TR-248

Multiple Reference Management in Flat GHC

by

T. Chikayama and Y. Kimura

March, 1987

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Multiple Reference Management in Flat GHC

Takashi Chikayama and Yasunori Kimura

ICOT
21F. Mita Kokusai bldg., 4-28, Mita 1, Minato-ku,
Tokyo 108, Japan

Abstract

This paper describes an implementation scheme of flat GHC which maintains information in pointers on whether the referenced data object has multiple references paths or not. With this multiple reference information, several optimization techniques become available: Reclamation of no longer required storage area without applying the general garbage collection, destructive array element update, etc. By keeping information on the pointers rather than in the referenced objects, no extra memory accesses are required for maintenance of the information. This is most advantageous in multiple processor implementations where pointers may point to an object in a non-local memory area. The representation and maintenance scheme for the multiple reference information is presented along with an abstract machine for flat GHC and its instruction set augmented with the feature. Preliminary results on its effectiveness and overhead measured on a experimental implementation are also given.

1 Introduction

In the execution of logic programming languages, data structures are used not so many times before being discarded as in procedural languages. As logic programming languages do not allow destructive updates, a new structure must be created each time a small modification to a structure is required. Because of this, memory consumption speed is much higher than in, for example, (impure) Lisp.

In case of sequential Prolog, this problem is partly solved by the backtracking feature. Backtracking undoes bindings inside structures, which can be bound to different values in the execution of alternatives; the same structure can be reused with some modification. Moreover, on backtracking, memory area can be reclaimed in a stack-like manner without the costly general garbage collection.

However, in case of so-called committed choice languages [1,5,6] without backtracking, memory consumption ratio is quite high and might be problematic from the following two viewpoints:

- As memory area is consumed up rapidly, garbage collection must be executed more frequently.
- The working set size becomes larger resulting in lower cache memory hit ratio or higher paging overhead.

There are two ways to solve this problem.

- Incremental reclamation of garbages.
- Incorporating destructive operations.

Lisp systems solve the problem by allowing explicit destructive operations (such as `RPLACD`) in source programs. However, putting responsibility to the programmers is quite dangerous in concurrent programming languages, as it is much harder to know which portion of the program has or has not completed its execution at a given time. It is desirable to recognize automatically when a certain storage area has become inaccessible from the program. With the knowledge, the memory area occupied by a no longer accessible data structure can be reclaimed, or, if *similar* data structure should be created at that time, the new one can be obtained by partially rewriting the original data structure in a destructive manner.

This paper proposes a method to maintain information during the execution on whether two or more reference paths exist for a data structure. When the *last* reference path to a structure is somehow known to have become no longer required, the structure can be reclaimed or destructively overwritten and reused immediately.

The rest of the paper concentrates on an implementation of so-called *flat* GHC (FGHC), a subset of GHC[6] in which no nested goals in the passive part are allowed. In FGHC, goal reductions can be regarded as an indivisible primitive operation, as far as maintenance of multiple reference information is concerned.

Only one bit in each pointer is used for keeping multiple reference information. This scheme has an admitted deficiency that, once multiple references are created,

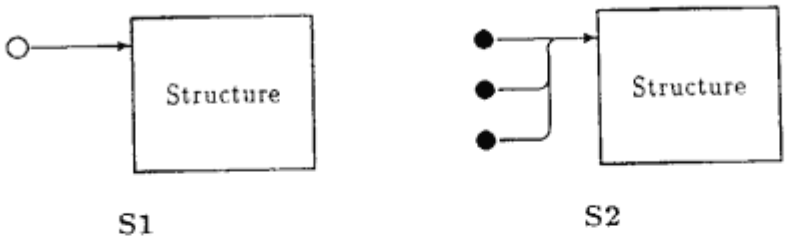

Figure 1: MRB of References to a Structure

it cannot detect when the number of references decreases. Conventional reference counting scheme is advantageous in this point. The merit of the MRB scheme is that it requires no extra memory accesses for maintaining the information. This is most effective in multiple processor implementations where pointers may point to an object in a non-local memory area.

2 Representation

One additional bit in pointers is used to keep the multiple reference information. It is called the *multiple reference bit*, or MRB in short.

