicates have a trigger argument that is used to generate literals and thresholds. In full GHC translations the trigger has a special value, being the colour of the calling/parent clause. This colour is used as a threshold value for determining when attempts to bind external variables should suspend. In line 4 the colour of the trigger is used to generate a threshold for the guard of the current clause, but in line 25 the trigger itself is passed to the body as its threshold upon commitment. Thus a committed body can bind variables up to the threshold of its parent clause. In flat GHC the threshold of every ancestor clause is always zero (and is thus not needed) since the path of commitment always extends back to the main goal, whereas in full GHC the path of commitment may terminate in the goal of an ancestor clause (recall that flat GHC does not allow user defined predicates to appear as guard goals).

Note the static evaluation of the arguments of =\= in lines 12-14. As part of this evaluation, variables are waited on in expectation of them becoming numbers (type checking occurs at run time). The second argument is evaluated and planted as a machine node literal in line 15. The conjunction of clause head unification and guard goal evaluation defines the clause guard and is sent as a commit request to an external manager via the clause output arc 'guard' (line 1 and figure 10). A boolean token will arrive on the clause input arc 'commit' indicating the success or failure of the commit request. This token controls the generation of body triggers and thresholds and the passage of parameters and guard variables into the body (lines 23-29).

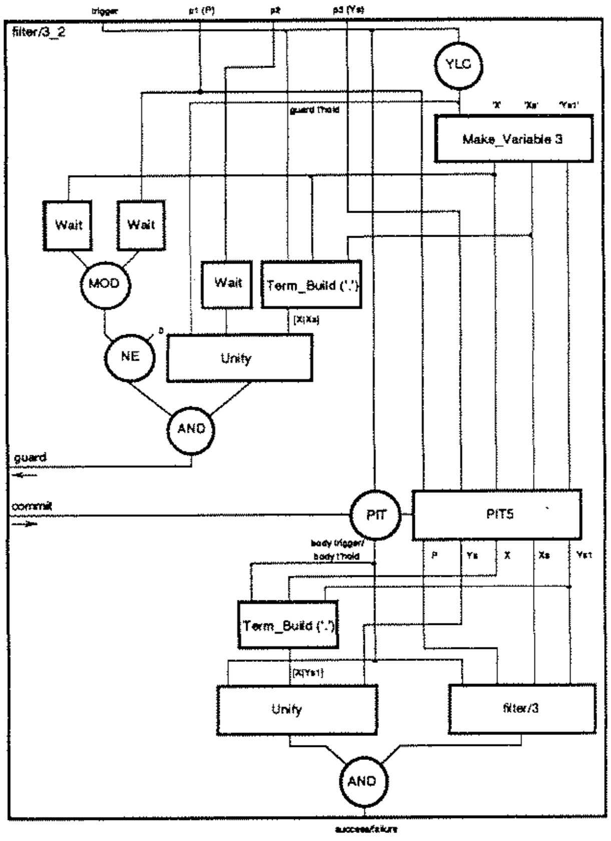

FIGURE 10 Dataflow Graph of Clause 'filter/3(2)'

Body variables (i.e, variables used only in the body of a clause) are created after commitment by the body threshold (thold', as used to create Ys1 in line 33). Waiting for commitment reduces the work done by clauses which fail to commit although it may eventually prove that a more eager approach is justified. One reason for the current lazy body evaluation is that a garbage collection scheme has not been implemented yet.

The conjunction of body goals forms the final result returned to the parent goal in the form of a simple boolean token (line 21). This and other clauses that make up the definition of the filter/3 predicate are called by a common handler which also includes the commit subgraph, see figures 1 and 4.

Conclusions / Future Work

The GHC implementation described here has been running programs on the RMIT/CSIRO dataflow machine simulators since early 1989. Early performance characteristics are promising with even trivial GHC programs generating useful run-time concurrency. However, several aspects of the system require further attention and templates are currently being designed for the following additions:

- Garbage Collection.
- Variable Dereferencing.
- Eager Body Evaluation.
- List Optimisation.
- Dynamic Evaluation.
- Compile Time Analysis (mode declarations, etc.).