When a data object is accessed from the program, it is through a certain chain of pointers starting from a root register. If *any* of the pointers in the chain have their MRB *set*, this reference path is called a *multiple reference path* or an MRP; otherwise, i.e., if all the pointers in the chain have their MRB *reset*, it is a *single reference path*, or an SRP.

The meaning of MRP or SRP depends on what kind of object the reference path is for. Primitive operations for execution of FGHC manipulates MRB so that the conditions described in this section should be maintained.

2.1 Data Structures

The MRB of pointers in reference paths to a data structure (not a variable cell) is maintained so that one of the following two conditions is satisfied.

S1: There exists only one reference path to the structure which is an SRP.

S2: All the reference paths to the structure are MRP's.

Figure 1 shows valid states of reference paths to a structure. Arrows starting with a white circle represent SRP's; those with a black circle represent MRP's

2.2 Uninstantiated Variable Cells

The MRB of pointers in reference paths to an uninstantiated variable cell is for knowing whether the referenced variable cell has two or less references to it. The MRB is maintained so that one of the following conditions is satisfied (figure 2).

U1: There exist exactly two SRP's to the cell.

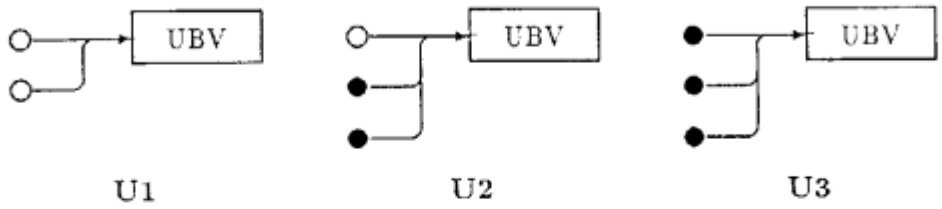

Figure 2: MRB of References to an Uninstantiated Variable

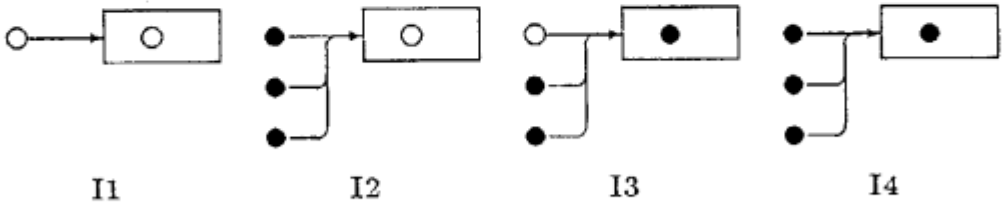

Figure 3: MRB of References to an Instantiated Variable

U2: There exists only one SRP to the cell. All the other paths, if any, are MRP's.

U3: All the reference paths are MRP's.

2.3 Instantiated Variable

The MRB of pointers in reference paths to an already instantiated variable is maintained in combination with the MRB of the data stored in the variable cell. For an already instantiated variable, one of the following conditions should apply.

I1: The MRB of the data in the cell is reset and there exists only one reference path to the cell which is an SRP.

I2: The MRB of the data in the cell is reset and all the reference paths to the cell are MRP's.

I3: The MRB of the data in the cell is set and there exists only one SRP to the cell. There may be other reference paths to the cell but all of them are MRP's.

I4: The MRB of the data in the cell is set and all the reference paths to the cell are MRP's.

Figure 3 shows valid states of an already instantiated variable. The MRB of the data stored in the cell are represented by a white or a black circle, which represent *reset* and *set* status, respectively.

3 Applications

Various optimization schemes become available using the multiple reference information. Here, two typical such optimizations, storage reclamation and destructive array element update are described.

3.1 Storage Reclamation

Certain memory area can be reclaimed without using the general garbage collection mechanism. Memory area may be reclaimed when the last reference path to the area is *consumed*. A reference path to a data structure is known to be the last path when it is an SRP, i.e., when **S1** applies. A reference path to an already instantiated variable cell is known to be the last path when **I1** applies, i.e., when it is an SRP and the MRB of the cell is *reset*.

Reference paths are *consumed* in the following ways.

Dereference: When variable values are required, reference pointers between variable cells are dereferenced until the final result (an atomic value, a pointer to a data structure, or a pointer to an uninstantiated variable cell) is reached. As GHC has no backtracking feature, the dereferenced result can be stored back to the original place (on a register or in a structure element). If the dereference result should be stored back, variable cells in the reference chain will never be used afterwards through this path. Thus, the reference paths to them used in this dereference are *consumed*.

Structure Unification: If a goal is reduced after one of its arguments is unified with some structure in the passive part, the reference path to the structure given as an argument is *consumed* in the unification. Consider an example:

```
p([X|Y]) :- true | q(X, Y).
```

The reference path to the list cell unified with `[X|Y]` will never be used after the reduction.

Unification with a Void: When an argument is unified with a void variable, the reference path passed as the argument is *consumed*.

Elements of Reclaimed Structures: When a data structure can be reclaimed, all the pointers in the elements of the structure can never be accessed through the structure. Thus, reference paths through them are *consumed*.

3.2 Destructive Update of Array Elements

If mutable arrays are to be introduced into GHC, its pure logical element update operation might be defined as follows:

`update(Old_array, Position, Data, New_array)`

A new array is created which is the same as `Old_array` except that the element at index `Position` is replaced by `Data`.

Without the multiple reference information, the implementation cannot know whether the version of the array before the update have other reference paths to it or not. Thus, two versions must be kept somehow. The most straightforward but costly solution is to copy the whole array. Sophisticated schemes such as [2] can reduce most of this cost, but it still has a considerable overhead.

Using the multiple reference information, whether the version of the array before the update has any other reference paths or not can be known immediately. If there are none, reusing the same structure by destructively rewriting the updated element will work correctly. This scheme can realize basically the same efficiency as the scheme proposed in [7] without any complicated analysis by the compiler nor introduction of new notions such as *built-in processes*.

4 Maintenance of MRB

This section describes primitive operations for FGHC execution designed to maintain the conditions described above.

4.1 Creation of Data Structures

In execution of FGHC, data structures are only created as arguments of body goals: Calls of usual predicates or the built-in unifier "=". Consider an example:

```
reconc([W|X],Y,Z) :- true | reconc(X,[W|Y],Z).
append([W|X],Y,Z) :- true | Z = [W|U], append(X,Y,U).
```

A reduction by the first clause creates a new list cell `[W|Y]`, and a reduction by the second clause creates a new `[W|U]`. Note that new structures are never created in the passive part in case of *flat* GHC.

When a new structure is created as an argument of a body goal, there can be no other reference paths to the structure and thus the MRB of the pointer to the structure can be reset (**S1**).

4.2 Creation of Variable Cells

Creation of a new variable cell is required when body goals have a variable not appearing in the head. For example, a reduction by the following clause creates a new variable cell for `I`.

```
compile(S, O) :- true | parse(S, I), generate(I, O).
```

When a new variable is created, there can be no other reference paths to the variable except those appearing in the same clause. Thus, the MRB of pointers to the variable cell is determined by the number of occurrences of the variable in the body part. Consider an example:

```
p :- true | q(X, Y), r(Y, Z), s(Z, Z).
```

Here, the MRB of the pointers to `X` and `Y` should be *reset* (**U1**) and that to `Z` should be *set* (**U3**). Note that two occurrences of the variable `Z` in the same goal `s(Z,Z)` are counted as they are, i.e., two.

4.3 Dereference

When reading the contents of an already instantiated variable cell (including variable cells where a pointer to another variable cell is stored), the MRB of the dereference result will be determined by the following rules.

I1: When the reference path to the variable cell is an SRP and the MRB of the cell is *reset*, MRB of the result should be *reset*. The reference path through the variable cell can never be accessed from the program after this dereference. Instead, a new path resulted from this dereference becomes accessible. As they cancel out each other, the number of reference paths does not change.

I2, I3 or I4: When the reference path to the variable cell is an MRP and/or the MRB of the cell is *set*, the MRB of the result should be *set*. In all such cases, there may be multiple reference paths to the variable cell before the dereference. Thus, there may remain other reference paths to the dereferenced result after the dereferencing procedure.

4.4 Instantiation of Variables

Here, the word *instantiation* means not only giving some concrete value (atomic or structured) to a variable, but also binding a variable to another variable by connecting them with a reference chain.

On instantiation of a variable, MRB of the data stored in the variable cell can be *reset* if and only if the reference to the variable cell is an SRP and the MRB of the data to be stored there is *reset*. In all other cases, the MRB of the stored data should be *set*. This can happen only when either **U1** or **U2** applies before the instantiation.