Implementation of these features is already well understood with the possible exception of eager body evaluation. Eager evaluation has significant implications for garbage collection and resource management. It is also related to partial evaluation within the dataflow framework where the extra code required to control resource utilisation and non-strict evaluation must be carefully weighed against possible efficiency gains.

There is scope for very significant performance improvements with detailed compile time analysis, particularly when aided by GHC's mode declaration system. The handling of variable unification, especially in full GHC, is clearly very costly compared to a simple dataflow approach, however, using compile time analysis and mode declarations it will be possible to identify and optimise dataflow variables (i.e., variables which are assigned once). A dataflow variable will not require a manager graph at all, instead its definition can be sent directly to its uses. The performance of GHC programs compiled in this way should be highly competitive with that of programs written in more traditional functional/dataflow languages.

Acknowledgements

The author is indebted to Dr. Kotagiri Ramamohanarao of Melbourne University for his invaluable assistance and advice in designing the GHC templates described here. Thanks also to Jacek Gibert for his contribution to the initial development of the GHC system described in this paper.

Glossary of Node Functions

CPT	Compare Type	Logical type comparison.
EQ	Equal	Logical value comparison.
NE	Not Equal	
AND	Logical AND	
SWI	Switch	Redirects arg0 according to boolean arg1.
PIP	Pass If Present	Passes arg0 on arrival of arg1 (for graph synchronisation and generation of literals).
PIF	Pass If False	Passes arg0 if arg1 is false, else consumes arg0.
PIT	Pass If True	Passes arg0 if arg1 is true, else consumes arg0.
TRD	Transmitted Read	Reads elements of vectors/compound tokens.
PRO	Proliferate	Generates arg1 copies of arg0.
ADD	Arithmetic Addition	

MOD	Modulus	Evaluates arg0 mod arg1.
SEQ	Sequence Generation	Generates arg1-1 'falses' followed by a 'true'.
SRL	Set Return Link	Creates a return address for function calls.
E	Exit	Sends a function result (arg0) to a return address (arg1).
STO	Storage	Permanent (non-destructive) storage.
STD	Deferred Storage	As above, but queues read requests until non-empty.
FTR	First Then Rest	Diverts the first token in a queue away from all others.
YLC	Yield Colour	Returns the dynamic colour of its argument.
STN	Set Name	Sends arg 0 to address arg 1 without altering the current colour.
TDS	Transmitted Distribute	Splits the compound on input 0 into *n* components where *n* is the number of outputs on the node. Unused fields are sent as a compound token to the last output.
CCM	Create Compound	Creates a two element compound token from its two inputs.
CAP	Compound Append	Appends arg0 to the end of the compound token arg1.

References

[1] K. Ueda, 'Guarded Horn Clauses',
D.Eng. thesis, University of Tokyo Graduate School, March 1986.

[2] M. W. Rawling, 'Implementation and Analysis of a Hybrid Dataflow System',
M. Eng. thesis, Royal Melbourne Institute of Technology, March 1988.

[3] D. Abramson and G.K. Egan, 'Design Considerations for a High Performance Dataflow Multiprocessor', RMIT technical report TR 112-73-R, Aug. 1988.

[4] G.K. Egan, M. W. Rawling and N.J. Webb, 'I2: An Intermediate Language for the RMIT Dataflow Computer', RMIT technical report TR 112-68-R, Dec. 1988.

[5] G.K. Egan, 'The RMIT Dataflow Computer: Token and Node Definitions',
RMIT technical report TR 112-60-R (internal), Dec. 1988.

[6] S. Gregory, 'Design, Application and implementation of a Parallel Logic Programming Language',
Ph. D. thesis, Imperial College of Science and Technology, London, Sep. 1985.