U1: There exists at most one other SRP to the variable cell besides one used for the instantiation. In this case, after the binding, there will remain only that SRP to the variable cell (**I1**).

U2: There may be many other reference paths to the variable cell but all of them are MRP's. In this case, after the binding, there may remain many reference paths to the variable cell and to the data stored in the variable cell, but all such paths are MRP's (**I2**).

4.5 Duplicating References

When a pointer given as an argument is *duplicated*, in other words, when a variable occurring in the head appears twice or more in the body of the clause, the reference to the object will be duplicated. Consider an example:

```
p(X) :- true | q(X), r(X).
```

Here, the reference path given as the argument `X` is distributed to both of the body goals `q` and `r`. In such cases, MRB of the duplicated pointers should be *set* to make reference paths through them an MRP. When the duplicated reference is originally an MRP, then the duplication only adds another MRP, which will never violate the conditions stated above. When it is originally an SRP, there are two cases depending on what is referenced.

- If a data structure is referenced, i.e., if **S1** originally applied, reference duplications is a state transition to **S2**.
- If an uninstantiated variable cell is referenced, then there can be two cases depending on other reference paths to the same variable.
  - If **U1** originally applies, i.e., if there exists only one other SRP, the reference duplication is a state transition to **U2**.
  - If **U2** originally applies, i.e., if all other reference paths to the same variable cell are MRP's, the reference duplication is a state transition to **U3**.
- If an already instantiated variable cell is referenced, then there can be two cases depending on the MRB of the data stored in the variable cell.
  - If **I1** originally applies, i.e., if the MRB of the referenced cell is *reset*, the reference duplication is a state transition to **I2**.
  - If **I3** originally applies, i.e., if the MRB of the referenced cell is *set*, the reference duplication is a state transition to **I4**.

4.6 Retrieval of Structure Elements

Retrieval of elements of a structure occurs when an unification with an explicit data structure is effected, as in:

```
p([X|_]) :- true | q(X).
```

If the reference path to the entire structure is an MRP, i.e., if S2 applies for the entire structure, there can be other paths to the same structure and thus creating a new reference to a structure element (X in the above example) is creating a new access path in addition to one via the structure. Thus, the MRB of the retrieved value should be *set* regardless of the MRB stored in the structure.

If the reference path to the entire structure is an SRP, i.e., if S1 applies, then the structure itself can never be accessed after the reduction. Retrieving a structure element will add one new reference path to the element, but, the reference path through the structure element will be discarded. As they cancel out each other, there is no change in the number of reference paths in total. Thus, in such cases, the MRB of the structure elements can be simply copied as the MRB of the retrieved result.

5 Abstract Machine

This section describes the characteristics of an abstract machine for FGHC implementation which realizes the above described multiple reference management. The abstract architecture is somewhat similar to one proposed by Warren for sequential Prolog execution [8], which will be called WAM in short in what follows. Principal differences from WAM are described here.

5.1 Multiple Reference Bit Management

All the memory words and all the argument/temporary registers have one extra bit called MRB, which is maintained to satisfy the conditions described in the previous sections.

A goal reduction is considered to be an indivisible unit as far as the MRB management is concerned, i.e., all the required conditions on MRB are maintained correctly *before* and *after* each reduction, but not necessarily *during* a goal reduction.

A dedicated register called S is used for unification of structure elements as in WAM. S also has its own MRB field. In the read mode structure element unification, it keeps the information on whether the reference path to the whole structure was an SRP or an MRP. It might also be reasonable to change the *opcode* dispatch address depending on the MRB of S. As there is no write mode unification in the passive part of FGHC, the same hardware mechanism used in implementing read/write mode in WAM may be utilized.

5.2 Storage Allocation

The MRB scheme enables the reclamation of areas for independent data structure without general garbage collection. Thus, the method of allocating new structures always at the heap top is inappropriate. Free lists for data structures and variable cells are maintained. It might be reasonable to use free lists only for frequently used structures and use different scheme (such as the buddy system) for larger

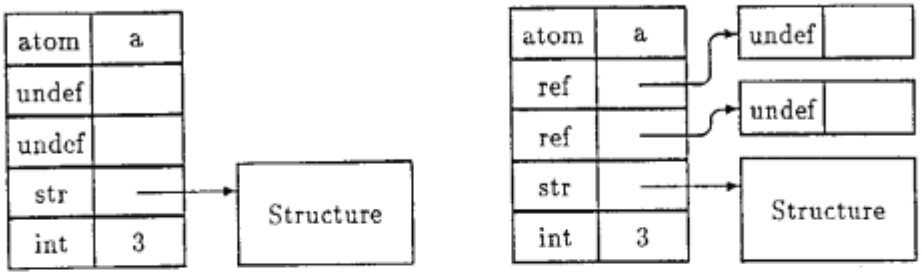

Figure 4: Variables as Structure Elements

structures. As structures are not allocated at the heap top, S register is used not only for *read* mode unifications but also for *write* mode unifications.

Unlike the original WAM, structure elements should not be used directly as variable cells to avoid fragmentation. When a structure element should be initiated as a variable, a new variable cell is allocated separately, and a pointer to the cell is stored in the element (figure 4).

5.3 Garbage Stack

When a garbage candidate is found in the passive part, its reclamation should be postponed until the end of the passive part, since the clause might not be selected due to suspension or failure afterwards.

When a data structure is explicitly given in the source code, appropriate instructions for reclamation can be generated at the top of the body part. However, when general unification (unification of two variables) is specified in the source code, it might unify two structures, during which unpredictable amount of garbage may be detected. Such garbages are pushed on to a stack called the *garbage stack*. The stack is emptied each time a goal reduction is initiated and each time a candidate clause fails or suspends. Its contents are reclaimed by an explicit instruction at the top of the body part.

In many FGHC programs, general unification of two structure is rather exceptional. Thus, implementations *without* this garbage stack mechanism might be good enough for most programs.

6 Abstract Instruction Set

The abstract instruction set described here is similar to that of WAM. Notable differences are:

- Passive unification instructions will suspend when instantiation of variables are required to accomplish the unification.
- The passive part is compiled so that argument registers are never destroyed before commitment.

Table 1: Passive Unification Instructions

`wait_value` X_j, A_i	`wait_reused_value` X_j, A_i
`wait_variable` X_j, A_i	`wait_constant` *Constant*, A_i
`wait_list` A_i	
`read_value` X_j	`read_reused_value` X_j
`read_variable` X_j	`read_constant` *Constant*

- Instructions are arranged so that the MRB information can be correctly maintained.

Only the instructions concerning unification and maintenance of MRB will be described here. For about structures, only the instructions handling lists are described. Extending them to treat general structures is straightforward.

6.1 Passive Unification

The passive unification instructions corresponding to `get` and `unify` instructions in WAM are called `wait` and `read`. These instructions are similar to their counterparts in WAM, but they suspend when instantiation of variables are required in the unification.

General unification instructions have two variants: With and without the word `reused`. In both of them, pointers to a structure in A_i or in a structure element used in the unification are pushed to the garbage stack if they are an SRP. Ones *without* the word `reused` also push such pointers in X_j; ones *with* the word will not. The latter is for when the unified data (X_j) is used later and thus its contents should not be reclaimed.

6.2 Storage Reclamation

Instructions described in this section are for reclaiming storage which are known to have become inaccessible.

`collect_stack`
: Reclaim storage area pointed from the garbage stack. This instruction will be generated only when some general passive unification instructions are in the passive part.

`collect_list` A_i
: Reclaim the list structure pointed from A_i if A_i is an SRP. This instruction is generated when A_i is unified with a list structure in the passive part.

`collect_value` A_i
: Reclaim storage area pointed from A_i if A_i is an SRP. This instruction is generated when A_i is unified with a void variable in the passive part. When A_i references a structure, not only that structure but also its substructures referenced through an SRP are reclaimed recursively.

Table 2: Argument Preparation Instructions

`put_variable` X_j, A_i	`set_variable` X_j, G_i
`put_constant` *Constant*, A_i	`set_constant` *Constant*, G_i
`put_marked_variable` X_j, A_i	`set_marked_variable` X_j, G_i
`put_list` A_i	`set_list` G_i
`put_value` X_j, A_i	`set_value` X_j, G_i
`put_marked_value` X_j, A_i	`set_marked_value` X_j, G_i
`write_variable` X_j	`write_marked_variable` X_j
`write_value` X_j	`write_marked_value` X_j
`write_constant` *Constant*	

6.3 Active Unification

For simplicity, only the most general instruction is introduced here for active unification. Its arguments should be prepared using the argument preparation instructions explained below.

`get_value` X_j, A_i

General unification instruction of a variable X_j with another variable A_i. During the unification, structures or variable cells known to have become inaccessible from the program are reclaimed.

For efficiency, actual implementations may provide instructions which combine argument preparation and unification. They will be very much like unification instructions of WAM except that the MRB maintenance should be taken into consideration.

6.4 Argument Preparation

Arguments of active unification and the body goal executed immediately after the reduction are prepared on argument registers designated by A_i; `put` and `write` instructions, which correspond to put and write mode `unify` instructions of WAM, are used. For other body goals, arguments are set in goal records in the goal queue designated by G_i; `set` instructions are used instead of `put`, in this case. This is similar to the "*goal stacking*" method mentioned in [8]. In case of Prolog, the goal stacking method may have considerable overhead when backtracking takes place; this is not a problem with backtrack-free GHC.

Instructions *with* the word "`marked`" in their names have the same functions as those *without* it except that the MRB of the destination is always set. They are for reference duplication.

6.5 Optimization

When a data structure is reclaimed and allocation of another structure of the same size is required at the same time, the reclaimed structure can be reused

```
p(X, X, Y, Y) :- true | q(X).
    wait_reused_value A1,A2
    wait_value A3,A4
    collect_stack
    execute q/1
```

```
p([X|Y]) :- true | q(X, Y).
    wait_list A1
    read_variable X3
    read_variable A2
    collect_list A1
    put_value X3,A1
    execute q/2
```

```
p(X) :- true | q([X]), r(X).
    enqueue_goal r/1
    set_list G1
    write_marked_value A1,G1
    write_constant []
    put_marked_value A1,A1
    execute q/1
```

```
p :- true | q(X), r(X), s(X).
    enqueue_goal s/1
    set_marked_variable A1,G1
    enqueue_goal r/1
    set_marked_value A1,G1
    put_marked_value A1,A1
    execute q/1
```

Figure 5: Compilation Examples

immediately. For example, two instructions `collect_list` and `put_list` can be combined into one.

`reuse_list` A_i, A_j

The same as "`put_list` A_i" except that if the MRB of A_j is *reset*, use the list cell A_j points to in stead of allocating a new one.

When some elements of the reused data structure can also be reused as they are, for example, when the *car* part of the reused list cell is the same as what should be stored in the newly allocated list cell, the following instruction might be useful.

`replace_cdr` A_i, A_j, A_k

The same as the instruction sequence "`reuse_list` A_i, A_j; `write_value` A_k", except that, if A_j can be reused, the *car* part is directly reused without rewriting.

Using this instruction, the object code as shown in Figure 6 can be obtained for list concatenation.

7 Comparison with Reference Counting

Reference counting is a widely used scheme for immediately detecting when a data structure becomes free. However, the general reference counting scheme has the following drawbacks.

- An extra word for counting references is required for each data object. In case of GHC, each variable cell must have an extra word which almost doubles the required memory space.
- Counting references up or down requires extra memory reads and writes, which may slow down the execution considerably.

```
append([W|X],Y,WZ) :- true | WZ=[W|Z], append(X,Y,Z).
```

before

```
wait_list A1
read_variable X4
read_variable X5
collect_list A1
put_list X6
write_value X4
write_variable X7
get_value X6,A3
put_value X5,A1
put_value X7,A3
execute append/3
```

after

```
wait_list A1
read_variable X4
read_variable X5
replace_cdr A1,X6,X4
write_variable X7
get_value X6,A3
put_value X5,A1
put_value X7,A3
execute append/3
```

Figure 6: Optimization of append

Time and space overhead of the MRB scheme is much smaller than the general reference counting scheme. One bit MRB in stead of one word count reduces the memory overhead considerably. By keeping the multiple reference information on *pointers* rather than in the data, no additional memory accesses are required. Especially, for shared objects in multiple processor systems, inter-processor communication with *locking* is required for reference counting; none such for MRB.

As the MRB scheme only distinguishes reference counts of 1 from *many*, once multiple references are made, the system cannot recognize even when all of them are disposed. However, in languages such as FGHC which do not have destructive assignment nor multiple environments, data objects often have only one reference paths to it. Thus, in implementations of such languages, this deficiency is not so serious as in procedural languages.

8 Using MRB for Other Languages

Unfortunately, using the MRB scheme for logic-based languages other than FGHC is not as easy or not as effective.

- In sequential or OR-parallel Prolog, multiple references are indispensable and probably much more frequent than in GHC.
- In *non-flat* languages such as *full* GHC, *full* CP or Parlog, goal reductions cannot be regarded as an indivisible operation. The management of MRB will be much more complicated.
- In *flat* CP, whether a unification is passive or active is determined at the execution time. The object code must be ready to handle all the possible cases, which may increase the code size and also slow down the execution.

The MRB scheme may be very effective for dataflow languages. Actually, FGHC *is* a dataflow language in some sense.

Table 3: Execution Statistics of Test Programs

cells	total	peak	ratio	left	ratio	extra†
primes list	5,372	499	.09	0	.00	—
variable	5,469	501	.09	0	.00	5,372
queens list	14,208	8,705	.61	8,515	.60	—
variable	17,561	15,164	.86	9,747	.56	6,544

† Number of extra variable cells required to separate elements from their parents.

9 Test Implementation

An FGHC system with the MRB scheme proposed in this paper has been implemented. Table 3 shows the execution statistics of a couple of test programs. The scheduling strategy is simple depth-first, and no suspension took place in their execution. Tested programs are:

Primes: A prime number generator which generates all the prime numbers not greater than 500. One process generates all the natural numbers and other dynamically created processes filter non-prime numbers out, and, finally, one process prints out the result. A typical application of the stream communication programming style.

Queens: A program which gives all solutions of the 8 queens problem. A typical all solution search program. Probably one of the worst cases for the MRB scheme.

In **primes**, all the allocated list cells could be collected during the execution. In **queens**, about 60% of them could not be collected. As no list cells are accessible when the program terminates, they are the cells which had multiple reference paths.

For about variable cells, 98% of them allocated in **primes** are additionally required to incorporate the MRB scheme for separating them from their parent list cell. In case of **queens**, it is 37%. Taking this and sizes of cells in consideration, the ratio of the peak memory requirement using the MRB scheme and the total memory requirement without the scheme are 13.8% for **primes** and 82.6% for **queens**. It might be said that:

- For stream parallel programs, the MRB scheme is very effective.
- For search programs, the MRB scheme is not so effective. Still, it is good enough to compensate the memory overhead imposed by separate allocation of variable cells.

10 Conclusion

A new implementation scheme of FGHC which maintains multiple reference information in pointers is described. Maintenance of the information requires no extra memory accesses. With the information, several optimizations such as storage reclamation without garbage collection or destructive array update become possible. From preliminary experiments, storage reclamation by the scheme is proven to be very effective for stream parallel programs.

Acknowledgements

This work is initiated being stimulated by the optimization effort for stream merging by Ito and Kuno[3]. Discussions with members of KL1, PIM and Multi-PSI groups of ICOT were very much profitable in developing the authors' initially vague idea.

References

[1] K. L. Clark and S. Gregory. *Parlog: A Parallel Logic Programming Language.* Research Report TR-83-5, Imperial College, 1983.

[2] L. H. Eriksson and M. Rayner. Incorporating mutable arrays into logic programming. In *Proceedings of the Second International Conference on Logic Programming*, pages 76–82, Uppsala, 1984.

[3] N. Ito, M. Kishi, E. Kuno, and K. Rokusawa. The dataflow-based parallel inference machine to support two basic languages in KL1. In *IFIP TC-10 Working Conference on Fifth Generation Computer Architecture*, UMIST, Manchester, 1985.

[4] J. Levy. A GHC abstract machine and instruction set. In *Proceedings of the Third International Conference on Logic Programming*, 1986.

[5] E. Y. Shapiro. *A Subset of Concurrent Prolog and Its Interpreter.* ICOT Technical Report TR-003, ICOT, 1983.

[6] K. Ueda. *Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard.* ICOT Technical Report TR-208, ICOT, 1986.

[7] K. Ueda and T. Chikayama. Efficient stream/array processing in logic programming language. In *Proceedings of FGCS'84*, ICOT, 1984.

[8] D. H. D. Warren. *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, 1983.